Real-Time Systems

Lecture 3

Introduction to Real-Time kernels

Task States
Generic architecture of Real-Time kernels
Typical structures and functions of Real-Time kernels

Last lecture (2)

- Computational models (**real-time model**)
- Real-time tasks: periodic, sporadic and aperiodic
- Temporal constraints of types: **deadline**, window, synchronization and distance
- Implementation of tasks using ***multitasking kernels***
- **Logic** and **temporal** control
- **Event-triggered** and **time-triggered** tasks

Task states

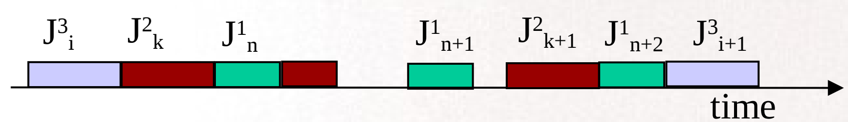
Task creation

Association between executable code (e.g. a “C” language function) to a private variable space (private stack) and a management structure - ***task control block (TCB)***

Task execution

Concurrent execution of the task's code, using the respective private variable space, under control of the kernel. The kernel is responsible for activating each one of the task's jobs, when:

- A period has elapsed (periodic)
- An associated external event has occurred (sporadic)

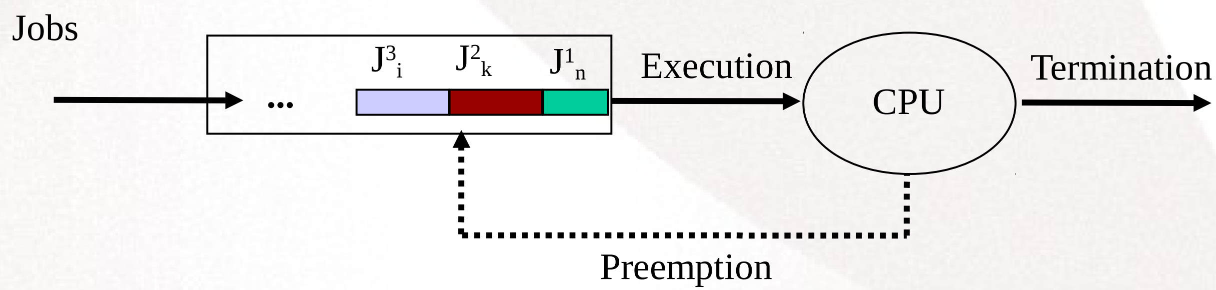


Task states

Execution of task instances (jobs)

After being activated, task's jobs wait in a queue (the **ready queue**) for its time to execute (i.e., for the CPU)

The ready queue is sorted by a given criterion (**scheduling** criterion). In real-time systems, most of the times this criterion is **not the arrival order**!

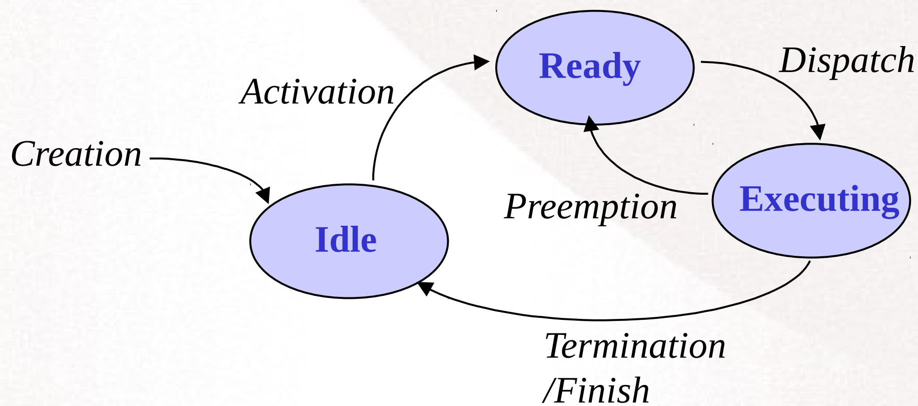


Task states

Task states

Task instances may be waiting for execution (**ready**) or executing. After completion of each instance, the task stay in the **idle** state, waiting for its **next activation**.

