Practical session on Cheddar and Marte OS

Master SE

September 2009

1 Case study

We consider the simplified Flight Control System of Fig. 1. This system controls the attitude, the trajectory and the speed of an airplane. It consists of 7 tasks which execute repeatedly at a periodic rate. The fastest sub-system executes at 10ms, it acquires the state of the system (angles, position, acceleration) and computes the feedback law of the system. The order is then sent to the flight control surfaces. The intermediate sub-system is the piloting loop, it executes at 40ms and determines the acceleration to apply.. The slowest sub-system is the navigation loop, it executes at 120ms and determines the position to reach. The required position of the airplane is acquired at the slow rate.

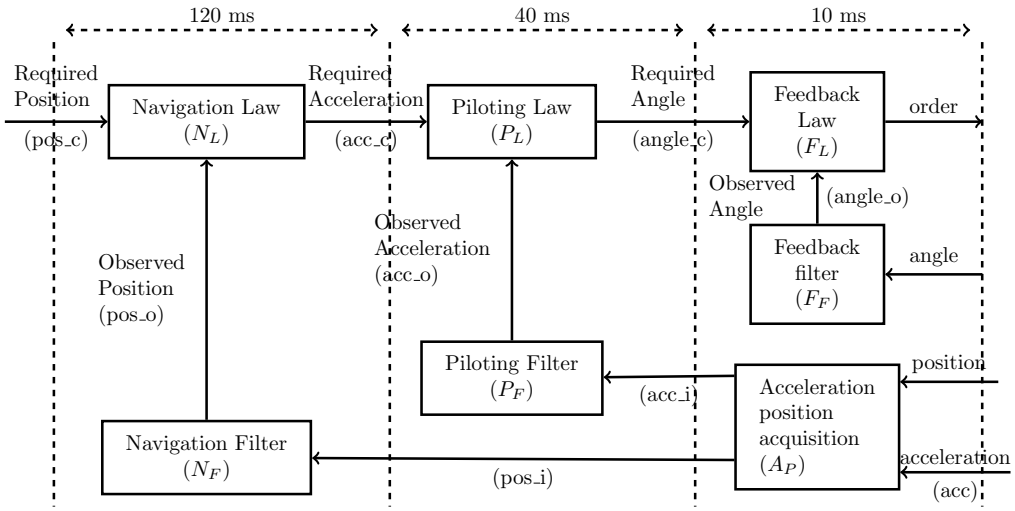


Fig. 1 – Flight control system

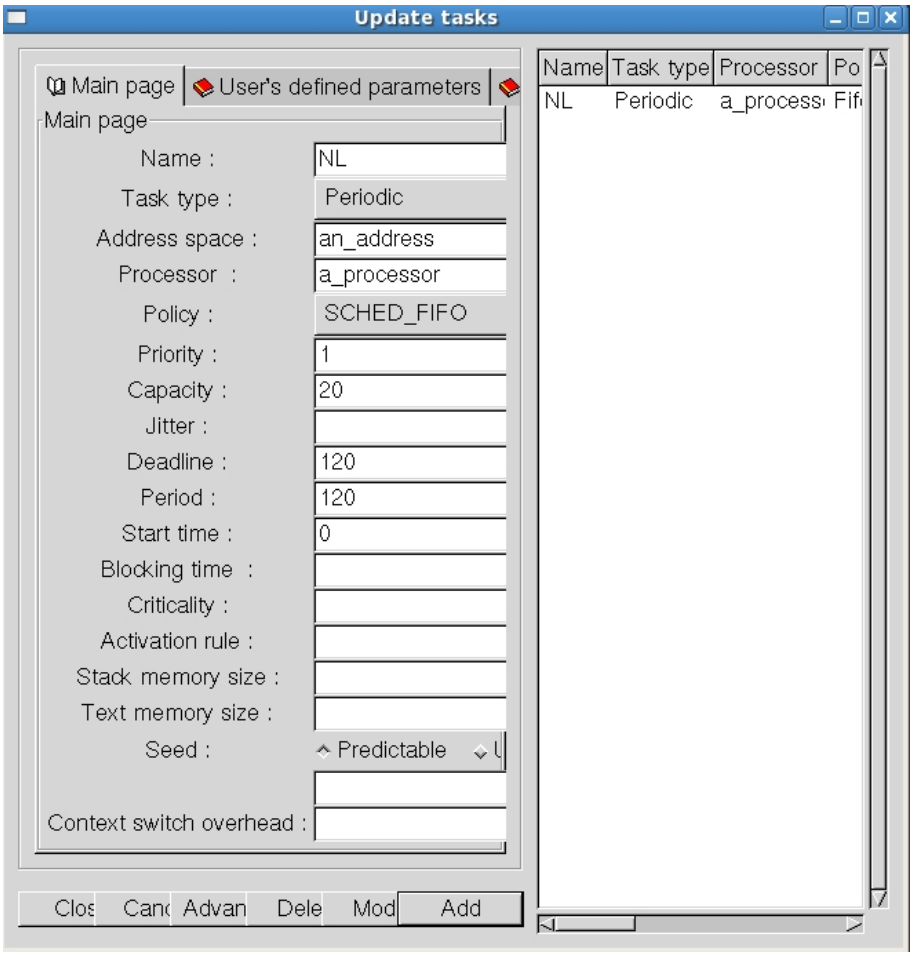
2 Cheddar

Cheddar is a free real-time scheduling tool which checks temporal properties on real-time systems. The tool provides a simulation engine and feasibility tests. All documents and binaries can be found in the url :
`http://beru.univ-brest.fr/~singhoff/cheddar/`
ant the tutorial is :
`http://beru.univ-brest.fr/~singhoff/cheddar/ug/cheddar-r2.html`

2.1 Monoprocessor scheduling analysis

Launch CHEDDAR : write cheddar in your terminal. The graphical interface appears. Follow the tutorial recommendation for programming the case study.

