Computation Models and Introduction to RTOS Architecture and Services

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

September 21, 2022

Last lecture

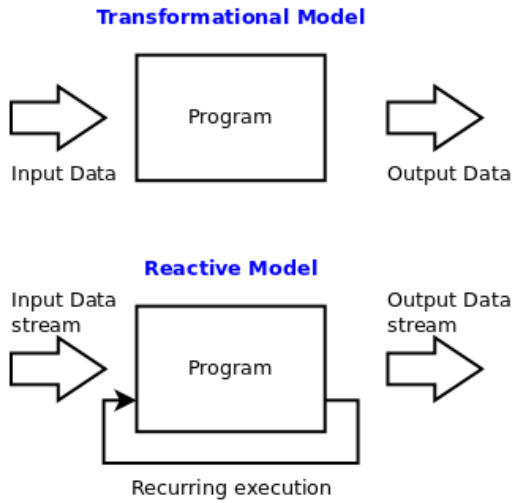
- Notion of real-time and real-time system
- Antagonism between real-time and best effort
- Aspects to consider: execution time, response-time and regularity of periodic events
- Requirements of RTS: functional, temporal and dependability
- Basic nomenclature
- Constraints: soft, firm and hard, and hard real time vs soft real time
- The importance of considering the worst-case scenario

Agenda for today

- Real Time model
- Additional task attributes
- Task states and execution
- Generic architecture of a RTOS
- RTOS examples

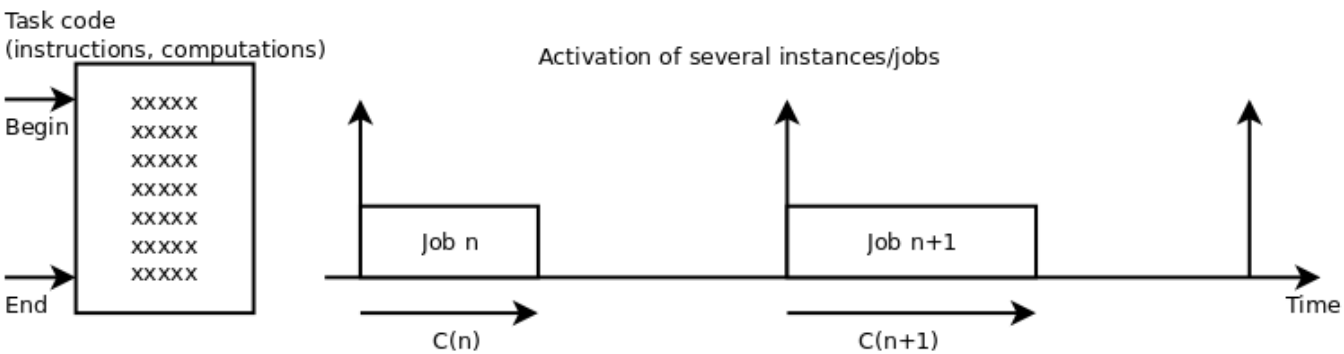
Real-time model

- **Transformational model**
  - According to which a program begins and ends, turning data into results or output data.
- **Reactive model**
  - According to which a program may execute indefinitely a sequence of instructions, for example operating on a data stream.
- **Real-time model**
  - Reactive model in which the program has to keep synchronized with the input data stream, which thus imposes time constraints to the program.