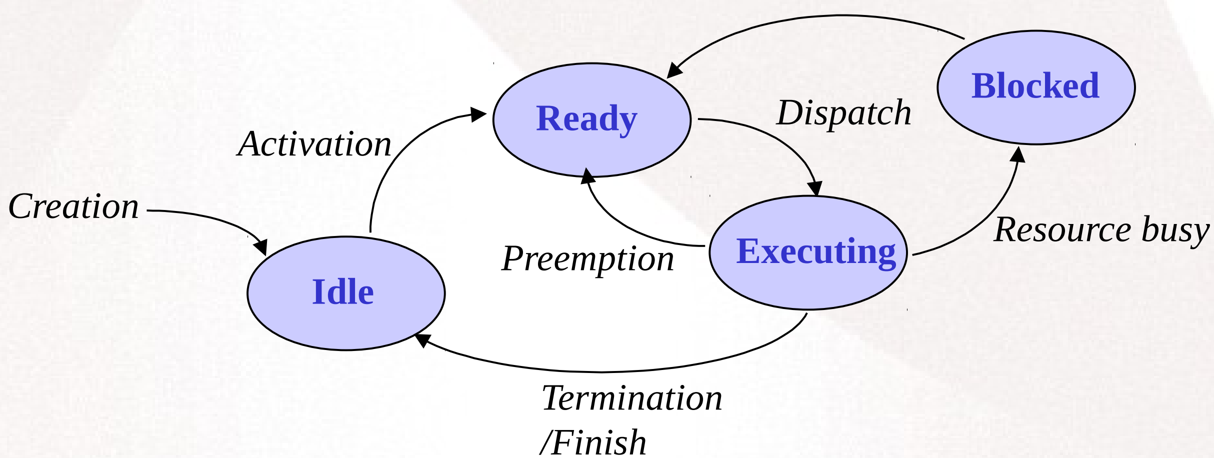
Thus, the basic set of dynamic states is: **idle**, **ready** and **execution**.



Task states

Other states: blocked

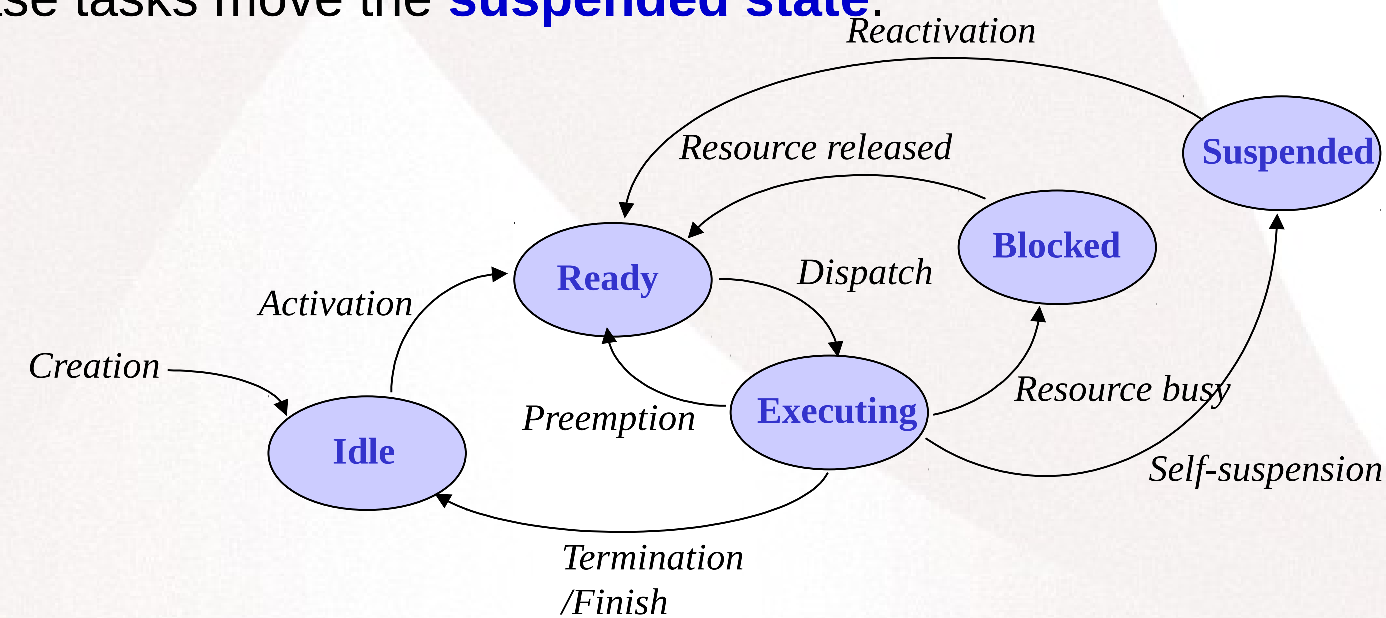
Whenever an executing task tries to use a **shared resource** (e.g. a memory buffer) that is already being used in **exclusive mode**, the task cannot continue executing. In this case it is moved to the **blocked state**. It remains in this state until the moment in which the **resource is released**. When that happens the task goes to the **ready state**.



