Real-Time Operative Systems Course

Course Presentation

Paulo Pedreiras

DETI/UA/IT

September 13, 2022

Real-Time Operative Systems Course

Faculty

- Paulo Pedreiras (Course Coordinator)
- pbrp@ua.pt
- http://ppedreiras.av.it.pt/

Preliminaries

What are real-time systems?

- Computational systems
- Subject to the evolution of real-time (or physical time, if you wish)

Systems in which it is required to do the right thing at the right time, otherwise things can go (very) wrong!

Course objectives

- Main topic:
  - **Operating Systems** , **Programming Techniques** and **Analysis tools** to support the design of systems that interact with (or simulate a) physical process (or environment) so that they can do the **right thing at the right time**
- We will address:
  - The source and characterization of the restrictions imposed by the environment to the temporal behavior of the computational system;
  - Approaches to allow the computational system to be aware of the state of the environment that surround them;
  - The scheduling theory of concurrent tasks associated with real-time processes;
  - The structure and functionality of real-time operating systems
  - **And apply all these knowledge in practice!**

Examples of Real-Time Systems

Agile Manufacturing

Traffic control system

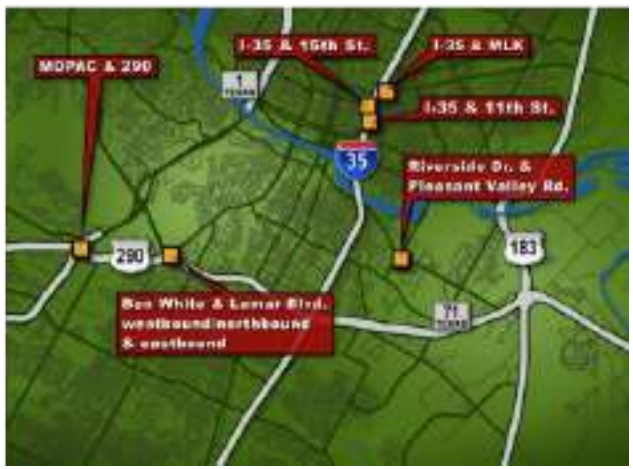


Image Credits: Realtime Robotics

Common misunderstandings

- **Isn't a quick processor sufficient?**
  - For a simple program, with a single task, that may work. However, when CPU(s) have to execute several tasks concurrently, just a quick processor may not be enough! Some tasks may block others and cause big and/or unpredictable delays.
- **Then, what do we need?**
  - Scheduling! Which means, select the right task to execute in every instant.
    There are scheduling algorithms that allow predicting and minimizing the maximum delays that tasks may suffer.

Common misunderstandings

- **So all this stuff applies only when we need multi-tasking... ?**
  - As stated above, we are considering situations where a computer has to perform several tasks simultaneously.
    It is normal that in such situations we use one multi-tasking operating system/kernel.
  - But often, even when the main body of the program is a simple endless loop, there are various pseudo-tasks, part of asynchronous interrupt routines, which lead to the same situation.
    The triggering of the interrupt routines may also be delayed, or even discarded. We must use proper techniques to bound and compute these delays.

Common misunderstandings

- **Are those delays so important?**
  - It depends on the kind of system. If we're talking e.g. about control systems on a plane, a car's X-by-wire or a robot that moves around other people and equipment ... **THEN YES!**
  - On the other hand, if we are talking about multimedia systems or routers in networks, delays in tasks shall not cause death to anyone, but there will be a loss of Quality-of-Service.

Course organization

- **Theoretical component:** presentation and discussion of concepts and techniques
  - Students should read selected parts of the base bibliography
  - Slides are available at the course website
  - Discussions are welcome!
- **Lab component:** learning about RTOS and tools, with focus on its practical use
  - Groups of 2 students
  - Tutorial classes to establish a basic set of practical competences: Linux (GPOS), Xenomai (RTOS-PC), Zephyr (RTOS-UC)
  - A final project

Grading

Normal period

- Theoretical component: 50%
  - 40% written exam + 10% quizzes, during classes
- Lab component: 50%:
  - 25% project + 10% for tutorials + 15% practical part of the written exam

Resit period

- Theoretical component: 50%
  - Written exam
- Lab component: 50%
  - Lab grade from the normal period or lab exam.

Class plan

Real-Time Operative Systems plan

- Class 1: Lecture 0+1: Course presentation + Basic concepts about real-time systems
- Class 2: Lecture 2: Computational models and RTOS architecture + Tutorial GPOS (1/2)
- Class 3: Lecture 3: Basic concepts on scheduling + Tutorial GPOS (2/2)
- Class 4: Lecture 4: Fixed-Priority scheduling + Tutorial Xenomai (1/2)
- Class 5: Lecture 5: Dynamic-Priority scheduling + Tutorial Xenomai (2/2)

Class plan

Real-Time Operative Systems plan (cont.)

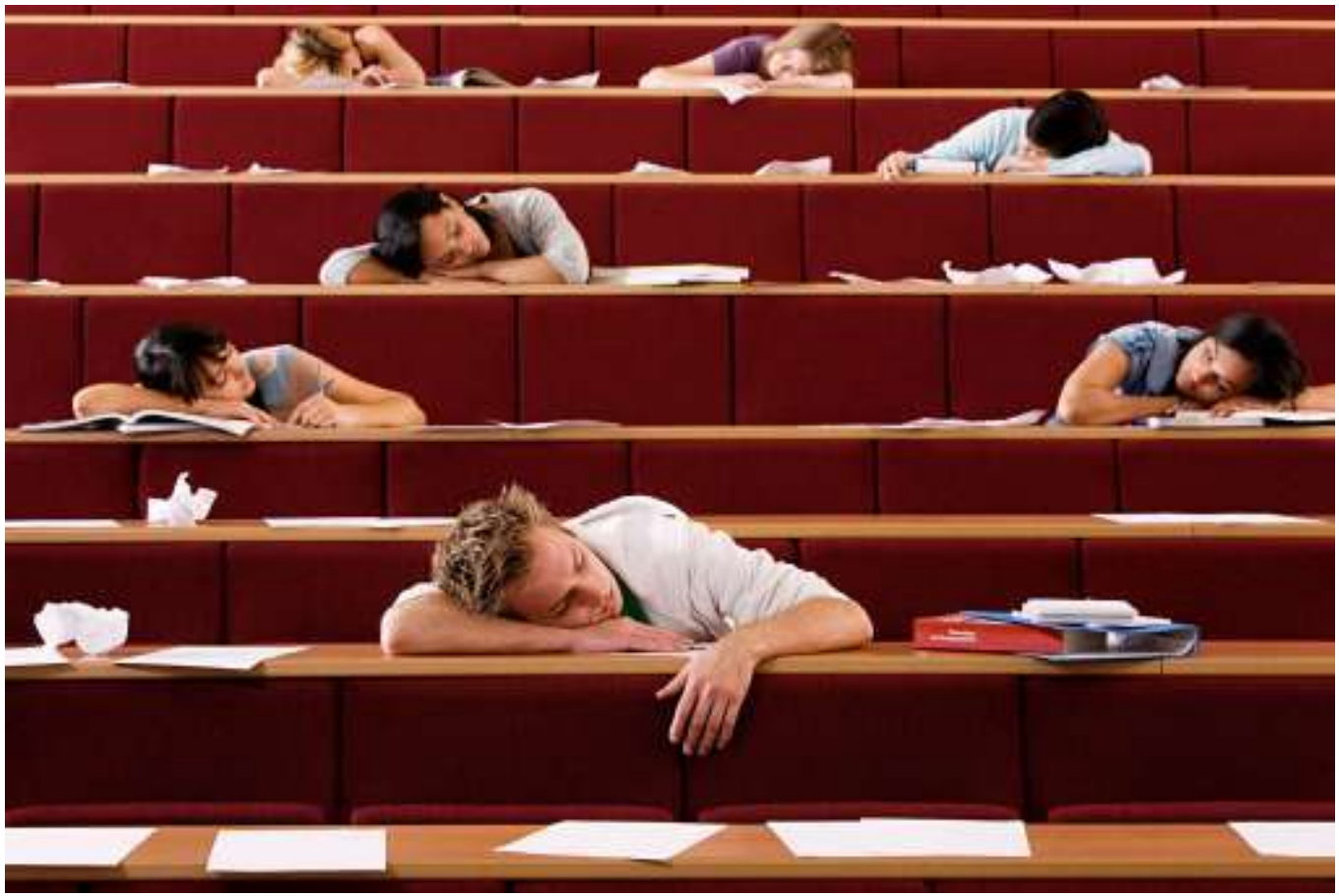
- Class 6: Lecture 6: Shared resources + Tutorial Zephyr (1/3)
- Class 7: Lecture 7: Aperiodic task scheduling + Tutorial Zephyr (2/3)
- Class 8: Lecture 8: Additional topics related with RT scheduling + Tutorial Zephyr (3/3)
- Class 9: Lecture 9: Optimizations + Project presentation
- Class 10: Lecture 10: Multiprocessor Scheduling + Project work
- Class 11 to Last-1: Project work
- Last Class: Project Demo and Presentation

Bibliography

- Base
  - Giorgio Buttazzo (2011). "HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications", Third Edition, Springer, 2011. (available at the course site
- Complementary
  - Peter Marwedel (2021), "Embedded System Desig-Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things", Springer, ISBN 978-3-030-60910-8
  - Kopetz, H. (2011). Real-Time Systems: Design Principles for Distributed Embedded Applications (Real-Time Systems Series), 2nd Edition , Springer, 2011.
  - Xiaocong Fan (2015). Real-Time Embedded Systems: Design Principles and Engineering Practices, 1st Edition, Springer, 2015
  - Jane W.S. Liu (2000). Real-Time Systems. Prentice Hall.
  - Richard Barry. Using the FreeRTOS Real-Time Kernel - A practical guide, Real-Time Engineers, Ltd. (available at the course site)

And now ....

- It is time to wake up and start working!



Basic Concepts on RTS

Real-Time Operative Systems Course

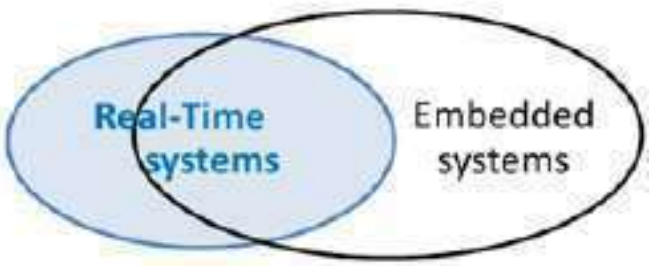
Paulo Pedreiras, DETI/UA/IT

September 13, 2022

Embedded vs Real-Time systems

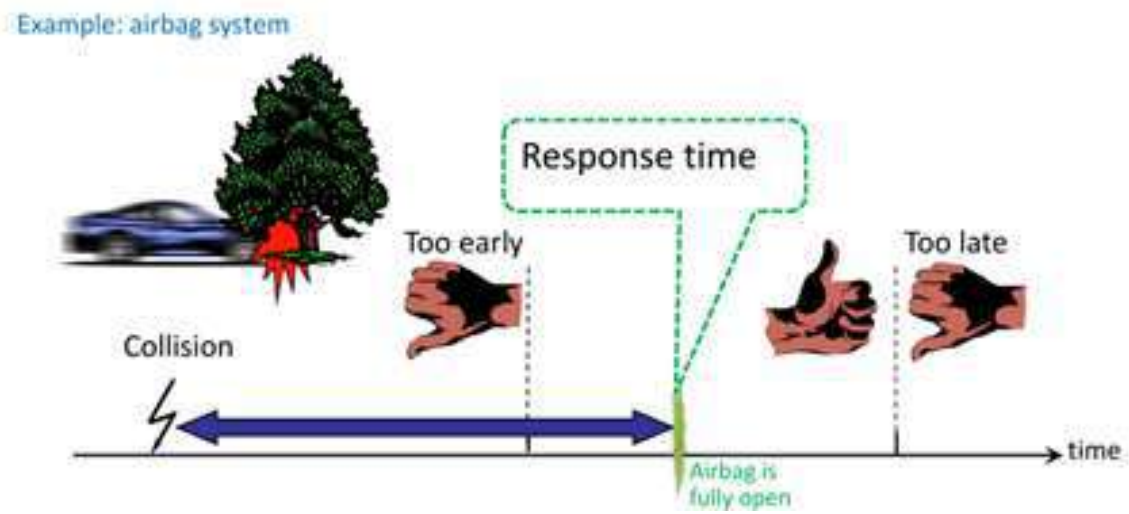
Real-Time and Embedded Systems are often confused. **But they are not the same ...**

- Real-time And Embedded:
  - Nuclear reactor control, Flight control, Mobile phone
- Real-time, but not embedded:
  - Stock trading system, Video streaming/processing
- Embedded, but not real-time:
  - Home temperature control, Washing machine, refrigerator, etc.



A few definitions related with "Real-Time"

- There is a wide variety of definitions related to Real-Time, systems dealing with Real-Time and the services that they provide.
- All of these definitions have in common the fact that express the **dependency of a computer system on the time, as it exists in a particular physical process** .



Definitions related with "Real-Time"

- Real-Time Service or Function
  - Which must be performed or provided within finite time intervals imposed by a physical process
- Real-Time System
  - One who contains at least one feature of real-time or at least providing a service of real-time
- Real-Time Science
  - Branch of computer science that studies the introduction of Real-Time in computational systems.

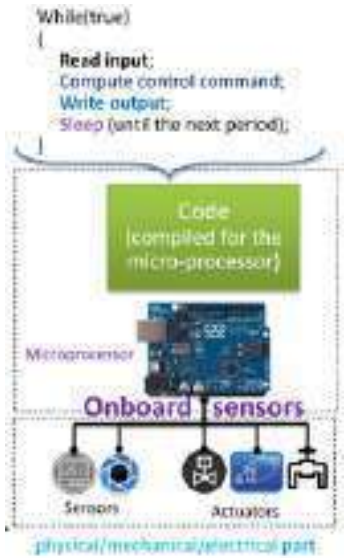
Real-Time Computation

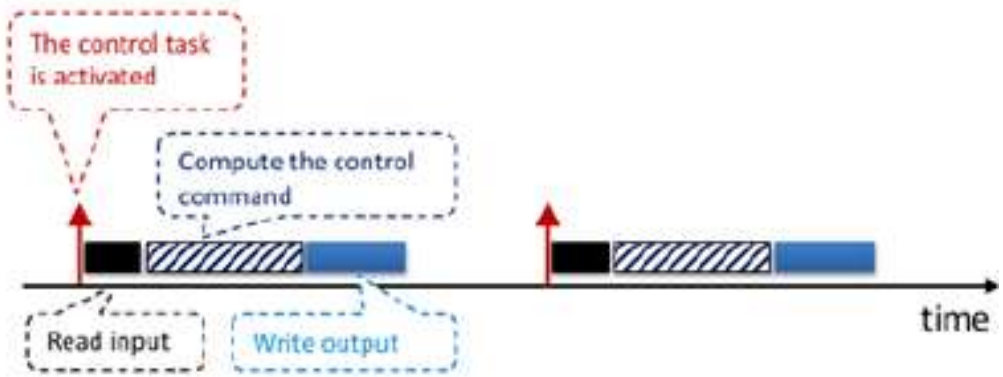
- The computation results must be
  - Logically correct
  - Delivered on time

(Stankovic, 1988)

Real-Time systems - implementation

- Simple case, with an infinite loop:

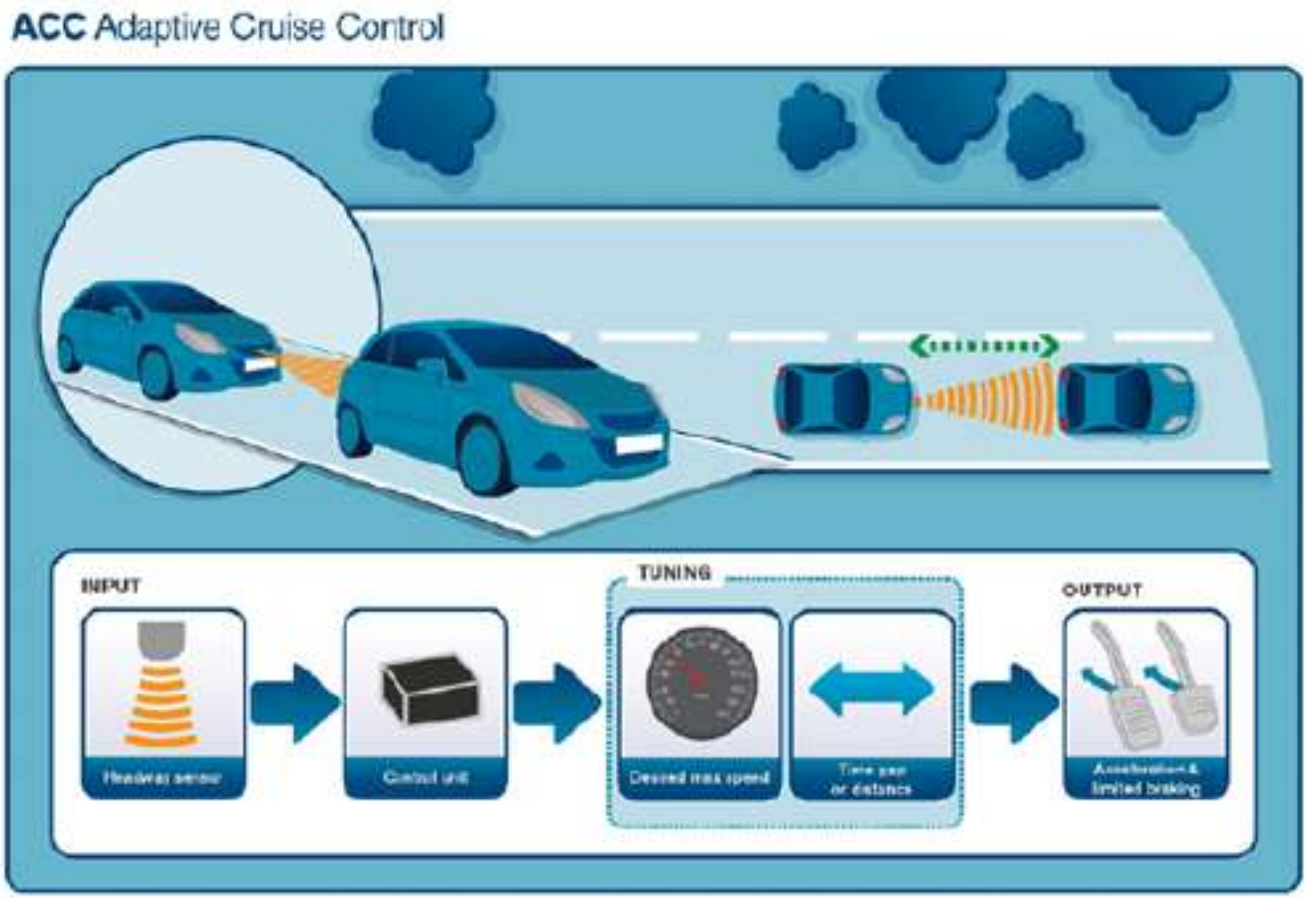




- Looks simple and predictable, right?

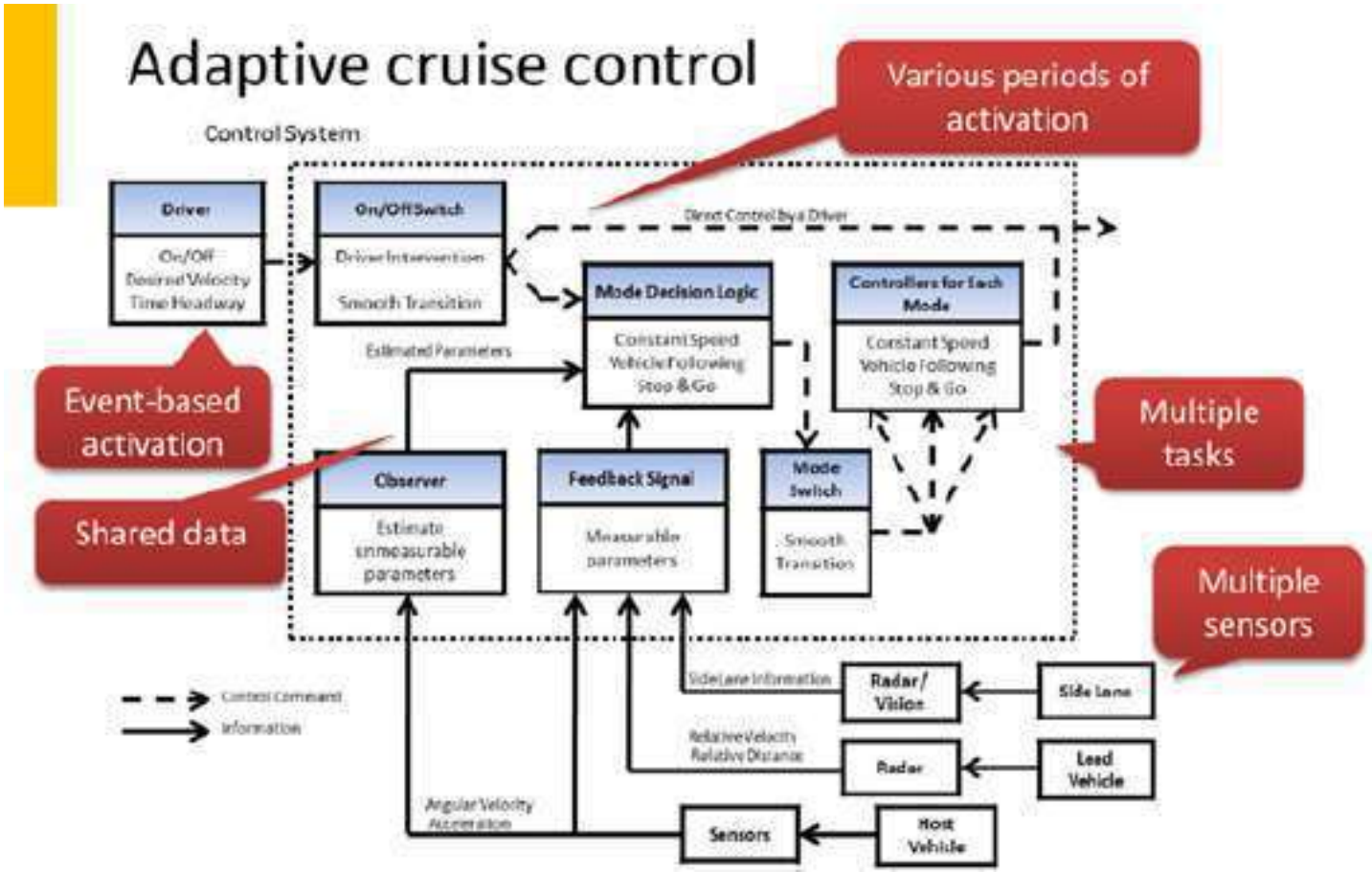
Real-Time systems - implementation

- But it can get **much more** complicated ...



Real-Time systems - implementation

- Does is still look simple and predictable?



https://codingcommunity.de/jaguar-cruise-control-diagram.html

Objective of the study of RTS

- Generally, when a computer system monitors the state of a given physical process and, if necessary, acts on it in time, then it is a real-time system.
- All living beings are real-time in relation to their natural habitats, which are its "real-time system".
- On the other hand, when we build (programmable) machines to interact with physical processes, we need to use Operative Systems, Programming Techniques and Analysis Methods that allow us to have confidence in its ability to carry out **correct and timely actions** .

Objective of the study of RTS

- The main objective of the study of Real-Time Systems is thus the development of techniques for
  - Design,
  - Analysis, and
  - Verification

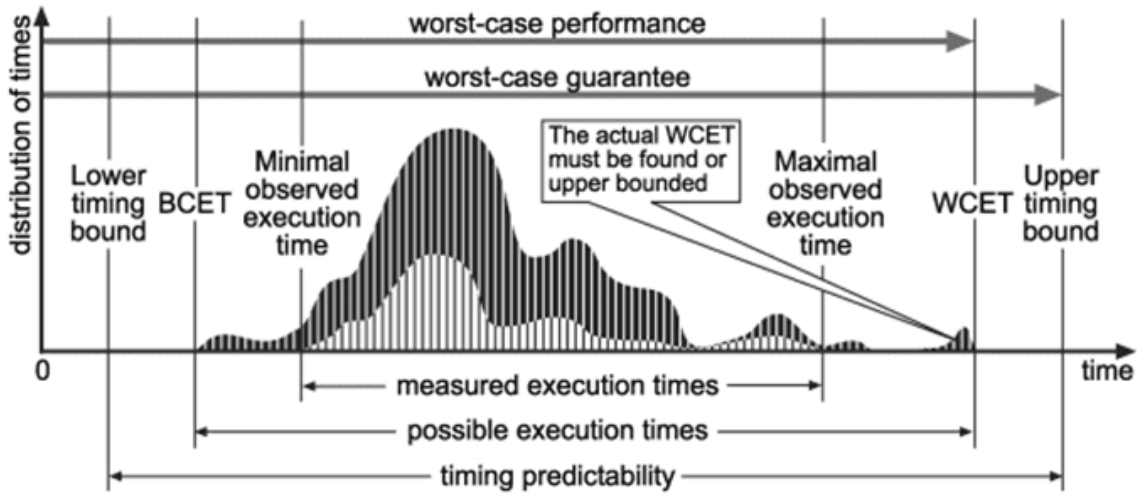
  that allow to assure that a given (computer, in our case) system has an appropriate timing behavior, satisfying the requirements imposed by the dynamics of the system with which it interacts

Objective of the study of RTS

- Regarding the computational activities of RTS, the main aspects to consider are:
  - Execution time
  - Response time
  - Regularity of periodic events

Objective of the study of RTS

Despite essential, these aspects are far from trivial:



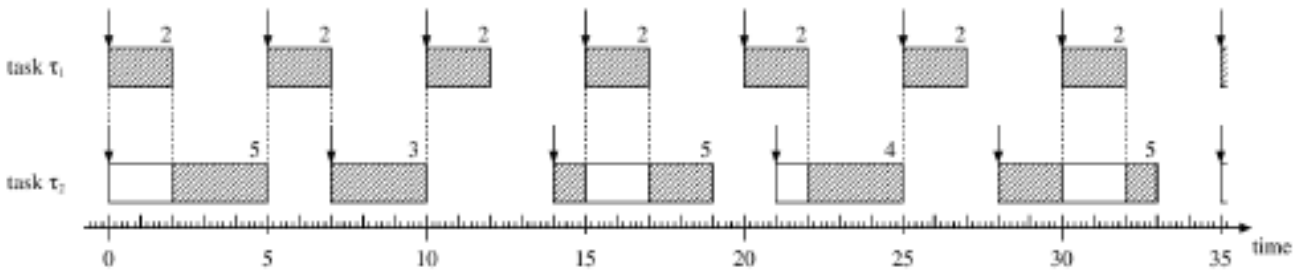
Analysing Extreme Value Theory Efficiency In Probabilistic Wcet Estimation With Block Maxima, ISSN:2448-0959

Reasons for this behaviour include:

- Execution time
  - Code structure (language, conditional execution, cycles)
  - DMA, cache, pipeline
  - Operative System or kernel (system calls)

Objective of the study of RTS

- Response time and regularity
  - Interrupts
  - Multi-tasking
  - Access to shared resources (buses, communication ports, ...)



Impact of interference on the Worst-case Response Time

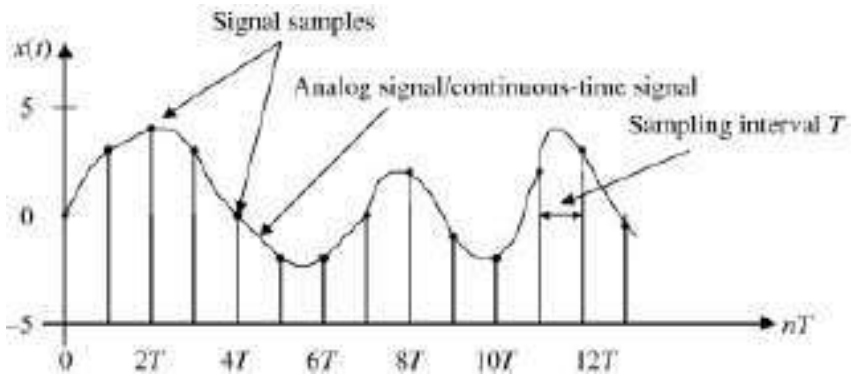
"Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited", Reinder J. Bril et al

Requirements of Real-Time Systems

- The requirements commonly imposed to real-time systems are of three types:
  - Functional
  - **Temporal** (our main focus)
  - Dependability

Functional requirements

- Data gathering
  - Sampling of system variables (real-time entities), both analog and discrete
- Supervise and Control
  - Direct access to sensors and actuators
- Interaction with the operator
  - System status information, logs, support to correct system operation, warnings, ...