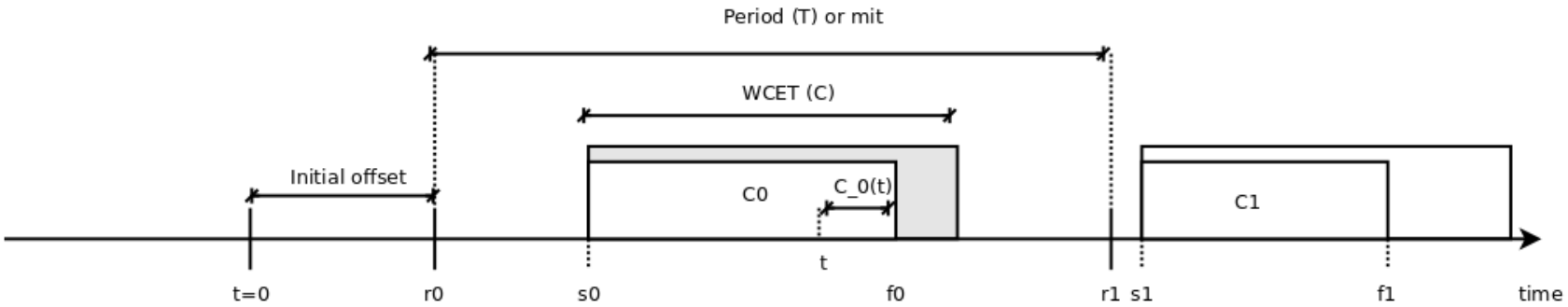
Processes, threads, tasks, activities and jobs

- **Definition of task (process, thread, activity)**
  - Sequence of activations (instances or jobs), each consisting of a set of instructions that, in the absence of other activities, is performed by the CPU without interruption.



Real-time model

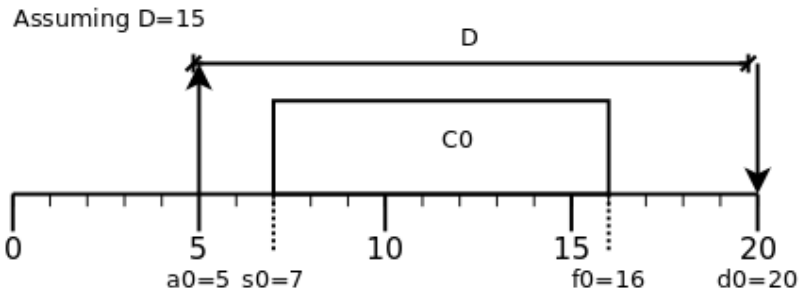
Temporal attributes



- Worst-Case Execution Time ($WCET$)
- Period (T) (if periodic, *mit* if sporadic)
- Relative phase or initial offset (ϕ)
- Release time (r_i)
- Finish/completion time (f_i)
- Residual execution time ($c_n(t)$): maximum remaining execution time required by job n at instant t

Deadline types

Deadline is the most common temporal requirements



D Relative deadline. Time measured from activation.

d_i Absolute deadline. Absolute time instant, regarding the i^{th} activation, in which the task must finish

$$d_i = r_i + D$$

- There are other types of task requirements: window, synchronization, distance, ...

Task temporal characterization

Example of task temporal characterization (there are other forms)

- **Periodic:**
  - $\Gamma = \tau_i(C_i, \phi_i, T_i, D_i)$
  - Examples: $\tau_1 = (1, 0, 4, 4), \tau_2 = (2, 1, 10, 8)$
- **Sporadic:**
  - Similar to periodic, but mit_i replaces T_i and ϕ_i usually is not used (tough it could be used to specify a minimum delay until the first activation)
  - $\Gamma = \tau_i(C_i, mit_i, D_i)$
  - Examples: $\tau_1 = (1, 4, 4), \tau_2 = (2, 10, 8)$

Temporal control

Triggering of activities

- **By time: Time-Triggered**
  - The execution of an activity (function) is triggered via a control signal based on the progression of time (e.g., through a periodic timer interrupt).
- **By events: Event-Triggered**
  - The execution of activities (functions) is triggered through an asynchronous control signal based on a change of system state (e.g., through an external interrupt).