Task states

Self suspension (sleep)

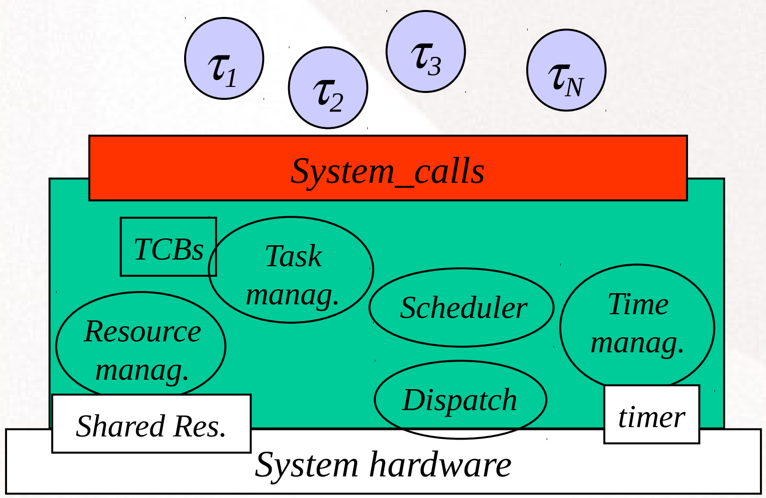
In certain applications tasks need to **suspend its execution** for a given amount of **time** (e.g. waiting a certain amount of time after requesting an ADC conversion), before completing its execution. In that case tasks move the **suspended state**.



Internal Architecture of a Real-Time Kernel

Basic services

- Task management (create, delete, initial activation, state)
- Time management (activation, policing, measurement of time intervals)
- Task scheduling (decide what jobs to execute in every instant)
- Task dispatching (putting jobs in execution)
- Resource management (mutexes, semaphores, etc.)



Management structures

TCB (*task control block*)

This is a fundamental structure of a kernel. It stores **all the relevant information about tasks**, which is then used by the kernel to manage their execution.

Common data (not exhaustive)

- Task identifier
- Pointer to the code to be executed
- Pointer to the private stack (for context saving, local variables, ...)
- Periodic activation attributes (task type (periodic/sporadic), period, initial phase, etc)
- Criticality (*hard*, *soft*, non *real-time*)
- Other attributes (*deadline*, priority)
- Dynamic execution state and other variables for activation control, e.g. SW timers, absolute deadline, ...

Management structures

RTKPIC's TCB

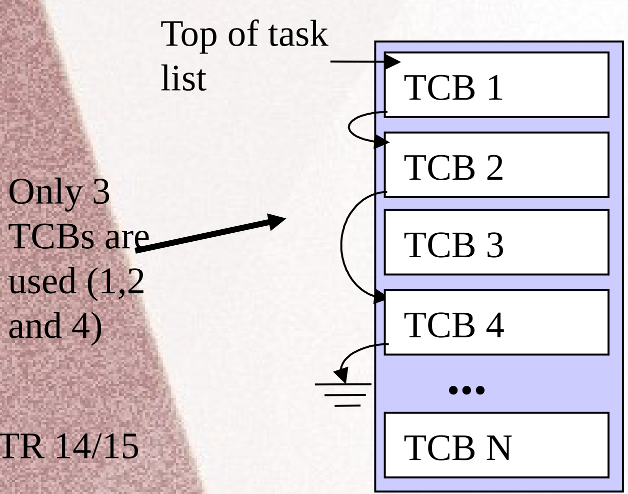
```
typedef struct {
        unsigned char id;               /* task id - 0..14 */
        void (*func_ptr)(void);         /* task first instruction address */
        unsigned char state;            /* task state */
        unsigned int period;            /* task period in ticks */
        unsigned int deadline;          /* task deadline relative to activation */
        unsigned long nx_activ;         /* task next activation in absolute ticks */
        unsigned long nx_deadline;      /* task next deadline in absolute ticks */
        unsigned char priority;         /* task priority */

} TASK;

/* Number of tasks */
#define NTASKS      14                  /* main + 13 user tasks */

/* Task Control Table */
TASK tcb[NTASKS]
```