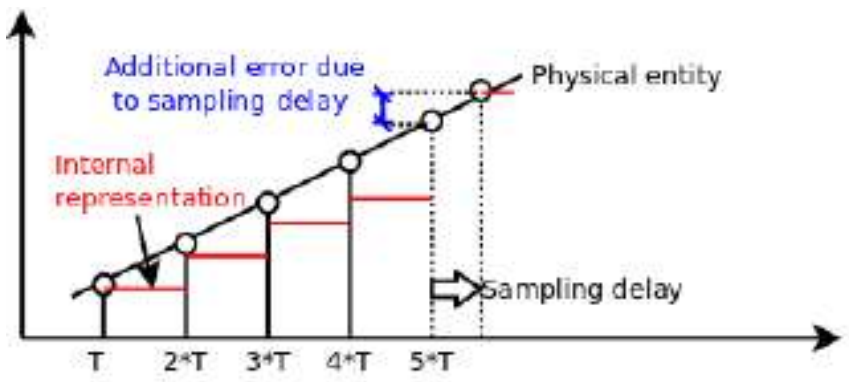


Periodic sampling illustration

Functional requirements

Data gathering

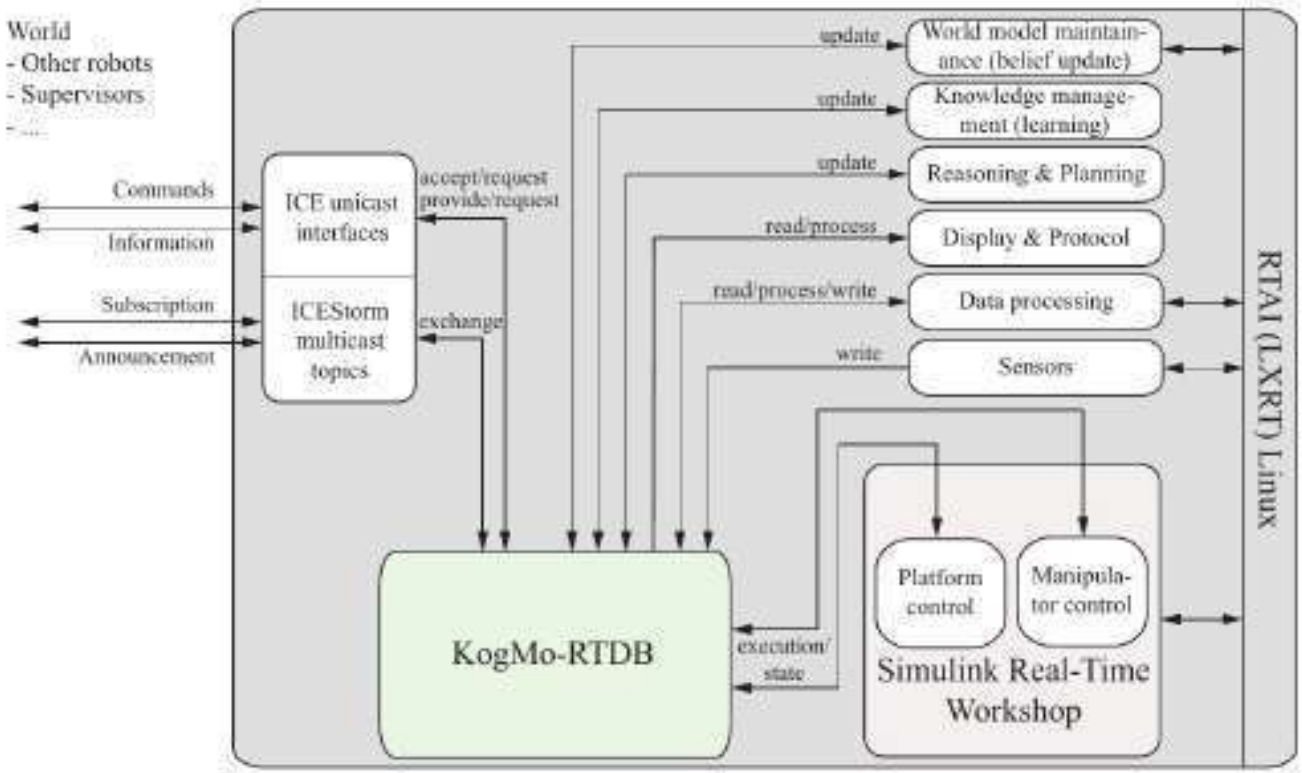
- The Real-Time computer operates on **local images** (internal variables) that represent the physical entities
- Each image of a real-time entity has a limited time validity, due to the temporal dynamics of the physical process
- The set of images of the real-time entities forms a **Real-Time Database**
- The real-time database must be updated to keep consistency between the physical world and its the internal representation



Impact of time delay in sampling error

Functional requirements

Illustration of a RTDB for Multi-Robot Systems



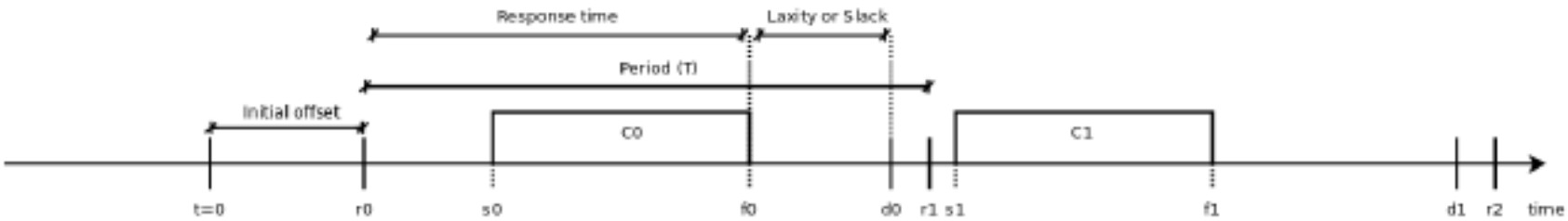
An Architecture for Real-Time Control in Multi-robot Systems, DOI:10.1007/978-3-642-10403-9_5

Temporal requirements

- Usually arise from the physical dynamics of the process to be managed or controlled
- Impose restrictions:
  - Delays the observation of the system state
  - Delays computing the new control values (acting)
  - Variations of previous delays (jitter)
- that must be followed in **all instances** (including the worst case) and not only on average

Temporal requirements

Terminology (1/2)



Initial offset (ϕ) : time before the first release/activation (job) of a task.

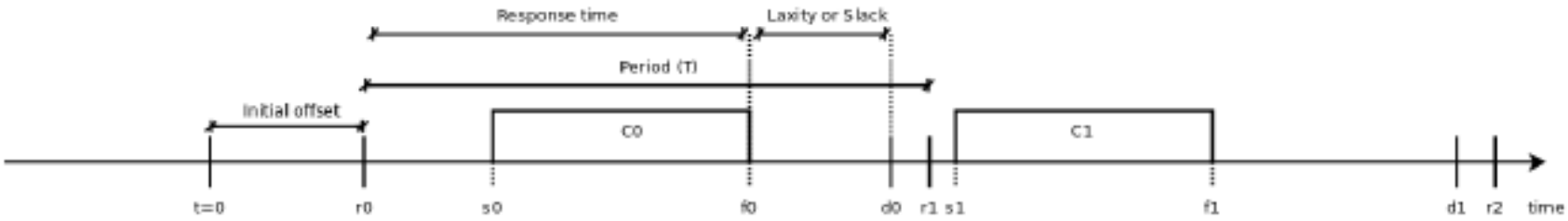
Period (T) : time between successive jobs of a task. Can be a Minimum Inter-Arrival time (MIT) for sporadic tasks.

Start/activation (r_i) : time instant of the i_{th} job of a task.
$r_i = \phi + k \cdot T_i$ for periodic tasks.

Finish/completion time (t_i) : time instant in which the i_{th} job of a task terminates.

Temporal requirements

Terminology (2/2)



Execution/computation time (C_i) :time necessary to the processor for executing the task instance without interruption.

Absolute deadline (d_i) : time instant by which the i_{th} execution of a task must complete

Response time (R_i) : time elapsed between the release of the i_{th} job of a task and its completion ($R_i = f_i - r_i$)

Slack/Laxity (L_i) : maximum time a task can be delayed on its activation to complete within its deadline ($L_i = d_i - r_i - C_i$)

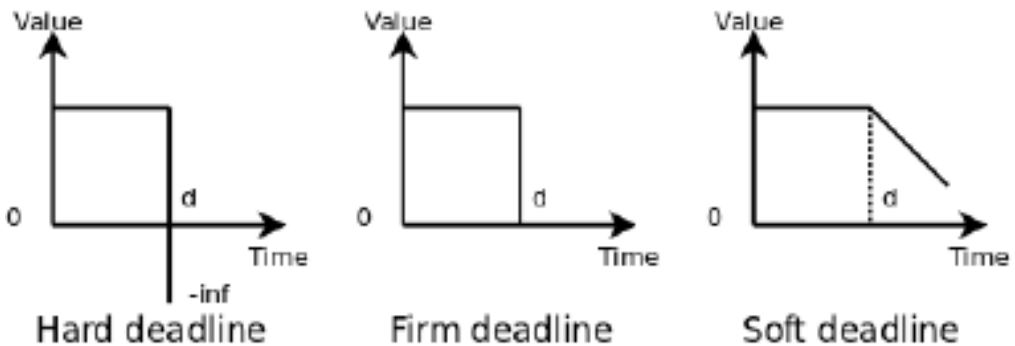
Temporal requirements

Classification of the temporal constraints, according with the **usefulness** of the result:

Soft: temporal constraint in which the result retains some utility to the application, even after a temporal limit D, although affected by a degradation of quality of service.

Firm: temporal constraint in which the result loses any usefulness to the application after a temporal limit D.

Hard: temporal restriction that, when not met, can lead to a catastrophic failure.



Temporal requirements

Classification of the Real-Time Systems, according with the **temporal** constraints:

Soft Real-Time: The system only has **firm or soft** real-time constraints (e.g., simulators, multimedia systems)

Hard Real-Time The system has **at least one hard real-time constraint** . These are the so-called safety-critical systems (e.g. airplane control, missile control, nuclear plants control, control of dangerous industrial processes)

Best Effort: The system **is not subject** to real-time constraints

Dependability requirements

- Real-time systems are typically used in critical applications, in which failures may endanger human lives or result in high economic impact/losses.
- This results in a requirement of **High Reliability**
  - Hard real-time systems have typically ultra-high reliability requirements ($< 10^{-9} failures/hour$)
  - Cannot be verified experimentally!
  - Validation requires solid analytic support (among other aspects)

Dependability requirements

- Important aspect to consider in safety-critical systems:
  - **Stable interfaces** between the critical and the remaining subsystems, in order to avoid error propagation between each other.
  - **Well defined worst case scenarios** . The system must have an adequate amount of resources to deal with worst case scenarios without resorting to probabilistic arguments, i.e. must provide service guarantees even in such scenarios.
  - Architecture composed of autonomous subsystems, whose properties can be checked independently of the others (**composability**).

Summary

- Notion of real-time and real-time system
- Antagonism between real-time and best effort
- Objectives of the study of RTS – how to guarantee the adequate temporal behavior
- Aspects to consider: execution time, response-time and regularity of periodic events
- Requirements of RTS: functional, temporal and dependability
- Constraints soft, firm and hard, and hard real-time vs soft real-time
- The importance of consider the worst-case scenario

Computation Models and Introduction to RTOS Architecture and Services

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

September 21, 2022

Last lecture

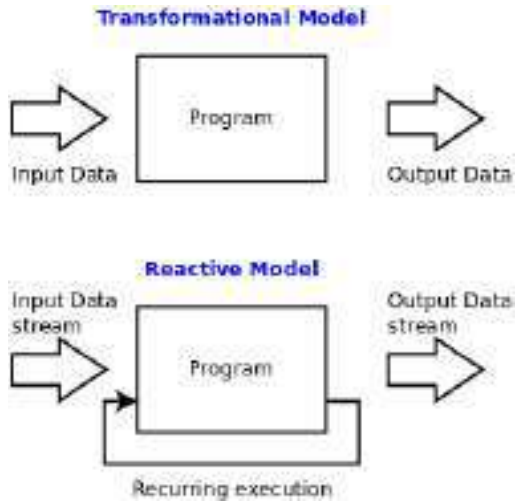
- Notion of real-time and real-time system
- Antagonism between real-time and best effort
- Aspects to consider: execution time, response-time and regularity of periodic events
- Requirements of RTS: functional, temporal and dependability
- Basic nomenclature
- Constraints: soft, firm and hard, and hard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

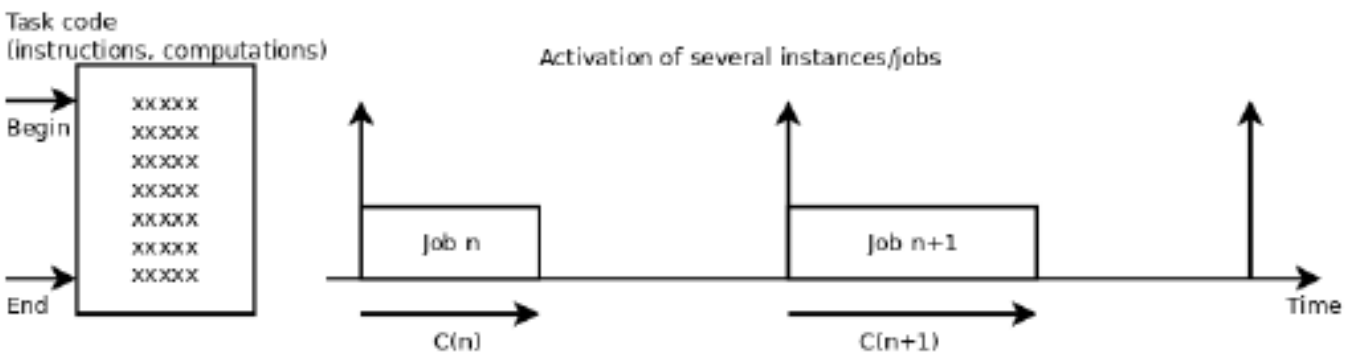
Real-time model

- **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.



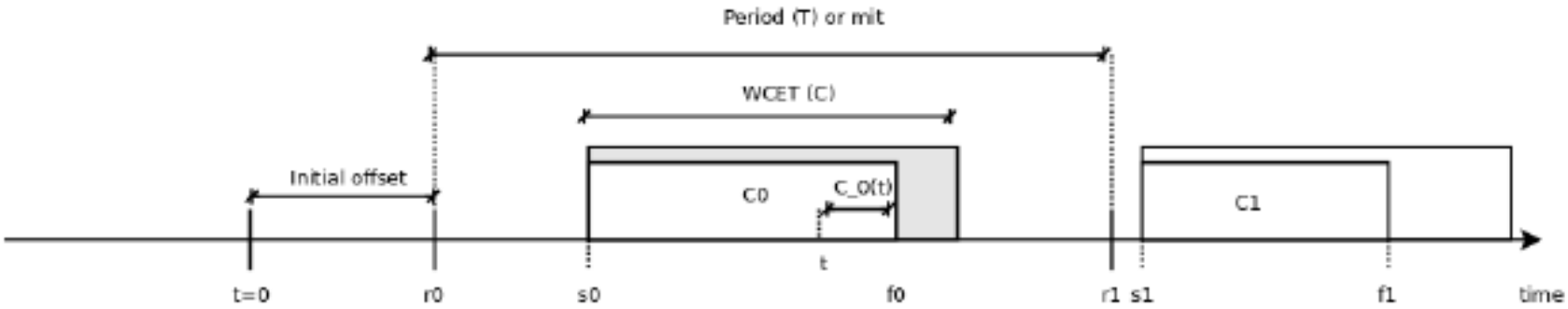
Processes, threads, tasks, activities and jobs

- **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

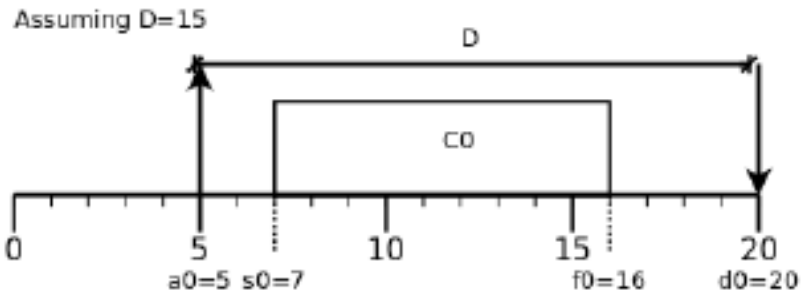
Temporal attributes



- 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: $U_i = \sum_{i=1}^{N} \frac{C_i}{T_i}$

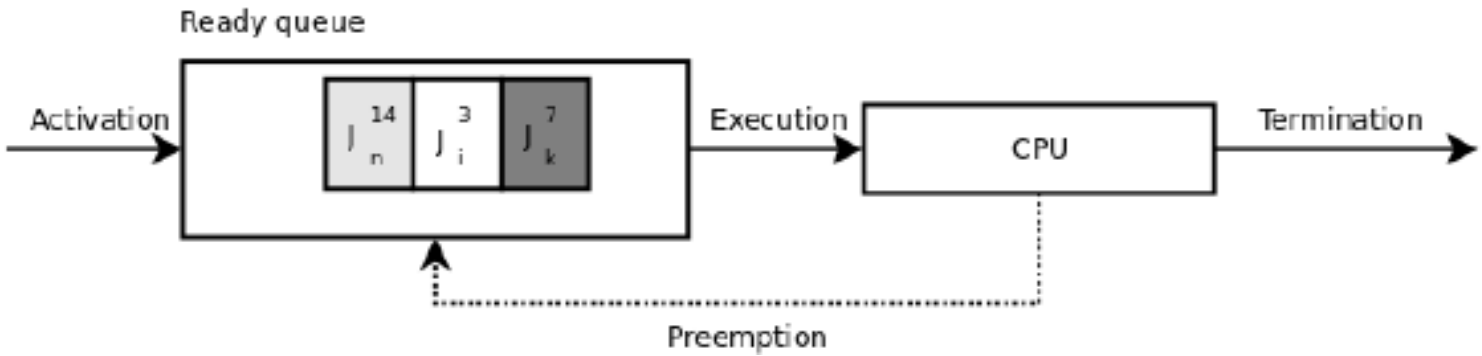
Task states and execution

- 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

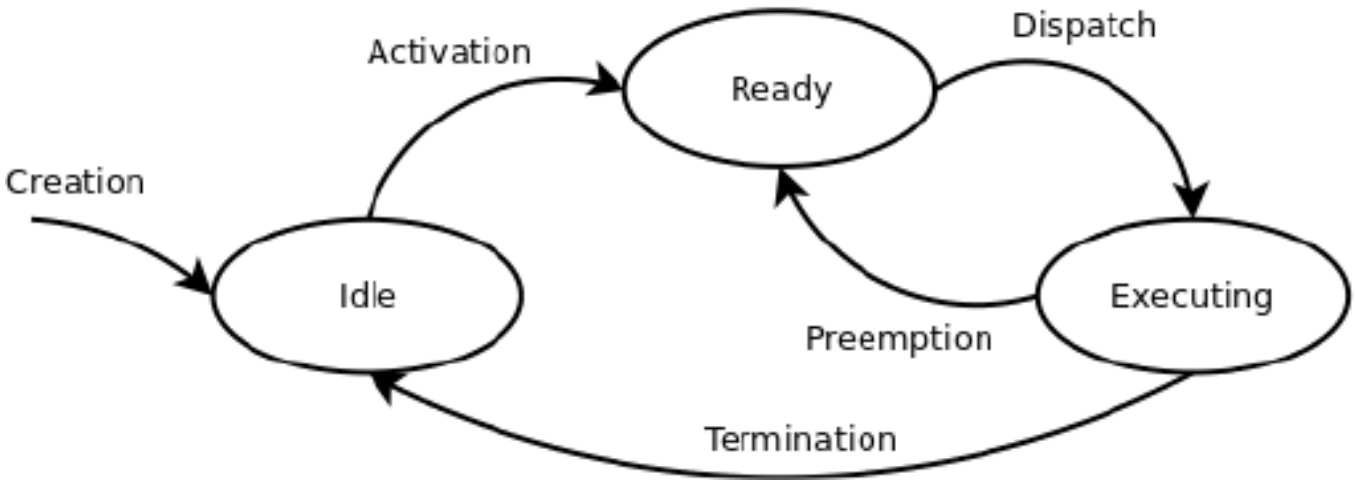
Execution of task instances (jobs)

- 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!



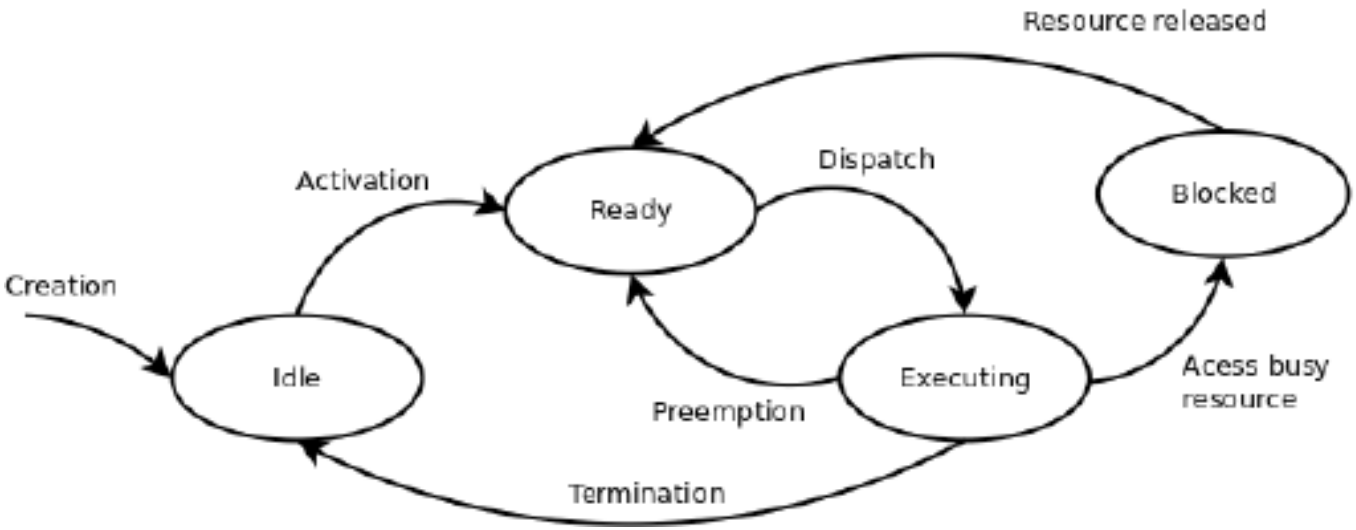
Task states

- 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** .



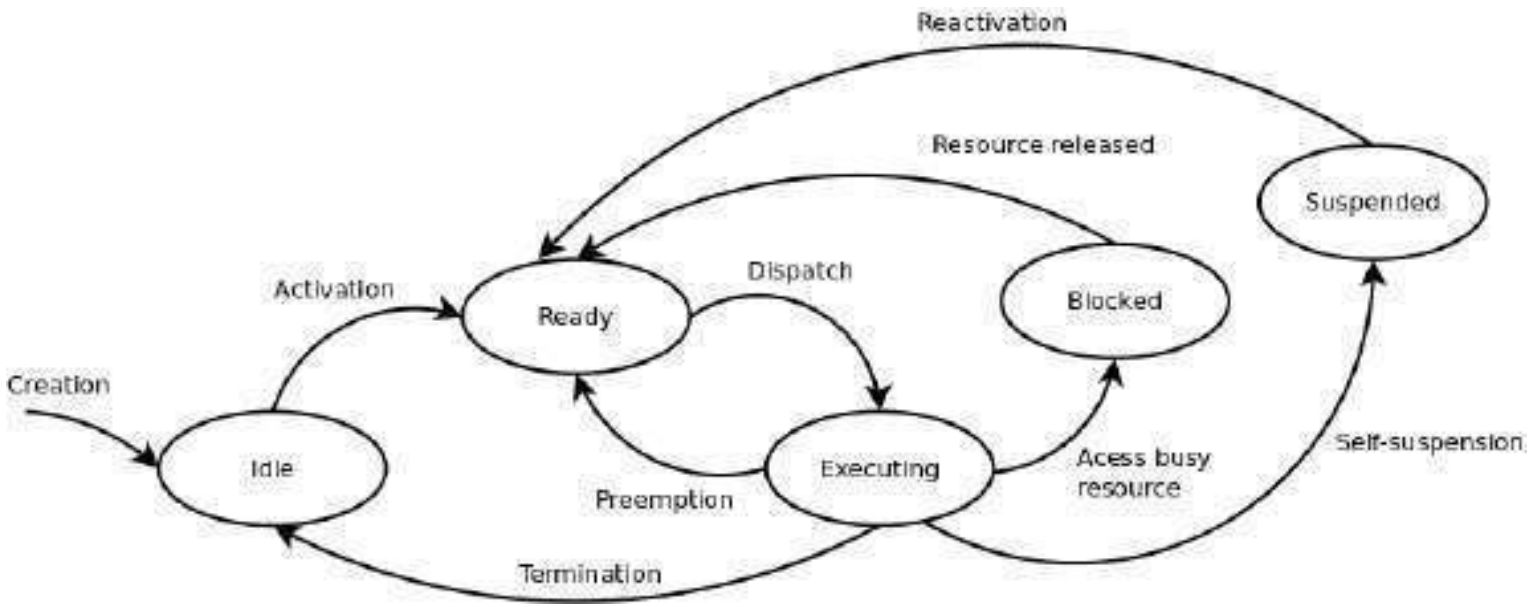
Task states

- 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.



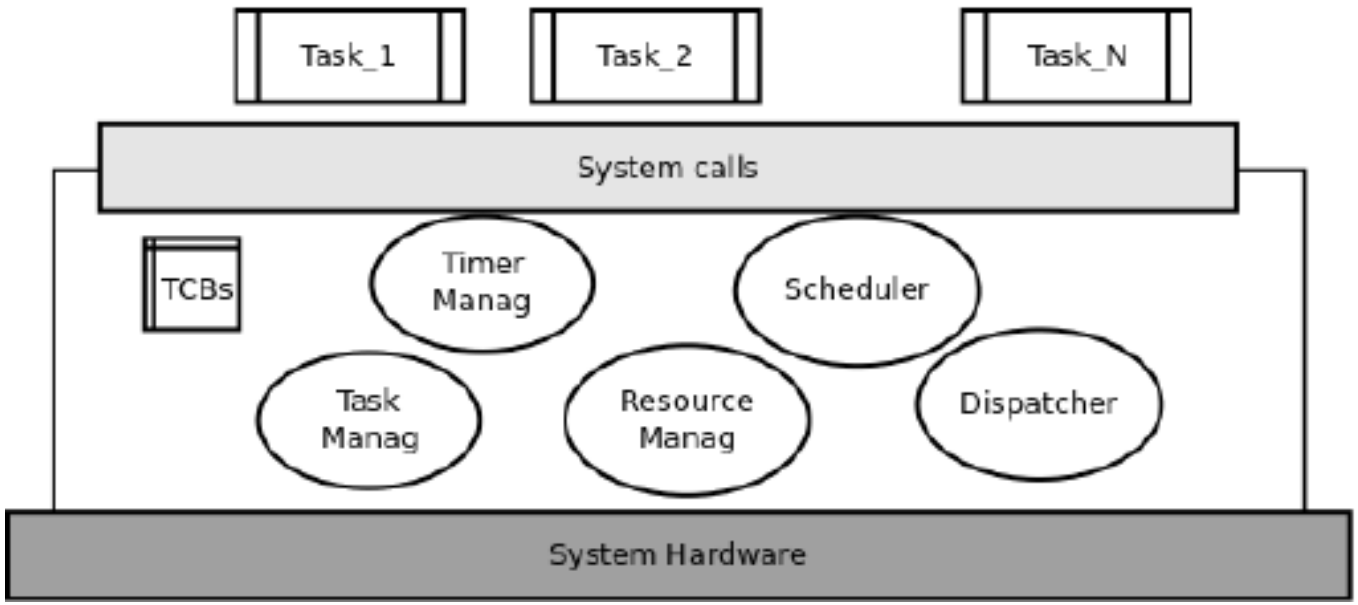
Task states

- 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

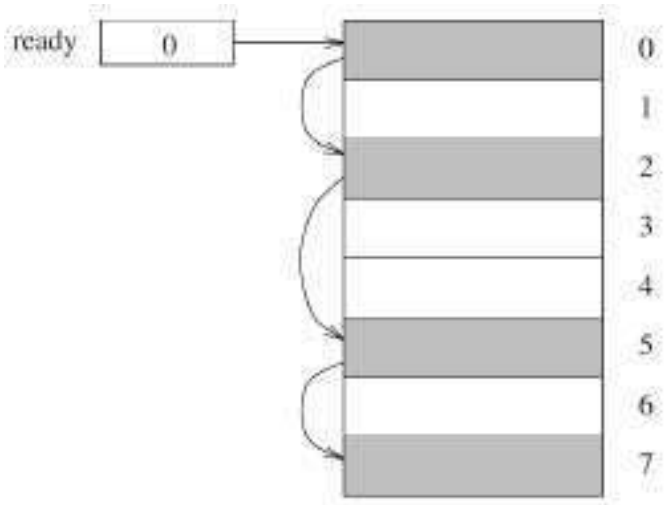
The **TCB** (task control block)

- 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

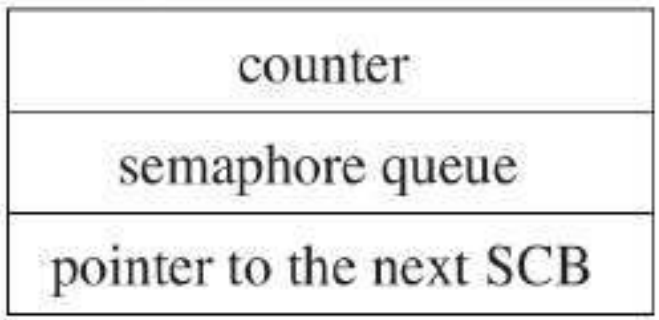
- TCBs are often defined in a static array, but are normally structured as linked lists to facilitate operations and searches over the task set.
- E.g., the ready queue (list of ready tasks sorted by a given criteria) is maintained as a linked list. These linked lists may be implemented e.g. through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

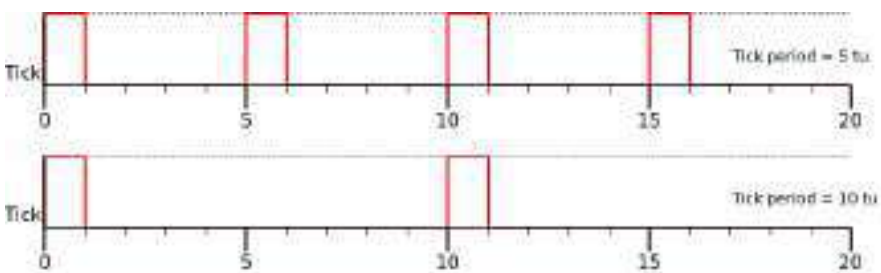
Time management functions

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

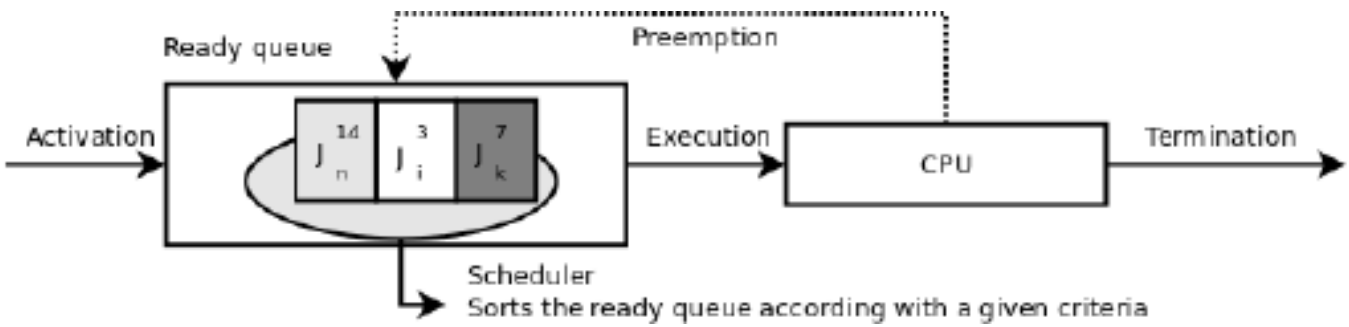
Measurement of time intervals

- 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

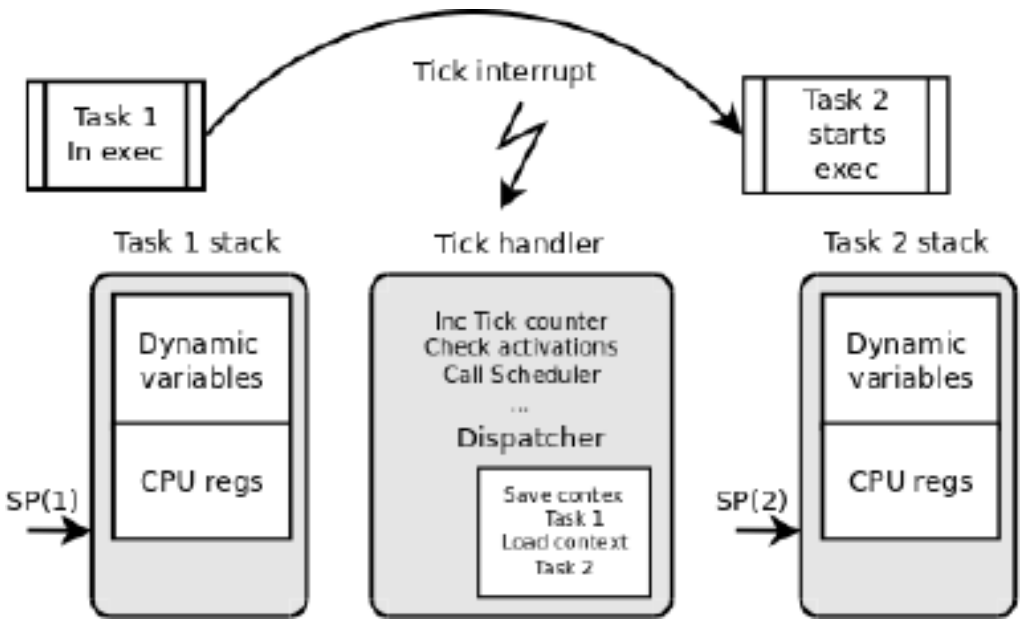
- The scheduler selects which task to execute among the (eventually) several ready tasks
- In real-time systems must be based on a deterministic criteria, which must allow computing an upper bound for the time that a given task may have to wait on the ready queue.