1. *add* a processor in the *Edit/Update processors*. For this, give a name, a scheduler policy and precise if it is preemptive or not.
2. *add* a memory address in the *Edit/Update address spaces*. Simply give a name.
3. *add* all the tasks in the *Edit/Update tasks*.



The real-time features of the set of tasks are given below :

Task	Period	WCET	release date	deadline
NL	120	20	0	120
NF	120	10	0	120
PL	40	5	0	40
PF	40	5	0	40
FL	10	2	0	10
FF	10	1	0	10
AP	10	1	0	10

Make a first scheduling analysis with RM and a second using EDF. For each one, try the simulator and the feasibility tests. Click on *scheduling simulation*. The result appears : a Gantt diagram illustrates an worst case execution. Click on *scheduling feasibility*. The tool check if the scheduling never misses any deadline. It computes the processor utilization factor and the response time for each task.

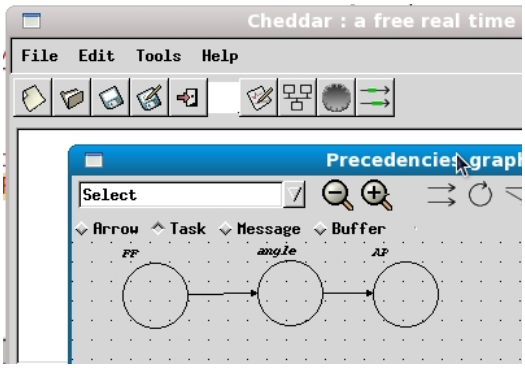
2.2 Distributed modelling

We assume in this section that there are 2 processors that host the functions. These processors use the rate monotonic policy.

Task	Processor	Period	WCET	release date	deadline
NL	1	120	20	0	120
NF	2	120	10	0	120
PL	1	40	5	0	40
PF	2	40	5	0	40
FL	2	10	2	0	10
FF	2	10	1	0	10
AP	1	10	1	0	10

We want to compute the response time taking into account the exchange of messages between task. There are 3 messages to consider : `angle_o`, `acc_o` and `pos_o`. We assume that the latency for each message is equal to 1.

