Integration of off-line and on-line scheduling for admission control
Implementation description and guide

Institutionen för Datateknik, Mälardalens högskola

Tomas Lennvall
2000
Integration of off-line and on-line scheduling for admission control
Abstract

In this paper we study the small os kernel called SHaRK in more detail to investigate if it's possible to use it as a testbed in the self-evolving dependable real-time systems project. As it turns out SHaRK is very suitable because it supports many of the demands we have on the platform that we are going to use. As a case study we also do the integration of the off-line and on-line scheduling policies, that is we have implemented the slot-shifting scheduling policy, which also supports admission control that is another important aspect of the self-evolving system.

The paper also contains a guide for how to implement further algorithms and other extensions we might want to make in the future.

1 Introduction

As stated earlier we went out to investigate for suitable platforms to use in the distributed system that is our testbed. This is for the self-evolving dependable real-time project where we need platforms that fulfills certain demands. These demands are support for many of the common scheduling algorithms, easy to extend, free source and free of charge or very low cost. We set out to investigate a few different platforms and SHaRK from Scuola Superiore S. Anna Pisa, Italy was one of them. As it turns out ShaRK supports all of our demands, its free of charge, open source and has many common scheduling algorithms already implemented. There was some other platforms investigated but this paper only deals with the ShaRK kernel, which we also found to be most suitable of the options.

Through the implementation of the slot shifting scheduling algorithm we got admission control into the kernel. This admission control is very important when we have tasks in the system that might migrate between the nodes seeking admission for scheduling. And we must make sure we only admit the tasks that we can guarantee before scheduling them.

The rest of the paper is organised as follows; Chapter 2 deals with the integration of off-line and on-line scheduling policies and also admission control. Chapter 3 deals with the implementation and how to implement further extensions in the kernel. Chapter 4 is an appendix with the code for the slot-shifting algorithm.

2 Integration of off-line and on-line scheduling and admission control

The slot shifting scheduling algorithm consist of two major parts; an off-line part and an on-line part. Off-line is where the static task schedules are created, this is done with an off-line scheduler which takes a taskset with complex constraints and produces a feasible schedule. Because finding an optimal solution to most sets of complex constraints is an NP hard problem the complexity reduction must be performed offline. NP hard problems has the consequence that the algorithms are heuristic in nature and only produces suboptimal solutions. The offline scheduler produces a schedule for each node in the system, and a schedule is simply the start times and deadlines of the tasks on the node.

Online part, or during runtime the algorithm uses a extended EDF algorithm to schedule the tasks in the schedules produced in the offline phase. This EDF algorithm, called two level EDF, has the basic idea as EDF and so schedules tasks with “normal” level, but give priority –“priority level” to a offline task when it need to start at latest. Thus the CPU is not completely available for runtime tasks, but reduced by the amount allocated for offline tasks. This leads to the requirement that we need to know the amount and location of resources available after the offline tasks are guaranteed.

This means we have to make some additional offline preparations. After offline scheduling, and all the tasks have start times and deadlines the deadlines are sorted on each node. The schedule is divided into a set of disjoint execution intervals for each node. Spare capacities are used to represent the amount of unused resources defined for these execution intervals. An interval has an end equal to the deadline of a task, so several tasks with the same deadline constitute one interval. The start is defined as the maximum of the end of the previous interval or the earliest start time of the task. Thus the intervals differ from an execution window which can overlap, where these intervals cannot because of the maximum condition. The amount of spare capacities in an interval equals the lengt of the interval minus the sum of execution time in that interval. Tasks may execute in an interval prior to the one they have been assigned to and then they “borrow” spare capacity from the earlier interval. Obviously the amount of unused resources in an interval cannot be less than zero, but we sometimes use negative spare capacity to increase runtime efficiency and flexibility. Thus we can represent information about the amount and distribution of free resources in the system in a flexible way.

The actual runtime scheduling is preformed locally on each node in the system. If the spare capacities for the current interval, sc(I) > 0 we use EDF in the “normal level”. But if the sc(I) = 0 that indicates that a guaranteed task has to be executed or else a deadline violation will occur. To take runtime tasks into account we decrease the spare
capacities in the intervals involved when guaranteeing that task. So guaranteed runtime tasks are treated exactly in the same way as offline tasks, and the spare capacity reflects this after the guarantee. The spare capacity is also updated after each scheduling decision; if the CPU remains idle or an aperiodic task executes the spare capacity is decreased. If a task from a later interval executes the spare capacity in the current interval is decreased and the spare capacity in the task interval is increased. If that task also “borrowed” spare capacity from a previous interval it must increase that intervals spare capacity since it no longer “borrows” it. This mechanism ensures that negative spare capacities become zero or positive at runtime.

2.1 Implementation

The implementation was straightforward just creating the interface as specified by SHaRK, but no guest functions needs to be created since slot shifting takes care of all tasks itself. And if it wouldn’t take care of all the tasks itself it requires a U=1.0 to handle the sc’s in a correct way. The tasks a slot-shifting module can accept are static, hard aperiodic and soft aperiodic tasks. The hard aperiodic and soft aperiodic model already existed in the SHaRK kernel, but the static task was added. It has three important parameters, release time, computation time and a relative deadline.

To represent the intervals an array was chosen foremost for the speed when accessing it. There is one drawback with it and that is when interval splitting occurs, then the array has to shifted so that the interval can be inserted which can be costly.

Slots are handled as events, at the end of a slot an event fires and an event handler takes over. The handler immediately sets the event for the next slot end so that the overhead is included in the current slot, namely in the beginning. Other than that the event handler takes care of all the business that needs to be done in slot shifting; look at the newly arrived aperiodic tasks, handle interval calculations and the cleanup at the end of each interval. The cleanup part can consist of interval merging if it has been splitted earlier; otherwise it just resets the variables to their original value.

When initialising the module the interval file is read and the structure created, here it should also be possible to specify the multiple of LCMs needed to cover the longest deadline.