Temporal control

Event-triggered systems

- Systems controlled by the occurrence of events on the environment
- Typical of sporadic conditions monitoring on the system state (e.g., alarm verification or asynchronous requests).
- Utilization rate of the the computing system resources (e.g. CPU) is variable, depending on the frequency of occurrence of events.
  - Ill-defined worst case situation. Implies either:
    - use of probability arguments, or
    - imposition of limitations on the maximum rate of events

Temporal control

Time-triggered systems

- Systems triggered by the progression of time
- Typical e.g. in control applications of cyber-physical systems
- There is a common time reference (allows establishing phase relations)
- CPU utilization is constant, even when there are no changes in the system state.
- Worst case situation is well-defined

Temporal control

Example

- For the following task sets compute the maximum response time that each task may experiment:
  - TT: $\Gamma = \tau_i(C_i = 1, \phi_i = i, T_i = 5, D_i = 5), i = 1...5$
  - ET: $\Gamma = \tau_i(C_i = 1, T_i = 5, D_i = 5), i = 1...5$
- Compute the average and maximum CPU utilization rate for both cases.
  - Admit that, on average, ET tasks are activated every 100 time units
  - Note: the CPU utilization is given by: $U_i = \sum_{i=1}^{N} \frac{C_i}{T_i}$

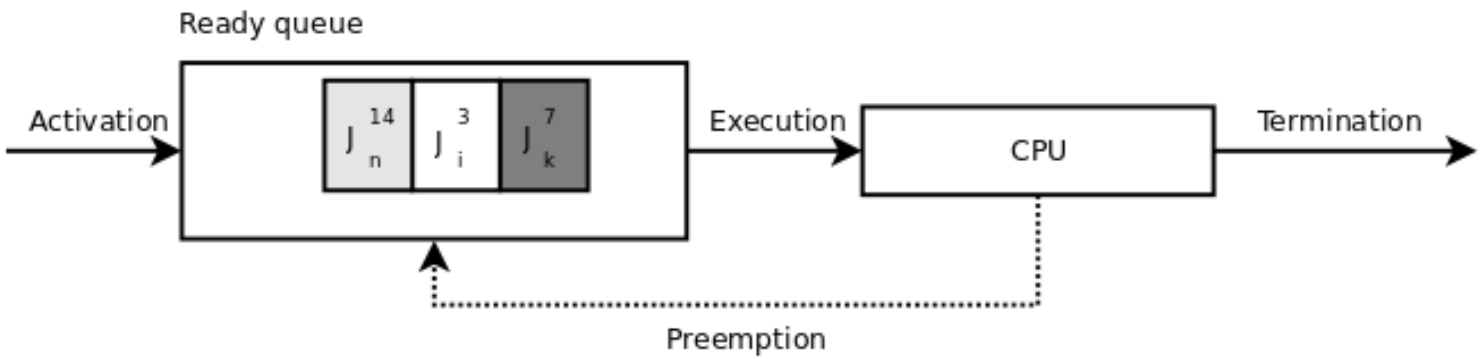
Task states and execution

- 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 RTOS kernel. The RTOS 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

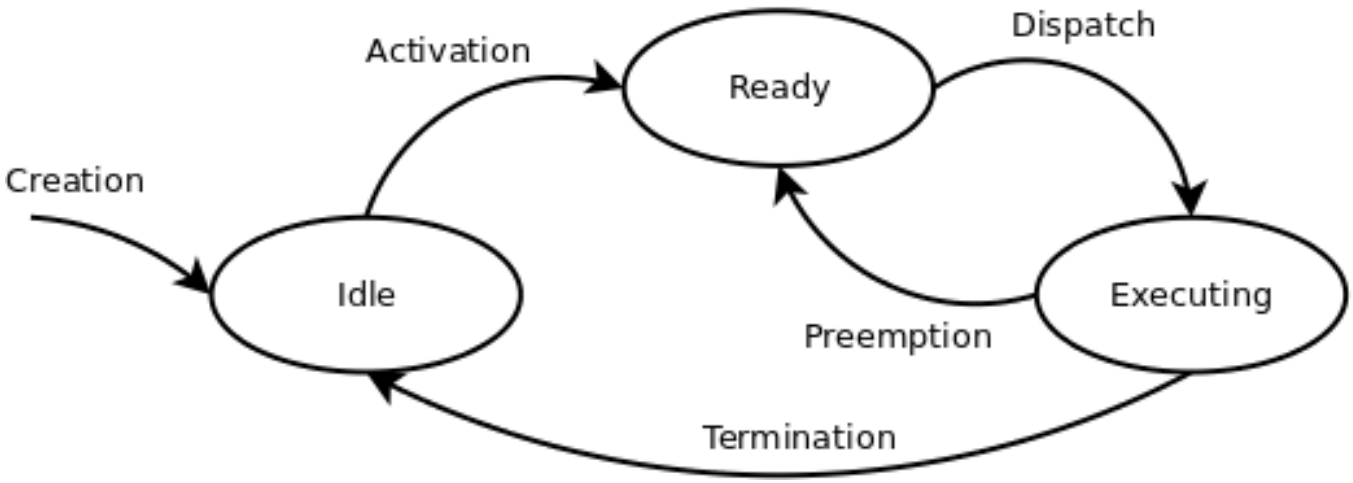
Execution of task instances (jobs)