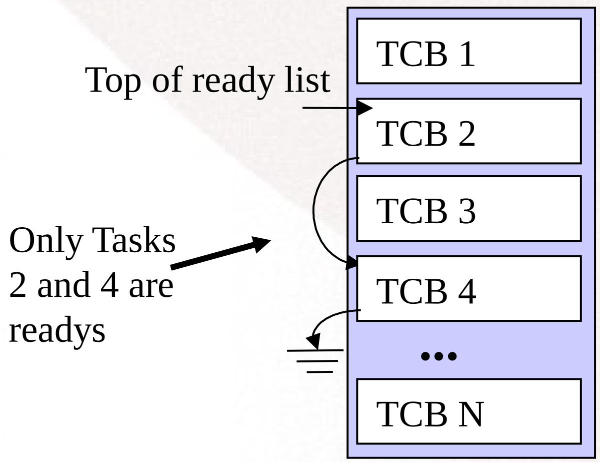
Management structures

TCB structure

TCBs are often defined in a **static array**, but are normally **structured** as **linked lists** to facilitate operations and searches over the task set.

E.g., the ready queue (list of ready tasks sorted by a given criteria) is maintained as a linked list. These linked lists may be implemented e.g. through indexes. **Multiple lists may (and usually do) coexist!**





Estruturas de gestão

Example: TCB structure of RTKPIC18

The RTKPIC18 was designed to handle applications with a **small number of tasks**. For this reason **no lists** were implemented. Consequently, whenever it is needed to do a search (e.g. to handle periodic task activations), the **whole TCB set must be checked**.

Serach all the TCB array

Check if next activation is duer

Check for deadline misses

Change state and call the scheduler

/* Tick handler code */

```
for (temporary_i = 1; temporary_i < n_task; temporary_i++)
{
        temporary_task_i = tcb + temporary_i;
        if (temporary_task_i->nx_activ == system_clock)
        {               /* new activation */
                if ((temporary_task_i->state == READY) || (temporary_task_i->state == RUN))
                     deadline_miss |= (0x01 << temporary_i);          /* deadline missing */

                temporary_task_i->state = READY;
                temporary_task_i->nx_activ += temporary_task_i->period;

                if ((preempt_sys == PREEMPT) || (run_task_id == 0))
                        Call_scheduler = 1;        /* New task READY: Call scheduler */

                if (sch_alg == EDF)
                        Call_EDF = 1;              /* Call EDF priority set */
        }
```

Management structures

Access to shared resources

Exclusive access shared resources (**critical sections**), have to be managed in an appropriate way, to allow access by only one task in any instant (as for the CPU). A simple way to do it is using **atomic *flags*** (*mutexes*), **monitors** (non preemptive execution) or **semaphores**.

For the case of semaphores it is needed a structure (**semaphore control block – SCB**) that holds its state as well as the list of tasks that are waiting for access.

SCB →

Counter (state)
Waiting tasks list

Management functions

Time management

Time management is another critical activity on kernels. It is required to:

- **Activate** periodic tasks
- **Check** if temporal constraints are met (e.g. deadline violations)
- **Measure** time intervals (e.g. **self-suspension**)

It is based on a **system timer**. This timer can be configured in two modes:

- ***Periodic tick***: generates periodic interrupts (**system ticks**). The respective ISR handles the time management. All **temporal attributes** (e.g. period, deadline, waiting times) must be **integer multiples of the clock tick**.
- ***Single-shot/One-shot/tickless***: the timer is configured for generating interrupts only when there are periodic task activations or other similar events (e.g. the termination of a task self-suspension interval).

Management functions

Tick-based systems

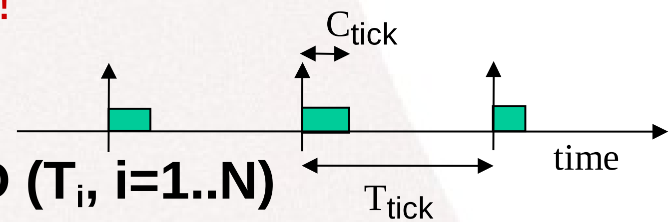
The **tick** defines the system's temporal resolution.

Smaller ticks corresponds to better resolutions.

E.g. 10ms tick => task periods: T_1=20ms, T_2=1290ms, T_3=~~25ms~~

The tick handler is code that is executed periodically. Thus it **consumes CPU time**, representing **overhead** (C_{tick}/T_{tick})

The bigger the tick, the lower the overhead!!