Since the application defines the tasks and then sends them to the scheduling module for creation it is easy to adjust it to different admission policy. If a task is rejected at creation or activation time the deadline can be extended and then the task can try to be accepted again. Other more refined methods may also be implemented if needed.

2.2 Problems

Besides the usual issues with learning a new system there was few problems that appeared.

One problem was that if a deadline is longer than a LCM a wraparound occurs, and that new deadline calculated might be in an interval that has not yes been visited. That is we may split an interval in the next LCM but actually we reuse the same data structures and this causes a serious problem. A solution is to allocate several LCMs, up to a multiple that can handle the longest possible deadline. Or we can say that dl < LCM but that can be a problem if we end up in the same interval as we currently are visiting.

Interval splitting in the array is probably costly, in the worst case the first interval is splitted and every other interval has to be shifted to make room for it. But a solution is to use a list instead. This has a slightly higher cost in accessing and traversing but a significantly lower cost for inserting intervals.

3 Implementation guide

3.1 Basic information

This part of the document will detail how to write a scheduling module in the SHARK kernel. First there is some basic explanation about task models and scheduling modules, and how they relate. Secondly there is more detail about scheduling modules, what to do when writing one. Lastly there are some tips and conventions that are good to use when writing a module. There is an appendix showing an example of the EDF scheduling module, both the ideas and the code.
3.2 Task models and scheduling modules

A task model is simply put a struct that defines some attributes for a specific type of task, like deadline, period and so on. There are several different task models already defined in the kernel, hard tasks, soft tasks and so on. They are listed in the model.h file in the include/kernel directory.

The kernel uses a scheduling module to schedule tasks; the kernel doesn’t know anything about what kind of algorithm the modules implements. Therefore all scheduling modules have a common interface, that interface is the only thing known by the kernel. The interface reflects the events that can occur to a task during its lifetime. The scheduling module is basically also a struct that is called the level descriptor; here all the internal information specific to the module is kept. Some examples of different scheduling modules exist in the include/modules and kernel/modules directories.

The relation between task models and scheduling modules is what kind of tasks a module can handle. This is seen in the interface implemented by a module, it reflects how a certain task is handled in different situations. So a scheduling module also tells what kind of tasks it can handle, so all tasks created knows what module it belongs to.

When scheduling occurs the kernel asks the scheduling module for a task to schedule, and the module returns a task according to its algorithm.

3.3 Scheduling modules

A scheduling module consists of an interface that can be grouped into three classes; Level calls, Task calls and Guest calls. The kernel uses Level calls to invoke module dependent operations, like scheduling, statistics and so on. Task calls are the modules way of handling different task events, like creation, activation and destruction. The kernel is the only one that can invoke task calls. Guest calls are for modules that need some task scheduling service from another module. This can be an aperiodic server that needs to put the tasks it handles into another module, called the master. This master takes care of the task and schedules it according to its algorithm, typically EDF or RM. In this case the aperiodic server is the guest module and it uses the guest calls in the master module.

The interface of a scheduling module is detailed below, and that is what needs to be implemented to create a scheduling module.

Note that all functions accept the LEVEL l in parameter. The LEVEL l parameter is a pointer to a level descriptor of the current scheduling module. A level descriptor is a data structure containing all the information about the scheduling module, similar to a task model. So all that information can be reached from within the functions. There also exists one or more registration functions to register the scheduling module, and if necessary to set some internal parameters.

3.3.1 Level calls

int level_accept_task_model(LEVEL l, TASK_MODEL *m);
This function is called when a task is created and checks if this scheduling model, l, accepts the task model, *m. It should return 0 if it can handle the task, and –1 otherwise.

void level_status(LEVEL l);
This function is not important, and if necessary it can print some information about the scheduling module to the console.

PID level_scheduler(LEVEL l);
This function is the actual scheduler of the scheduling module, it should return the PID of a task chosen to run by the algorithm. The function should not modify the pointer to the running task, handle timers for deadline, capacity exhaustions, and ... and lastly not change the ready queue or something similar. This is because it’s not guaranteed that the task chosen by the algorithm is the one that is going to be executed by the kernel (in this lab it is, but these conventions should be followed anyway).

int level_guarantee(LEVEL l, bandwidth_t *freebandwidth);
This function represents the on-line guarantee for the tasks owned by the scheduling module (not guest tasks). The freebandwidth parameter represents the remaining available bandwidth left by modules with a level number less than l. The guarantees are of the type that uses the U factor, and used bandwidth must be subtracted from the freebandwidth parameter. The function should return 0 if guaranteed, and –1 otherwise.
3.3.2 Task calls

Task calls are associated with the life of a task, and to easier understand the task part of the interface a simple view of the events occurring in a tasks lifetime is shown.

![Task lifetime](image)

Figure 3.2.1: Task lifetime.

Figure 3.2.1 shows a basic example in a task life, it is created, activated and then pre-empted by another task. The events generated are:

- **create** This event is generated at task creation. The scheduling module initiates the data structures for the task activation.
- **activate** This event is generated when a task is explicitly activated through a call to a user primitive. The event authorizes the scheduling module to insert the task into the set of schedulable tasks.
- **dispatch** This event is generated when a task, after it has been scheduled (not shown in figure) is really executed. The scheduling module updates the data structures to register the execution of the task.
- **epilogue** This event is generated all the times the task is pre-empted by another task. The scheduling module usually reinserts the task into the set of schedulable tasks.
- **end** This event is generated when a task ends its execution. The scheduling module is authorized to free the task descriptor and any data allocated for the task.

The following figure shows the events involved in the lifetime of a periodic task.

![Periodic task](image)

Figure 3.2.2: A periodic task.

The new events introduced from the periodic task are:

- **endcycle** The endcycle event is the termination of the current activation of a task. A primitive inserted into the task code generates it. If the task is periodic the scheduling module will reactivate it when the next...
period starts. If the task is aperiodic this event implies that the task will await an explicit activation (that is an activate event).