- 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!



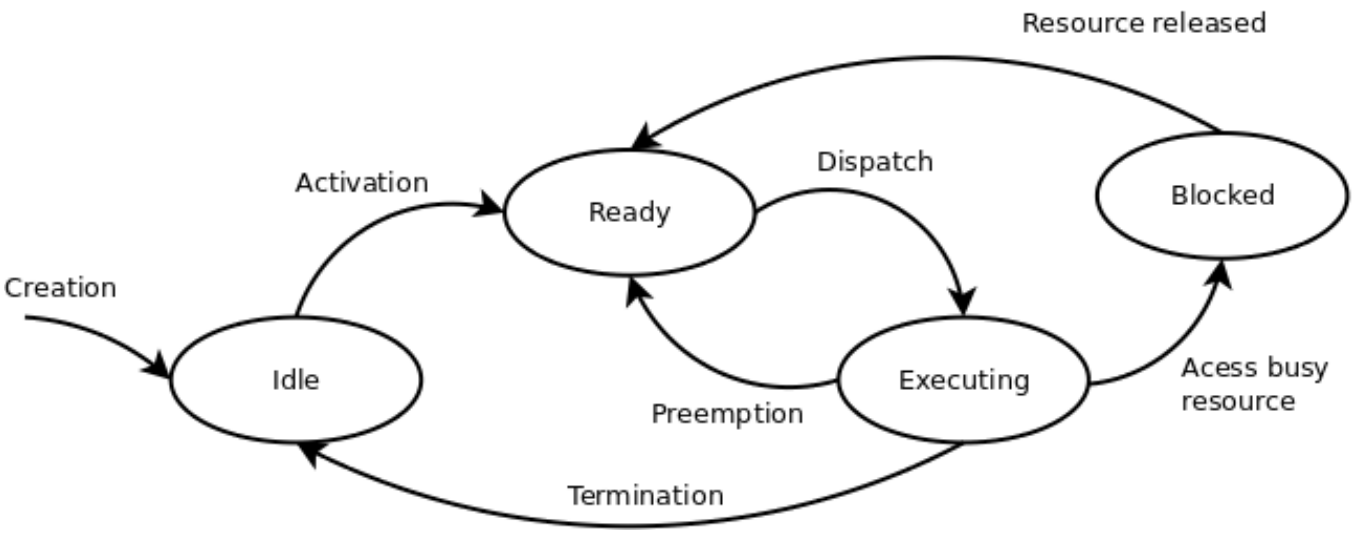
Task states

- Task instances may be waiting for execution (ready) or executing. After completion of each instance, tasks stay in the idle state, waiting for its next activation.
- Thus, the basic set of dynamic states is: **idle** , **ready** and **execution** .