Compromise is needed: ***tick* = GCD (T_i, i=1..N)**

E.g. T_1=20ms, T_2=1290ms, T_3=25ms => GCD(20,1290,25)=**5ms**

However it mus be assured that tick > min_tick, which is imposed by the CPU processing capacity!

Management functions

Measurement of time intervals

In tick-based systems, the kernel keeps a variable that counts the number of ticks since the system boot.

- e.g. in the RTKPIC kernel “unsigned long system_clock”, accessed by the macro get_sys_time()
- **With tick=10ms, this variable wraps around after 1.6 years**

Better precision is achieved if the timer is read directly. However, bigger time ranges may imply using SW managed variables.

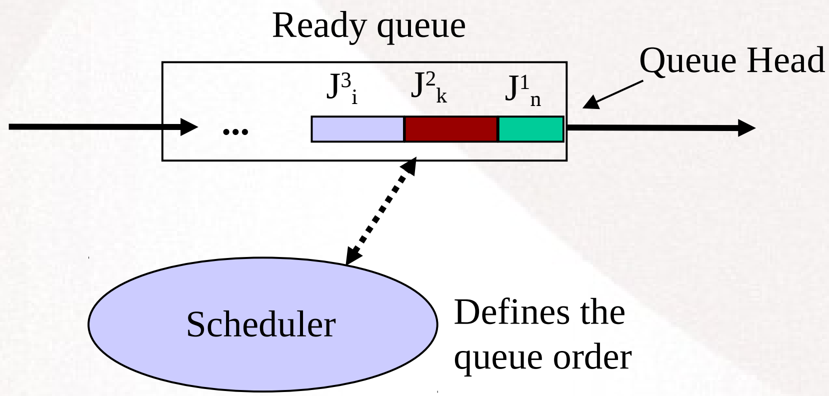
E.g. in Pentium CPUs, with a 1GHz clock, the TSC *wraps around* after 486 years !!!

Management functions

Scheduler

The scheduler selects **which task to execute** among the (eventually) several ready tasks

In real-time systems must be based on a deterministic criteria, which must allow computing an upper bound for the time that a given task may have to wait on the ready queue.