**sleep**  The sleep event ends the current activation, like the **endcycle** event. The difference is that the task will await an explicit activation in any case. The internal reactivations programmed by the scheduling module have to be erased.

**reactivation**  This event is not created directly by the user with a primitive, but is handled internally by the scheduling module that handles the task. This event has to reactivate the task, and it usually delivered at the end of a period.

At a first look the sleep and endcycle events look exactly the same, but there are is a small semantic difference. It has to do with pending activations, that is an activate event is delivered before the task has ended its previous instance. If the pending activations are saved (which is up to the scheduling module) the difference between the events are:

- The endcycle event does not stop the execution of the task if it has a pending activation. It tries to execute a lost instance.
- The sleep event always stops the task, ignoring any pending activation.

The PID parameter is the task descriptor that gets allocated for each new task when the task is created.

```c
int task_create(LEVEL l, PID p, TASK_MODEL *m);
```

The kernel calls this function to create a task into the scheduling module. The m parameter is the task model of the task (this model is guaranteed to be of an accepted model type for this scheduling module). The function should return 0 if the task can be created, and -1 otherwise.

```c
void task_detach(LEVEL l, PID p);
```

This function is called when an error has occurred during task creation. This function should free internal data structures allocated during the creation of the task.

```c
int task_eligible(LEVEL l, PID p);
```

This function is only used width aperiodic servers. It should return 0 if the task chosen by the level scheduler can be schedule, -1 otherwise.

```c
void task_dispatch(LEVEL l, PID p, int nostop);
```

The kernel calls this function when a task is about to be executed. The function should prepare the task for execution, for example by removing it from the ready queue, and so on. The nostop parameter is the result of the expression `exec == exec_shadow`. The function should not change the state of the task or the `exec` and `exec_shadow` variables.

```c
void task_epilogue(LEVEL l, PID p);
```

The kernel calls this function when a running task is pre-empted or the capacity of task is exhausted (execution is finished). The function may reinsert the task into the ready queue, change the state of the task, and so on if it still is active.

```c
void task_activate(LEVEL l, PID p);
```

This function represents the activation of a task, which is making the task schedulable. Typically this functions sets the initial status of a task, insert the task into a ready queue, save activations (if current activation is not finished), set deadline events and so on.

```c
void task_extract(LEVEL l, PID p);
```

This function is used if the task becomes blocked (not possible in this laboration).

```c
void task_insert(LEVEL l, PID p);
```

This function is also used if the task becomes blocked.

```c
void task_endcycle(LEVEL l, PID p);
```

This function implements the end of a task instance, if the task has no pending activations the effect is to disable the schedulability of the task (remove it from the ready queue). Then the task may be reactivated at the next period, or explicitly reactivated by the user depending on task type.
**void task_sleep(LEVEL l, PID p);**

This function also implements the end of a task instance exactly like the `task_endcycle` function, but with the difference that pending activations are discarded.

**void task_end(LEVEL l, PID p);**

This function implements the task termination, meaning when a task is finished. So this function should cleanup everything in the scheduling module that has to do with the terminated task. The task should be inserted into the freedesc queue so that the PID can be reused for other tasks.

**void task_delay(LEVEL l, PID p, TIME tickdelay);**

This function implements a delay in the execution of a task. Tickdelay represents the minimum delay of the task. Typically the task is set to SLEEP state, removed from the ready queue, and resource reclaiming may be done.

### 3.3.3 Guest calls

The guest calls will not be used in this laboration so they will not be shown.

### 3.3.4 Registration function(s)

The registration function is called when the system starts, and it should initiate all the internal data structures of the scheduling module. The registration function typically consists of four parts:

- **Part 1:** Allocates a level descriptor that will be used to initiate the scheduling module.
- **Part 2:** Initiates the kernel interface of the scheduling module.
- **Part 3:** Initiates the private data structures of the scheduling module, and if needed posts a function to be called just after the system has gone into multitasking mode.
- **Part 4:** This is the part that executes the function posted in part 3, if it was posted.

The function itself is typically named `register_level()` but that is not necessary, and it’s also possible to have some in parameters.

### 3.4 Tips and conventions when writing a module

After all the functions have been created the init file and functions must be created, in this lab an init file already exists. All that’s needed is to do some small changes in that file to init the correct kernel module. The init file is located in the `kernel/init` directory and is named `artc_init.c`. In the skeleton file all the calls have the prefix ARTC_ in their name, but that is not necessary and it can be changed to whatever.

An application file also exists, and in it is the main function. The main function starts the kernel with the modules specified in the init file and creates and adds all the tasks to the module. The application file exists in the `examples/kernel` directory and is called `artc_app.c`.

When writing a module there are some functions and data structures that can be used to save time. One example is the queue data structure, the queue is a linked list and it has methods that operate on the queue already defined. There are more examples, to many to list here but they exist in the `func.h` file in the `include/kernel` directory.

The scheduling module created should exist in both an .h and a .c file. The .c file should be located in the `kernel/modules` directory, and the .h file is in the `include/modules` directory. The names of internal functions have to be static and have the form MODULENAME_FUNCTIONNAME; in the skeleton file (artc.c) it is ARTC_FUNCTIONNAME.

To create a scheduling module there are certain functions that must be defined, these functions define the interface for the module. In this laboration the scheduling module should only handle one task model and so the guest functions can be ignored. That means all functions with a name consisting of guest should be left unchanged from the skeleton file, where these functions generate an exception if called.
The task model used in this lab is the **VALUE_HARD_TASK_MODEL** it is a firm aperiodic task with a wcet, drel (relative deadline) and a value that can be used for scheduling purposes.