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

Scheduling Basics

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

October 10, 2022

Last lecture

- Computation models and Real-Time Kernels and Operating Systems
- Computation models:
  - Tasks with specific temporal constraints;
  - The Event- and Time-Trigger paradigms
- Real-Time OSs and kernels:
  - General architecture
  - Task states
  - Basic components

Agenda for today

Scheduling basics

- Task scheduling - basic concepts and taxonomy
- Basic scheduling techniques
- Static cyclic scheduling

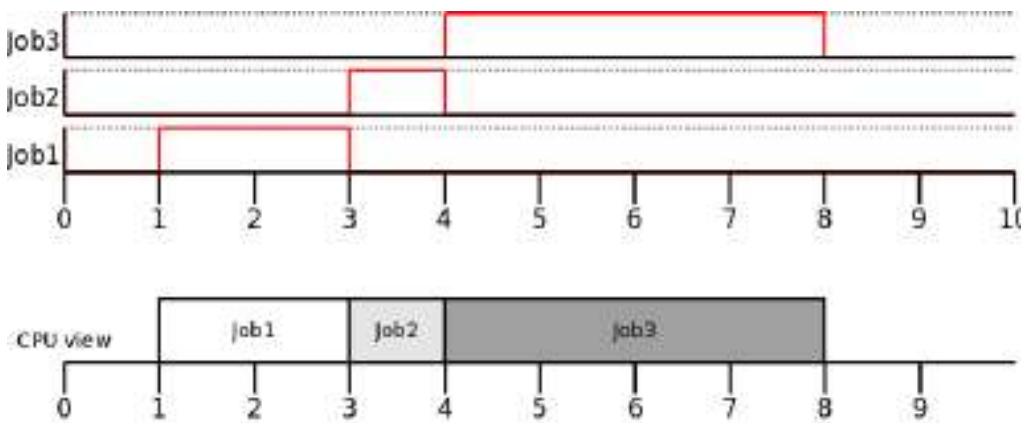
Scheduling Definition

Task scheduling (also applies to messages, with due adaptations)

- Sequence of task executions (jobs) in one or more processors
- Application of $\mathbb{R}^+$ (time) in $\mathbb{N}_0$ (task set), assigning to each time instant "t" a task/job "i" that executes in that time instant

  $\sigma(t) : \mathbb{R}^+ \rightarrow \mathbb{N}_0$

  $i = \sigma(t), t \in \mathbb{R}^+$, i=0 means that the processor is idle
- $\sigma(t)$ is a step function, which can be expressed e.g. as a Gantt graph



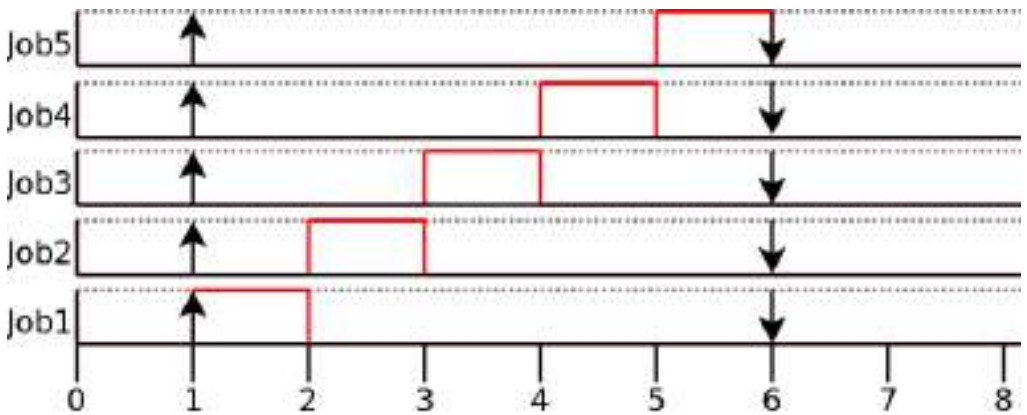
Scheduling Definition

- A schedule is called **feasible** if it fulfills all the task requirements
  - temporal, non-preemption, shared resources, precedences, ...
- A task set is called **schedulable** if there is at least one feasible schedule for that task set

The scheduling problem: easy to formulate, but hard to solve!

- Given:
  - A task set
  - Requirements of the tasks (or cost function)
- Find a time attribution of processor(s) to tasks so that:
  - Tasks are completely executed, and
  - Meet they requirements (or minimize the cost function)

E.g. $J = J_i(C_i = 1, a_i = 1, D_i = 5), i = \{1..5\}$



Scheduling problem

Exercise:

- Build a Gantt diagram for the execution of the following periodic tasks, admitting $D_i = T_i$ and no preemption.
  - $\tau = \{(1, 5)(6, 10)\}$
- Is the execution order important? Why?

Scheduling algorithms

- A **scheduling algorithm** is a method for solving the scheduling problem.
  - Note: don't confuse scheduling algorithm (the process/method) with schedule (the result)
- Classification of scheduling algorithms:
  - Preemptive vs non-preemptive
  - Static vs dynamic (priorities)
  - Off-line vs on-line
  - Optimal vs sub-optimal
  - With strict guarantees vs best effort

A short note on temporal complexity

- Measurement of the **growth** of the execution time of an algorithm as a function of the problem size (e.g. the number of elements of a vector, the number of tasks of a real-time system)
- Expressed via the $O()$ operator (big O notation)
- $O()$ arithmetic, n=problem dimension, k=constant
  - $O(k) = O(1)$
  - $O(kn) = O(n)$
  - $O(k_1 \cdot n^m + k_2 \cdot n^{m-1} + ... + k_{m+1}) = O(n^m)$

Compl. $= O(N)$

```
for (k=0;k<N;k++)
    a[k]=0;
```

Compl. $= O(N^2)$

```
for (k=0;k<N-1;k++)
    for (m=k;m<N;m++)
        if a[k]<a[m]
            swap(a[k],a[m]);
```

Compl. $= O(N^N)$

Computation of the permutations of a set $A = a_i, i = 1..N$

Basic algorithms

EDD - Earliest Due Date (Jackson, 1955)

- Single instance tasks fired synchronously:$J = Ji(Ci, (ai = 0,)Di), i = 1..n$
- Executing the tasks by non-decreasing deadlines minimizes the maximum lateness ($L_{max}(J) = max_i(f_i - d_i)$)
- Complexity: O(n.log(n))

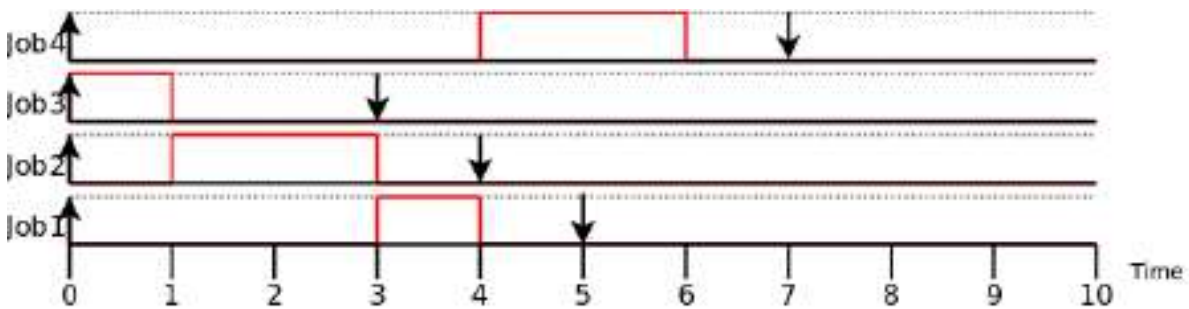
E.g. $J = \{J_1(1,5), J_2(2,4), J_3(1,3), J_4(2,7)\}$
Determine the maximum lateness!

Basic algorithms

EDD - Earliest Due Date (Jackson, 1955)

- Single instance tasks fired synchronously: $J = Ji(Ci, (ai = 0,)Di), i = 1..n$
- Executing the tasks by non-decreasing deadlines minimizes the maximum lateness ($L_{max}(J) = max_i(f_i - d_i)$)
- Complexity: O(n.log(n))

E.g. $J = \{J_1(1,5), J_2(2,4), J_3(1,3), J_4(2,7)\}$
Determine the maximum lateness!



Lmax,EDD(J) = -1

Basic algorithms

EDF - Earliest Deadline First (Liu and Layland, 1973; Horn, 1974)

- Single instance or periodic, asynchronous arrivals, preemptive: $J = J_i(C_i, a_i, D_i), i = 1..n$
- Always executing the task with shorter absolute deadline minimizes the maximum latency $L_{max}(J) = max_i(f_i - d_i)$
- Complexity: $O(n.log(n))$, **Optimal** among all scheduling algorithms of this class

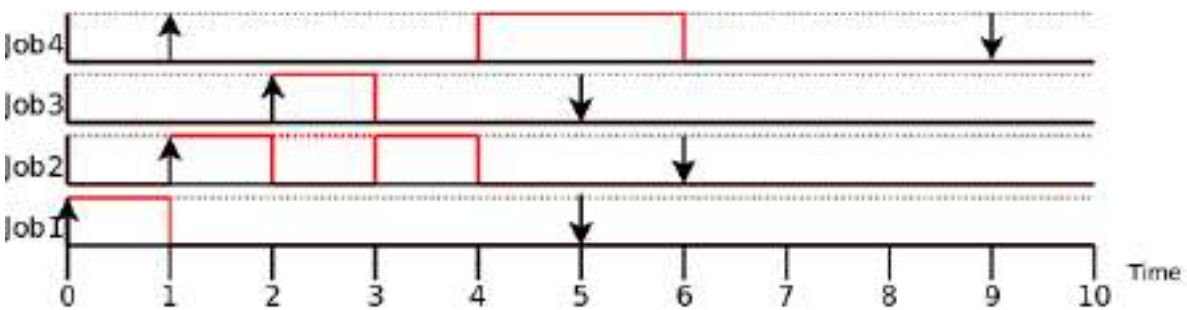
E.g. $J = \{J_1(1, 0, 5), J_2(2, 1, 5), J_3(1, 2, 3), J_4(2, 1, 8)\}$
Determine the maximum lateness!

Basic algorithms

EDF - Earliest Deadline First (Liu and Layland, 1973; Horn, 1974)

- Single instance or periodic, asynchronous arrivals, preemptive: $J = J_i(C_i, a_i, D_i), i = 1..n$
- Always executing the task with shorter absolute deadline minimizes the maximum latency $L_{max}(J) = max_i(f_i - d_i)$
- Complexity: $O(n.log(n))$, **Optimal** among all scheduling algorithms of this class

E.g. $J = \{J_1(1, 0, 5), J_2(2, 1, 5), J_3(1, 2, 3), J_4(2, 1, 8)\}$
Determine the maximum lateness!



Lmax,EDD(J) = -2

Basic algorithms

BB – Branch and Bound (Bratley, 1971)

- Single instance or periodic tasks, asynchronous arrivals, non-preemptive: $J = J_i(C_i, a_i, D_i), i = 1..n$
- Based on building an exhaustive search in the permutation tree space, finding all possible execution sequences:
- Complexity: O(n!)

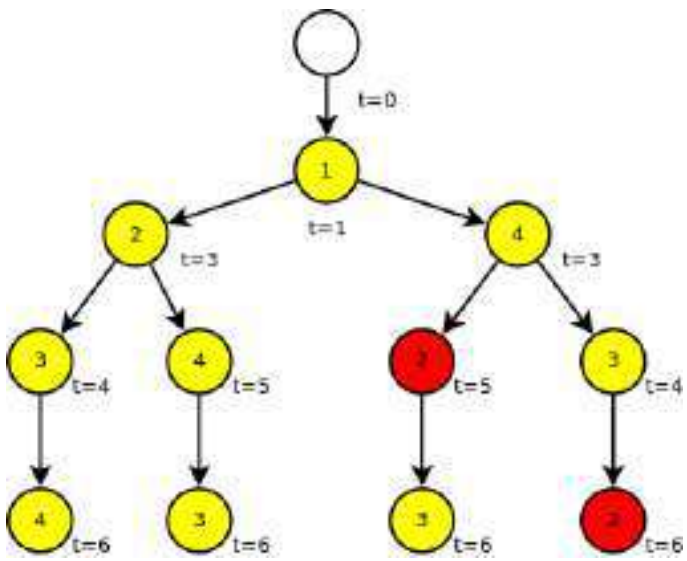
E.g. $J = \{J_1(1, 0, 5), J_2(2, 1, 3), J_3(1, 2, 4), J_4(2, 1, 7)\}$

Basic algorithms

BB – Branch and Bound (Bratley, 1971)

- Single instance or periodic tasks, asynchronous arrivals, non-preemptive: $J = J_i(C_i, a_i, D_i), i = 1..n$
- Based on building an exhaustive search in the permutation tree space, finding all possible execution sequences:
- Complexity: O(n!)

E.g. $J = \{J_1(1, 0, 5), J_2(2, 1, 3), J_3(1, 2, 4), J_4(2, 1, 7)\}$



Periodic task scheduling

Periodic tasks

- The release/activation instants are known **a priori**
- $\Gamma = \{\tau_i(C_i, \phi_i, T_i, D_i)\}, i = \{1...n\}$
- $a_{i,k} = \phi_i + (k-1) \cdot T_i, k = 1, 2, ...$

Thus, in this case the schedule can be built:

Online Tasks to execute are selected as they are released and finish, during normal system operation (addressed in next class)

Offline The task execution order is computed before the system enters in normal operation and stored in a table, which is used at runtime time to execute the tasks (static cyclic scheduling).

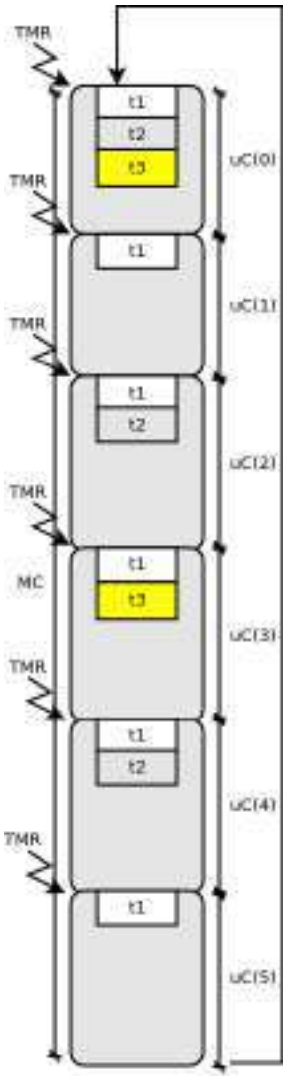
Static cyclic scheduling

- The table is organized in micro-cycles (μC) with a fixed duration. This way it is possible to release tasks periodically
- The micro-cycles are triggered by a Timer
- Scanning the whole table repeatedly generates a periodic pattern, called macro-cycle (MC)

$$\Gamma = \{\tau_i(C_i, \phi_i, T_i, D_i)\}, i = \{1...n\}$$
$$\mu C = GCD(T_i); MC = mCM(T_i)$$

Example:

- $\phi_i = 0$
- $T_1 = 5ms$; $T_2 = 10ms$; $T_3 = 15ms$



Static cyclic scheduling

Pros

- Very simple implementation (timer+table)
- Execution overhead very low (simple dispatcher)
- Permits complex optimizations (e.g. jitter reduction, check precedence constraints)

Cons

- Doesn't scale (changes on the tasks may incur in massive changes on the table. In particular the table size may be prohibitively high)
- Sensitive to overloads, which may cause the "domino effect", i.e., sequence of consecutive tasks failing its deadlines due to a bad-behaving task.

Static cyclic scheduling - Algorithm

How to build the table:

- Compute the micro and macro cycles (μC and MC)
- Express the periods and phases of the tasks as an integer number of micro-cycles
- Compute the cycles where tasks are activated
- Using a suitable scheduling algorithm, determine the execution order of the ready tasks
- Check if all tasks scheduled for a give micro-cycle fit inside the cycle. Otherwise some of them have to be postponed for the following cycle(s)
- It may be necessary to break a task in several parts, so that that each one of them fits inside the respective micro-cycle

Summary

- The concept of temporal complexity
- Definition of schedule and scheduling algorithm
- Some basic scheduling techniques (EDD, EDF, BB)
- The static cyclic scheduling technique

Homework

Consider the following message set (syntax (C,T), with D=T):

$$\Gamma = \{(1,5);(2,10);(3,10);(4,20)\}$$

- Compute the utilization of each task and the global utilization
- Compute the micro and macrocycle
- Build the scheduling table

Fixed Priority Scheduling

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

October 17, 2022

Last lecture

- The concept of temporal complexity
- Definition of schedule and scheduling algorithm
- Some basic scheduling techniques (EDD, EDF, BB)
- The static cyclic scheduling technique

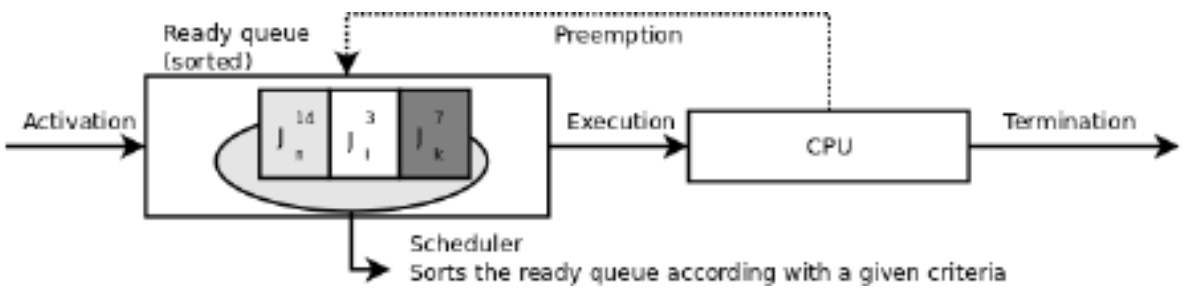
Agenda for today

Fixed-priority online scheduling

- Rate-Monotonic scheduling
- Deadline-Monotonic and arbitrary priorities
- Analysis:
  - The CPU utilization bound
  - Worst-Case Response-Time analysis

Online scheduling with fixed priorities

- The schedule is built while the system is operating normally (**online**) and is based on a **static criterium** (priority)
- The ready queue is **sorted by decreasing priorities** . Executes first the task with highest priority.
- If the system is preemptive, whenever a task job arrives to the ready queue, if it has higher priority than the one currently executing, it starts executing while the latter one is moved to the ready queue
- Complexity: $O(n)$



Online scheduling with fixed priorities

Pros (with respect to Static Cyclic Scheduling)

- Scales
- Changes on the task set are immediately taken into account by the scheduler
- Sporadic tasks are easily accommodated
- Deterministic behavior on overloads
  - Tasks are affected by priority level (lower priority are the first ones)

Cons (with respect to Static Cyclic Scheduling)

- More complex implementation (w.r.t. static cyclic scheduling)
- Higher execution overhead (scheduler + dispatcher)
- On overloads (e.g. due to SW errors or unpredicted events) an higher priority tasks may block the execution of lower priority ones

Online scheduling with fixed priorities

Rules for priority assignment to tasks

- Inversely proportional to period (RM – Rate Monotonic)
  - Optimal among fixed priority scheduling criteria if D=T
- Inversely proportional to deadline (DM – Deadline Monotonic)
  - Optimal if $D \leq T$
- Proportional to the task importance
  - Typically **reduces the schedulability** – not optimal
  - But very common is industry cases – e.g. automotive CAN

Online scheduling with fixed priorities

Schedulability tests

- As the schedule is built online it is fundamental to know a priori if a given task set is schedulable (i.e., its temporal requirements are met)
- There are two basic types of schedulability tests:
  - Based on CPU utilization rate
  - Based on response time

RM Scheduling

Schedulability tests for RM based on task utilization

- Valid with preemption, n independent tasks, D=T
- Liu&Layland's (1973), **Least Upper Bound (LUB)**

$$U(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1) \Rightarrow \text{One execution per period guaranteed}$$

- Bini&Buttazzo&Buttazzo's (2001), **Hyperbolic Bound**

$$\prod_{i=1}^{n} (\frac{C_i}{T_i} + 1) \leq 2 \Rightarrow \text{One execution per period guaranteed}$$

RM Scheduling

Interpretation of the Liu&Layland test

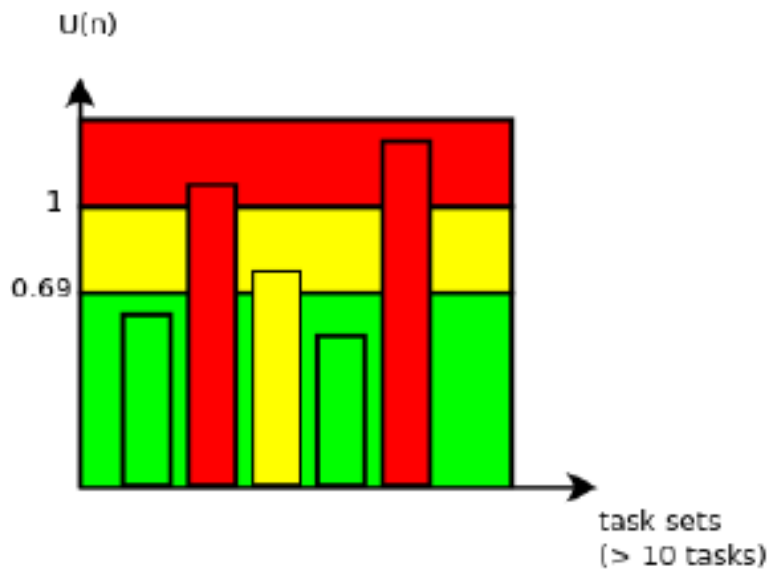
$U(n) > 1 \Rightarrow$ task set not schedulable (overload) – **necessary condition**

$U(n) \leq U_{lub} \Rightarrow$ task set is schedulable – **sufficient condition**

$1 \geq U(n) \geq U_{lub} \Rightarrow$ the test is **indeterminate**

Some U_{lub} values

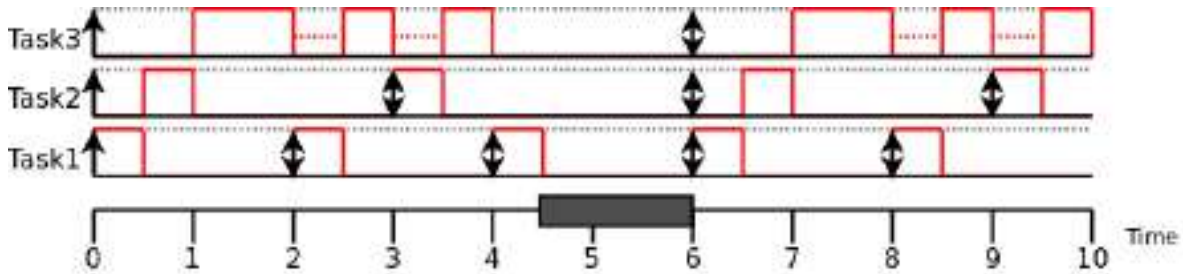
- $U_{lub}(1) = 1$
- $U_{lub}(2) = 0.83$
- $U_{lub}(3) = 0.78$
- ...
- $U_{lub}(n) \rightarrow ln(2) \approx 0.69$



RM Scheduling – example 1

Task properties

τ	C	T
1	0.5	2
2	0.5	3
3	2	6



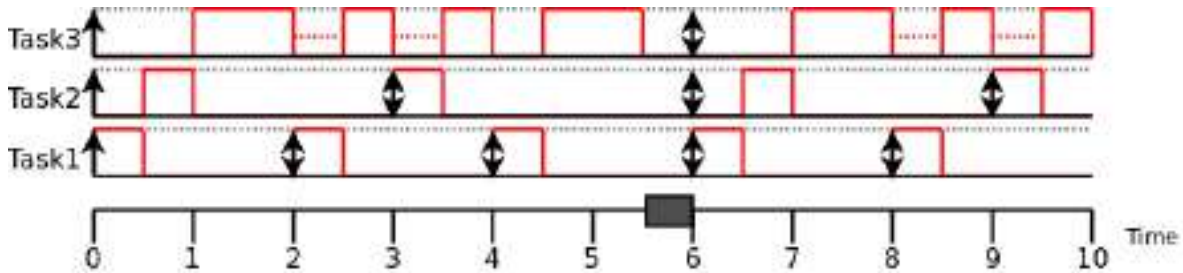
- $U = \frac{0.5}{2} + \frac{0.5}{3} + \frac{2}{6} \approx 0.75$
- $U_{lub}(3) = 0.78$. As $0.75 < 0.78$ one execution per period is guaranteed

RM Scheduling – example 2

Task properties

Same task set, but C_3 increases from 2 to 3 tu.

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

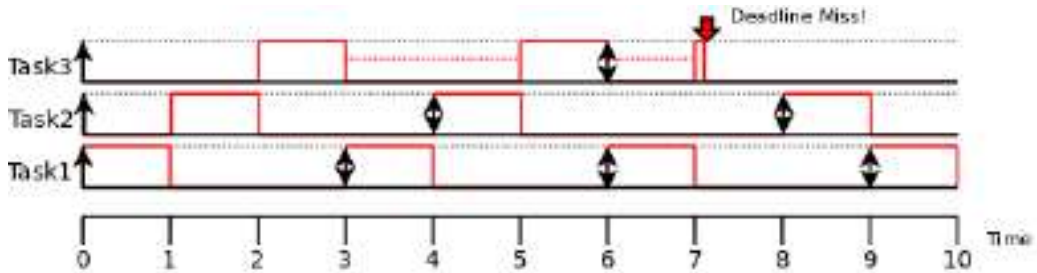


- $U = \frac{0.5}{2} + \frac{0.5}{3} + \frac{3}{6} \approx 0.92$
- $U_{lub}(3) = 0.78$. As $0.92 > 0.78$ one execution per period is **NOT guaranteed**
- But the task set is schedulable (see Gantt chart)

RM Scheduling – example 3

Task set properties

τ	C	T
1	1	3
2	1	4
3	2.1	6

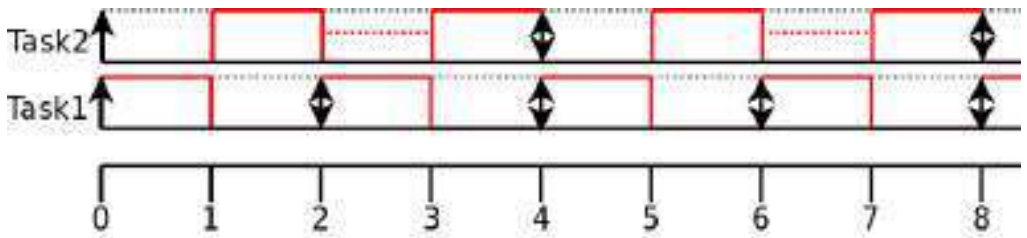


- $U = \frac{1}{3} + \frac{1}{4} + \frac{2.1}{6} \approx 0.93$
- $U_{lub}(3) = 0.78$. As $0.93 > 0.78$ one execution per period is **NOT guaranteed**
- And the task set indeed is **not schedulable** (see Gantt chart)

RM Scheduling – Harmonic Periods

Particular case: if the task **periods are harmonic** then the task set is schedlable iif $U(n) \leq 1$

- E.g. $\Gamma = \{(1,2);(2,4)\}$



Deadline Monotonic Scheduling DM)

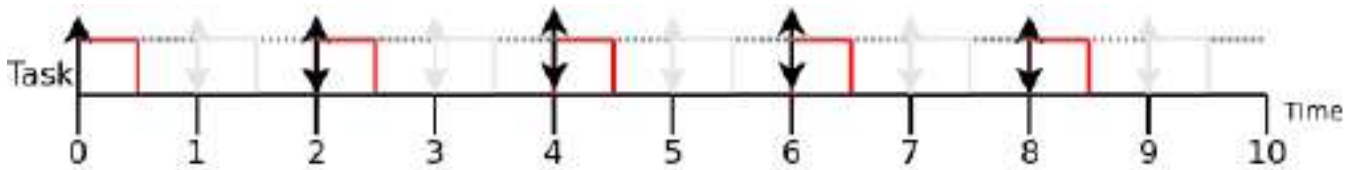
Schedulability tests for DM

- In some cases tasks may have large periods (low frequency) but require a short response time.
- In these cases we assign a deadline shorter than the period, and the scheduling criteria is the deadline.
- It is possible to use utilization-based tests.
  - The adaptation is simple, but the test is **very pessimistic** .