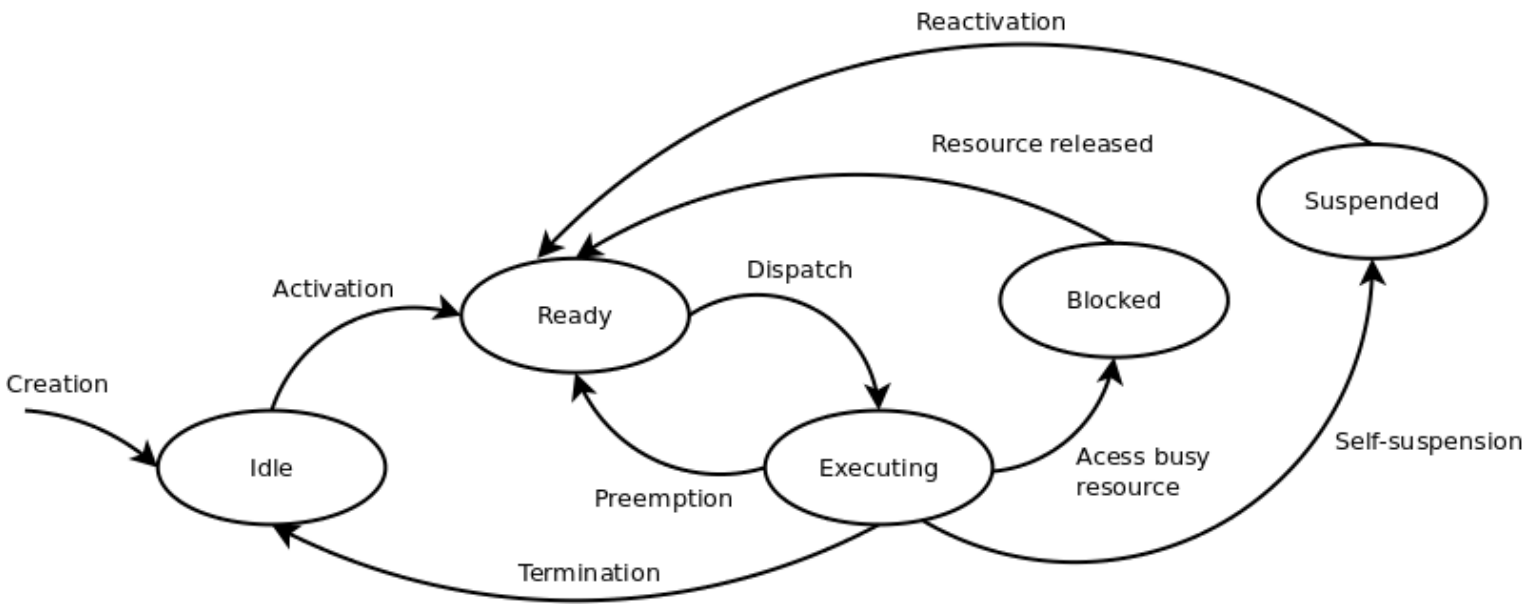
Task states

- 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.