### 3.4.1 Debugging

Debugging in the SHARK kernel can be quite tricky since there is no way of stepping through the code (like Borland C++ and Visual C++). Instead printouts have to be done, but there is also one issue with the printouts. If you print long enough and reach the bottom of the screen, the kernel will get an exception and terminate the program. This is because of not yet completed graphics drivers and unfortunately they will not be completed before the course. So you have to print more information on each row instead of always ending with a newline in the print call. To print the `kern_printf()` function must be used which works exactly as the ordinary printf() but it writes directly to the screen memory.

### 3.4.2 How to run an application

In order to run an application the DOS extender X must be used. X takes the application .exe file as an in parameter. After the application has finished execution DOS will be in control again. So when applications are crated the result is an .exe file that can be executed with X.

### Appendix

This is the source code for the slot shifting module both the .c file and .h file.

```c
#include <modules/slash.h>
#include <ll/stdio.h>
#include <ll/stdlib.h>
#include <ll/string.h>
#include <ll/math.h> /* for ceil(...) */
#include <ll/ll.h> /* for memcpy(...) */
#include <kernel/model.h>
#include <kernel/descr.h>
#include <kernel/var.h>
#include <kernel/func.h>

#define eslsh_printf kern_printf
#define slsh_printf printk

/* Keeps information about static and guaranteed tasks */
typedef struct { int est; int dabs; int interval; } SLSH_task;

/* Status used in the level */
#define SLSH_READY MODULE_STATUS_BASE
#define SLSH_WAIT MEMBER_STATUS_BASE + 1
#define SLSH_IDLE MODULE_STATUS_BASE + 2
#define SLSH_WCET_VIOLATED MODULE_STATUS_BASE + 3

/* defines */
#define MAX_INTERVALS 1000 /* 1000 intervals is max, for now */

/* the level redefinition for the SLOT SHIFT level */
typedef struct { level_des l; /* the standard level descriptor */

/* task lists */
```

---

9
SLSLH_task tasks[MAX_PROC]; /* est and dl's for static and guaranteed task */

QUEUE idle_statics; /* finished static tasks */
QQUEUE unspecified; /* tasks with only a wcet */

/* the Intervals list */
SLSLH_interval intervals[MAX_INTERVALS];
int current; /* current interval */
int last; /* last interval */

int slot; /* slot shifting time */
TIME slot_length; /* slot length in real system time */
int LCM; /* length (in slots) of offline schedule */

int SLH_slot_event; /* save the event */
}

} SLSLH_level_des;

/* Which task models the Slot-Shifting module accepts */
static int SLSLH_level_accept_task_model(LEVEL l, TASK_MODEL* m)
{
    HARD_TASK_MODEL* h;
    SOFT_TASK_MODEL* s;

    /* Check the models */
    switch(m->pclass)
    {
        case STATIC_PCLASS: /* offline scheduled tasks */
            return 0;
        case HARD_PCLASS: /* hard aperiodic tasks */
            h = (HARD_TASK_MODEL*) m;
            if(h->drel != 0 && h->wcet != 0) /* must be set */
                return 0;
            break;
        case SOFT_PCLASS: /* soft aperiodic tasks */
            s = (SOFT_TASK_MODEL*) m;
            if(s->wcet != 0) /* must be set */
                return 0;
            break;
        default:
            return -1; /* Not accepted model */
    }

    return -1; /* Not accepted model */
}

static void SLSLH_level_status(LEVEL l)
{
    kern_printf("Level status not implemented\n");
}

/* check if some tasks are ready, return 0 if ready, -1 otherwise */
static int SLSLH_R(SLSLH_task* tasks)
{
    int s;

    /* for all static tasks */
    for(s = 0; tasks[s].est != -1; ++s)
    {
        
    }
if(proc_table[s].status == SLSH_READY)
    return 0;
return -1;

/* check if unspecified exists, return 0 if it exists, -1 otherwise */
static int SLSH_T(QQUEUE unspecified)
{
    if(unspecified.first != NIL)
        return 0;
    else
        return -1;
}

/* return the sc in an interval */
static int SLSH_sc(SLSH_interval* intervals, int i)
{
    return intervals[i].sc;
}

/* return a static task from current interval or a guaranteed task */
static PID SLSH_staticOrGuaranteed(SLSH_level_des* lev)
{
    int lowest_dl = 0; /* lowest dl found */
    PID pid = 0;      /* static or guaranteed task */
    int t;

    /* Decide according to EDF, go through all static & guaranteed tasks */
    for(t = 0; t < MAX_PROC; ++t)
    {
        /* static tasks */
        if(proc_table[t].pclass == STATIC_PCLASS)
        {
            /* static task must belong to current interval */
            if(lev->tasks[t].interval == lev->current)
            {
                /* only ready tasks */
                if(proc_table[t].status == SLSH_READY)
                {
                    /* a new lower dl was found */
                    if(lev->tasks[t].dabs < lowest_dl)
                    {
                        lowest_dl = lev->tasks[t].dabs;
                        pid = t;
                    }
                }
            }
        }
        /* guaranteed tasks */
        else if(proc_table[t].pclass == HARD_PCLASS)
        {
            /* only ready tasks */
            if(proc_table[t].status == SLSH_READY)
            {
                /* a new lower dl was found */
                if(lev->tasks[t].dabs < lowest_dl)
                {
                    lowest_dl = lev->tasks[t].dabs;
                    pid = t;
                }
            }
        }
    }
}
/* for all tasks */
return pid;
}

/* return a static task among the candidates, all ready statics */
static PID SLSH_candidates(SLSH_task* tasks)
{
    int lowest_dl = 0;
    PID pid;
    int t;

    /* Use the EDL algorithm again to decide which task to run */
    for(t = 0; t < MAX_PROC; ++t)
    {
        /* only static tasks */
        if(proc_table[t].pclass == STATIC_PCLASS)
        {
            /* only ready tasks */
            if(proc_table[t].status == SLSH_READY)
            {
                /* a new lower dl was found */
                if(tasks[t].dabs < lowest_dl)
                {
                    lowest_dl = tasks[t].dabs;
                    pid = t;
                }
            }
        }
    }

    return pid;
}

/* decrease the sc in a interval by amount */
void SLSH_decSc(SLSH_interval* intervals, int i, int amount)
{
    intervals[i].sc -= amount;
}

void SLSH_incSc(SLSH_interval* intervals, int i, int amount)
{
    intervals[i].sc += amount;
}