Utilization-based test

$$\sum_{i=1}^{n}\left(\frac{C_i}{D_i}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

E.g. $\{C = 0.5, T = 2, D = 1\}$ (Assumed utilization is doubled)



Response-time analysis

- For arbitrary fixed priorities, including RM, DM and others, the **Response Time Analysis** is an **exact** test (i.e., a necessary and sufficient condition) in the following conditions:
  - full preemption, synchronous release, independent tasks and $D \leq T$
- Worst-case response time (WCRT, Rwc, R,...) = maximum time interval between arrival and finish instants.
  - $Rwc_i = max_k(f_{i,k} - a_{i,k})$

Schedulability test based on the WCRT

$Rwc_i < D_i, \forall i \Leftrightarrow$ task set is schedulable

Response-time analysis

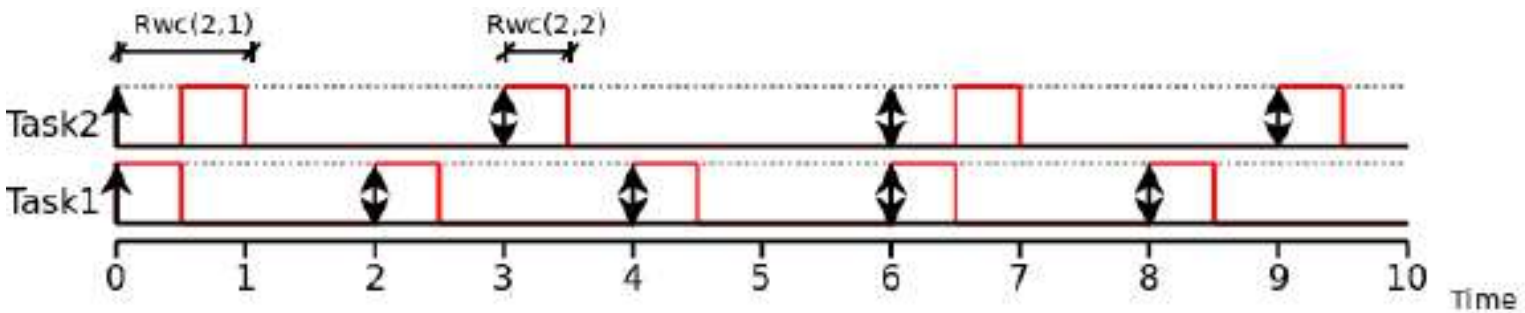
- The WCRT of a given task occurs when the task is activated at the same time as all other high-priority tasks (critical instant)
- I_i – interference caused by the execution of higher priority tasks

Computing Rwc_i

$$\forall i, Rwc_i = I_i + C_i$$

$$I_i = \sum_{k \in hp(i)} \lceil \frac{Rwc_i}{T_k} \rceil \cdot C_k$$

$\lceil \frac{Rwc_i}{T_k} \rceil$ represents the number of activations of hp task "k"



Response-time analysis

The equation is solved iteratively. Stop conditions are:

- A **deadline is violated**
  - $Rwc_i(m) > D_i$
- **Convergence** (two successive iterations yield the same result)
  - $Rwc_i(m+1) = Rwc_i(m)$

Algorithm

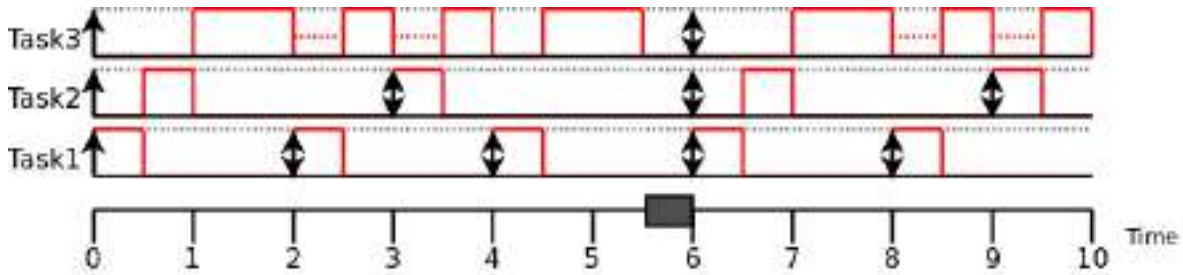
1. $Rwc_i(0) = \sum_{k \in hp(i)} (C_k) + C_i$
2. $Rwc_i(1) = \sum_{k \in hp(i)} \lceil \frac{Rwc_i(0)}{T_k} \rceil \cdot C_k + C_i$
3. ...
4. $Rwc_i(m) = \sum_{k \in hp(i)} \lceil \frac{Rwc_i(m-1)}{T_k} \rceil \cdot C_k + C_i$

Repeat Step 3 until convergence or deadline violation

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

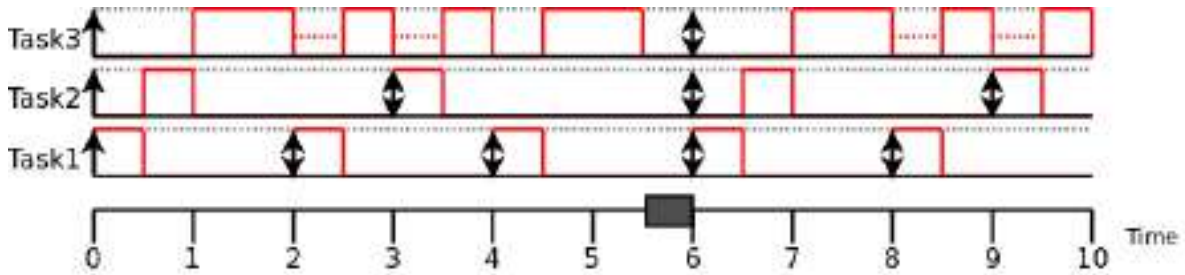


- $Rwc_1 = ?$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

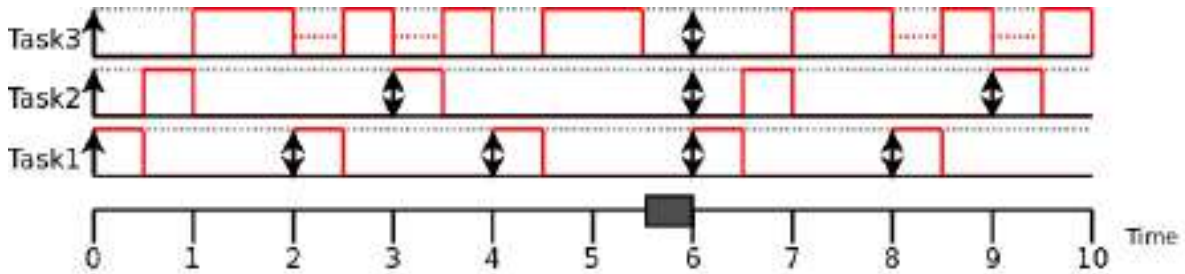


- $Rwc_1 = ?$
  - $Rwc_1(0) = C_1 = 0.5tu$
- $Rwc_2 = ?$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6



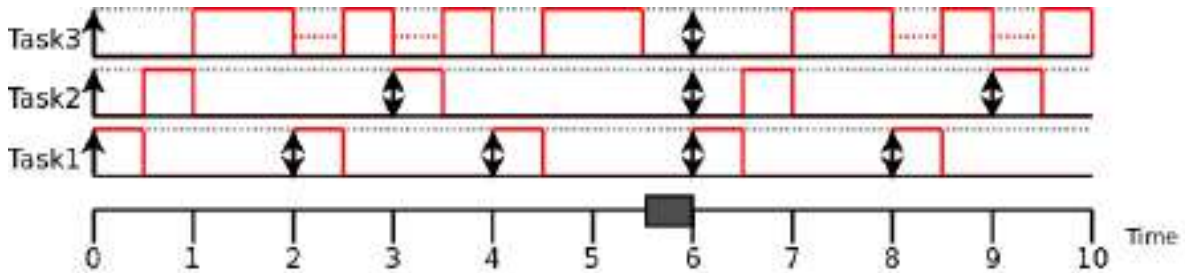
- $Rwc_1 = ?$
  - $Rwc_1(0) = C_1 = 0.5tu$
- $Rwc_2 = ?$
  - $Rwc_2(0) = C_1 + C_2 = 0.5 + 0.5 = 1tu$
  - $Rwc_2(1) = \sum_{k \in \{1\}} \lceil \frac{Rwc_i(0)}{T_k} \rceil \cdot C_k + C_2 = \lceil \frac{1}{2} \rceil \cdot 0.5 + 0.5 = 1tu$
  - **Converged** , thus $Rwc_2 = 1tu$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

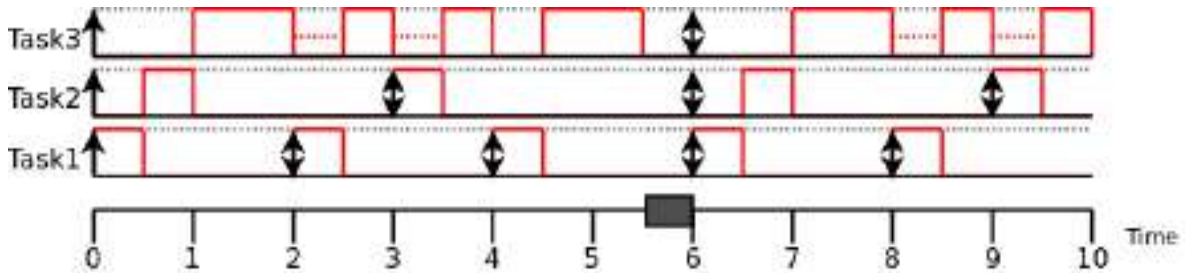
- Rwc_3=?



Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

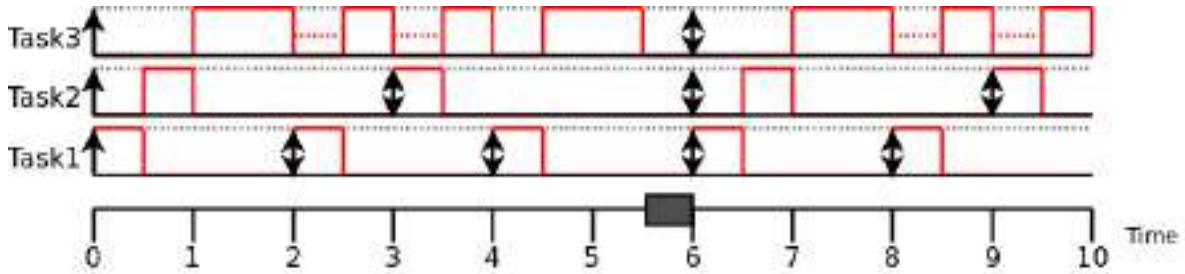


- Rwc_3=?
  - $Rwc_3(0)=C_1+C_2+C_3=4tu$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

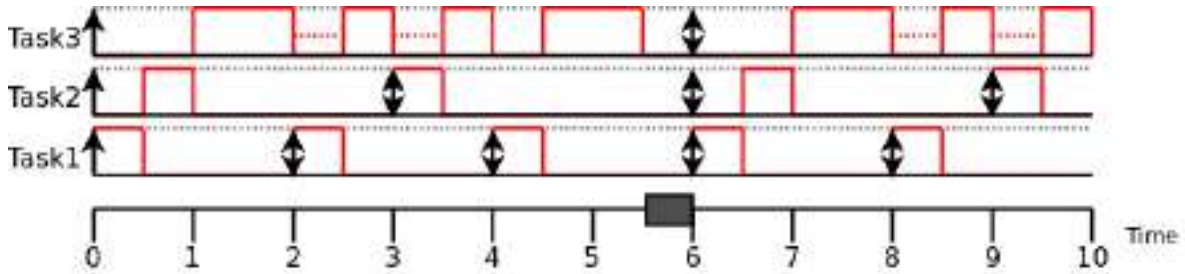


- Rwc_3=?
  - $Rwc_3(0)=C_1+C_2+C_3=4tu$
  - $Rwc_3(1)=\sum_{k\in\{1,2\}} \lceil \frac{Rwc_i(0)}{T_k} \rceil \cdot C_k+C_3=\lceil \frac{4}{2} \rceil \cdot 0.5+\lceil \frac{4}{3} \rceil \cdot 0.5+3=5tu$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

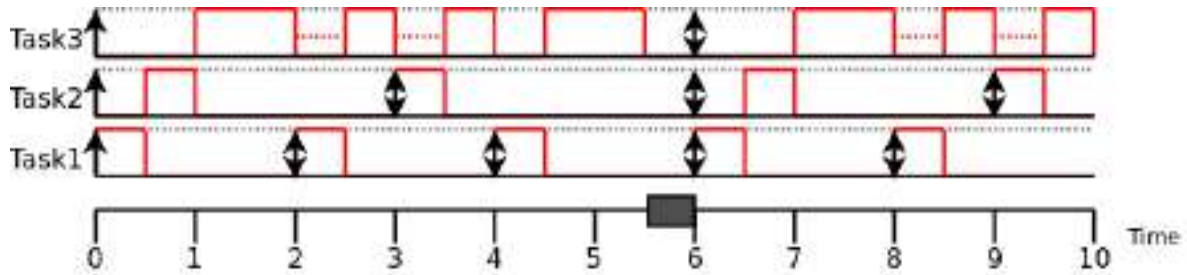


- Rwc_3=?
  - $Rwc_3(0)=C_1+C_2+C_3=4tu$
  - $Rwc_3(1)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(0)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{4}{2}\rceil\cdot 0.5+\lceil\frac{4}{3}\rceil\cdot 0.5+3=5tu$
  - $Rwc_3(2)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(1)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{5}{2}\rceil\cdot 0.5+\lceil\frac{5}{3}\rceil\cdot 0.5+3=5.5tu$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6



- Rwc_3=?
  - $Rwc_3(0)=C_1+C_2+C_3=4tu$
  - $Rwc_3(1)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(0)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{4}{2}\rceil\cdot 0.5+\lceil\frac{4}{3}\rceil\cdot 0.5+3=5tu$
  - $Rwc_3(2)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(1)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{5}{2}\rceil\cdot 0.5+\lceil\frac{5}{3}\rceil\cdot 0.5+3=5.5tu$
  - $Rwc_3(3)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(2)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{5.5}{2}\rceil\cdot 0.5+\lceil\frac{5.5}{3}\rceil\cdot 0.5+3=5.5tu$
  - **Converged** , thus $Rwc_3 = 5.5tu$

As $Rwc(i) \leq D_i \forall i$, the task set is schedulable!

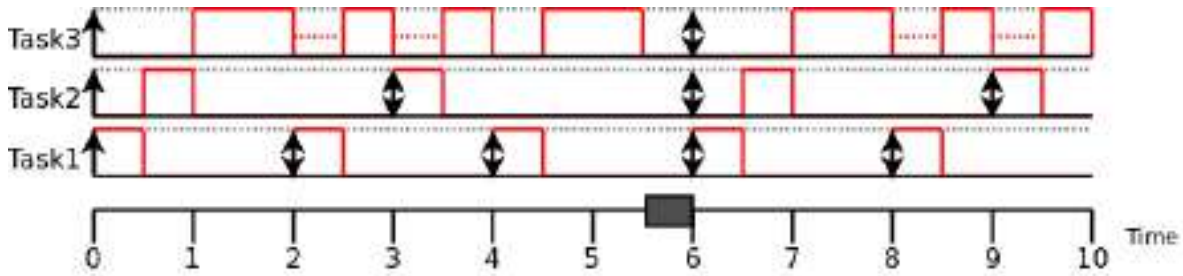
Restrictions to the schedulability tests previously presented

- The previous schedulability tests **must be modified** in the following cases:
  - Non-preemption
  - Tasks not independent
    - Share mutually exclusive resources
    - Have precedence constrains
  - It is also necessary to take into account the overhead of the kernel, because the scheduler, dispatcher and interrupts consume CPU time

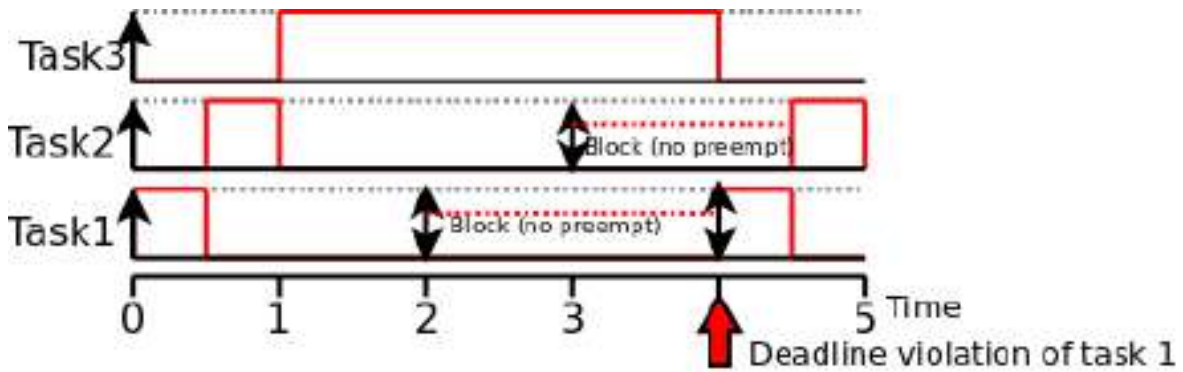
Impact of non-preemption

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6



With preemption



Without preemption

Summary

- On-line scheduling with fixed-priorities
- The Rate Monotonic scheduling policy – schedulability analysis based on utilization
- The Deadline Monotonic and arbitrary deadlines scheduling policies
- Response-time analysis

Dynamic Priority Scheduling

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

October 24, 2022

Last lecture

Fixed-priority online scheduling

- Rate-Monotonic scheduling
- Deadline-Monotonic and arbitrary priorities
- Analysis:
  - The CPU utilization bound
  - Worst-Case Response-Time analysis

Agenda for today

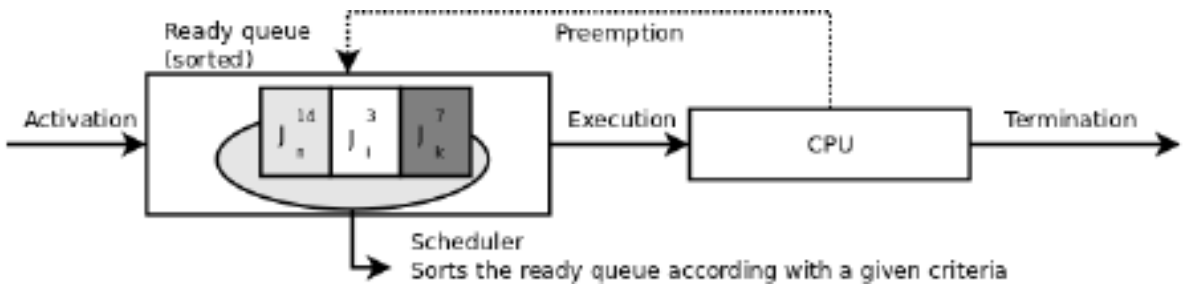
- Online scheduling with dynamic priorities
  - Earliest Deadline First scheduling
  - Analysis: CPU utilization bound and CPU Load Analysis
- Optimality and comparison with RM
  - Schedulability level, number of preemptions, jitter and response time
- Other dynamic priority criteria
  - Least Slack First, First Come First Served

On-line scheduling with dynamic priorities

Dynamic priorities

Ready Queue sorted by instantaneous priorities and dynamically re-sorted (if/when necessary)

- Scheduling is based in dynamic criteria, i.e. one that is known only at run-time
- The dynamic parameter used to sort the ready tasks can be understood as a dynamic priority
- The ready queue is (re)sorted according with decreasing priorities whenever there is a priority change. Executes first the task that has the greater instantaneous priority
  - Complexity O(n.log(n))



On-line scheduling with dynamic priorities

Pros

- Scales well (wrt SCS)
  - Changes made to the task set are immediately seen by the scheduler
- Accommodates easily sporadic tasks (wrt SCS)
- Allows higher utilization levels than Fixed Priorities

Cons

- More complex implementation (wrt SCS and FP)
- Higher overhead (wrt SCS and FP)
  - Re-sorting of ready queue; depends on the algorithm
- "Unpredictability" on overloads (wrt FP)
  - It is not possible to know a priory which tasks will fail deadlines

On-line scheduling with dynamic priorities

Priority allocation approaches

- Inversely proportional to the time to the deadline
  - EDF – Earliest Deadline First
    - Optimal among all dynamic priority criteria
- Inversely proportional to the laxity or slack
  - LSF (LST or LLF) – Least Slack First
    - Optimal among all dynamic priority criteria
- Inversely proportional to the service waiting time
  - FCFS –First Come First Served
    - Not optimal with respect to meet deadlines; extremely poor real-time performance
- etc.

On-line scheduling with dynamic priorities

Schedulability tests

- Since the schedule is built online it is important to determine a priori if a given task set meets or not its temporal requirements
- There are three types of schedulability tests:
  - Based on CPU utilization
  - Based on CPU load (processor demand)
  - Based on response time

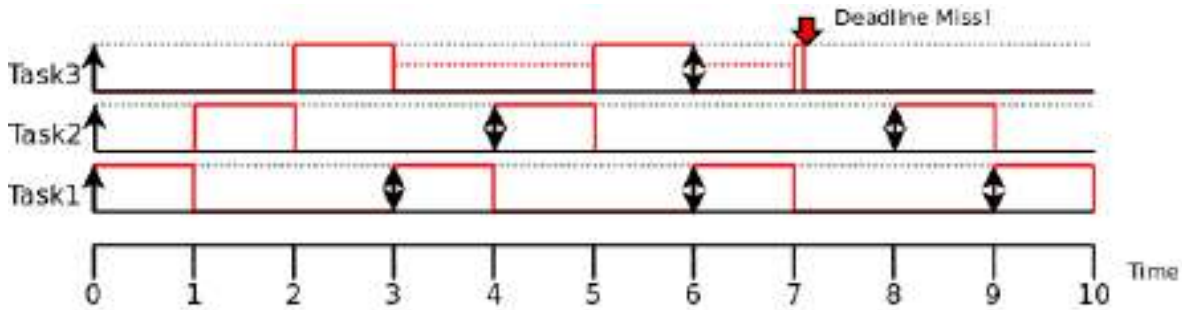
EDF Scheduling

EDF tests based on CPU utilization (n independent tasks, with full preemption)

- $D = T$
  - $U(n) = \sum_{i=1}^{n} (\frac{C_i}{T_i}) \leq 1 \Leftrightarrow$ Task set is schedulable
  - Allows using 100% of CPU with timeliness guarantees
- $D < T$
  - $U(n) = \sum_{i=1}^{n} (\frac{C_i}{D_i}) \leq 1 \Rightarrow$ Task set is schedulable
  - Sufficient condition, only.
  - Pessimistic test (as for RM, inflates the utilization)

RM Scheduling - example

τ	C	T
1	1	3
2	1	4
3	2.1	6

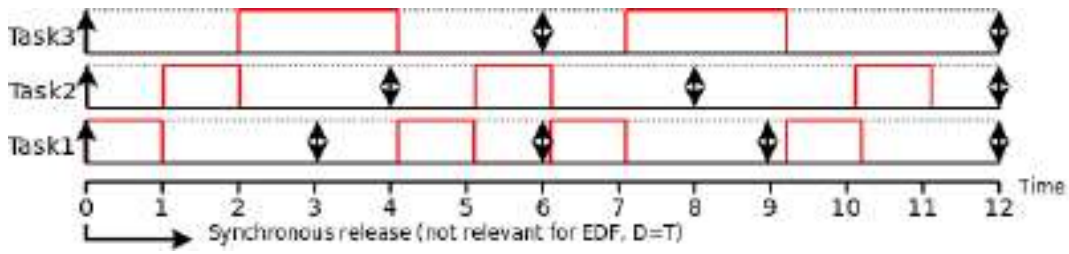


- $U = \frac{1}{3} + \frac{1}{4} + \frac{2.1}{6} = 0.93 > 0.78 \Rightarrow$ 1 activation per period NOT guaranteed.
- In fact τ_3 fails a deadline!

EDF Scheduling – same example

Same example as before!

τ	C	T
1	1	3
2	1	4
3	2.1	6

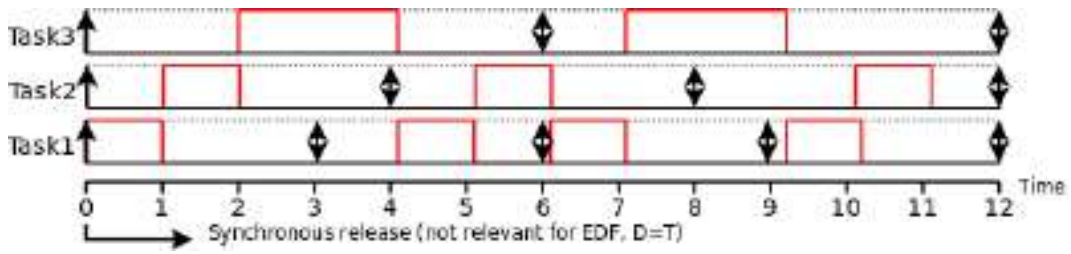


- $U = \frac{1}{3} + \frac{1}{4} + \frac{2.1}{6} = 0.93 \leq 1 \Leftrightarrow$ one activation per period IS guaranteed.

EDF Scheduling – same example

Same example as before!

τ	C	T
1	1	3
2	1	4
3	2.1	6



Note:

- No deadline misses
- Less preemptions
- Higher jitter on quicker tasks
- The worst-case response time does not coincide necessarily with the synchronous release

EDF Scheduling

Notion of **fairness**

- Be fair on the attribution of resources (e.g. CPU)
- EDF is intrinsically fairer than RM, in the sense that tasks see its relative deadline increased as the absolute deadline approaches, independently of its period or any other static parameter.
- Consequences:
  - Deadlines are easier to met
  - As the deadline approaches preemptions suffered by a given task are reduced
  - The slack of tasks that are quick but have large deadlines can be used by other task (higher jitter on tasks with shorter periods)

CPU Load Analysis

- For $D < T$, the biggest period during which the CPU is permanently used (i.e. without interruption, idle time) corresponds to the scenario in which all tasks are activated synchronously. This period is called **synchronous busy period** and has duration L
- L can be computed by the following iterative method, which returns the first instant since the synchronous activation in which the CPU completes all the submitted jobs

Computation of L

$$L(0) = \sum_i (C_i)$$

$$L(m+1) = \sum_i (\lceil \frac{L(m)}{T_i} \rceil \cdot C_i)$$

Stop condition: $L(m+1) = L(m)$

CPU Load Analysis

- Knowing L, we have to guarantee the **Load Condition** , i.e.

$$\boxed{h(t) \leq t, \forall t \in [0, L[\Rightarrow \textit{ Task set is schedulable}}$$

- The Load Condition refers to all the work that must be **completed by t** .
- How to compute h(t)?

- Example:
  - Task set $\Gamma = \{(2,4),(2,8),(3,16)\}$
- Draw the Gantt chart and mark the points where tasks must complete
- Write a suitable equation

CPU Load Analysis

Knowing L we have to guarantee the load condition, i.e.

$$h(t) \leq t, \forall t \in [0, L[$$

And $h(t)$ can be computed as follows:

$$h(t) = \sum_{i=1..N} max(0, 1 + \lfloor \frac{t - D_i}{T_i} \rfloor) \cdot C_i$$

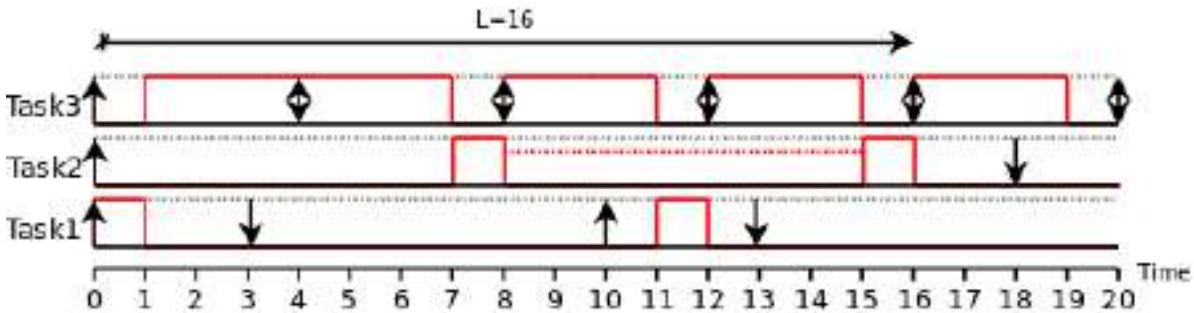
- The computation of h(t) for all values of $t \in [0, L[$ is unfeasible.
  - However it is enough computing the load condition for the instants in which the load function varies, i.e.

$$S = \bigcup_i (S_i), S_i = \{m \cdot T_i + D_i\}, m = 0, 1, ...$$

- Note: there are other, possibly shorter, values for L

EDF Scheduling

τ	C	D	T
1	1	3	10
2	2	18	20
3	3	4	4



- $\sum_{i=1..N} \frac{C_i}{D_i} = \frac{1}{3} + \frac{2}{18} + \frac{3}{4} = 1.194 > 1 \Rightarrow$ Schedulability not guaranteed
- But the CPU load analysis indicates that the task set is schedulable!

LSF Scheduling

Least Slack First : Executes first the task with smaller slack $\left(L_i(t)=d_i-c_i(t)\right)$

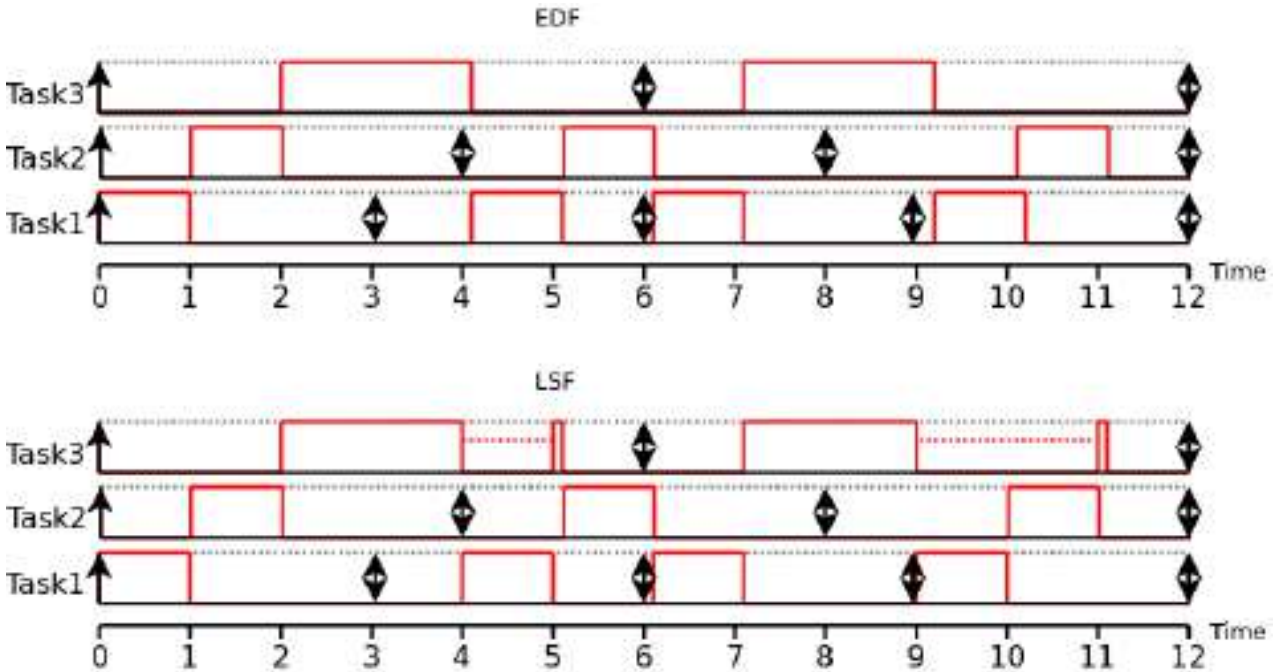
LSF vs EDF short comparison

- LS is optimal (as EDF)
- As the slack $\uparrow$ the priority $\downarrow$
- Priority of ready tasks increases as time goes by
  - Rescheduling only on instants where there are activations or terminations. **Why?**
- Priority of the task in the running state does not change
  - In EDF the priorities of all tasks (ready and executing) increase equally as time goes by
- Causes an higher number of preemptions than EDF (and thus higher overhead)
- No significant advantages with respect to EDF!