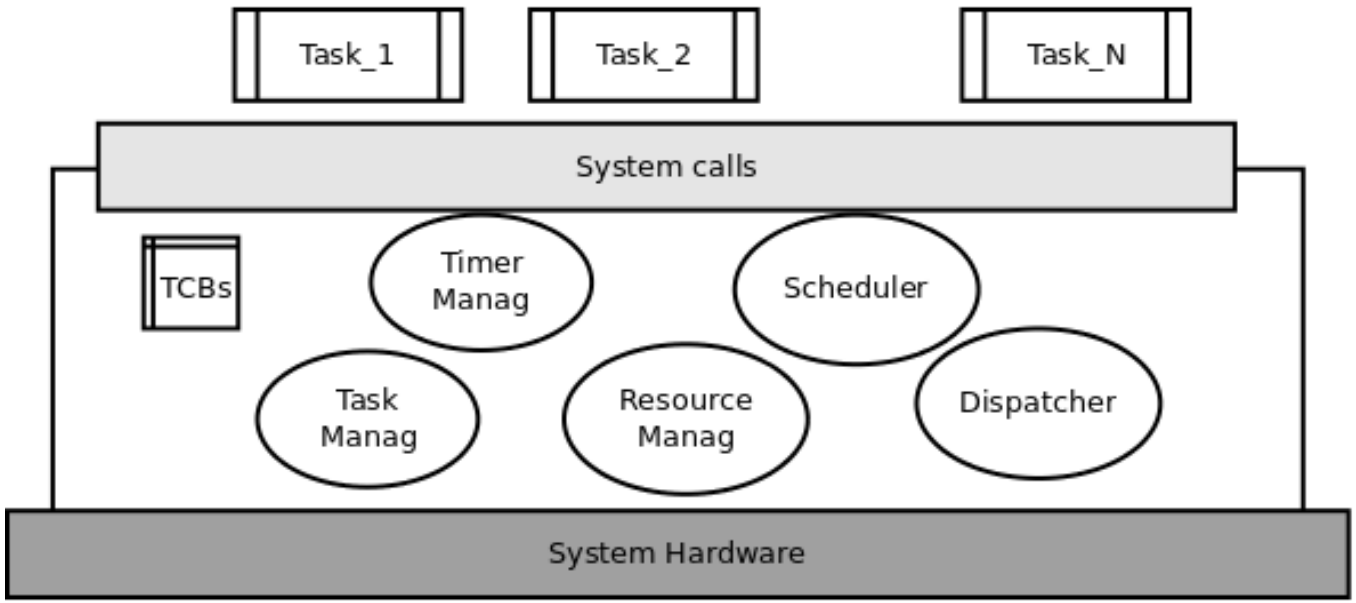
Task states

- Other 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 for a packet acknowledgment), before completing its execution. In that case tasks move to the **suspended** state.



Internal Architecture of a Real-Time OS/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

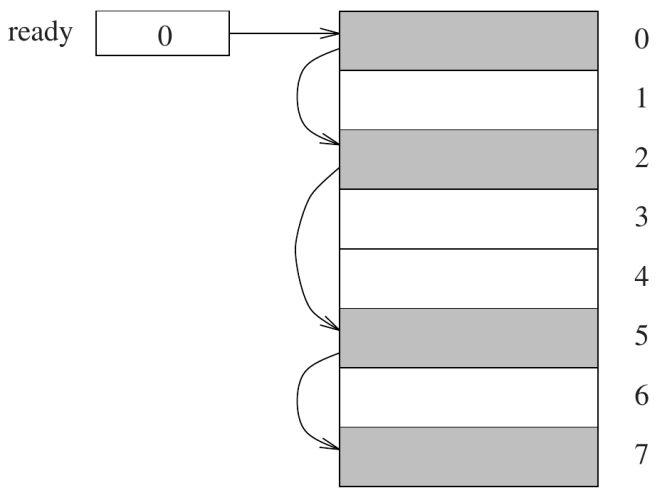
The **TCB** (task control block)

- 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

TCB structure

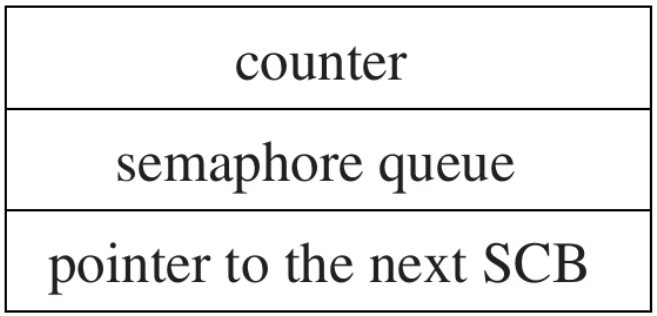
- TCBs are often defined in a static array, but are normally structured as linked lists to facilitate operations and searches over the task set.
- E.g., the ready queue (list of ready tasks sorted by a given criteria) is maintained as a linked list. These linked lists may be implemented e.g. through indexes. Multiple lists may (and usually do) coexist!



Management structures

SCB structure

- Similarly, the information concerning a semaphore is stored in a **Semaphore Control Block (SCB)** , which contains at least the following three fields:
  - A counter, which represents the value of the semaphore
  - A queue, for enqueueing the tasks blocked on the semaphore
  - A pointer to the next SCB, to form a list



Time management functions

- 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. There are two common 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/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).