/* swap the sc between intervals, also consider intervals with negative sc */
void SLSH_swapSc(SLSH_interval* intervals, int current, int task_interval)
{
    /* decrease the sc in the current interval */
    SLSH_decSc(intervals, current, 1);

    /* update the other interval(s) */
    if(intervals[task_interval].sc < 0) /* negative sc */
    {
        /* special case, increase next interval sc by 1 and also current interval (borrowing) */
        if(task_interval == current + 1)
        {
            SLSH_incSc(intervals, task_interval, 1);
            SLSH_incSc(intervals, current, 1);
        }
    }
}
else /* increase every interval sc that is negative between current and 
task_interval */
{
    while(task_interval > current && intervals[task_interval].sc < 0)
    {
        SLSH_incSc(intervals, task_interval, 1);
        task_interval--;
    }
}
else /* ordinary swapping */
    SLSH_incSc(intervals, task_interval, 1);

/* The scheduler, decides which task to run. */
static PID SLSH_level_scheduler(LEVEL l)
{
    SLSH_level_des* lev = (SLSH_level_des *)(level_table[l]);
    PID pid;

    /* The scheduler choses among static, guaranteed (hard aperiodic) and 
    unspecified (soft aperiodic) tasks */
    /* no ready tasks and no sc, execute idle task */
    if(SLSH_R(lev->tasks) == 0 && SLSH_sc(lev->intervals, lev->current) == 0)
        return NIL;
    /* must execute a static from current intervall or a guaranteed task */
    else if(SLSH_R(lev->tasks) > 0 && SLSH_sc(lev->intervals, lev->current) == 0)
    {
        /* If unspecified exist, execute it according to FIFO order */
        if(SLSH_T(lev->unspecified) == 0)
        {
            SLSH_decSc(lev->intervals, lev->current, 1); /* decrease sc by 1 */
            return (PID)qq_getfirst(&lev->unspecified);
        }
        else /* No unspecified, execute task from candidates (statics) */
        {
            pid = SLSH_candidates(lev->tasks);

            /* sc needs to be swapped */
            if(lev->tasks[pid].interval != lev->current)
                SLSH_swapSc(lev->intervals, lev->tasks[pid].interval, lev->current);

            return pid;
        }
    }

    kern_printf("(SLSH s)" Jessie
        return NIL;
}

/* not used, slot-shifting handles all guarantees itself, it handles all 
bandwidth */
static int SLSH_level_guarantee(LEVEL l, bandwidth_t *freebandwidth)
{
    *freebandwidth = 0;
    return 1;
}
/* get the interval that x is in */
static int SLSH_getInterval(SLSH_interval* intervals, int x, int last)
{
    int i;

    /* search through the intervals */
    for(i = 0; i <= last; ++i)
    {
        /* I is in the interval where start is smaller or equal and end is bigger */
        if(intervals[i].start <= x && x < intervals[i].end)
            return i;
    }
    return -1;
}

/* get the start of the interval I */
static int SLSH_intervalStart(SLSH_interval* intervals, int I)
{
    return intervals[I].start;
}

/* split interval I into two parts, slow because of copying. OBS!!! no check if
 there is
  enough space in the intervals array */
static void SLSH_splitInterval(SLSH_level_des* lev, int I, int dabs)
{
    SLSH_interval left_interval;
    int i;

    lev->last++;

    /* move every interval above and including I */
    for(i = lev->last; i > I; --i)
        memcpy(&lev->intervals[i], &lev->intervals[i - 1], sizeof(SLSH_interval));

    /* Left interval start, end and length */
    left_interval.start = lev->intervals[I].start;
    left_interval.end = dabs;
    left_interval.length = left_interval.end - left_interval.start;

    /* Right interval (uses old interval struct) start and length end remains as
 the old value */
    lev->intervals[I + 1].start = dabs;
    lev->intervals[I + 1].length = lev->intervals[I + 1].end - lev->intervals[I + 1].start;

    /* check if sc still exists in the right interval */
    if(lev->intervals[I + 1].length - lev->intervals[I + 1].maxt > 0)
    {
        lev->intervals[I + 1].sc = lev->intervals[I + 1].length - lev->intervals[I + 1].maxt;
        left_interval.sc = left_interval.length; /* the whole interval is free, for now... */
    }
    else /* no sc in the right interval */
    {
        lev->intervals[I + 1].maxt = lev->intervals[I + 1].length;
        left_interval.sc = lev->intervals[I + 1].sc; /* all sc in left interval */
lev->intervals[I + 1].sc = 0;
}

/* insert the new interval */
memcpy(&lev->intervals[I], &left_interval, sizeof(SLSH_interval));

/* Reduce the sc from back to front by the wcet amount, interval splitting may be neccessary */
static void SLSH_updateSc(SLSH_level_des* lev, HARD_TASK_MODEL* h)
{
    int dabs = ceil((lev->slot + h->drel)/lev->slot_length); /* absolute
deadline of request */
    int dabs_interval = SLSH_getInterval(lev->intervals, dabs, lev->last); /*
interval where dabs is */
    int C = ceil(h->wcet/lev->slot_length); /* amount of sc to reduce */
    int sc = 0;
    int i;

    /* check if interval splitting is neccessary */
    if(lev->intervals[dabs_interval].end != dabs)
        SLSH_splitInterval(lev, dabs_interval, dabs);