LSF scheduling example

LSF vs EDF

τ	C	T
1	1	3
2	1	4
3	2.1	6

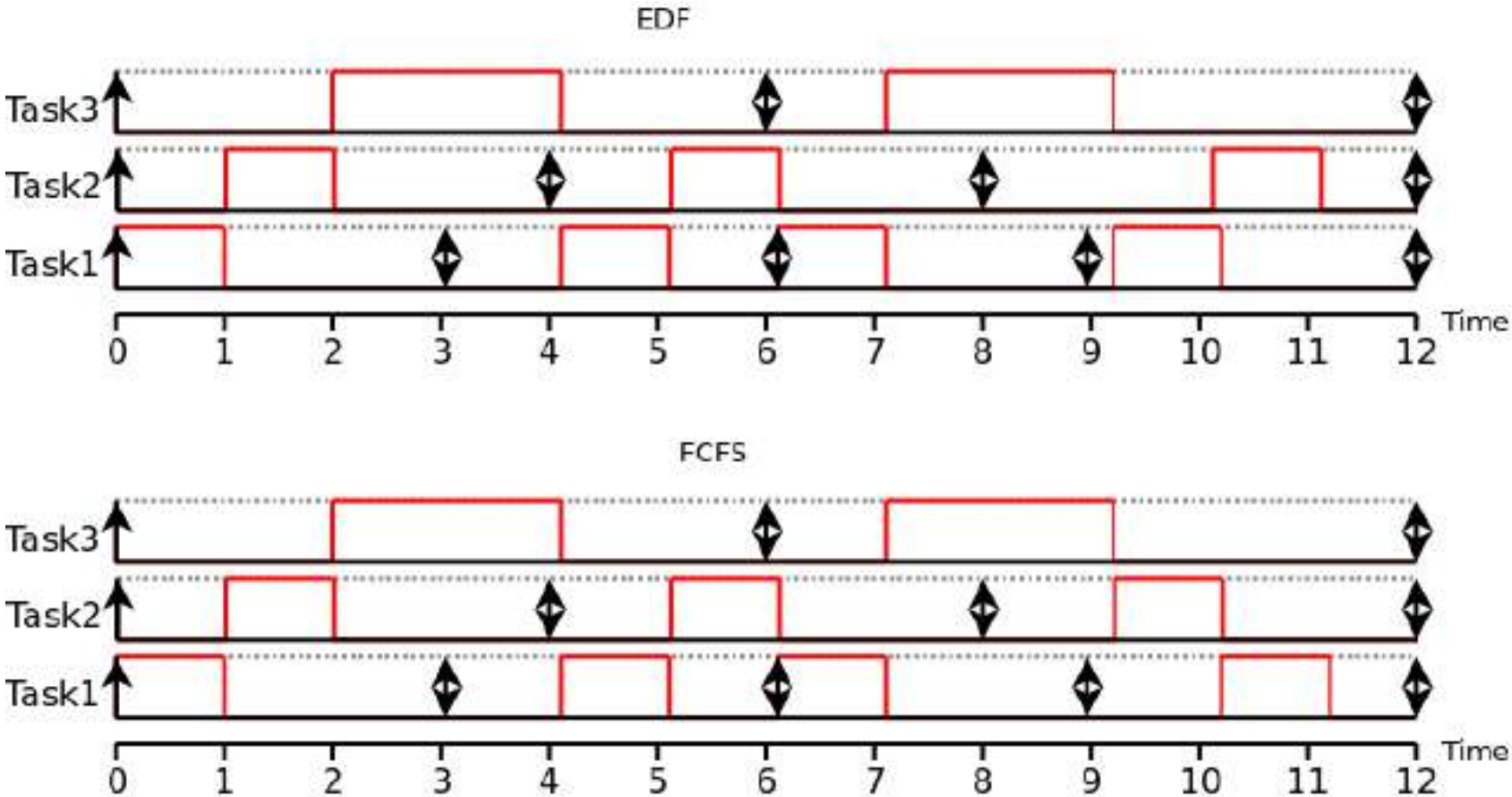


FCFS Scheduling

Execute tasks as they arrive. Priority depends on the arrival order.
A brief comparison between FCFS and EDF/LLF

- Non optimal
  - May lead to deadline misses even with very low CPU utilization rates
- Job age $\uparrow$ Priority $\uparrow$
- Priority of the ready and running tasks increases as time goes by (an in EDF)
- New jobs always get the lower priority
- There are no preeemptions (smaller overhead and facilitates the implementation)
- Very poor temporal behavior!

FCFS – same example



- When the "age" is the same the tie break criteria is decisive!

Summary

- On-line scheduling with dynamic priorities
- The EDF - Earliest Deadline First criteria: CPU utilization bound and CPU Load Analysis
- Optimality of EDF and comparison with RM:
  - Schedulability level, number of preemptions, jitter and response time
- Other dynamic priority criteria:
  - LLF (LST) - Least Laxity (Slack) First
  - FCFS - First Come First Served

Exclusive Access to Shared Resources

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

October 31, 2022

Last lecture

- Online scheduling with dynamic priorities
  - Earliest Deadline First scheduling
  - Analysis: CPU utilization bound and CPU Load
- Optimality and comparison with RM and SCS
  - Schedulability level, number of preemptions, jitter and response time
- Other dynamic priority criteria
  - Least Slack First, First Come First Served

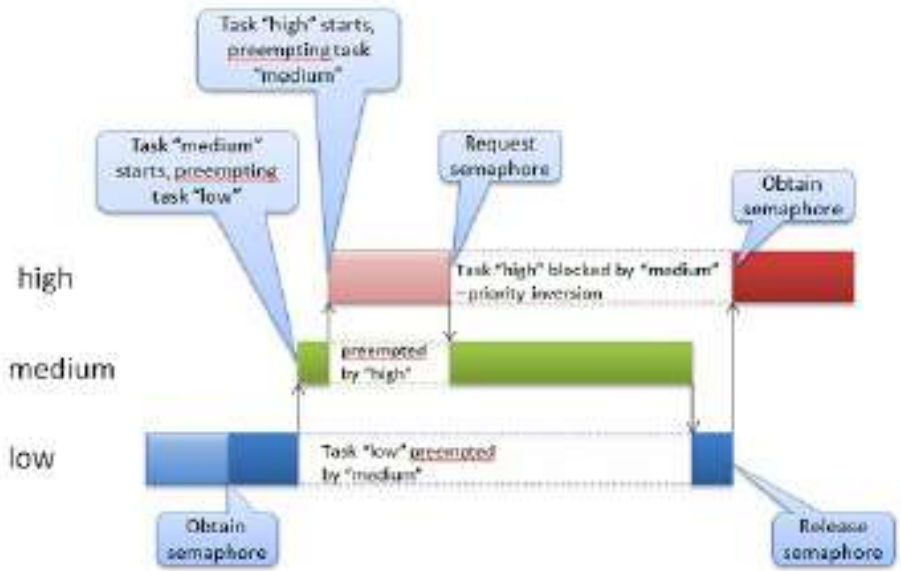
Agenda for today

- Priority inversion as a consequence of blocking
- Basic techniques to enforce exclusive access to shared resources:
  - Priority Inheritance Protocol – PIP
  - Priority Ceiling Protocol – PCP
  - Stack Resource Protocol- SRP

The priority inversion problem in not "academic"!

What really happened to the software on the Mars Pathfinder spacecraft?

- *(July 4th 1997)... the Mars Pathfinder landed to a media fanfare and began to transmit data back to Earth. Days later and the flow of information and images was interrupted by a series of total systems resets. ...*



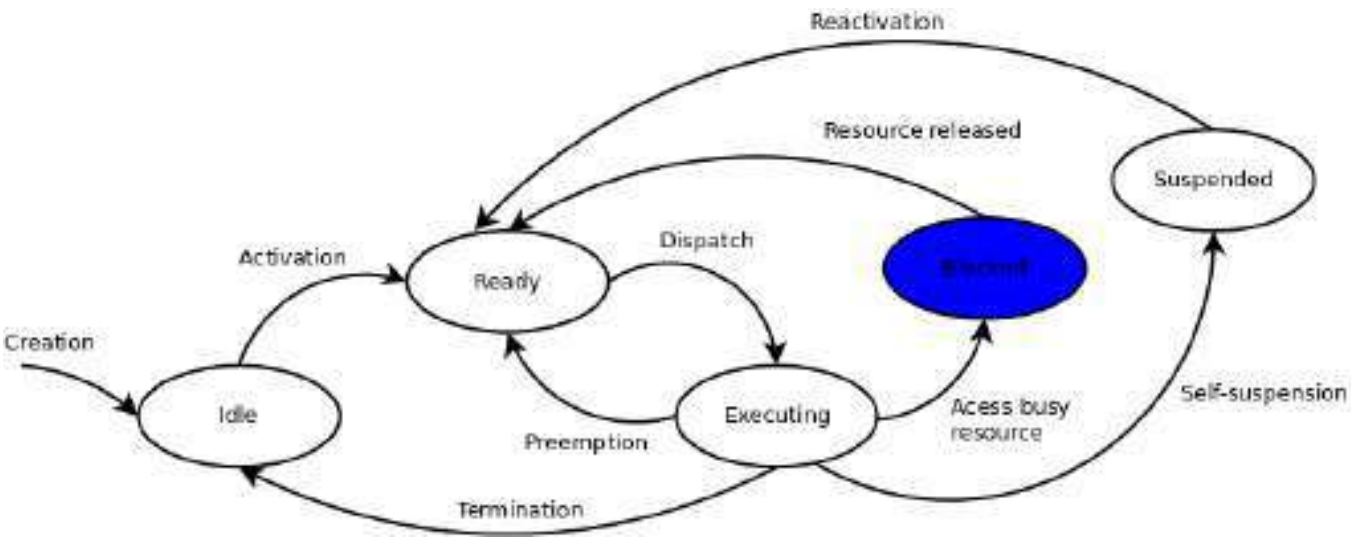
Source:

https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

Shared resources with exclusive access

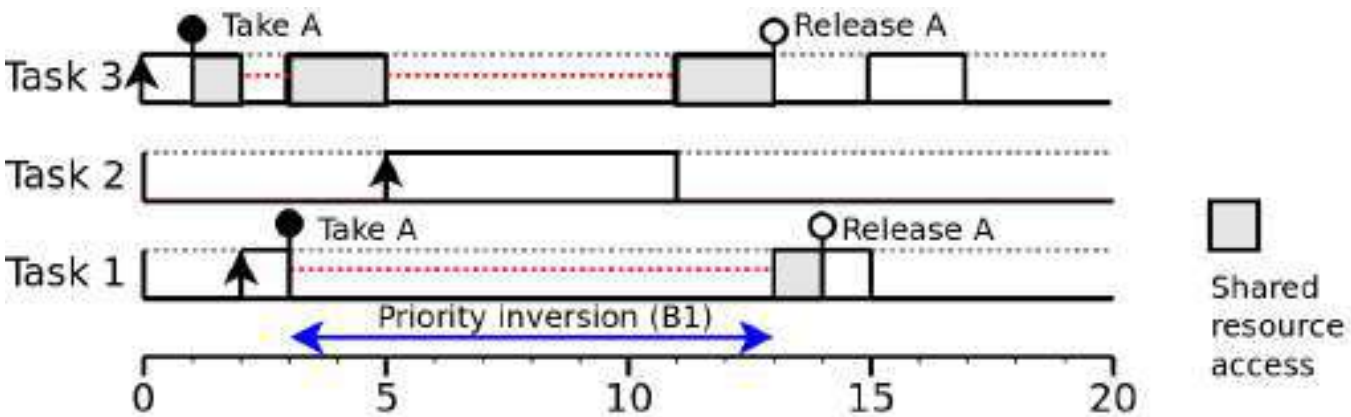
Tasks: the Blocked state

- When a running task tries to access a shared resource (e.g. a buffer, a communication port) that is **already taken** (i.e. in use) by another task, it gets **blocked** . When the resource becomes free, the blocked task becomes again ready for execution. To handle this scenario the state diagram is updated as follows:



The priority inversion phenomenon

- On a real-time system with preemption and independent tasks, the highest priority ready task is always the one in execution
- However, when tasks share resources with exclusive access, the case is different. An higher priority task may be **blocked** by another (**lower priority**) task, whenever this latter one owns a resource needed by the first one. In such scenario it is said that the higher priority task is blocked.
- When the blocking task (and eventually other tasks with intermediate priority) execute, there is a **priority inversion** .



The priority inversion phenomenon

- The priority inversion is an unavoidable phenomenon in the presence of shared resources with exclusive access.
- However, in real-time systems, it is of utmost importance to **bound and quantify** its worst-case impact, to allow reasoning about the schedulability of the task set.
- Therefore, the techniques used to guarantee the exclusive access to the resources (synchronization primitives) must **restrict the duration of the priority inversion and be analyzable** , i.e., allow the quantification of the maximum blocking time that each task may experience in any shared resource.

Techniques to allow exclusive access

Possible synchronization approaches

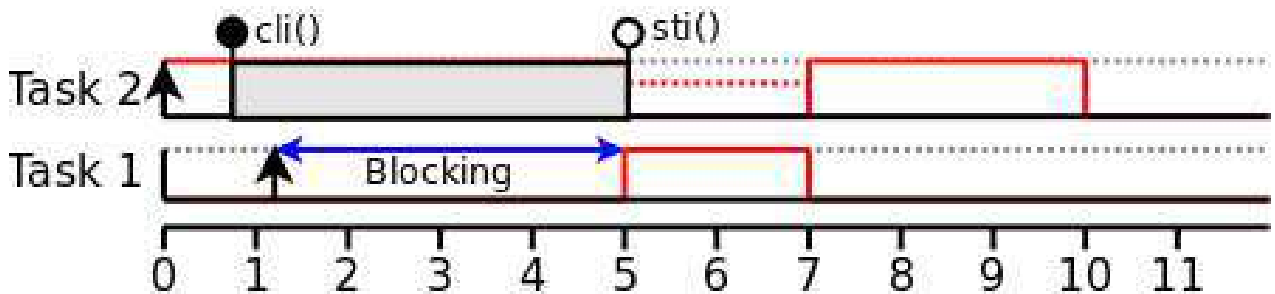
- **Disable Interrupts**
  - *disable*()/*enable*() or *cli*()/*sti*()
- **Inhibit preemption**
  - *no_preemp*()/*preempt*()
- **Locks or atomic flags**
  - *lock*()/*unlock*()
- **Use of semaphores**
  - Counter + task list
  - *P*()/*V*(), *wait*()/*signal*(), *take*()/*give*(), ...

Let's take a look at them ...

Techniques to allow exclusive access

Interrupt inhibit

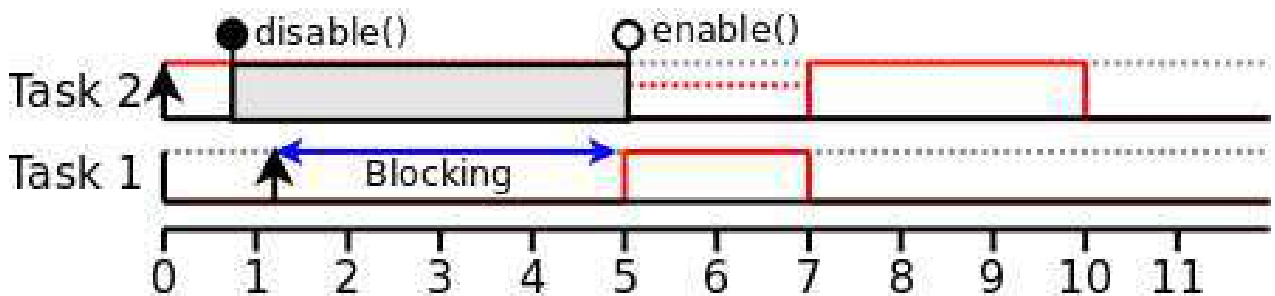
- All other system activities are blocked, not just other tasks, but also interrupt service routines, **including the system tick handler** .
- Very easy to implement but should only be used with very short critical regions (e.g. access to a few HW registers, read-modify-write of a variable)
- Each task can only be blocked once and for the maximum duration of the critical region of lower priority tasks (or smaller relative deadline for EDF), even if they don't use any shared resource!!



Techniques to allow exclusive access

Preemption inhibiting

- All other tasks are blocked. However, contrarily to disabling the interrupts, in this case the **interrupt service routines** , including the system tick, are **not blocked** !
- Very easy to implement but not efficient, as it causes unnecessary blocking.
- Each task can only be blocked once and for the maximum duration of the critical region of lower priority tasks (or smaller relative deadline for EDF), even if these don't use any shared resource!!



Techniques to allow exclusive access

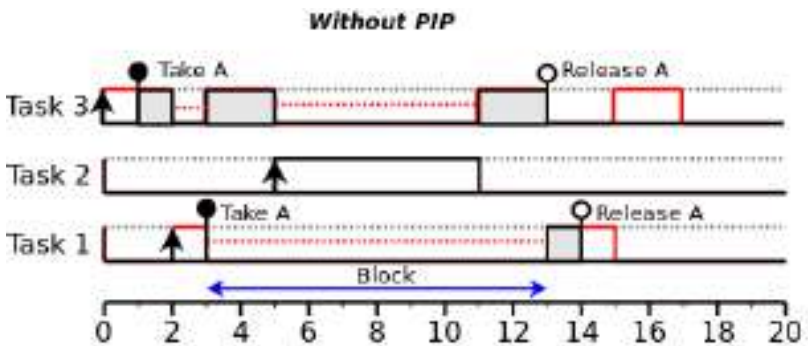
Semaphores

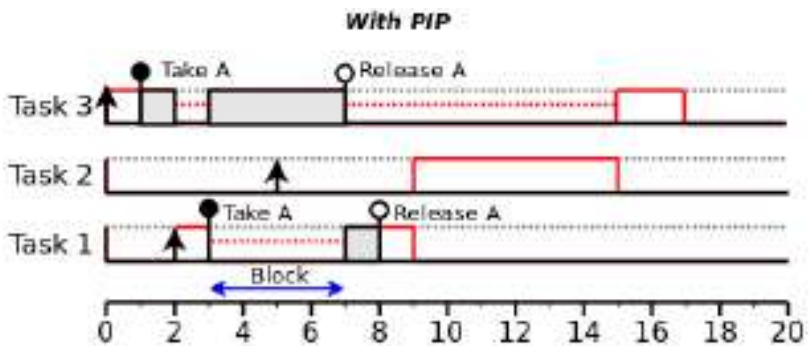
- More **complex and costly** to implement
- In general much **more efficient** resource management
- However, the blocking duration depends on the specific protocol used to manage the semaphores
- These protocols should prevent:
  - Indeterminate blocking
  - Chain blocking
  - Deadlocks

Let's see a few protocols commonly used in RTS

PIP – Priority Inheritance Protocol

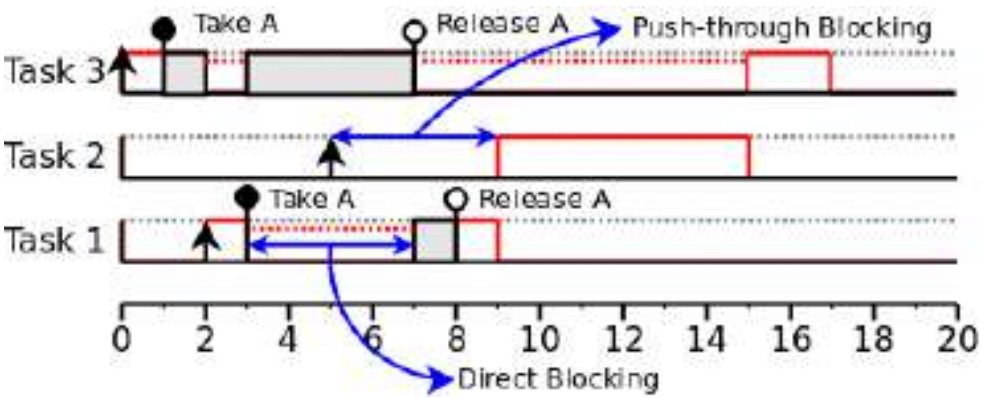
- The blocking task (lower priority) **temporarily inherits** the priority of the blocked task (the one with higher priority).
- Limits the blocking duration, **preventing the execution of intermediate priority tasks** while the blocking tasks owns the critical region. The priority of the blocking task returns to its nominal value when leaving the critical region.





PIP – Priority Inheritance Protocol

- To bound the blocking time (B) it is important to note that a task can be blocked by any lower priority task that:
  - Shares a resource with it - **Direct blocking** , or
  - Can block a task with higher priority - **Push-through** or **Indirect blocking**
- Note also that in the absence of chained accesses:
  - Each task can block any other task just once
  - Each task can block only once in each resource



τ	S_1	S_2	S_3	B_i
τ_1	1	2	0	?
τ_2	0	9	3	?
τ_3	8	7	0	?
τ_4	6	5	4	?

PIP – Priority Inheritance Protocol

Worst-case blocking times

τ	S_1	S_2	S_3
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

$\Rightarrow$

τ	C_i	T_i	B_i
τ_1	5	30	17
τ_2	15	60	13
τ_3	20	80	6
τ_4	20	100	0

Schedulability analysis with PiP

Utilization test, for Rate Monotonic

$$\forall_{1 \le i \le N} \sum_{h:P_h > P_i} \left(\frac{C_h}{T_h}\right) + \frac{C_i + B_i}{T_i} \le i \cdot (2^{\frac{1}{i}} - 1)$$

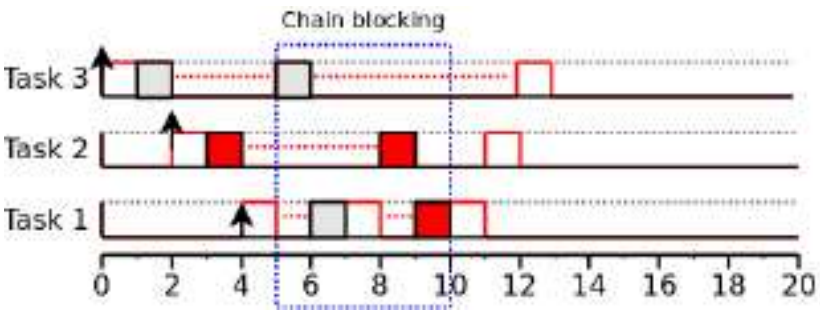
Response Time Analysis (Fixed Priority)

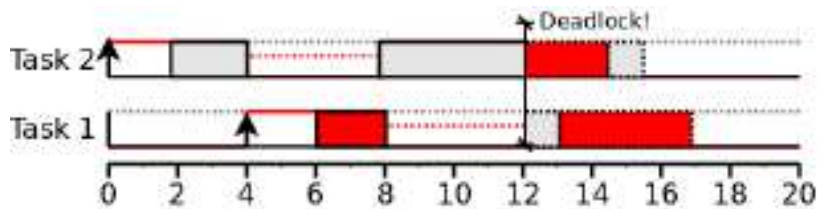
$$R_i(0) = C_i + B_i$$
$$R_i(m) = C_i + B_i + \sum_{h:P_h > Pi} \left\lceil \frac{R_i(m-1)}{T_h} \right\rceil C_h$$

PIP – Priority Inheritance Protocol

PiP evaluation:

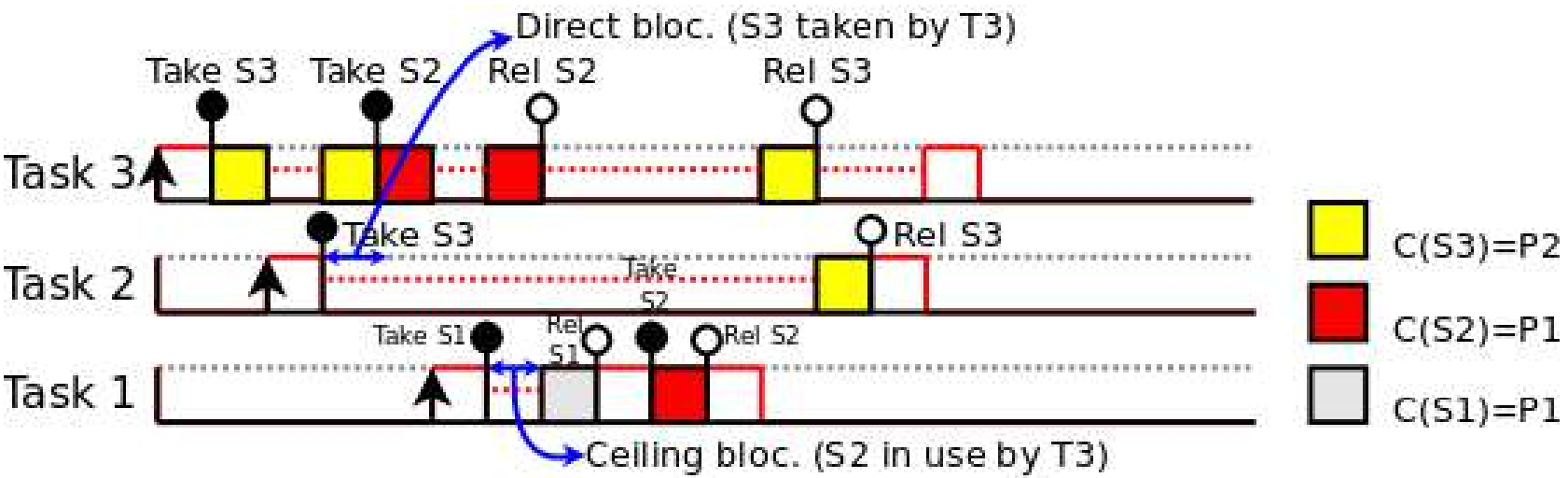
+ Relatively easy to implement
  - Only one additional field on the TCB, the inherited priority
+ Transparent to the programmer
  - Each task only uses local information
- Suffers from chain blocking and does not prevent deadlocks





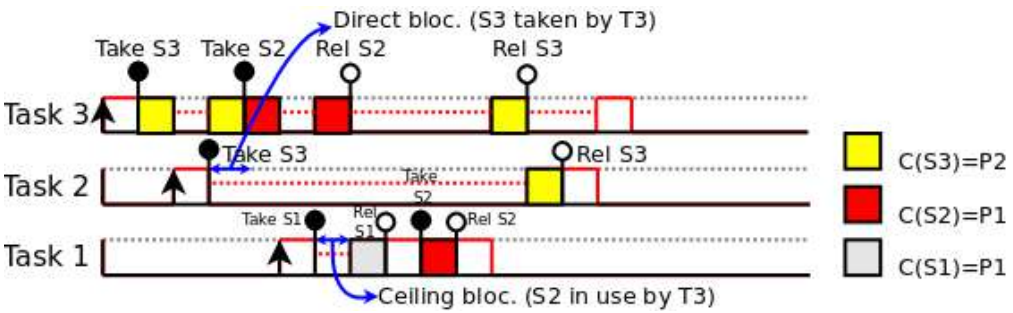
PCP – Priority Ceiling Protocol

- Extension of PIP with one additional rule about access to free semaphores, inserted to guarantee that all required semaphores are free.
- For each semaphore it is defined a **priority ceiling** , which equals the priority of the maximum priority task that uses it.
- A task can only **take a semaphore** if this one is free and if its **priority** is **greater than the ceilings** of all semaphores currently taken.



PCP – Priority Ceiling Protocol

- The PCP protocol only allows the access to the first semaphore when **all** other semaphores that a task needs are **free**
- To bound the blocking time (B) note that a task can be **blocked only once by lower priority tasks** . Only lower priority tasks that use semaphores which have a ceiling at least equal to the higher priority task can cause blocking.
- Note also that each task can only be blocked once



τ	S_1	S_2	S_3	B_i
τ_1	1	2	0	?
τ_2	0	9	3	?
τ_3	8	7	0	?
τ_4	6	5	4	?

PCP – Priority Ceiling Protocol

Worst-case blocking times

τ	S_1	S_2	S_3
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

$\Rightarrow$

τ	C_i	T_i	B_i
τ_1	5	30	9
τ_2	15	60	8
τ_3	20	80	6
τ_4	20	100	0

Schedulability analysis with PCP

- Same equations as for PIP, **only B_i computes differently**

Utilization test, for Rate Monotonic

$$\forall_{1 \le i \le N} \sum_{h:P_h > P_i} \left(\frac{C_h}{T_h}\right) + \frac{C_i + B_i}{T_i} \le i \cdot \left(2^{\frac{1}{i}} - 1\right)$$

Response Time Analysis (Fixed Priority)

$$R_i(0) = C_i + B_i$$
$$R_i(m) = C_i + B_i + \sum_{h:P_h > Pi} \left\lceil \frac{R_i(m-1)}{T_h} \right\rceil C_h$$

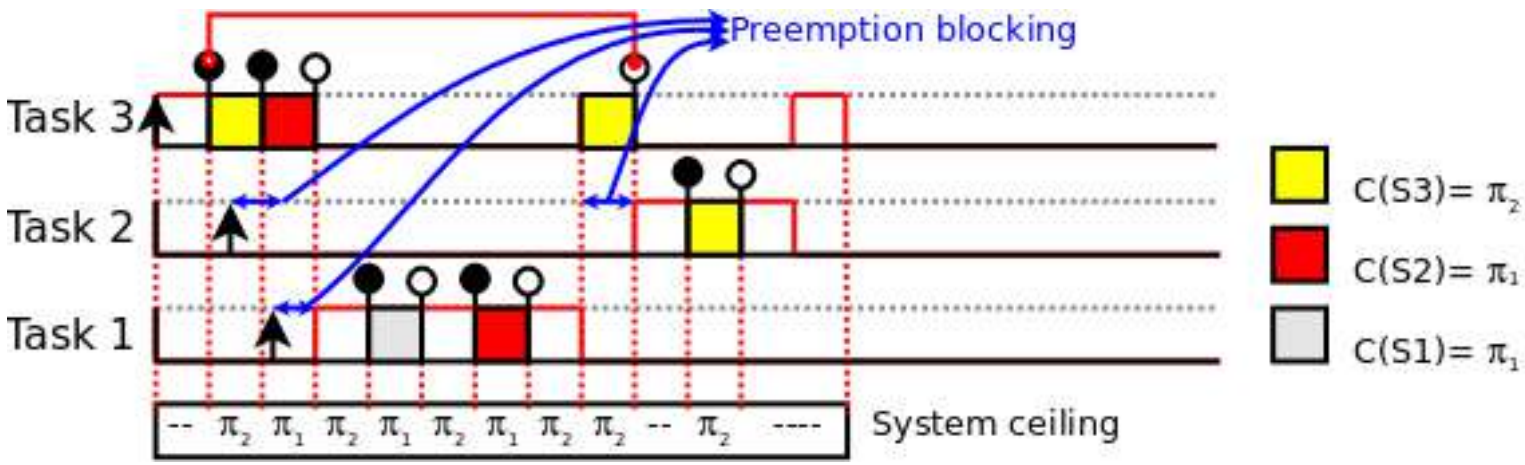
PCP – Priority Ceiling Protocol

PCP Evaluation

+ **Smaller blocking** than PIP, **free of chain blocking and deadlocks**
- Much **harder to implement** than PiP. On the TCB it requires one additional field for the inherited priority and another one for the semaphore where the task is blocked. To facilitate the transitivity of the inheritance it also requires a structure to the semaphores, their respective ceilings and the identification of the tasks that are using them
- Moreover, it is **not transparent to the programmer** as the semaphore ceilings are not local to the tasks

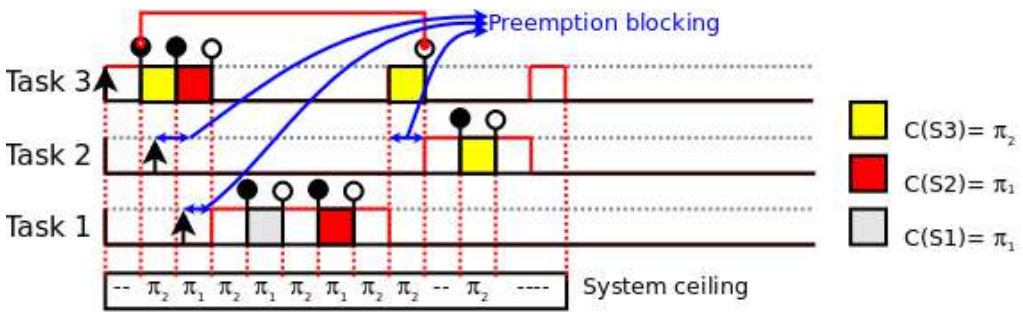
SRP – Stack Resource Policy

- Similar to PCP, but with one rule about the **beginning of execution** , to guarantee that all required semaphores are free
- Uses also the concept of priority ceiling
- Defines the preemption level (π) as the capacity of a task to cause preemption on another one (static parameter).
- A task may only start executing when its own **preemption level is higher than** the one of the executing task and also higher than the ceilings of all the semaphores in use (**system ceiling**).