For modelling this in CHEDDAR, add a third processor that will model the messages. Add for each message a task on this processor, the period of which is the period of the sender and the capacity is equal to 1. Add also a precedence for each message. For instance, we have drawn the precedence between F_F and `angle_o`, since F_F produces the message ; and the precedence between `angle_o` and A_P.



Make a simulation and verify that the precedences imposed by the messages are respected. To compute the response time, use the menu : Tools → precedencies → end-to-en response time → compute and update tasks : one step. Apply this computing several times, until you find a fixed point. The underlying algorithm is the *holistic method* [TC94]. How do you interpret the results ?

3 Programming with MARTE OS

MARTE OS is a hard real-time operating system for embedded applications. It provides a framework for developing multi-thread real-time applications. It can be used as a native kernel or as an emulator. Today, we will use MARTE OS as an emulator. You can find the documents relative to MARTE OS on the web page : `http://marte.unican.es/`

You must connect on a virtual machine. For this, open a terminal and

command	ssh etu1@c104-01	or etu2, ..., etu8
pwd	etu1	or etu2, ..., etu8
modify the PATH	`export PATH=/usr/gnat/bin:$PATH` `export PATH=/usr/marte/utils:$PATH`	
create a directory	`mkdir myname`	
enter the directory	`cd myname`	
Copy the examples	`cp -R /usr/marte/examples/appsched .`	

Enter in the folder `cd appsched` and open the files concerning the edf scheduler : `edf_sched.c` and `edf_threads.c`. The user scheduler is programed in the file `edf_sched.c`. The tasks are described in the file `edf_threads.c`. We will modify these two files in order to program the case study with MARTE OS and the policy EDF. For compiling, modify the `Makefile` by removing `../../misc/load.o` in the edf compilation. Then, after saving, write the command `make edf`. It produces an executable `a.out`. Launch this executable `./a.out`. The example supplied by MARTE OS contains two tasks t_1 and t_2.

3.1 Programming the tasks

We will first code the example using the edf scheduler defined by the original file `edf_sched.c`. For this, modify the file `edf_threads.c`.

In the proposed example `edf_threads.c`, there are two tasks t_1 and t_2 : t_1 has period 2s and wcet 0.5, while t_2 has period 3s and wcet 1.5. At execution, the RQ (ready queue) is depicted. At first we have `RQ: EMPTY`, then all the tasks are created. In the example, the two tasks are created and inserted in the RQ :

```
Event: POSIX_APPSCHED_NEW                       at 0.0s
    Add new thread (id:1, period:2.0s)
    RQ: (id:1, deadline:2.0s)  Activate:1
Event: POSIX_APPSCHED_NEW                       at 0.0s
    Add new thread (id:2, period:3.0s)
    RQ: (id:1, deadline:2.0s) (id:2, deadline:3.0s)
```

Once the tasks are created, the nominal scheduling is running and is showed in the shell :

```
Event: POSIX_APPSCHED_EXPLICIT_CALL             at 0.5s
    RQ: (id:2, deadline:3.0s)  Activate:2  Suspend:1
Event: POSIX_APPSCHED_SIGNAL                    at 2.0s
    RQ: (id:2, deadline:3.0s) (id:1, deadline:4.0s)
Event: POSIX_APPSCHED_EXPLICIT_CALL             at 2.0s
    RQ: (id:1, deadline:4.0s)  Activate:1  Suspend:2
Event: POSIX_APPSCHED_EXPLICIT_CALL             at 2.5s
    RQ: EMPTY Suspend:1
Event: POSIX_APPSCHED_SIGNAL                    at 3.0s
    RQ: (id:2, deadline:6.0s)  Activate:2
```

We notice that at time 0.5s, the task 1 ends its execution (and becomes suspend). Thus the second task executes. At date 2s, the task 1 is awaken and task 2 ends its execution (indeed 0.5 + 1.5 = 2). And so on.