    /* decrease sc in all intervals that are neccessary from dabs_interval o
current */
    for(i = dabs_interval; i >= lev->current && C > 0; --i)
    {
        if((sc = SLSH_sc(lev->intervals, i)) >= 0) /* only decrease where sc
exists */
        {
            if(sc > C) /* the last sc dec */
            {
                SLSH_decSc(lev->intervals, i, C);
                C = 0;
            }
            else /* to little sc in this interval, decrease it to 0 */
            {
                C -= SLSH_sc(lev->intervals, i);
                SLSH_decSc(lev->intervals, i, SLSH_sc(lev->intervals, i));
            }
        }
    } /* for all intervals */
}

/* the guarantee algorithm for hard aperiodic requests */
static int SLSH_guarantee(SLSH_level_des* lev, HARD_TASK_MODEL* h)
{
    int total_sc = 0;
    int temp, i;
    int dabs = ceil((lev->slot + h->drel)/lev->slot_length); /* absolute
deadline of request */
    int dabs_interval = SLSH_getInterval(lev->intervals, dabs, lev->last); /*
interval where dabs is */

    /* check if the sc up until request deadline is >= request wcet */
    /* 1. the sc of the current interal */
    total_sc = SLSH_sc(lev->intervals, lev->current);

    /* 2. the sc for all whole intervals between current and the interval
with the request deadline */
    for(i = (lev->current) + 1; i < dabs_interval; ++i)
{ if((temp = SLSH_sc(lev->intervals, i)) > 0)
   total_sc += temp;
}

/* 3. the min of sc or the execution need in the last interval */
total_sc += min(SLSH_sc(lev->intervals, dabs_interval),
   dabs - SLSH_intervalStart(lev->intervals,
   dabs_interval));

if(total_sc >= h->wcet)
{ /* update the sc in the intervals from back to front */
   SLSH_updateSc(lev, h);
   return 0;
}
else
   return -1;
}

/* check if task model is accepted and store nessecary parameters */
static int SLSH_task_create(LEVEL l, PID p, TASK_MODEL *m)
{
   SLSH_level_des *lev = (SLSH_level_des *)(level_table[l]);
   STATIC_TASK_MODEL* s;
   HARD_TASK_MODEL* h;
   SOFT_TASK_MODEL* u;

   /* if the SLSH_task_create is called, then the pclass must be a
   valid pclass. Slot-shifting accepts STATIC_TASK, HARD_TASK
   and SOFT_TASK models with some restrictions */

   /* est, dl and wcet is saved in slotlengths */
   switch(m->pclass)
   {
   case STATIC_PCLASS: /* offline scheduled tasks */
      s = (STATIC_TASK_MODEL *) m;
      lev->tasks[p].est = ceil(s->est/lev->slot_length);
      lev->tasks[p].dabs = ceil(s->dabs/lev->slot_length);
      lev->tasks[p].interval = s->interval;
      proc_table[p].avail_time = s->wcet;
      proc_table[p].wcet = s->wcet;
      break;
   case HARD_PCLASS: /* hard aperiodic tasks */
      h = (HARD_TASK_MODEL *) m;
      if(SLSH_guarantee(lev, h) == 0)
      { /* convert drel to dabs */
         lev->tasks[p].dabs = ceil((lev->slot + h->drel)/lev->slot_length);
         proc_table[p].avail_time = h->wcet;
         proc_table[p].wcet = h->wcet;
      }
      else /* task not guaranteed */
         return -1;
      break;
   case SOFT_PCLASS:
      u = (SOFT_TASK_MODEL *) m;
      proc_table[p].avail_time = u->wcet;
      proc_table[p].wcet = u->wcet;
      qq_insertlast(p, &lev->unspecified); /* respect FIFO order */
      break;
   default: /* a task model not supported */
return -1;
/* enable wcet check in the kernel */
proc_table[p].control |= CONTROL_CAP;
return 0;
}

static void SLSH_task_detach(LEVEL l, PID p)
{
    /* do nothing */
}

/* check if a task chosen by scheduler is correct */
static int SLSH_task_eligible(LEVEL l, PID p)
{
    return 0; /* if the task p is chosen, it is always eligible */
}

/************* The slot end event handler *************
static void SLSH_slot_end(void* p)
{
    SLSH_level_des* lev = (SLSH_level_des *) p;
    PID pid;
    int i;

    /* increase slot "time" by 1 */
    if(lev->slot < lev->LCM)
    {
        lev->slot++;
        /* check if new statics are ready */
        for(i = 0; lev->tasks[i].interval != -1; ++i)
        {
            if(lev->tasks[i].est <= lev->slot && proc_table[i].status == SLSH_WAIT)
                proc_table[i].status = SLSH_READY;
        }

        /* check if current (interval) needs updating */
        if(lev->current < SLSH_getInterval(lev->intervals, lev->slot, lev->last))
            lev->current++;
    }
    else /* restart from the beginning of the offline schedule */
    {
        lev->slot = 0;
        while((pid = q_getfirst(&lev->idle_statics)) != NIL)
        {
            if(lev->tasks[pid].est <= lev->slot)
                proc_table[pid].status = SLSH_READY;
            else
                proc_table[pid].status = SLSH_WAIT;
        }

        /* call for a rescheduling, reset event flag and increase slot by 1 */
        lev->slot_event = -1;
        kern_printf("*");
        event_need_reschedule();
    }
}
/* when a task becomes executing (EXE status) */
static void SLSH_task_dispatch(LEVEL l, PID pid, int nostop)
{
    SLSH_level_des *lev = (SLSH_level_des *) (level_table[l]);
    struct timespec t;

    /* the task state is set EXE by the scheduler()
    we extract the task from the unspecified queue.
    NB: we can't assume that p is the first task in the queue!!! */
    if(proc_table[p].pclass == SOFT_PCLASS)
        qq_extract(pid, &lev->unspecified);
    /* also start the timer for one slot length */
    lev->slot_event = kern_event_post(&TIME2TImESPEC(lev->slot_length, t),
                                       SLSH_slot_end, (void*) lev);
}