SRP – Stack Resource Policy

- The SRP protocol only allows that a task **starts executing** when all **resources that it needs are free**
- The upper bound of the blocking time (B) is equal to the one of the PCP protocol, but it occurs in a different time - at the beginning of the execution instead of at the shared resource access.
- Each task can block only once by any task with a lower preemption level that uses a semaphore whose ceiling is at least equal to its preemption level.



τ	S_1	S_2	S_3	B_i
τ_1	1	2	0	?
τ_2	0	9	3	?
τ_3	8	7	0	?
τ_4	6	5	4	?

SRP – Stack Resource Policy

Worst-case blocking time

τ	S_1	S_2	S_3
τ_1	1	2	0
τ_2	0	9	3
τ_3	8	7	0
τ_4	6	5	4

$\Rightarrow$

τ	C_i	T_i	B_i
τ_1	5	30	9
τ_2	15	60	8
τ_3	20	80	6
τ_4	20	100	0

Schedulability analysis with SRP

- Same equations as for PIP and PCP. B_i **computes as for PCP**

Utilization test, for Rate Monotonic

$$\forall_{1 \le i \le N} \sum_{h:P_h > P_i} \left(\frac{C_h}{T_h}\right) + \frac{C_i + B_i}{T_i} \le i \cdot \left(2^{\frac{1}{i}} - 1\right)$$

Response Time Analysis (Fixed Priority)

$$R_i(0) = C_i + B_i$$
$$R_i(m) = C_i + B_i + \sum_{h:P_h > Pi} \left\lceil \frac{R_i(m-1)}{T_h} \right\rceil C_h$$

SRP – Stack Resource Policy

SRP Evaluation

+ Smaller blocking than PiP, free of chain blocking and deadlocks
+ Smaller number of preemptions than PCP, intrinsic compatibility with fixed an dynamic priorities and to resources with multiple units (i.e., that allow more than one concurrent access, e.g. buffer arrays)
- The hardest to implement (preemption test much more complex, requires computing the system ceiling, etc.)
- Not transparent to the programmer (semaphore ceilings, etc.)

Notes (1/2)

These policies are more complex to understand and implement than it seems! E.g.:

"I observed in the kernel code (to my disgust), the Linux PIP implementation is a nightmare: extremely heavy weight, involving maintenance of a full wait-for graph, and requiring updates for a range of events, including priority changes and interruptions of wait operations."

"An additional reason is that the original specification of PIP [24], despite being informally "proved" correct, is actually flawed.

Quoted from "Zhang, X., Urban, C. & Wu, C. Priority Inheritance Protocol Proved Correct. J Autom Reasoning 64, 73-95 (2020)."

Notes (2/2)

Equations in Buttazzo's book

- In the equations for computing the blocking times, Buttazzo's book has a "-1" factor affecting the critical region duration
- This is due to the fact that he considers a clock resolution of 1 t.u.
- This means that if a $t = t_k$ a resource is busy then it was taken at least at $t = t_k - 1$, so the task holding it already executed at least one time unit
- in these slides I'm considering that the clock resolution is much smaller than the duration of tasks and critical sections, so that factor is neglected.
- Other than that the equations are similar

Summary

- Access to shared resources: blocking
- The priority inversion problem: need to bound and analyze
- Basic techniques to allow exclusive access to shared resources
  - Disable interrupts, disable preemption
- Advanced techniques to allow exclusive access to shared resources
  - The Priority Inheritance Protocol – PIP (Linux, Xenomai, VXWorks...)
  - The Priority Ceiling Protocol – PCP (Mostly academic)
  - The Stack Resource Protocol - SRP (Mostly academic)

Aperiodic Servers

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

November 4, 2022

Last lecture

- Priority inversion as a consequence of blocking
- Basic techniques to enforce exclusive access to shared resources:
  - Priority Inheritance Protocol – PIP
  - Priority Ceiling Protocol – PCP
  - Stack Resource Protocol- SRP

Agenda for today

Aperiodic task scheduling

- Joint execution of periodic and sporadic tasks
- Use of aperiodic task servers
  - Fixed-priority aperiodic task servers
  - Dynamic-priority aperiodic task servers

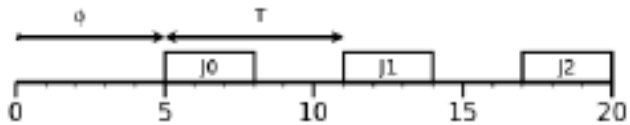
Joint scheduling of periodic and aperiodic tasks

Classification of tasks found in practical systems

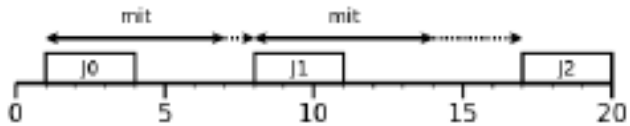
- **Periodic tasks**
  - Suitable e.g. to applications where it is required sampling regularly a given physical entity (e.g. acquire an image, a temperature, pressure, torque, speed), actuate regularly on the system, etc.
- **Aperiodic and Sporadic tasks**
  - Suitable to scenarios where the event activation instants cannot be forecast, e.g. alarms, human-machine interfaces, external asynchronous interrupts.
- **Hybrid systems**
  - Applications which contain both types of tasks.
  - Many (most?) real systems contain naturally both periodic and aperiodic events/tasks

Joint scheduling of periodic and aperiodic tasks

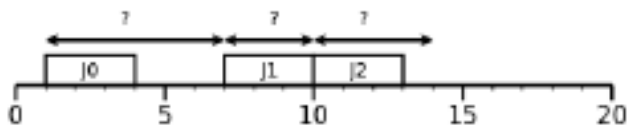
- Periodic tasks
  - k_{th} task instance activated at $a_k = k \cdot T_k + \Phi_k$
    Worst-case is well defined



- Sporadic tasks
  - In worst-case it behaves **as a periodic task** , with period = mit



- Aperiodic tasks

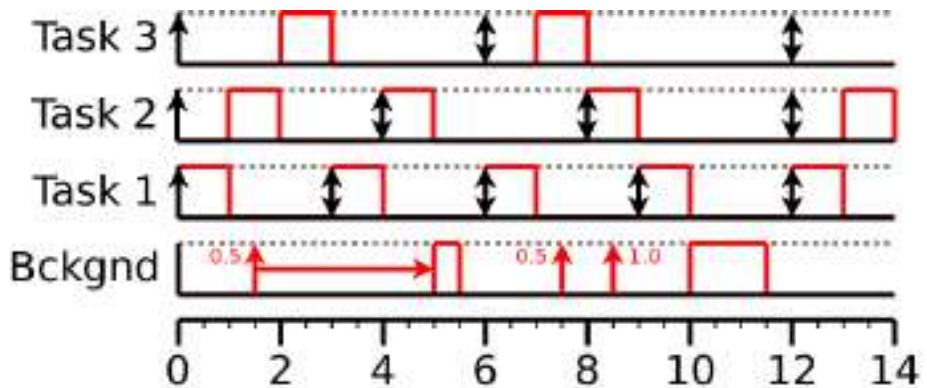


  - Only characterizable via probabilistic methods
  - How to **bound the interference** on periodic tasks?
  - How to guarantee an acceptable/best possible **quality of service** (QoS)?

Background execution

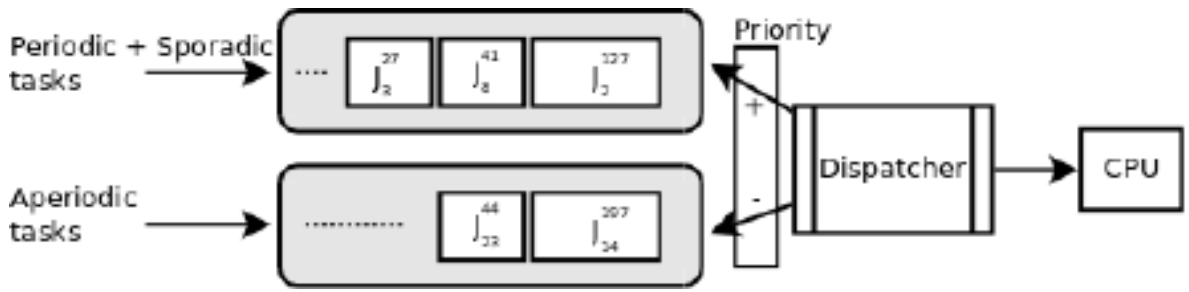
- A simple way of combining both task types is giving higher priority to the periodic/sporadic tasks than to the aperiodic ones.
- Thus the aperiodic tasks only execute when **there are no ready periodic/sporadic tasks** .
- Aperiodic tasks are executed in background with respect to the periodic/sporadic ones – background execution.

τ_i	C_i	T_i
1	1	3
2	1	4
3	1	6



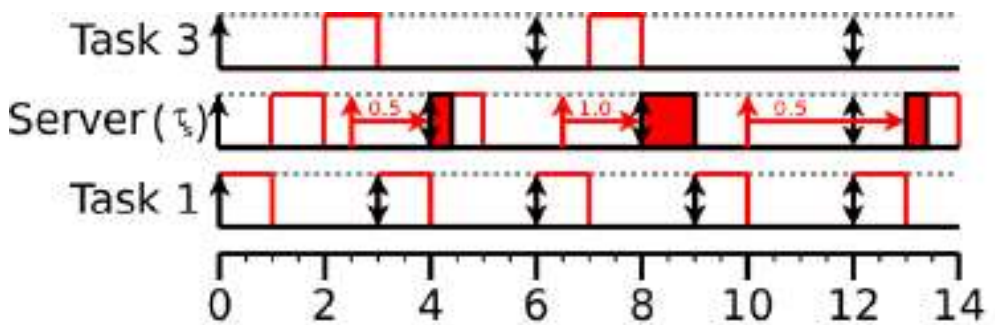
Background execution

- The background execution is very easy to implement and does not interfere directly with the periodic system/tasks.
  - However, interference may still occur indirectly, via ISR, non-preemptive system calls, etc.
- On the other hand, aperiodic tasks may suffer big delays, depending on the periodic load.
  - This delay may be upper-bounded considering the aperiodic tasks as the lowest priority task.
- The performance is poor for aperiodic tasks that have some level of real-time requirements, tough it can be acceptable to the ones that don't have such requirements.



Aperiodic servers

- When the background execution service does not allow meeting the real-time constraints of aperiodic tasks, the response time of these can be improved by using a **pseudo-periodic task** whose only function is to execute the active aperiodic tasks.
- This pseudo-task is designated **aperiodic server** and is characterized by a **period** TS and a **capacity** CS.
- It is now possible to insert the aperiodic server in the set of periodic tasks and assign it sufficient priority to provide the required QoS.



Aperiodic servers

- There are many types of aperiodic servers, both based on **fixed and dynamic priorities** , which vary in terms of:
  - Impact on the schedulability of the periodic tasks
  - Average response time to aperiodic requests
  - Computational cost/overhead, memory and implementation complexity.
- Fixed priority: **Polling Server** , **Deferable Server** , Priority Exchange Server, **Sporadic Server** ,...
- Dynamic priorities: Adapted fixed-priority servers, **Total Bandwidth Server** , **Constant Bandwidth Server** , ...

Worst-case response time to aperiodic requests

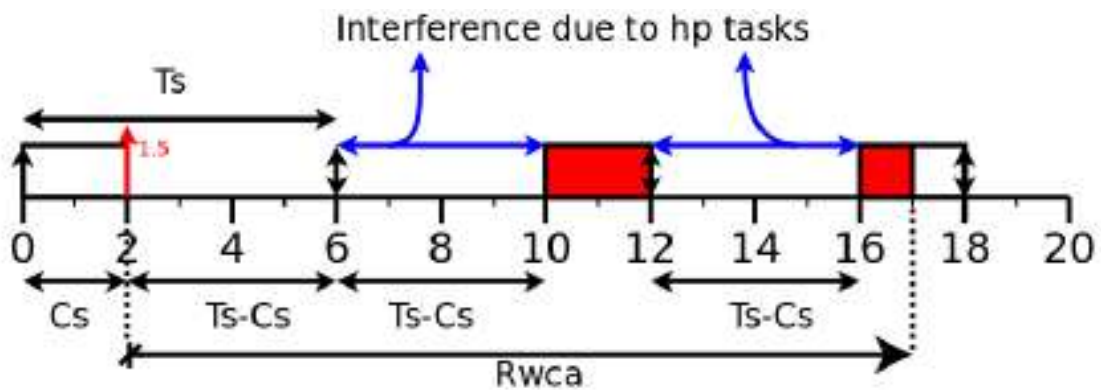
Worst-case response time upper bound:

- Equal to all servers that can be modeled by a periodic task
- It is assumed that (worst-case scenario):
  - The server is a periodic task $\tau_s(C_S, T_S)$
  - Suffers maximum jitter on the instant of the aperiodic request
  - Suffers maximum delay in all successive instances

WCRT of an aperiodic request of task τ_i

$$Rwc_i^a = Ca_i + (T_s - C_s) \cdot (1 + \lceil \frac{Ca_i}{C_s} \rceil)$$

where Ca_i is the worst-case execution time of aperiodic task i



Worst-case response time to aperiodic requests

Considering several aperiodic requests:

- If there are several aperiodic requests queued for the same server i (Na_i), sorted by a suitable criteria, the worst-case response time is:

WCRT of a set of aperiodic requests Ca_k to a server

$$\forall i = 1..Na, Rwc_i^a = \sum_{k=1}^{i}(Ca_k) + (T_s - C_s) \cdot (1 + \left\lceil \frac{\sum_{k=1}^{i} Ca_k}{C_s} \right\rceil)$$

- It is assumed that all requests are issued at the same instant, which corresponds to the worst-case scenario.

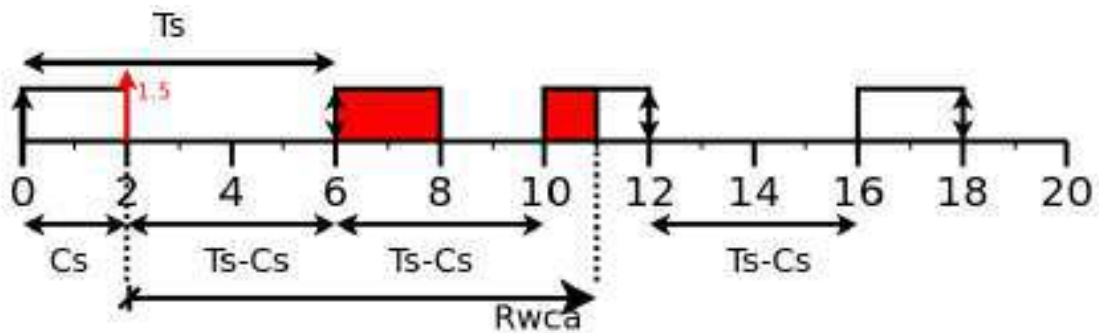
Worst-case response time to aperiodic requests

If the server has the **highest priority** :

- If, in a fixed priority system, the aperiodic server has the highest priority, the interference term, due to higher priority tasks, disappears, and the worst-case response time is:

WCRT of a set of aperiodic requests Ca_k to a server

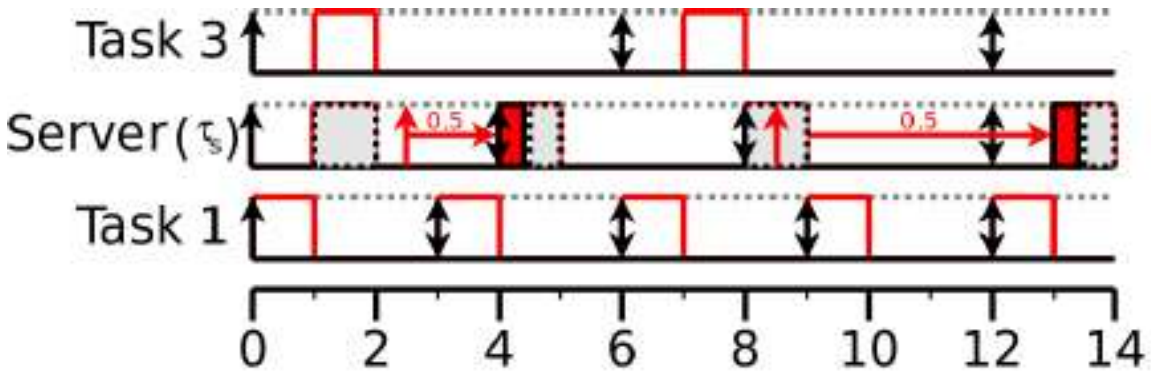
$$Rwc_i^a = Ca_i + (T_s - C_s) \cdot \lceil \frac{Ca_i}{T_s} \rceil$$



Polling server (PS)

- This fixed priority server is completely equivalent to the execution of a periodic task. The aperiodic requests are served only during the execution intervals granted to the server by the periodic task scheduler.

E.g. Polling server with $(C, T) = (1, 4)$



Polling server (PS)

- The implementation of a polling server is relatively simple. It only requires a queue for the aperiodic requests and control of the capacity used.
- The average response time to aperiodic requests is better than the one obtained with background execution, since it is possible to elevate the priority of the server. However it has relatively long unavailability periods.
- The impact on the periodic task set is exactly the same as the one of a periodic task.

Utilization test for RM + PS

$$U_p + U_s \leq (n+1) \cdot (2^{\frac{1}{n+1}} - 1)$$

Up: utilization of n periodic tasks

Polling server (PS)

Tighter tests

- The previous test (Liu & Layland bound) is independent of the utilization of each task. It is possible to improve the test (i.e. obtain tighter bounds).
- Let U_p and U_s be the utilization factors of the periodic task set and polling server, resp.

RM + PS

$$U_p \leq n \cdot [(\frac{2}{U_s+1})^{\frac{1}{n}} - 1]$$

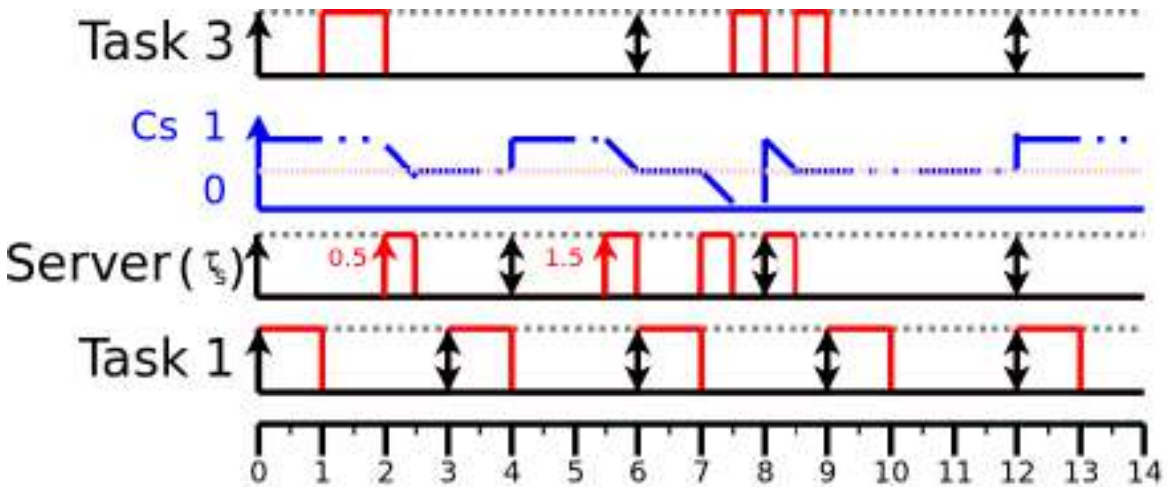
RM + PS, Hyperbolic Bound

$$\prod_{i=1}^{n} (U_i + 1) \leq \frac{2}{U_s+1}$$

Deferrable server (DS)

- The basic idea of this fixed-priority server is to handle aperiodic requests from the beginning of its execution until:
  - End of its period (TS) or
  - Its capacity (CS) gets exhausted
- The capacity is replenished at the beginning of each period.

E.g. Deferrable server with $(C, T) = (1, 4)$



Deferrable server (DS)

+ Simple implementation (similar to a PS).
+ The average response time to aperiodic requests is improved with respect to the PS, since it is possible to use the capacity of the DS during the whole period, provided that its capacity is not exhausted.
- However, there is a negative impact on the schedulability of the periodic tasks. The reason for this impact is that the delayed executions increase the load on the future. E.g., it is possible having two consecutive executions (back-to-back execution).

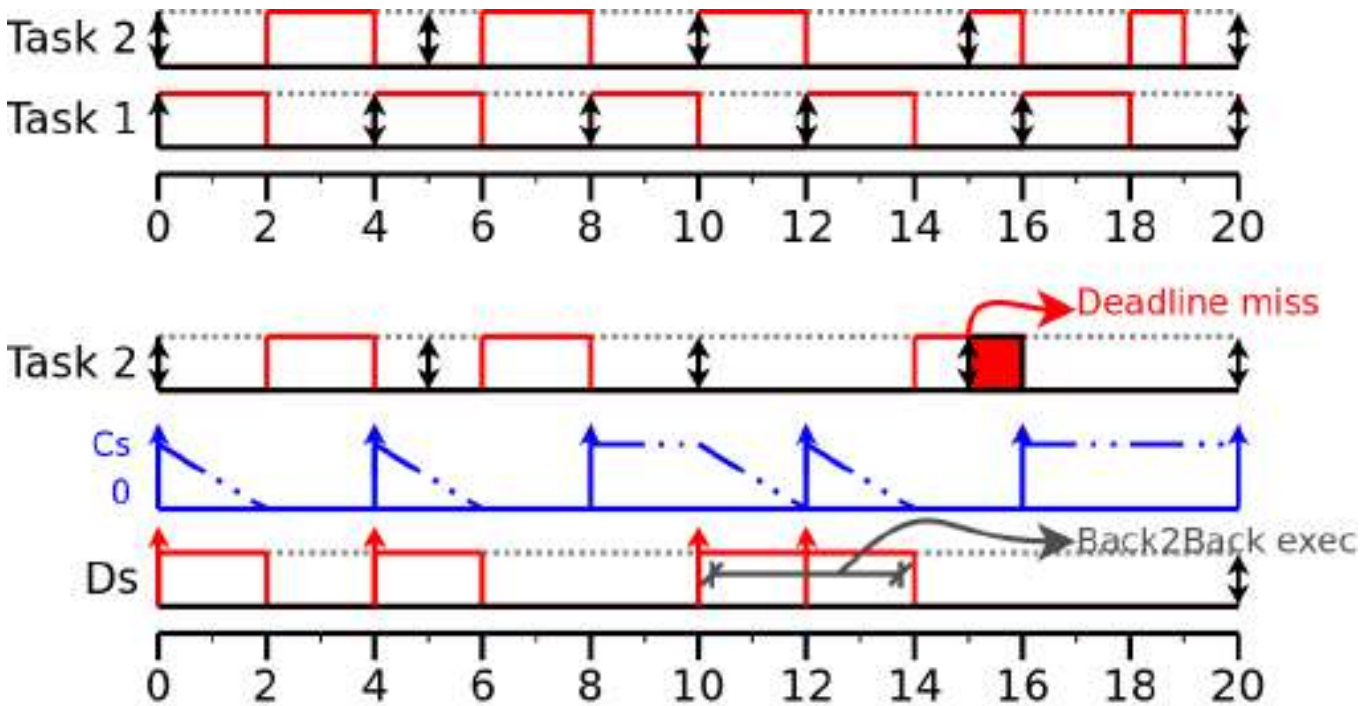
RM + DS: Least Upper Bound

$$U_{lub} = U_s + n \cdot [(\frac{U_s+2}{2 \cdot U_s+1})^{\frac{1}{n}} - 1]$$

Deferrable server (DS)

Illustration of a scenario in which replacing a periodic task by a DS causes deadline misses

- Periodic tasks
- Task 1 replaced by a DS



Sporadic server (SS) [Sprunt, Sha and Lehoczky, 89]

- The basic idea of this fixed-priority server is also allow the **execution** of the server at **any instant** (as the DS), however **without penalizing the schedulability** of the periodic system (as the PS).
- The SS replenishes the capacity not at the end of the period but instead **according with the time instants in which the capacity is actually used** .
- Definitions

  SS active : the server or $hep(SS)$ tasks are executing
  SS idle : processor idle or $lp(SS)$ task executing
  RT/RA : replenishing time/amount
- Replenishment rules:
  - RT is set when SS becomes active and $C_s > 0$ (t_A)
  - RA is computed when the SS becomes idle or C_s is exhausted (t_I). RA is the capacity used in the interval $[t_a, t_I]$

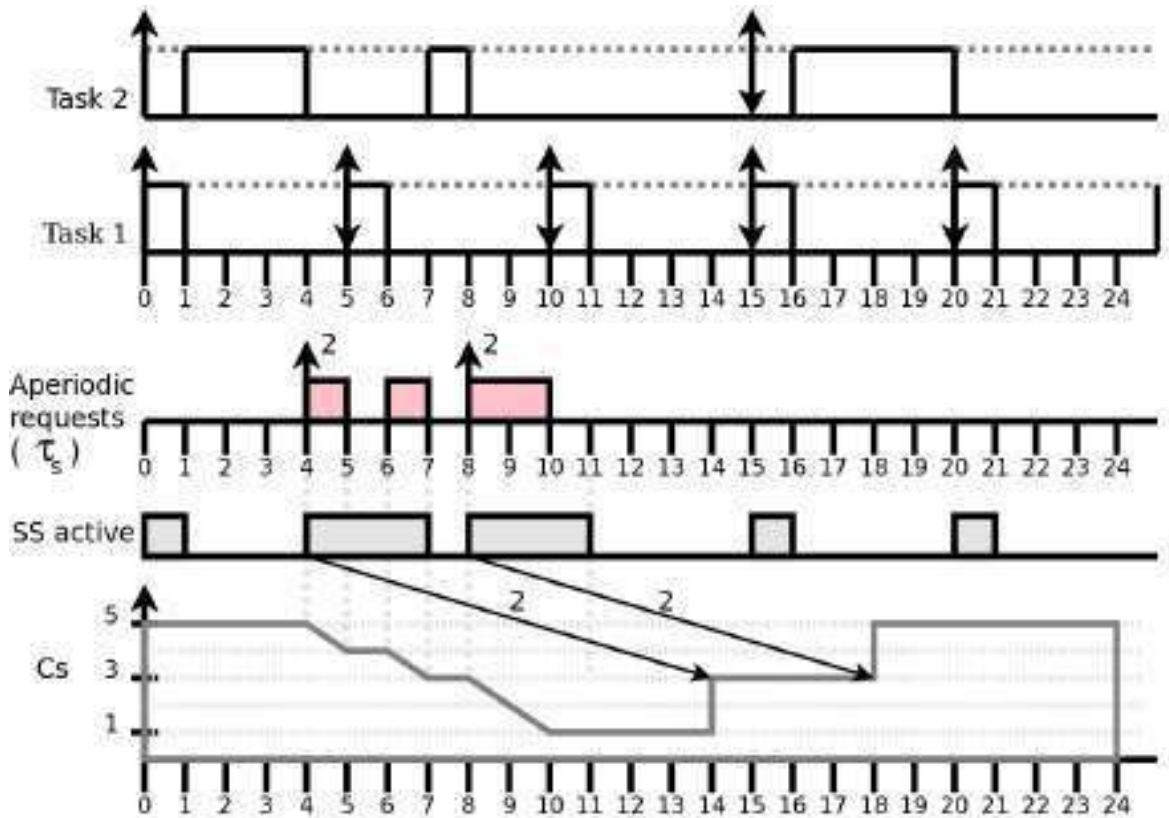
Sporadic server (SS) - Illustration

Periodic tasks

τ_i	C_i	T_i
1	1	5
2	4	15

Sporadic server: $SS : C_s = 5, T_s = 10$

Priority assignment: RM
(thus $Pr(\tau_1) > Pr(\tau_s) > Pr(\tau_2)$)



Sporadic server (SS)

- The implementation **complexity** of a sporadic server is **higher** than the one of PS and DS, due to the computation of the replenishment instants and, more importantly, to the complex timer management
+ The average response time to aperiodic requests is similar to the one of the DS
+ The impact on the **schedulability** of the periodic tasks is **exactly the same as the one of the PS**
  - The SS executes as soon as it has capacity, but the technique used to replenish the capacity preserves the timing behavior and bandwidth (unlike the DS).