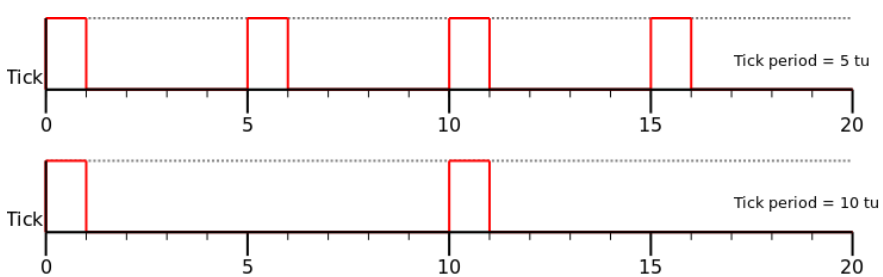
Time management functions

Tick-based systems

- The tick defines the system's temporal resolution.
  - Smaller ticks corresponds to better resolutions
  - E.g. if tick=10 ms and task periods are: T1=20ms, T2=1290ms, ~~T3=25ms~~
- The tick handler is code that is executed periodically, thus it consumes CPU time ($U_{tick} = \frac{C_{tick}}{T_{tick}}$).
- A bigger tick lowers the overhead; compromise!
  - $tick = GCD(T_i), i = 1..N$
  - E.g. T1=20 ms, T2=1290 ms, T3=25 ms, then GCD(20,1290,25)=5 ms

.

  - Works but U_{tick} doubles!



Time management functions

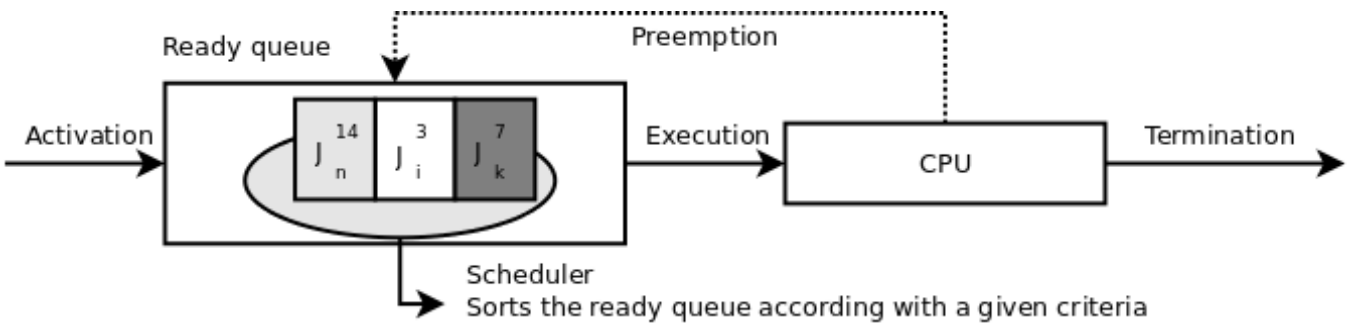
Measurement of time intervals

- In tick-based systems, the kernel keeps a variable that counts the number of ticks since the system boot.
  - In FreeRTOS the system call xTaskGetTickCount() returns the number of ticks since the scheduler was initiated (via vTaskStartScheduler()).
  - The constant portTICK_RATE_MS can be used to calculate the (real) time from the tick rate with the resolution of one tick period
  - Better resolutions require e.g. the use of HW timers.
- Be careful with overflows
- E.g. in Pentium CPUs, with a 1GHz clock, the TSC wraps around after 486 years !!!