/* called when task is moved from EXE status */
static void SLSH_task_epilogue(LEVEL l, PID pid)
{
    SLSH_level_des *lev = (SLSH_level_des *) (level_table[l]);

    /* check if the wcet is finished... */
    if (proc_table[p].avail_time <= 0)
    {
        /* if it is, raise a XWCET_VIOLATION exception */
        kern_raise(XWCET_VIOLATION, pid);
        proc_table[p].status = SLSH_WCET_VIOLATED;
    }
    else /* the end of a slot. the task returns into the ready queue... */
    {
        if(proc_table[p].pclass == SOFT_PCLASS)
            qq_insertfirst(pid, &lev->unspecified);
        proc_table[p].status = SLSH_READY;
    }
}

/* when task go from SLEEP to SLSH_READY or SLSH_WAIT */
static void SLSH_task_activate(LEVEL l, PID pid)
{
    SLSH_level_des *lev = (SLSH_level_des *) (level_table[l]);
    WORD type = proc_table[p].pclass;

    /* Test if we are trying to activate a non sleeping task */
    /* Ignore this; the task is already active */
    if (proc_table[p].status != SLEEP && proc_table[p].status !=
       SLSH_WCET_VIOLATED)
        return;

    /* make task ready or waiting, dependong on slot (the time) for static tasks
    only*/
    if(type == STATIC_PCLASS && lev->tasks[p].est <= lev->slot)
        proc_table[p].status = SLSH_READY;
    else
        proc_table[p].status = SLSH_WAIT;

    if(type == HARD_PCLASS)
        proc_table[p].status = SLSH_READY;
}
/* insert unspecified tasks in QQUEUE and make it ready */
if (type == SOFT_PCLASS)
{
    qq_insertlast(pid, &lev->unspecified);
    proc_table[pid].status = SLSH_READY;
}

/* when a task is returned to the module from a semaphore, mutex ... */
static void SLSH_task_insert(LEVEL l, PID pid)
{
    SLSH_level_des *lev = (SLSH_level_des *)(level_table[l]);
    /* change status of task */
    proc_table[pid].status = SLSH_READY;
    if (proc_table[pid].pclass == SOFT_PCLASS)
        qq_insertfirst(pid, &lev->unspecified);
}

/* when a semaphore, mutex ... tasks a task from module */
static void SLSH_task_extract(LEVEL l, PID pid)
{
    /* Extract the running task from the level
     * we have already extract it from the ready queue at the dispatch time.
     * the capacity event have to be removed by the generic kernel
     * the wcet don't need modification...
     * the state of the task is set by the calling function
     * the deadline must remain...
     *
     * So, we do nothing!!!
     */
}

/* task has finished execution for this period */
static void SLSH_task_endcycle(LEVEL l, PID pid)
{
    /* do nothing */
}

/* the task has finished its wcet, kill task (dont kill static tasks) */
static void SLSH_task_end(LEVEL l, PID pid)
{
    SLSH_level_des *lev = (SLSH_level_des *)(level_table[l]);
    if (proc_table[pid].pclass == SOFT_PCLASS)
    {
        if (proc_table[pid].status == SLSH_READY)
            qq_extract(pid, &lev->unspecified);
    }
    else if (proc_table[pid].pclass == HARD_PCLASS)
    {
        if (proc_table[pid].status == SLSH_READY)
            lev->tasks[pid].dabs = 0;
    }
    /* static tasks: put them in idle QUEUE, reset status and avail_time */
    else if (proc_table[pid].pclass == STATIC_PCLASS)
    {
        proc_table[pid].avail_time = proc_table[pid].wcet;
        proc_table[pid].status = SLSH_IDLE;
    }
q_insert(pid, &lev->idle_statics);
)

proc_table[pid].status = FREE;
)

/* called when a task should sleep but not execute for awhile, mabe a mode change */
static void SLSH_task_sleep(LEVEL l, PID pid)
{
    /* the task has terminated his job before it consume the wcet. All OK! */
    proc_table[pid].status = SLEEP;

    /* we reset the capacity counters... only for static tasks */
    if (proc_table[pid].pclass == STATIC_PCLASS)
        proc_table[pid].avail_time = proc_table[pid].wcet;
}

static void SLSH_task_delay(LEVEL l, PID p, TIME usdelay) {
}

/** Guest Functions, slot shifing accepts no guests, so all generates exceptions **/
static int SLSH_level_accept_guest_model(LEVEL l, TASK_MODEL* m) { return -1; }

static int SLSH_guest_create(LEVEL l, PID p, TASK_MODEL *m) { kern_raise(XUNVALID_GUEST,exec_shadow); return 0; }

static void SLSH_guest_detach(LEVEL l, PID p) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_dispatch(LEVEL l, PID p, int nostop) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_epilogue(LEVEL l, PID p) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_activate(LEVEL l, PID p) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_insert(LEVEL l, PID p) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_extract(LEVEL l, PID p) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_endcycle(LEVEL l, PID p) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_end(LEVEL l, PID p) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_sleep(LEVEL l, PID p) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

static void SLSH_guest_delay(LEVEL l, PID p, TIME usdelay) {
    kern_raise(XUNVALID_GUEST,exec_shadow);
}

/****** Registration functions ******/
/*+ Registration function: */
void SLSH_register_level()
{
    LEVEL l; /* the level that we register */
    SLSH_level_des *lev; /* for readableness only */
    PID i; /* a counter */

    kern_printf("SLSH_register_level\n");

    /* request an entry in the level_table */
    l = level_alloc_descriptor();

    /* alloc the space needed for the EDF_level_des */
    lev = (SLSH_level_des *)kern_alloc(sizeof(SLSH_level_des));

    /* update the level_table with the new entry */
    level_table[l] = (level_des *)lev;