RM + SS utilization tests

$$U_p \leq n \cdot [(\frac{2}{U_s+1})^{\frac{1}{n}} - 1] \; ; \; \prod_{i=1}^{n}(U_i + 1) \leq \frac{2}{U_s+1}$$

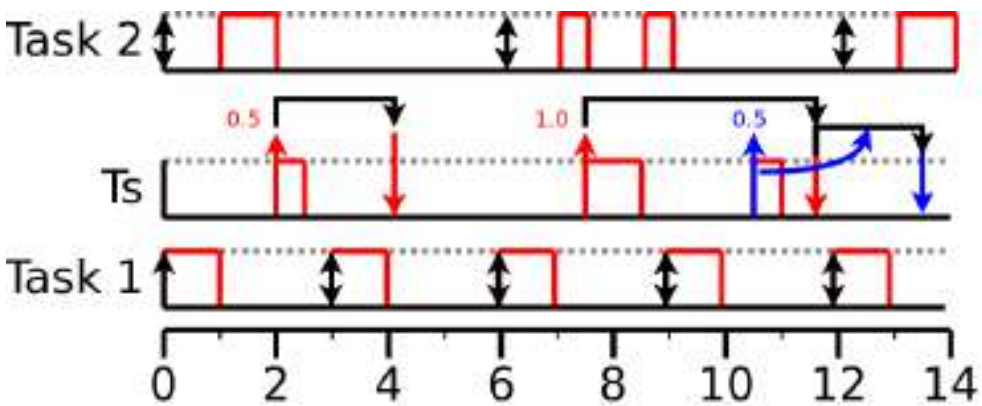
Total Bandwidth Server (TBS)

- The Total Bandwidth Server is a dynamic priority server which has the objective of executing the aperiodic requests **as soon as possible while preserving the bandwidth** assigned to it, to not disturb the periodic tasks. It was developed for EDF systems.
- When an aperiodic request arrives (r_k), it receives a deadline d_k,

TBS - deadline computation

$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

E.g. for $U_s = 25\%$



Total Bandwidth Server (TBS)

+ TBS is simple to implement and has low overhead, since it only requires a simple computation (deadline for each arrival). Then the aperiodic request is inserted in the ready queue and handled as any other task.
+ The average response time to aperiodic requests is smaller than the one obtained with dynamic- priority versions of fixed-priority servers.
+ The impact on the schedulability of the periodic task set is equal to the one of a periodic task with utilization equal to the one granted to the server. Using EDF+TBS:
  - Simple test: $U_P + U_S \leq 1$
- **Requires a priori knowledge of Ck and is vulnerable to overruns** .
  - After starting executing, a task may execute more time than the one declared

Constant Bandwidth Server (CBS)

- The Constant Bandwidth Server (CBS) is a dynamic priority server that was created to **solve the robustness problem of TBS** , enforcing bandwidth isolation.
- This goal is achieved by managing the execution time based on a budget/capacity (QS ,TS) scheme.
  - When an aperiodic request r_k arrives, it is computed a server deadline d_s, as follows:

$$\boxed{\textit{If } r_k + \frac{c_s}{U_s} < d_s^{actual}, \textit{ then } d_s^{actual} \textit{ does not change.}}$$

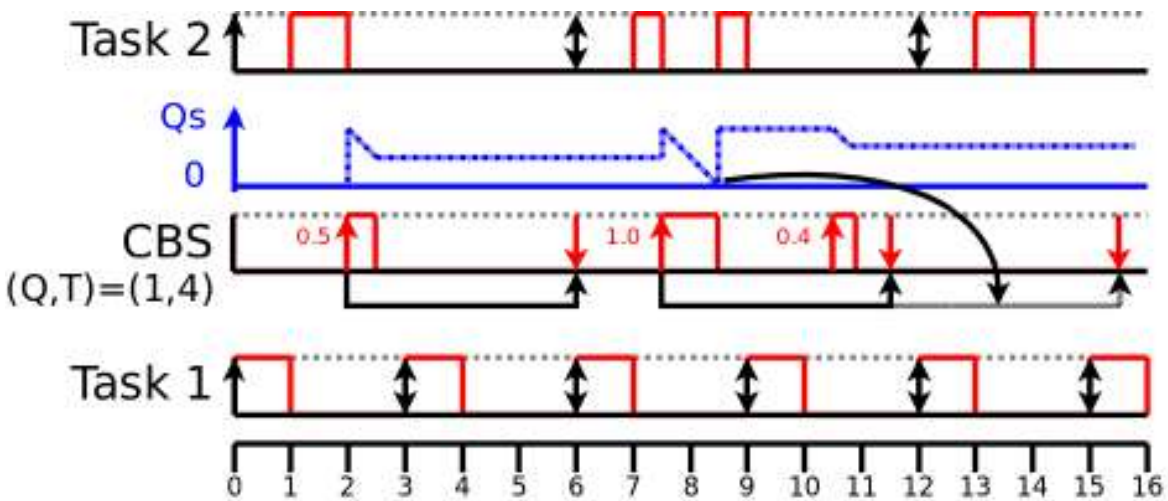
$$\boxed{\textit{Otherwise, } d_s = r_k + T_s \textit{ and } c_s = Q_s}$$

  - When the instantaneous capacity (c_s) gets exhausted, the capacity is replenished and the deadline postponed:

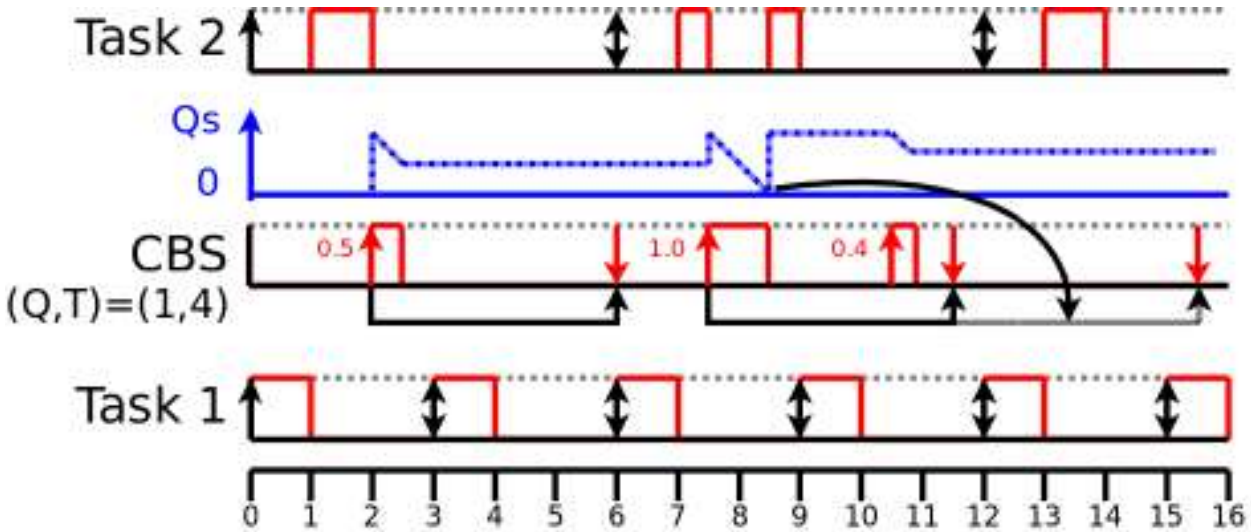
$$\boxed{d_s = d_s + T_s \textit{ and } c_s = Q_s}$$

Constant Bandwidth Server (CBS)

- The CBS assigns deadlines in such a way that prevents the **bandwidth given to the server** from being **higher than the one assigned to it** .
- If a task executes for longer than expected, its deadline is automatically postponed, lowering the priority of the task. This can also be seen as if the task period was artificially increased, in such a way the the utilization is maintained.



Constant Bandwidth Server (CBS)



Rules

[Arrival]

- If $r_k + \frac{c_s}{U_s} < d_s^{actual}$, then d_s^{actual} doesn't change [R1].
- Otherwise, $d_s = r_k + T_s$ and $c_s = Q_s$ [R2]

[c_s exhausted]

- $d_s = d_s + T_s; c_s = Q_s$ [R3]

- $t = 2.0$: $d_s^{actual} < r_k$, thus R2 applies
  $d_s = r_k + T_s = 2.0 + 4 = 6.0; c_s = 1$
- $t = 7.5$: $d_s^{actual} < r_k$, thus R2 applies
  $d_s = r_k + T_s = 7.5 + 4 = 11.5; cs = 1$
- $t = 8.5$: cs exhausted, thus R3 applies
  $d_s = d_s + T_s = 11.5 + 4 = 15.5; cs = 1$
- $t = 10.5$:
  $r_k + \frac{c_s}{U_s} = 10.5 + \frac{1}{0.25} = 14.5 < d_s^{actual}$, thus R1 applies: d_s^{actual} does not change

Constant Bandwidth Server (CBS)

- The implementation **complexity** of CBS is somehow **higher** than the one of TBS, due to the need to dynamically manage the capacity. Other than that, aperiodic tasks are put in the ready queue and handled as any regular periodic task.
+ The average response time to aperiodic requests is similar to TBS.
+ The impact on the **schedulability** of the periodic task set is **equal to the one of a periodic** task with an utilization equal to the one given to the server. Using EDF+CBS:

$$\boxed{U_p + U_s \leq 1}$$

Constant Bandwidth Server (CBS)

- The **big advantage** of CBS is that it provides **bandwidth isolation**
- If a task is served by a CBS with bandwidth U_s, in any time interval Δt that task will never require more than $\Delta t \cdot U_s$ CPU time.
- Any task $\tau_i(C_i, T_i)$ schedulable with EDF is also schedulable by a CBS server with $Q_s = C_i$ and $T_s = T_i$
- A CBS may be used to:
  - Protect the system from possible overruns in any task
  - Guarantee a minimum service to soft real-time tasks
  - Reserve bandwidth to any activity

Summary

- Joint execution of periodic and aperiodic tasks
  - Background execution of aperiodic tasks
- Notion and characteristics of aperiodic task servers
  - Fixed priority servers
    - Polling Server - PS
    - Deferrable Server - DS
    - Sporadic Server - SS
  - Dynamic priority servers
    - Total Bandwidth Server – TBS
    - Constant Bandwidth Server - CBS

Limited preemption, release jitter and overheads

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

November 14, 2022

Last lecture

Aperiodic task scheduling

- Joint execution of periodic and sporadic tasks
- Use of aperiodic task servers
  - Fixed-priority aperiodic task servers
  - Dynamic-priority aperiodic task servers

Agenda for today

- Non-preemptive scheduling
- Impact of Release Jitter
- Accounting for overheads
  - Cost of tick handler
  - Cost of context switching
  - Cost of ISR
- Considerations about the WCET

Non-preemptive scheduling

- Non preemptive scheduling consists in executing the jobs until completion, without allowing its suspension for the execution of higher priority jobs
- Main characteristics/ **advantages** :
  - Very simple to implement, as it is not necessary to save the intermediate job's state.
  - Stack size much lower (equal to the stack size of the task with higher requirements + interrupt service routines)
  - No need for any synchronization protocol to access shared resources, since tasks execute inherently with mutual exclusion

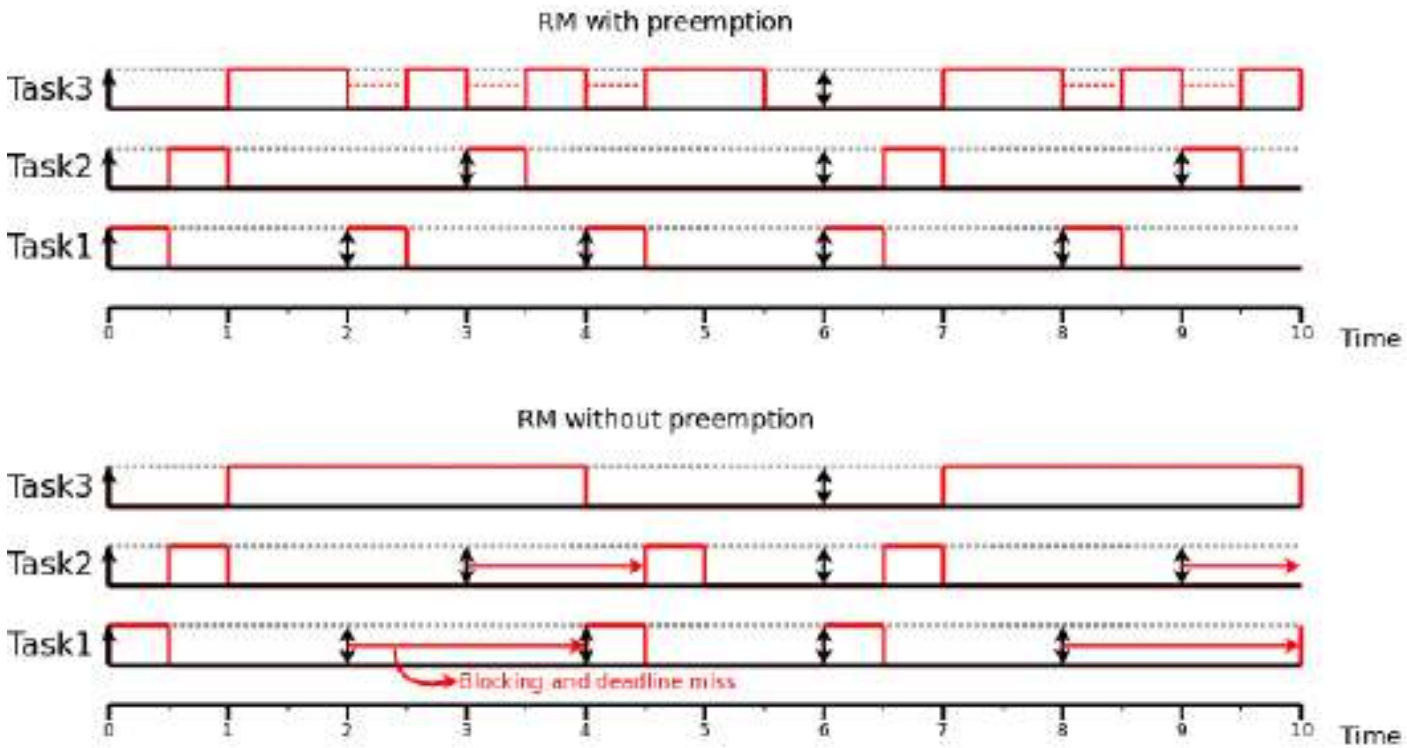
Non-preemptive scheduling

Main characteristics/ **disadvantages** :

- **Penalizes the system schedulability** , mainly when there are tasks with long execution times.
- This penalization may be excessive when, simultaneously, the system has tasks with high activation rates (short periods).
- The penalization can be seen as a blocking on the access of a shared resource, in the case the CPU.

Non-preemptive scheduling

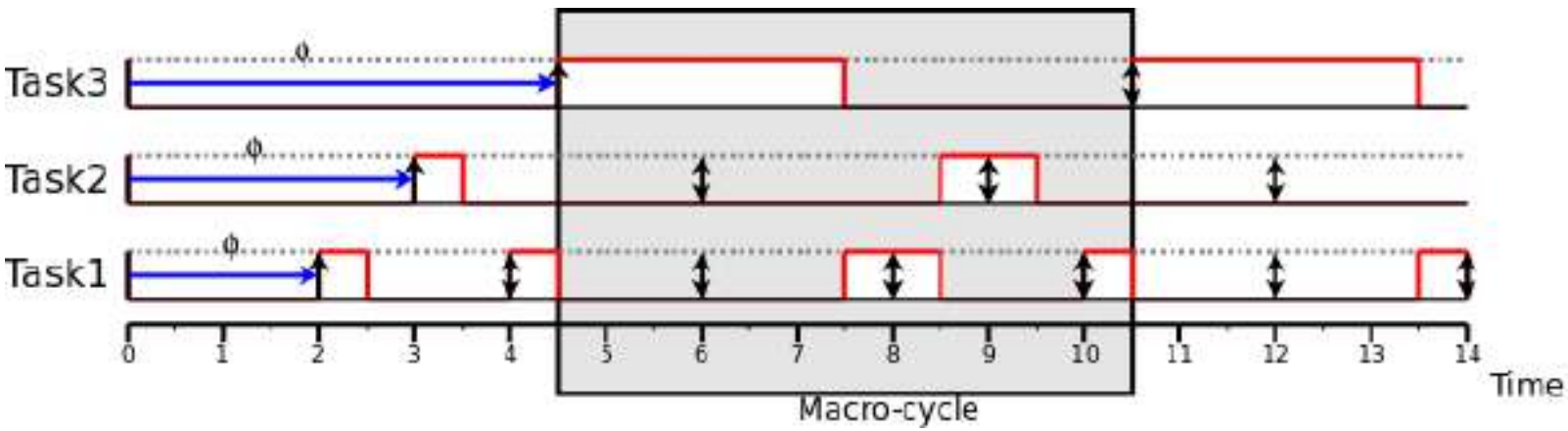
τ	C	T
1	0.5	2
2	0.5	3
3	3	6



Non-preemptive scheduling

- Sometimes the use of offsets may turn a system schedulable.
- However, finding these offset is usually is very complex ...

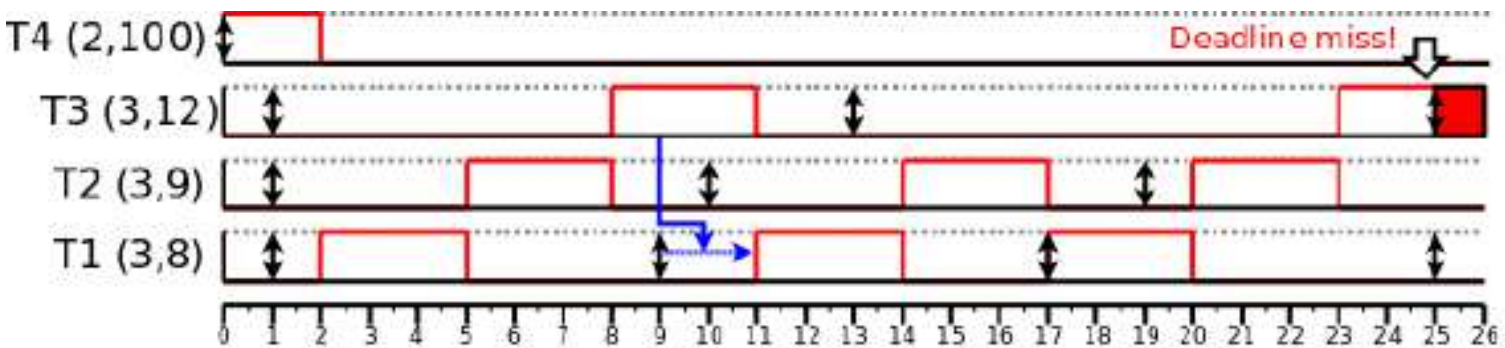
τ	C	T	ϕ
1	0.5	2	2
2	0.5	3	3
3	3	6	4.5



Non-preemptive scheduling

Computation of the Rwc_i with fixed priorities:

- The feasibility analysis must be adapted and becomes more complex.
- In non-preemptive scheduling **the longest response time does not necessarily coincide with the first job after the critical instant** .
  - **Self-pushing phenomenon** : high priority jobs activated during the first instance of τ_i are pushed ahead to the following jobs of τ_i, which may experience higher interference



Non-preemptive scheduling

- Therefore the response time analysis must be performed until the processor finishes executing all tasks with priority greater than or equal to P_i: **Level-i Active Period (L_i)**

$$B_i = \max_{j:P_j<P_i} \{C_j - \delta\}, \delta = \textit{clock resolution}$$

$$L_i(0) = B_i + C_i$$

$$L_i(s) = B_i + \sum_{h:P_h \geq P_i} \left\lceil \frac{L_i(s-1)}{T_h} \right\rceil \cdot C_h$$

L_i is the smallest value for which $L_i(s) = L_i(s-1)$

- For a generic task τ_i we must compute the response time for all jobs $\tau_{i,k}$, where $k = 1, \ldots K_i$ and

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil$$

Non-preemptive scheduling

The starting times of the relevant jobs can be computed as follows:

$$\boxed{s_{i,k}^{(0)} = B_i + \sum_{h:P_h>P_i} C_h}$$

$$\boxed{s_{i,k}^{(l)} = B_i + (k-1)\cdot C_i + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{s_{i,k}^{(l-1)}}{T_h} \right\rfloor + 1 \right) \cdot C_h}$$

Since once started a job cannot be preempted, the finishing time is

$$\boxed{f_{i,k} = s_{i,k} + C_i}$$

And finally the worst-case response time is

$$\boxed{Rwc_i = \max_{k\in[1,K_i]} \{f_{i,k} - (k-1)\cdot T_i\}}$$

Non-preemptive scheduling

Exercise:

Given the task set below, compute the worst-case response time, without preemption, of each tasks. Assume RM priority assignment.

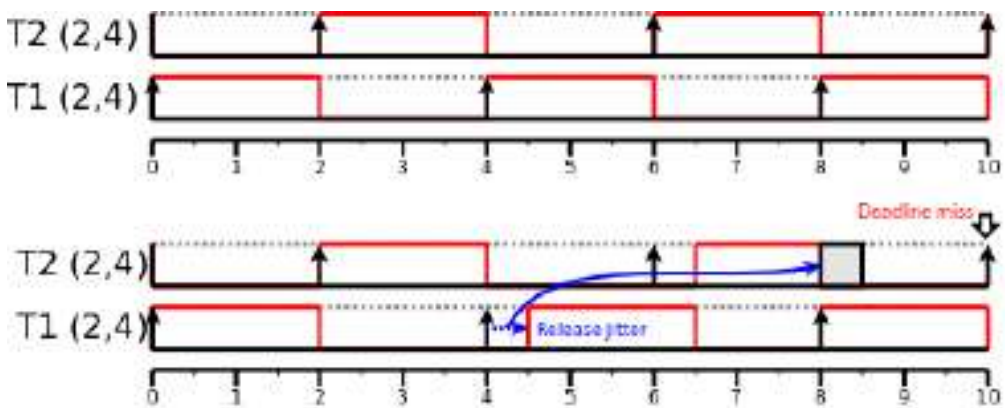
Is the task set schedulable without preemption? And with preemption?

τ_i	C_i	$T_i (= D_i)$
1	1	6
2	3	8
3	5	18

Impact of release jitter

Impact of the variations on the tasks' activation instants

- Tasks may suffer deviations on the respective activation instants, e.g. when a task is activated by the completion of another one, by an external interrupt or by the reception of a message on a communication port. In such cases the real time lapse between consecutive activations may vary with respect to the predicted values – **release jitter**
- The existence of release jitter must be taken into account in the schedulability analysis, as in such cases the tasks can **execute during time instants different from the assumed ones** .



Impact of release jitter

Schedulability tests including the impact of Relese Jitter:

- The presence of release jitter can be modeled by the anticipation of the activation instants of the following task instances.

Computing Rwc_i with release jitter (J_k) for preemptive systems with fixed priorities:

$$\forall i, Rwc_i = I_i + C_i, \text{ with } I_i = \sum_{k \in hp(i)} \left\lceil \frac{Rwc_i + J_k}{T_k} \right\rceil \cdot C_k$$

Solved iteratively, as usual:

$$Rwc_i(0) = \left(\sum_{k \in hp(i)} C_k \right) + C_i$$

$$Rwc_i(m+1) = \left(\sum_{k \in hp(i)} \left\lceil \frac{Rwc_i(m) + J_k}{T_k} \right\rceil \cdot C_k \right) + C_i$$

System overheads

- When developing real applications, system overheads may not be always negligible.
- In such cases it may be necessary to consider the impact of:
  - The processing cost of internal mechanisms (e.g. **tick handler**)
  - The overhead due to **context switching**
  - **Interrupt Service Routines**

System tick cost

Evaluating the computational cost of the system tick

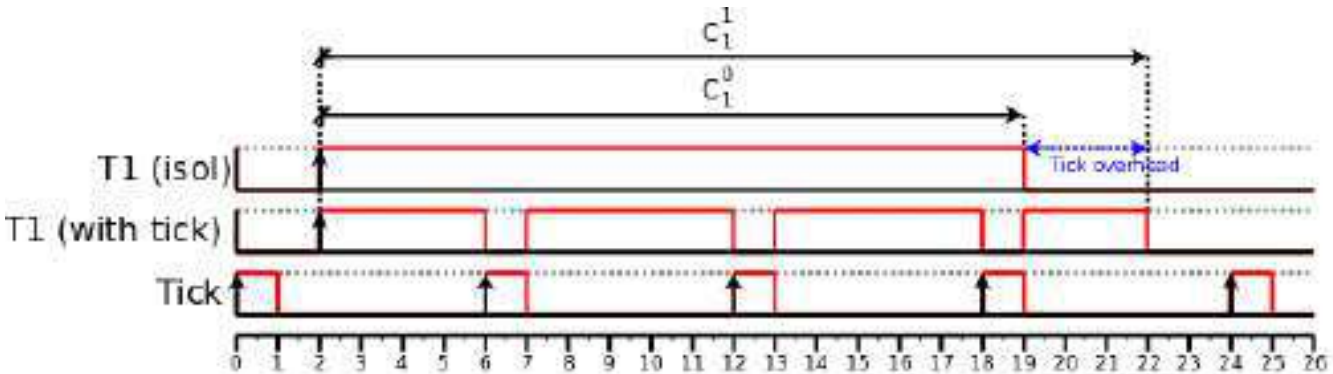
- The service to the system tick uses CPU time (overhead), which is **taken from the tasks' execution** .
- It is the highest priority activity on the system and can be modeled by a **periodic task** .
- The respective overhead (σ) may have a substantial impact on the system, as it is a part of the CPU availability that is not available to the application tasks.

System tick cost

Evaluating the computational cost of the system tick

- Can be measured either directly or via the timed execution of a long function, executed with and without tick interrupts (period T_{tick}) and measuring the difference on the execution times (C_1^0 and C_1^1 respectively).
- In this case the average value for σ is as follows:

$$\sigma = \frac{C_1^1 - C_1^0}{\left\lceil \frac{C_1^1}{T_{Tick}} \right\rceil}$$

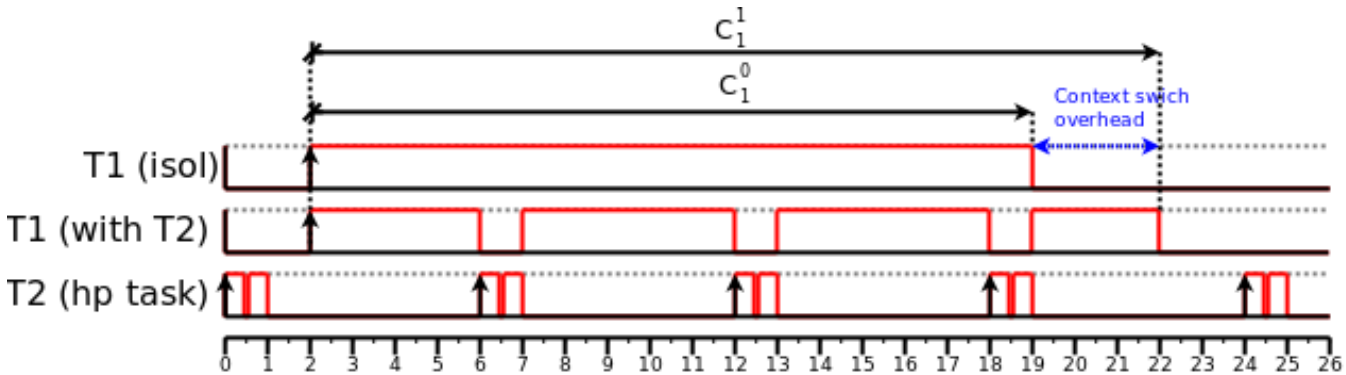


Context switch costs

Evaluating the cost of context switches

- Context switches also require CPU time to save and restore the tasks' context.
- A simple way of measuring this overhead (δ) consists in using two tasks, a long one (τ_1) and another one with higher priority (τ_2), quicker period (T_2) and empty (no code). Then it is only required measuring the execution time of the first task alone (C_1^0) and together with the second one (C_1^1).
- In this case, the average value for δ is:

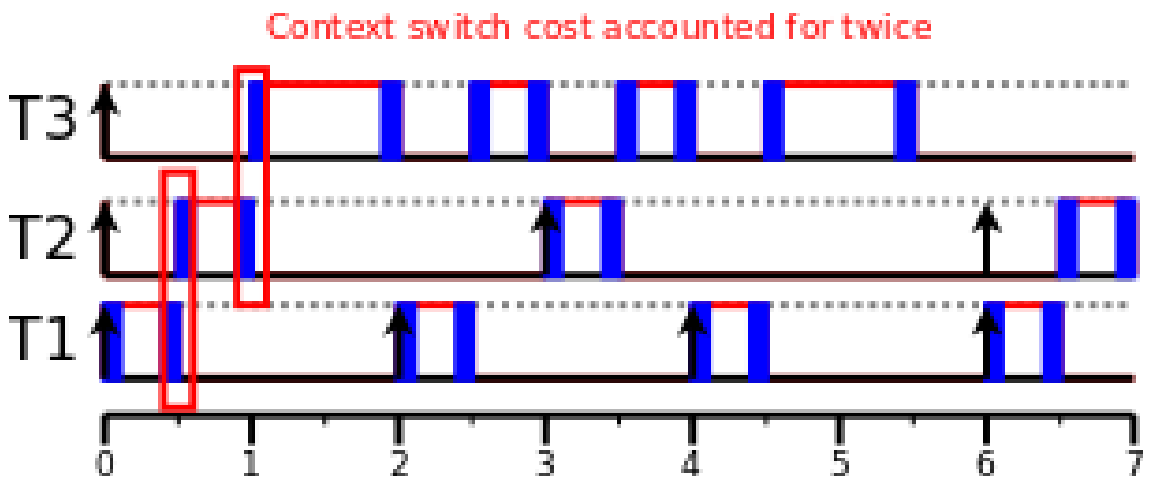
$$\delta = \frac{C_1^1 - C_1^0}{\left\lceil \frac{C_1^1}{T_2} \right\rceil}$$



Context switch costs

Evaluating the cost of context switches (cont.)

- A simple (but pessimistic) way of taking into account the overhead due to context switching (δ) consists in adding that time to the execution time of the tasks. This way it is taken into account not only the context switching overhead due to the task itself as well as the one relative to all context switches that may occur.
- Simple but pessimistic, as the overhead is take into account twice



Cost of ISR's

Impact of Interrupt Service Routines

- Generally, the **Interrupt Service Routines** (ISR) execute with an **higher priority** level than all other system tasks.
- Therefore, on a fixed priority system, the respective impact can be taken directly into account by **including these ISR as tasks** in the schedulability analysis.
- In systems with dynamic priorities the situation is much more complex (e.g. how to assign deadlines?). In these cases it is usually considered that the time windows in which such ISR execute are not available for normal tasks execution. This can be taken into account in the CPU load analysis.

Task's WCET

Evaluating the task's execution time

- Can be made via source code analysis, to determine the **longest execution path** , according with the input data.
  - Then the corresponding object code is analyzed to determine the required number of CPU cycles
- Note that the **execution time of a task may vary** from instance to instance, according with the input data or internal state, due to presence of conditionals and cycles, etc.

Task's WCET

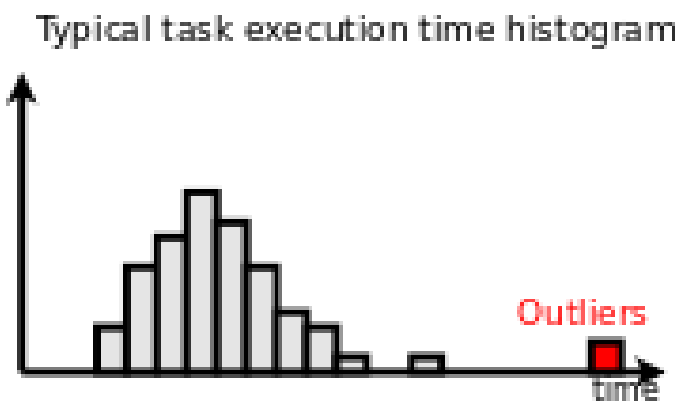
Evaluating the task's execution time (cont.)

- It is also possible execute tasks in isolation and in a controlled fashion, feeding them with **adequate input data** and measuring its execution time on the target platform.
  - This experimental method requires extreme care to make sure that the **longest execution paths are reached** , a necessary condition to obtain an upper bound on the execution time!
- **Modern processors** use features like pipelines and caches (data and/or instructions) that improve dramatically the average execution time but that present an **increased gap between the average and the worst-case scenarios** .
  - For these cases are used specific analysis that try to reduce the pessimism, e.g. by bounding the maximum number of cache misses and pipeline flushes, according with the particular instruction sequences.

Task's WCET

Evaluating the task's execution time (cont.)

- Nowadays there is an growing interest on stochastic analysis of the execution times and respective impact in terms of interference.
- The basic idea consists in determining the distribution of the probability of the execution times and use an estimate that covers a given target (e.g. 99% of the instances).
- In many cases (mainly when the worst case is infrequent and much worst than the average case) this technique allows reducing drastically the impact of the gap between the average execution time and the WCET (higher efficiency)



Summary

- Non-preemptive scheduling
- Impact of Release Jitter
- Accounting for overheads
  - Cost of tick handler
  - Cost of context switching
  - Cost of ISR
- Considerations about the WCET

Profiling and Code Optimization

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

November 21, 2022

Last lecture

Other Topics Relevant for Real-Time Operative Systems

- Non-preemptive scheduling
- Practical aspects related with the implementation of applications on a RTOS
  - Cost of tick handler
  - Cost of context switching
  - Measuring of WCET
  - Cost of ISR
  - Impact of Release Jitter

Agenda for today

Profiling and Code Optimization

- Code optimization techniques
  - Introduction
  - Processor-independent techniques
  - Techniques dependent on memory architecture
  - Processor-dependent techniques
- Profiling tools
  - Objectives and methodologies
  - Tools

Why optimizing code?

- Many real-time systems are used in applications:
  - Highly cost-sensitive
  - Where energy consumption must be minimized
  - Where physical space is restricted, ...
- If the execution time or footprint is too large just buying a faster processor IS NOT a solution!
- Code optimization allows to produce:
  - Faster programs:
    - Enables the use of slower processors, with lower cost, lower energy consumption.
  - Shorter programs:
    - Less memory, therefore lower costs and lower energy consumption

Low-level vs high-level languages

Is using assembly the solution?