The start routine is the same for both tasks and is called `periodic`. The code is the following :

```
void * periodic (void * arg)
{
  float amount_of_work = *(float *) arg;
  while (1) {
    /* do useful work */
    eat (amount_of_work);
```

```
    posix_appsched_invoke_scheduler (0);
  }
}
```

The creation of a thread in the main process is realised by the instruction (for t_1) :

```
  /* Creation of one scheduled thread */
  pthread_attr_init (&attr);
  pthread_attr_setschedpolicy (&attr, SCHED_APP);
  CHK( pthread_attr_setappscheduler (&attr, sched) );
  user_param.period.tv_sec  = 2;
  user_param.period.tv_nsec = 0;
  load1 = 0.5;
  CHK( pthread_attr_setappschedparam (&attr, &user_param,
      sizeof(user_param)) );
  param.sched_priority = MAIN_PRIO - 1;
  CHK( pthread_attr_setschedparam (&attr, &param) );
  CHK( pthread_create (&t2, &attr, periodic, &load2) );
```

Exercise 1 *Modify the main function in order to create 7 threads (NL, NF, PL, PF, FF, AP, FL) for modelling the case study.*

Exercise 2 *Modify the start routine* `periodic` *such that each activated thread prints its name.*

3.2 Modification of the scheduler

The implemented scheduler does not provide the management of the deadline (they assume deadline = period). We want to modify `edf_sched.c` to allow the management of deadlines. For testing your new scheduler, choose for A_P a deadline of 9 and for P_F a deadline of 39.

In the `edf_sched.h`, you must change the struct

```
struct edf_sched_param {
  struct timespec period;
  struct timespec relative_deadline;
};
```

In the `edf_sched.c`, you must change the struct

```
/* Thread-specific data */
typedef struct thread_data {
  struct thread_data * next;
  th_state_t th_state;
  struct timespec period;
  struct timespec next_deadline; /* absolute time */
  int id;
  timer_t timer_id;
  pthread_t thread_id;

//add the fields
  struct timespec next_period; /* absolute time of the next activation*/
  struct timespec relative_deadline;
} thread_data_t;
```

You must modify the functions :

1. `add_to_list_of_threads` : for this, you must
   (a) read the relative deadline in the t_data (adapt the period treatment),
   (b) compute the next absolute deadline and the next absolute activation of the task
   (c) modify the iterators in this way

   ```
   timer_prog.it_value = t_data->next_period;
   ```

   (d) print the deadline when adding the task ;
2. `make_ready` : for this, you must
   (a) compute the next absolute deadline and the next absolute activation of the task
   (b) modify the iterator `timer_prog.it_value = t_data->next_period;`
3. `reached_activation_time` :

```
void reached_activation_time (thread_data_t *t_data)
{
  switch (t_data->th_state) {
  case TIMED:
    t_data->th_state = ACTIVE;
    add_timespec (&t_data->next_deadline, &t_data->next_period, &t_data->relative_deadline);
    incr_timespec (&t_data->next_period, &t_data->period);
    enqueue_in_order (t_data, &RQ, more_urgent_than);
    break;
  case BLOCKED:
    break;
  case ACTIVE:
    // Deadline missed
    printf (" Deadline missed in thread:%d !!", t_data->id);
    add_timespec (&t_data->next_deadline, &t_data->next_period, &t_data->relative_deadline);
    incr_timespec (&t_data->next_period, &t_data->period);
    break;
  default:
    printf (" Invalid state:%d in thread:%d !!", t_data->th_state, t_data->id);
  }
}
```

You must then modify `edf_thread.c` to add the relative deadlines.

Références

[TC94] Ken Tindell and John Clark. Holistic schedulability for distributed hard real-time systems. *Microprocessing & Microprogramming*, 40 :117–134, 1994.