    /* fill the standard descriptor */
    strncpy(lev->l.level_name, SLSH_LEVELNAME, MAX_LEVELNAME);
    lev->l.level_code = SLSH_LEVEL_CODE;
    lev->l.level_version = SLSH_LEVEL_VERSION;
    lev->l.level_accept_task_model = SLSH_level_accept_task_model;
    lev->l.level_accept_guest_model = SLSH_level_accept_guest_model;
    lev->l.level_status = SLSH_level_status;
    lev->l.level_scheduler = SLSH_level_scheduler;
    lev->l.level_guarantee = SLSH_level_guarantee;
    lev->l.task_create = SLSH_task_create;
    lev->l.task_detach = SLSH_task_detach;
    lev->l.task_eligible = SLSH_task_eligible;
    lev->l.task_dispatch = SLSH_task_dispatch;
    lev->l.task_epilogue = SLSH_task_epilogue;
    lev->l.task_activate = SLSH_task_activate;
    lev->l.task_insert = SLSH_task_insert;
    lev->l.task_extract = SLSH_task_extract;
    lev->l.task_endcycle = SLSH_task_endcycle;
    lev->l.task_end = SLSH_task_end;
    lev->l.task_sleep = SLSH_task_sleep;
    lev->l.task_delay = SLSH_task_delay;

    /* fill the SLSH descriptor part */
    for(i = 0; i < MAX_PROC; i++)
    {
        lev->tasks[i].est = -1;
    }
lev->tasks[i].dabs = 0;
lev->tasks[i].interval = -1;

for(i = 0; i < MAX_INTERVALS; i++)
{
    lev->intervals[i].start = -1;
    lev->intervals[i].end = -1;
    lev->intervals[i].length = 0;
    lev->intervals[i].maxt = 0;
    lev->intervals[i].sc = 0;
}

lev->current = 0;
lev->last = NIL;
lev->slot = 0;
lev->slot_length = 0;
lev->slot_event = -1;

void SLSH_set_interval(LEVEL l, int start, int end, int maxt)
{
    SLSH_level_des* lev = (SLSH_level_des *)(level_table[l]);
    static int i = -1;

    i++;
    lev->intervals[i].start = start;
    lev->intervals[i].end = end;
    lev->intervals[i].length = end - start;
    lev->intervals[i].maxt = maxt;
    lev->intervals[i].sc = lev->intervals[i].length - maxt;

    lev->last = i;
}

void SLSH_set_variables(LEVEL l, TIME length)
{
    SLSH_level_des* lev = (SLSH_level_des *)(level_table[l]);

    lev->slot_length = length;
}

This file contains the scheduling module for Slot shifting.

Title:
Slot Shifting

Task Models Accepted:
STATIC_TASK_MODEL - Periodic Hard tasks that are scheduled by
an off-line scheduler, so that all guarantees regarding precedence, mutex
deadline violation is taken care of. The tasks are in an executione
schedule,
that is the order in when they become ready. They have the following
fields:
est (earliest start time), wcet and absolute deadline.

HARD_TASK_MODEL - Hard Tasks (Hard aperiodic requests)
wcet field and drel field must be != 0. They are used to set the wcet
and deadline of the tasks.
periodicity field must be APERIODIC
mit field is ignored.

SOFT_TASK_MODEL - Soft Tasks (Unspecified tasks)
wcet field must be != 0. periodicity filed must be APERIODIC
period and met filed is ignored.

Guest Models Accepted:
NONE - Slot shifting handles all tasks by itself (at this moment).

Description:
This module schedules the offline scheduled tasks according to the slot-
shifting paradigm, dividing time into slots of a fixed length and assigning
tasks to execute in those slots. Slot-shifting also keeps track of the free
bandwidth in the schedule by using disjoint intervals and sc (spare
capacity).
Each interval has a sc nr that represents the free bandwidth in that
interval,
the sc can be used by hard aperiodic tasks, static tasks from later interval
or
soft aperiodic tasks. Hard aperiodic tasks are guaranteed an incorporated in
the schedule by reduction of sc before they execute. No guarantee is
performed on the soft aperiodic tasks, they are run when no other task wants
to execute and sc is available.

Description:
This module implements the Slot shifting algorithm, by Gerhard Fohler. Slot
shifting
schedules off-line scheduled tasks and also handles hard aperiodic requests
by the
guarantee algorithm. Slot shifting can also handle soft aperiodic tasks,
called unspecified. That is tasks without a deadline.

Exceptions raised:
These exceptions are pclass-dependent...
XDEADLINE_MISS
If a task miss his deadline, the exception is raised.

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 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...

Restrictions & special features:
- This level doesn't manage the main task.
- At init time we can choose if the level have to activate
  . the wcet check
    (If a task require more time than declared, it is stopped and put in
     the state EDF_WCET_VIOLATED; a XWCET_VIOLATION exception is raised)
  . the task guarantee algorithm
    (when all task are created the system will check that the task_set
     will not use more than the available bandwidth)
- The level use the priority and timespec_priority fields.
- A function to return the used bandwidth of a level is provided.
- The guest tasks don't provide the guest_endcycle function

/**

#ifndef __SLSH_H__
```c
#define __SLSH_H__
#include <ll/ll.h>
#include <kernel/config.h>
#include <sys/types.h>
#include <kernel/types.h>
#include <modules/codes.h>

/*#define min(a, b) ((a) < (b) ? (a) : (b))*/
#define TIME2TIMESPEC(T, TS) 
  ((TS).tv_sec = ((T)/1000000)), 
  ((TS).tv_nsec = (((T)%1000000) * 1000)), 
  (TS)
/* define the interval struct */
type def struct {
  int start; /* start of interval */
  int end; /* end of interval */
  int length; /* Length of interval */
  int maxt; /* maximum execution time in interval */
  int sc; /* spare capacity in interval */
} SLSH_interval;

/** Registration function: */
void SLSH_register_level();
void SLSH_set_intervals(LEVEL l, int start, int end, int maxt);
void SLSH_set_variables(LEVEL l, TIME length);
#endif
```