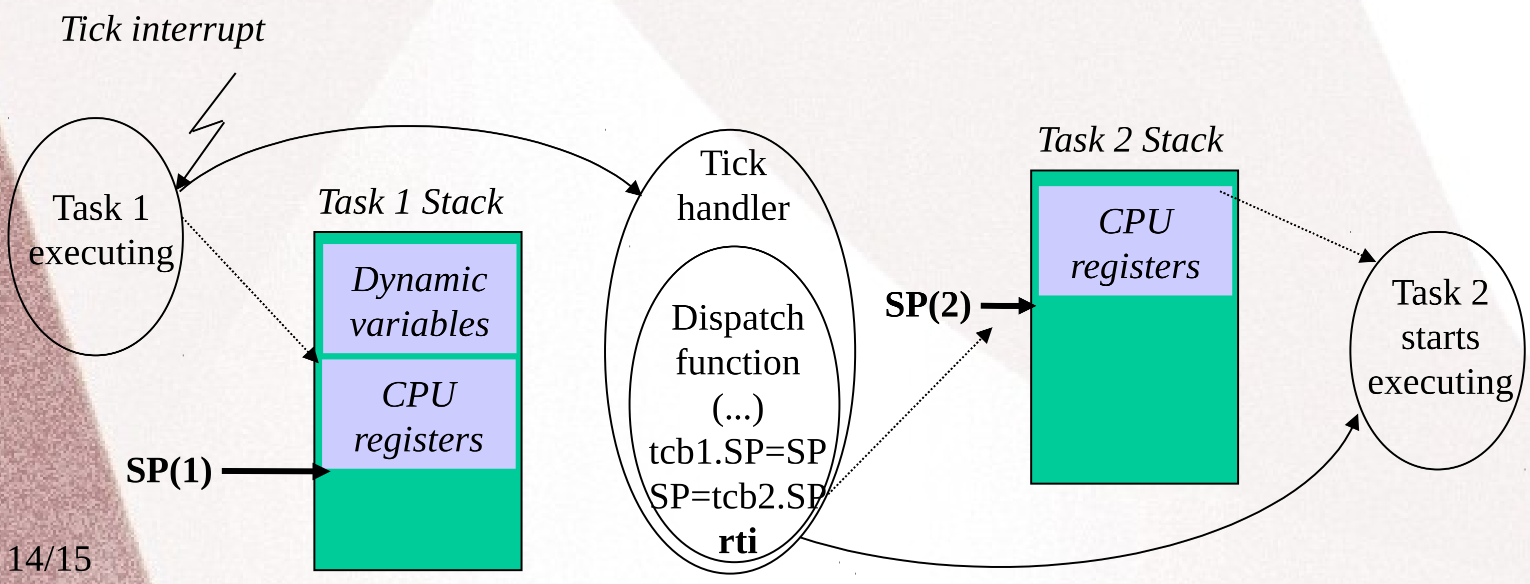


Management functions

Dispatch

•Puts in execution the task selected by the scheduler

•For preemptive systems it may be needed to preempt (suspend the execution) of a running task. In these cases the dispatch mechanism must also manipulate the stack.



RTKPIC – Real-Time Kernel for PIC18

Based on a older RTOS for X86 developed at the UA (ReTMiK)

- Tick based
- For PIC18FXXX
- Task code is cyclic
- Scheduler is part of the kernel
- Allows preemption control
- IPC via global variables
- Monolithic application (kernel + application code in a single executable file)

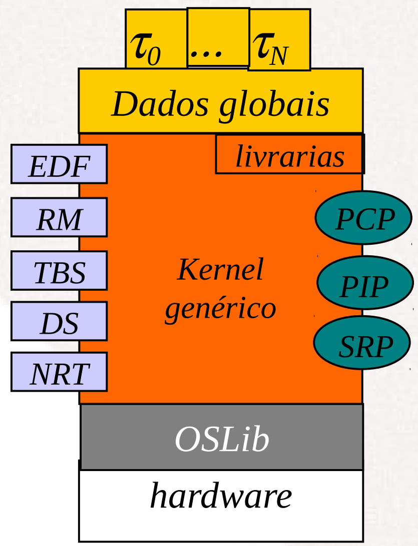
```
main( )
{
   create_system(... );
   /* For each task */
  create_task (...);

   config_system();
   release_system( );
   while(1)
   {
        /* background */
   }
}
```

```
task_n( )
{

   task_init();

   while(1) {
      /* Task code */

        .....
   }

}
```

SHaRK – Soft and Hard Real Time Kernel

http://shark.sssup.it/

- Research kernel, main objective is flexibility in terms of scheduling and shared resource management policies
- POSIX (partially compat.)
- For x86 (>= i386 with MMU) architectures
- Cyclic task
- Several IPC methods
- Concept of *Task Model* (HRT, SRT, NRT, per, aper) and *Scheduling Module*
- Policing, admission control
- Monolithic application



InitFile
(module declaration)

tarefa __init__
(initializations and call main( ))

```
int main ( ){
/* Other inits */
/* Define tasks */
   task_create ( );
   task_activate ( );
/* May terminate or wait to stop
the system */
while (keyb_getchar( )!=ESC);
sys_end ( );}
```

```
void * TaskBody (void *arg){
/* Init code */
while (cond) {
     /* Task code */
     (...)
   task_endcycle( );}
/* terminates if “cond” */
return my_val;}
```

Xenomai: Real-Time Framework for Linux

http://www.xenomai.org/

- Allow the use of Linux for Real-Time applications
- Dynamically loadable modules
- Tasks may execute and kernel or user space
- POSIX (partially compat.)
- Cyclic tasks
- Support to several IPC mechanisms, both between RT and NRT tasks (pipe, queue, buffer, ...)

```
// A task
void task_a(void *cookie) {
        /* Set task as periodic */
        err=rt_task_set_periodic(NULL, TM_NOW, TASK_A_PERIOD_NS);
        for(;;) { // Forever
                err=rt_task_wait_period(&overruns);
                // Task load
        }
        return;
}
// Main
int main(int argc, char *argv[]) {
        .... // Init code
        /* Create RT task */
        err=rt_task_create(&task_a_desc, "Task a", TASK_STKSZ,
TASK_A_PRIO, TASK_MODE);
        rt_task_start(&task_a_desc, &task_a, 0);
        ....
        /* wait for termination signal */
        wait_for_ctrl_c();
        return 0;
}
```

Summary of lecture 3

- The **task states**
  - States and transition diagram
- The **generic architecture** of a RT kernel
- The basic components of a RT kernel, its structure and functionalities
- Some examples: RTKPIC18, SHaRK and XENOMAI