Management functions - Scheduler

Scheduler

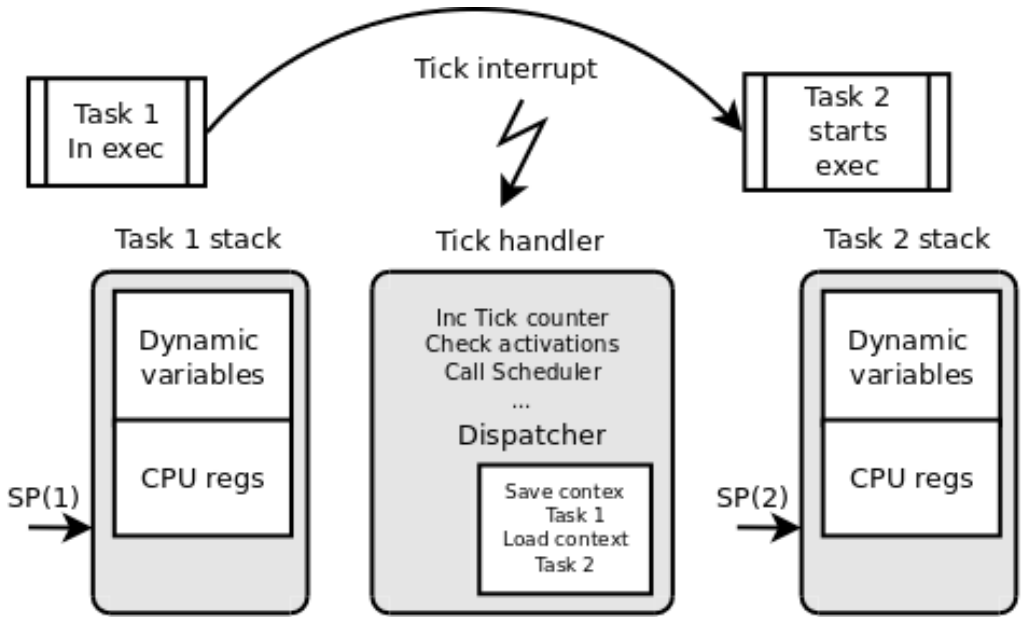
- The scheduler selects which task to execute among the (eventually) several ready tasks
- In real-time systems must be based on a deterministic criteria, which must allow computing an upper bound for the time that a given task may have to wait on the ready queue.



Management functions - Dispatcher

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.



FreeRTOS

- Many CPU architectures (8 to 32 bit)
- Tick and tickless operation
- Task code is cyclic
- Scheduler is part of the kernel
- Allows preemption control
- IPC: queues, buffers
- Synchronization: semaphores, mutex, ...
- Monolithic application (kernel + application code in a single executable file)


```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}

int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - most small microcontrollers will use
                             much less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
```

Xenomai: Real-Time Framework for Linux

Xenomai, https://xenomai.org/

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

```
// Main
int main(int argc, char *argv[]) {
    .... // Init code
    /* Create RT task */
    err=rt_task_create(\&task_a_desc, "Task a", ...);
    rt_task_start(\&task_a_desc, \&task_a, 0);
    ....
    /* wait for termination signal */
    wait_for_ctrl_c();
    return 0;
}

--------------------------------

    // A task
    void task_a(void *cookie) {
        /* Set task as periodic */
        err=rt_task_set_periodic(NULL, TM_NOW, ...);
        for(;;) { // Forever
            err=rt_task_wait_period(\&overruns);
            .... // Task work
        }
        return;
    }
```

SHaRK: Soft and Hard Real Time Kernel

Academic 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 or above) architectures
- Cyclic tasks
- Several IPC methods
- Concept of Task Model(HRT, SRT, NRT, per, aper) and Scheduling Module
- Policing, admission control
- Monolithic application

```
 main( )
{
    create_system(... );
    /* For each task */
    create_task (...);
    config_system();
    release_system( );
    while(1)
    {
          /* background */
    }
}
-----------------------
    task_n( )
    {
        task_init();
        while(1) {
            /* Task code */
       .....
        }
    }
```

Summary

- Computational models (real-time model)
- Implementation of tasks using multitasking kernels
- Logic and temporal control
- Event-triggered and time-triggered tasks
  - States and transition diagram
- The generic architecture of a RT kernel
- The basic components of a RT kernel, its structure and functionalities
- RTOS examples