- Assembly language programming
  - It potentially allows a very high level of efficiency, but:
    - Difficult debugging and maintenance
    - Long development cycle
    - Processor dependent - non-portable code!
    - Requires long learning cycles
- Programming in high-level languages (e.g. "C")
  - Easier to develop and maintain, relatively processor independent, etc. but:
    - Generated code potentially less efficient than code written in assembly

"C" vs Assembly

- Assembly programming, while potentially more efficient, in general turns out not to be a good approach:
  - Focus on implementation details and not on fundamental algorithmic issues, where the best optimization opportunities usually reside
    - E.g .: instead of spending hours building an "ultra-efficient" library for manipulating lists, it will be preferable to use "hash-tables";
    - Good quality compilers produce more efficient code than the one produced by an "average" programmer;
    - And finally:

John Levine, on Comp.Compilers

"Compilers make it a lot easier to use complex data structures, compilers don't get bored halfway through, and generate reliably pretty good code."

Then what is the best approach?

As in almost everything in life, "virtue is in the middle", or otherwise, in the "war" between "C" and Assembly wins ... whoever chooses both!

- General approach:
  - The programmer writes the application in a high-level language (e.g. "C")
  - The programmer uses tools to detect "hot spots" (points where the application spends more resources)
  - The programmer analyzes the generated code and ...
    - Re-write critical sections in assembly
    - and/or restructures high-level code to generate more suitable assembly code

CPU independent optimization techniques

Elimination of common sub-expressions

- Formally, the occurrence of an expression "E" is called a common sub-expression if "E" was previously calculated and the values of the variables in "E" have not changed since the last calculation of "E".
- The benefit is obvious: less code to run!

Code before (left) and after(right) optimization

```
b:
   t6 = 4 * i // E1
   x  = a[t6]
   t7 = 4 * i // E1
   t8 = 4 * j // E2
   t9 = a[t8]
   a[t7]  = t9
   t10    = 4 * j // E2
   a[t10] = x
   goto b
```

```
b:
    t6 = 4* i
    x  = a[t6]
    t8 = 4 * j
    t9 = a[t8]
    a[t6] = t9
    a[t8] = x
    goto b
```

CPU independent optimization techniques

Elimination of "dead code"

- If a certain set of instructions is not executed under any circumstances, it is called "dead code" and can be removed
- E.g. replacing the "if" with "#ifdef" allows the compiler to remove debug code at the pre-processing stage (no memory and CPU wasted)

```
...
debug = 0;
...
if (debug){
    print .....
}
```

```
...
#define DEBUG
...
#ifdef DEBUG
    print .....
#endif
```

CPU independent optimization techniques

Induction and force reduction variables

- An "X" variable is called an "L" cycle induction variable if each time "X" is changed in cycle "L", it is increased or decreased by a constant value
  - When there are two or more induction variables in a cycle, it may be possible to remove one of them
  - Sometimes it is also possible to reduce its "strength", i.e., its cost of execution
  - **Benefits**: lower and/or less costly computations

```
    j = 0
label_XXX:
    j  = j + 1
    t4 = 11 * j // t4 depends on j
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

```
    t4 = 0
label_XXX:
    t4 += 11 // J removed
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

CPU independent optimization techniques

Cycle expansion

- It consists of making multiple iterations of the calculations in each iteration of the cycle
- **Benefits**: reduction of overhead due to the cycle
- **Problems**: increased amount of memory
- Suitable for short cycles

Before:

```
int checksum(int *data, int N){
    int i, sum=0;
    for(i=0;i<N;i++)
    {
        sum += *data++;
    }
    return sum;
}
```

After:

```
int checksum(int *data, int N){
    int i, sum=0;
    for(i=0;i<N;i+=4)
    {
        sum += *data++;
        sum += *data++;
        sum += *data++;
        sum += *data++;
    }
    return sum;
}
```

Cycle expansion example

Before:

```
0x00:    MOV      r3,#0 ; sum =0
0x04:    MOV      r2,#0 ; i= 0
**********************************
0x08:    CMP      r2,r1 ; (i < N) ?
0x0c:    BGE      0x20 ; go to 0x20 if i >= N
0x10:    LDR      r12,[r0],#4 ; r12 <- data++
0x14:    ADD      r3,r12,r3 ; sum = sum + r12
**********************************
0x18:    ADD      r2,r2,#1 ; i=i+1 (N times)
**********************************
0x1c:    B        0x8 ; jmp to 0x08
0x20:    MOV      r0,r3 ; sum = r3
0x24:    MOV      pc,r14 ; return
```

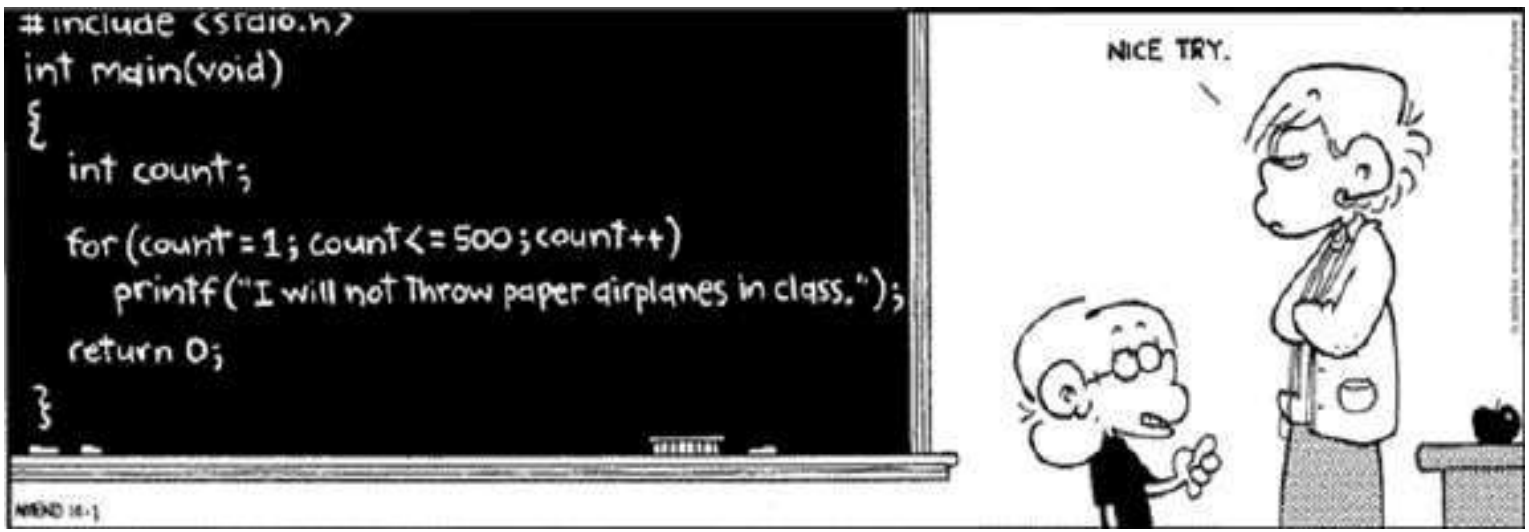
After:

```
0x00:    MOV      r3,#0   ; sum = 0
0x04:    MOV      r2,#0   ; i = 0
0x08:    B        0x30    ; jmp to 0x30
****************************************
0x0c:    LDR      r12,[r0],#4   ; r12 <- data++
0x10:    ADD      r3,r12,r3     ; sum = sum + r12
0x14:    LDR      r12,[r0],#4   ; r12 <- data++
0x18:    ADD      r3,r12,r3     ; sum = sum + r12
0x1c:    LDR      r12,[r0],#4   ; r12 <- data++
0x20:    ADD      r3,r12,r3     ; sum = sum + r12

0x24:    LDR      r12,[r0],#4   ; r12 <- data++
0x28:    ADD      r3,r12,r3     ; sum = sum + r12

****************************************
0x2c:    ADD      r2,r2,#4      ; i = i + 4 (N/4 times)
****************************************
0x30:    CMP      r2,r1   ; (i < N) ?
0x34:    BLT      0xc     ; go to 0x0c if i < N
0x38:    MOV      r0,r3   ; r0 <- sum
0x3c:    MOV      pc,r14  ; return
```

CPU independent optimization techniques

Cycle expansion (cont.)

- As we will see later on, there may be problems associated with this technique ...



CPU independent optimization techniques

Function inlining

- Replace a function call with the function code
  - **Benefits**: reduced overhead associated with calling a function
  - **Problems**: (possible) increased code size
  - Suitable when small functions are called **multiple times** from a small number of locations

Before:

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
```

After:

```
__inline int max(int a, int b)
{
int x;
x=(a>b ? a:b);
return x;
}
```

Inlining function example (without)

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}

int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
max
$a
0x00:    CMP      r0,r1; (x > y) ?
0x04:    BGT      0x0c; return if (x > y)
0x08:    MOV      r0,r1;  else r0 <- y
0x0c:    MOV      pc,r14 return
t


0x10:    STMFD    r13!,{r4,r14}; save registers
0x14:    MOV      r2,r0;    r2 <- x
0x18:    MOV      r3,r1;    r3 <- y
0x1c:    MOV      r1,r3;    r1 <- y
0x20:    MOV      r0,r2;    r0 <- x
0x24:    BL       max  ;  r0 <- max(x,y)
0x28:    MOV      r4,r0;     r4 <- a1
0x2c:    MOV      r1,r3;    r1 <- y


0x30:    ADD      r0,r2,#1; r0 <- x+1
0x34:    BL       max  ;          r0 <- max(x+1,y)
0x38:    MOV      r1,r0 ;          r1 <- a2
0x3c:    ADD      r0,r4,#1 ; r0 <- a1+1


0x40:    LDMFD    r13!,{r4,r14} ; restore


0x44:    B        max ;
```

Inlining function example (with)

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}

__inline int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
0x00:    CMP     r0,r1 ; (x<= y) ?
0x04:    BLE     0x10  ; jmp to 0x10 if true
0x08:    MOV     r2,r0 ; a1 <- x
0x0c:    B       0x14  ; jmp to 0x14
0x10:    MOV     r2,r1 ; a1 <- y if x <= y
0x14:    ADD     r0,r0,#1; generate r0=x+1
0x18:    CMP     r0,r1   ;   (x+1 > y) ?
0x1c:    BGT     0x24    ;jmp to 0x24 if true
0x20:    MOV     r0,r1   ; r0 <- y
0x24:    ADD     r1,r2,#1 ; r1 <- a1+1
0x28:    CMP     r1,r0 ; (a1+1 <= a2) ?
0x2c:    BLE     0x34  ; jmp to 0x34 if true
0x30:    MOV     r0,r1 ; else r0 <- a1+1
0x34:    MOV     pc,r14
```

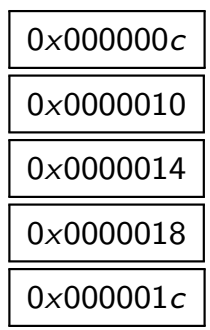
Cache impact

The use of techniques such as cycle expansion or inline functions can cause **performance degradation in systems with cache** !

Before cycle expansion:

```
int checksum(int *data, int N)
{
    int i;
    for(i=N;i>=0;i--)
    {
        sum += *data++;
    }
    return sum;
}
```

```
0x0000000c:    LDR     r3,[r2],#4
0x00000010:    ADD     r0,r3,r0
0x00000014:    SUB     r1,r1,#1
0x00000018:    CMP     r1,#0
0x0000001c:    BGE     0xc
```



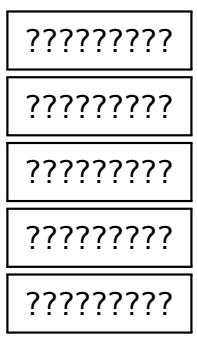
Instructions cache

Cache impact (cont.)

After cycle expansion:

```
int checksum(int *data, int N)
{
    int i;
    for(i=N;i>=0;i-=4)
    {
        sum += *data++;
        sum += *data++;
        sum += *data++;
        sum += *data++;
    }
    return sum;
}
```

```
0x00000008:    LDR     r3,[r0],#4
0x0000000c:    ADD     r2,r3,r2
0x00000010:    LDR     r3,[r0],#4
0x00000014:    ADD     r2,r3,r2
0x00000018:    LDR     r3,[r0],#4
0x0000001c:    ADD     r2,r3,r2
0x00000020:    LDR     r3,[r0],#4
0x00000024:    ADD     r2,r3,r2
0x00000028:    SUB     r1,r1,#4
0x0000002c:    CMP     r1,#0
0x00000030:    BGE     0x8
```



Cache capacity smaller than cycle code

Successive cache misses and updates!

Optimization techniques dependent on memory architecture

Memory access order

- In matrices, the "C" language defines that the rightmost index defines adjacent memory positions
- Significant impact on cache memory data in structures with high dimension

Array p[j][k]

...	...
j=0	k=0 k=1 k=2
j=1	k=0 k=1 k=2
j=2	k=0 k=1 k=2
...	...

Optimization techniques dependent on memory architecture

Better performance when the internal cycle corresponds to the rightmost index

```
//Poor performance with cache
for (k=0; k<=m; k++)
    for (j=0; j<=n; j++) )
        p[j][k] = ...
```

```
//Better performance with cache
for (j=0; j<=n; j++)
    for (k=0; k<=m; k++)
        p[j][k] = ...
```

- For homogeneous memory access, the performance is identical.
- But in the presence of **cache** the **performance can be very distinct** !
- Thus it depends on the memory architecture

Architecture-dependent optimization techniques

Depending on the processor family used as well as the type of coprocessors available, several optimizations are possible:

- Conversion from floating point to fixed point in the **absence of math co-processor**
  - **Benefits**
    - Lower computational cost,
    - Lower energy consumption,
    - Sufficient signal-to-noise ratio if correctly scaled,
    - Suitable e.g. for mobile applications.
  - **Problems**:
    - Dynamic range reduction,
    - Possible overflows.

Architecture-dependent optimization techniques

Use of assembly specifics

- Example: on ARM architecture it is possible to set flags when doing an arithmetic operation

Before:

```
int checksum_v1(int *data)
{
   unsigned i;
   int sum=0;

   for(i=0;i<64;i++)
    sum += *data++;

   return sum;
}
----------------------------
MOV  r2, r0; r2=data
    MOV  r0, #0; sum=0
    MOV  r1, #0; i=0
L1  LDR  r3,[r2],#4;   r3=*(data++)
****
    ADD  r1, r1, #1;   i=i+1 (a)
    CMP  r1, 0x40;     cmp r1, 64 (b)
****
    ADD  r0, r3, r0;   sum +=r3
    BCC  L1;       if i < 64, goto L1
    MOV  pc, lr;   return sum
```

After:

```
int checksum_v2(int *data)
{
   unsigned i;
   int sum=0;

   for(i=63;i >= 0;i--)
    sum += *data++;

   return sum;
}
----------------------------
    MOV  r2, r0; r2=data
    MOV  r0, #0; sum=0
    MOV  r1, #0x3f; i=63
L1  LDR  r3,[r2],#4;   r3=*(data++)
    ADD  r0, r3, r0;   sum +=r3
***
    SUBS  r1, r1, #1;  i--, set flags (rep. a and b)
***
    BGE  L1;  if i >= 0, goto L1
    MOV  pc, lr;       return sum
```

Architecture-dependent optimization techniques

There are many other techniques that, due to time limitations, are not covered in this course unit:

- Some classes:
  - Controlling resource use (e.g. variables assigned to registers)
  - Exploring parallelism
  - Multiple memory banks
  - Multimedia instructions
  - ....

Profiling

Task

Given the source code of a program, possibly written by someone else, perform its optimization!

Where to start?

- Analyze the source code and detect inefficient "C" code
- Re-write some sections in assembly
- Use more efficient algorithms

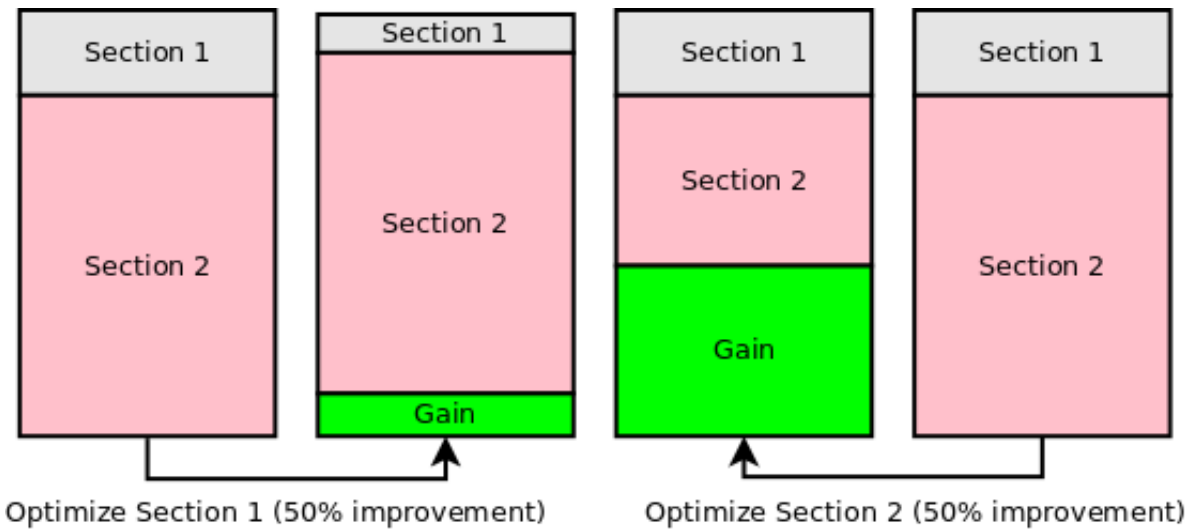
How to determine which sections to optimize?

- A typical application consists of many functions spread over different source files
- **Manual inspection** of the entire application code to determine which sections to optimize is in many cases **unpractical** !

Profiling

Amdahl's law

The performance gain that may be obtained when optimizing a section of code is limited to the fraction of the total time that is spent on that particular section.



- But how to determine the parts of code that consume the more significant share of CPU?

Profiling

Profiling

Collection of statistical data carried out on the execution of an application

- Fundamental to determine the relative weight of each function
- Approaches:
  - **Call graph profiling**: function invocation is instrumented
    - Intrusive, requires access to the source code, computationally heavy (overhead can reach 20%)
  - **Flat profiling**: the application status is sampled at regular time intervals
    - Accurate as long as functions execution time much bigger than the sampling period

Profiling

Example:

```
Routine          % of Execution Time
function_a       60%
function_b       27%
function_c        4%
...
function_zzz 0.01%
```

"80/20 Law"

In a "typical" application about 80% of the time is spent in about 20% of the code.

Profiling - tools

GNU Gprof

(https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html)

- Profiling requires several steps
  - Compilation and "linking" of the application with debug and profiling active
    - gcc **-pg** -o sample sample.c
  - Run the program to generate statistical data (profiling data)
    - ./sample
  - Run the gprof program to analyze the data
    - gprof ./sample [> text.file]

"-pg" : Generate extra code to write profile information suitable for the analysis program gprof.

Profiling - tools

GNU Gcov

(https://gcc.gnu.org/onlinedocs/gcc/Gcov.html)

- Coverage test, complementary to gprof.
- Indicates the number of times each line is executed
  - Must compile and link with "-fprofile-arcs -ftest-coverage" to generate additional information needed by gcov
    - gcc **-pg -fprofile-arcs -ftest-coverage** -o sample sample.c -lm
  - Run the program to generate statistical data and then run gcov
    - ./sample
    - gcov sample.c
  - File "sample.c.cov" contains the execution data

Profiling - tools

```
 ...   (main) ...
1:                5:{
-:                6: int i;
1:                7: int colcnt = 0;
200000:           8:for (i=2; i <= 200000; i++)
199999:           9: if (prime(i)) {
17984:           10:  colcnt++;
17984:           11:  if (colcnt%9 == 0) {
1998:            12:    printf("%5d\n",i);
1998:        13:    colcnt = 0;
-:               14:  }
-:               15: else
15986:           16:   printf("%5d ", i);
-:               17: }
1:               18: putchar('\n');
1:               19: return 0;
-:               20:}
199999:          21:int prime (int num) {
-:               22: /* check to see if the number is a prime? */
-:               23:  int i;


1711598836:      24:  for (i=2; i < num; i++)      Number of executions very high - optimization
1711580852:      25:  if (num \%i == 0)           target


182015:          26:   return 0;
17984:           27:  return 1;
-:               28:}
```

Profiling - tools

- Analyzing the code, an optimization was identified ...

```
....
199999:     22:int prime (int num) {
-:          23: /* check to see if the number is a prime? */
-:          24:  int i;


7167465:    25:  for (i=2; i < (int) sqrt( (float) num); i++)


7149370:    26:  if (num %i == 0)
181904:     27:   return 0;
....
```

Number of executions reduced by a factor of 238!!!

Profiling - tools

- Results with gprof

```
--------------------
Before optimization:
--------------------
Call graph
granularity: each sample hit covers 4 byte(s) for 0.02\% of 40.32 seconds
....
index   % time    self      children    called            name

[1]      100.0     0.01       40.31                      main  [1]
                  40.31        0.00    199999/199999      prime [2]

--------------------
After optimization:
--------------------
Call graph
granularity: each sample hit covers 4 byte(s) for 2.63\% of 0.38 seconds
...
index   % time    self      children    called            name

[2]      100.0    0.00        0.38                      main  [2]
                  0.38        0.00     199999/199999     prime [1]
```

Execution time reduced by a factor of 106!!!!

Profiling - tools

There are many other profiling tools. E.g. "perf":

- Performance counters for Linux ("perf" or "perf_events"): Linux tool that shows performance measurements in the command line interface.
- Can be used for finding bottlenecks, analysing applications' execution time, wait latency, CPU cycles, etc.
- Events of interest can be selected by the user ("perf list" allows to see the supported events)
- E.g.:

```
sudo perf stat  ls -al
... (command output ommited)
Performance counter stats for 'ls -al':

          4,97 msec task-clock                #    0,882 CPUs utilized
             2      context-switches          #    0,403 K/sec
             0      cpu-migrations            #    0,000 K/sec
           152      page-faults               #    0,031 M/sec
       7399469      cycles                    #    1,489 GHz
       6022842      instructions              #    0,81  insn per cycle
       1247081      branches                  #  251,024 M/sec
         37966      branch-misses             #    3,04% of all branches

   0,005634326 seconds time elapsed

   0,005555000 seconds user
```

Summary (1/2)

- Improving application performance
  - Why optimize
  - Assembly programming vs. "C" programming
  - Architecture-independent optimizations
    - Elimination of common sub-expressions, elimination of dead code, reduction of induction variables, expansion of cycles, inlining
  - Cache impact
  - Memory access

Summary (2/2)

- Architecture-dependent optimizations
  - Conversion of floating to fixed point arithmetic, assembly specifics
- Optimization / profiling
  - General methodology
  - Case study: use of gprof and gconv

Real-Time Operative Systems

Module: Multiprocessor Scheduling, V1.2

Paulo Pedreiras

DETI/UA/IT

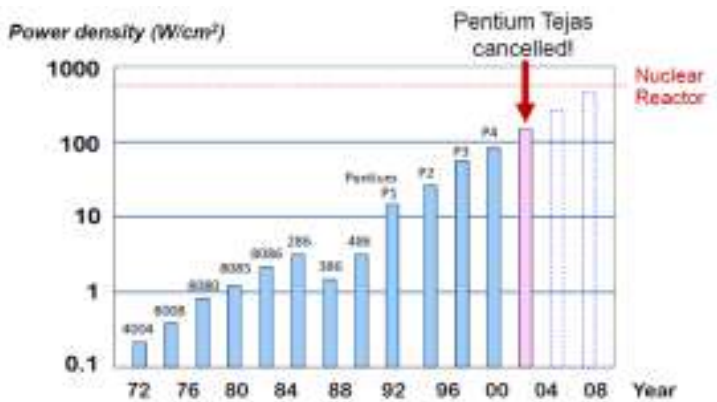
November 28, 2022

Outline

Why multicore?

Multicore systems are becoming increasingly popular for developing RT systems. **Why?**

- For years processing power increases arose (mainly) from using higher clock frequencies
- Higher clock frequencies and smaller transistors lead to higher dynamic and static power consumption
- Chip temperature eventually reached levels beyond the capability of cooling systems



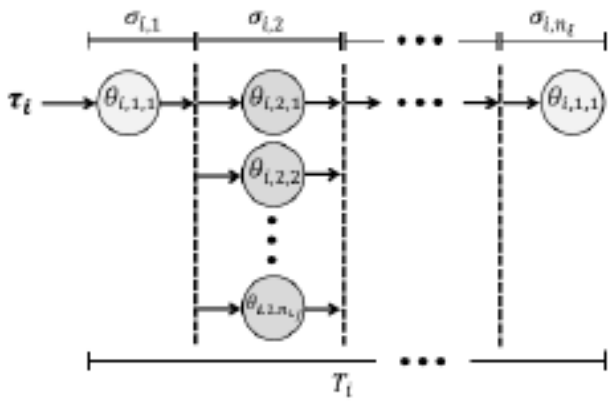
Why multicore?

... and the demand for processing power in real-time applications never ceased to increase ...

Problems and Challenges

But ...

- Moving/developing applications to multicore platforms is not simple nor straightforward
- Simple use of sequential languages hides the **intrinsic concurrency** that must be exploited to allow benefiting from the hardware redundancy
- Programmers have to take approaches that allow to split the code in segments that can be executed in parallel, on different cores (special languages, annotations, ...)



Problems and Challenges

Some problems and challenges include:

- How to split the code into segments/jobs that can be executed in parallel?
- How to allocate code segments to different cores?
- How to assess the schedulability on multicore platforms?
- How to handle dependencies?
- How to cope with the impact of shared resources on the WCET
- etc.

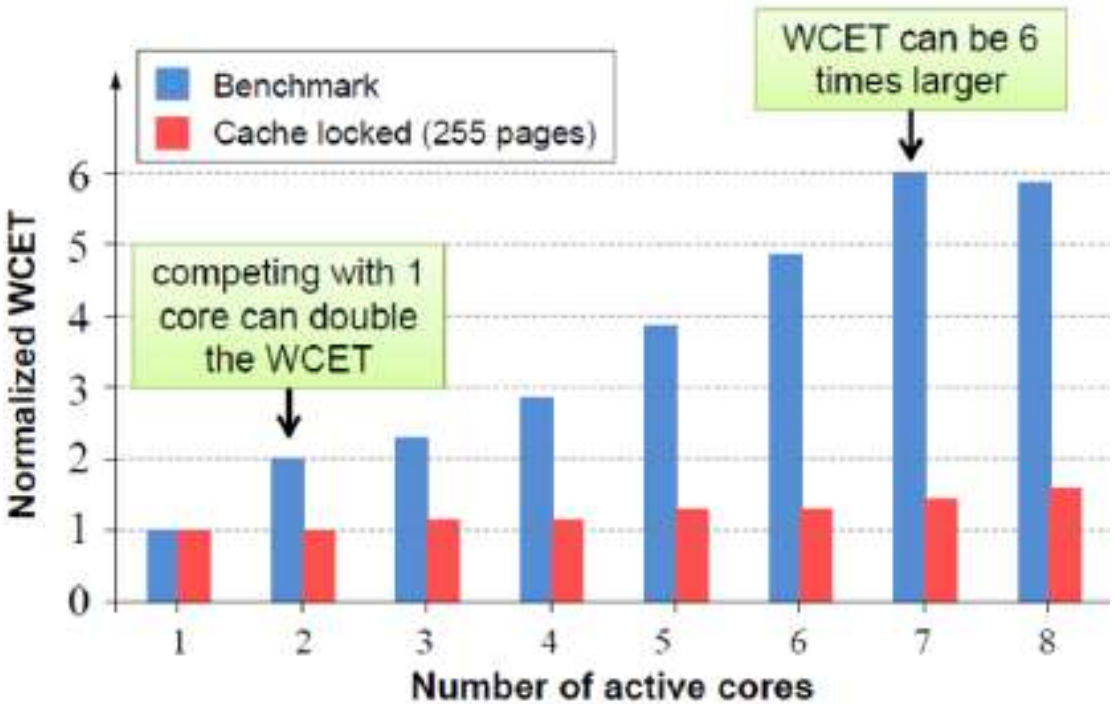
Is it that bad?

The WCET problem

- WCET is fundamental for RT analysis
- Existing RT analysis assumes that the **WCET of a task is constant** when it is executed alone or together with other tasks
- While this assumption is reasonable for single-core chips, it is **NOT true for multicore chips** !

The WCET problem

- Example: **Impact of shared resources on the WCET** of code on an 8-core platform, by Lockheed Martin Space Systems
- Shared resources include main memory, memory bus, cache, etc.



Example: impact on WCET

Reasons for the impact on the WCET

- In a **single core system** , concurrent tasks are **sequentially executed** on the processor.
  - Access to physical resources is implicitly serialized. E.g., two tasks can never cause a contention for a simultaneous memory access

.

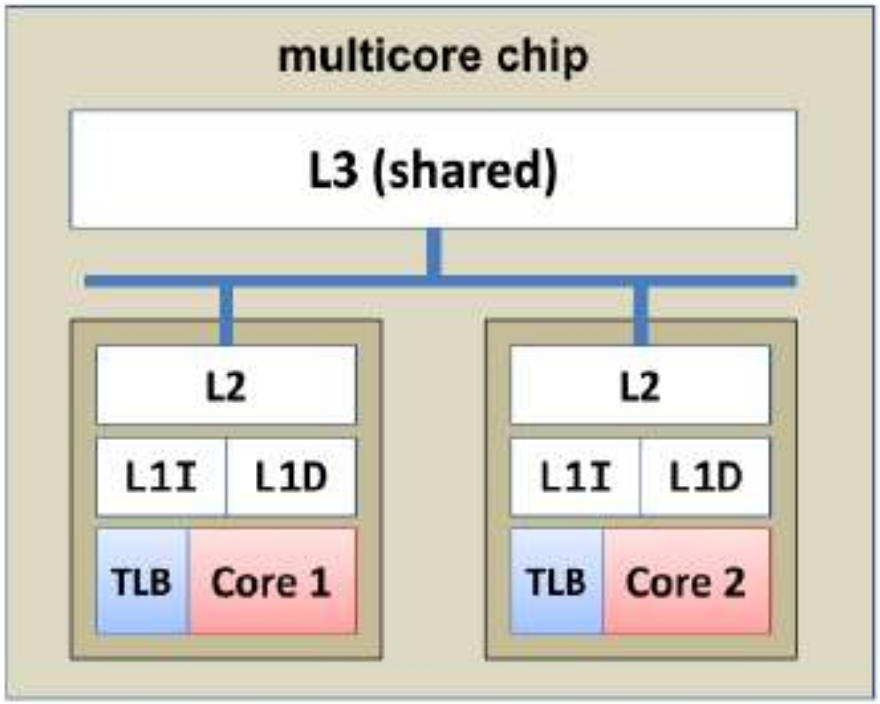
- In a **multicore platform** , different tasks can **run simultaneously** on different cores
  - several conflicts can arise while accessing physical resources

.

- Important issue, as existing RT analysis assumes that WCET is constant an known!

Example: impact on WCET

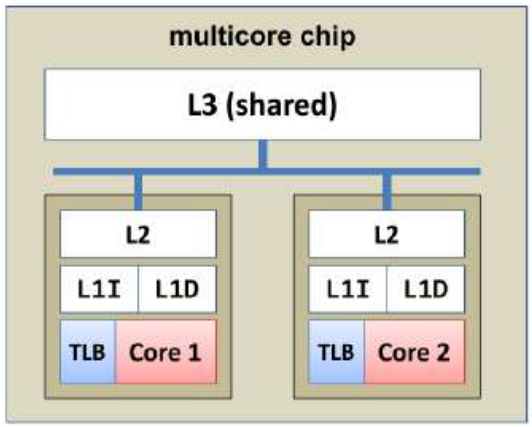
Cache in multicore systems



- L3 cache is typically shared by all cores ⇒ **cache conflicts**

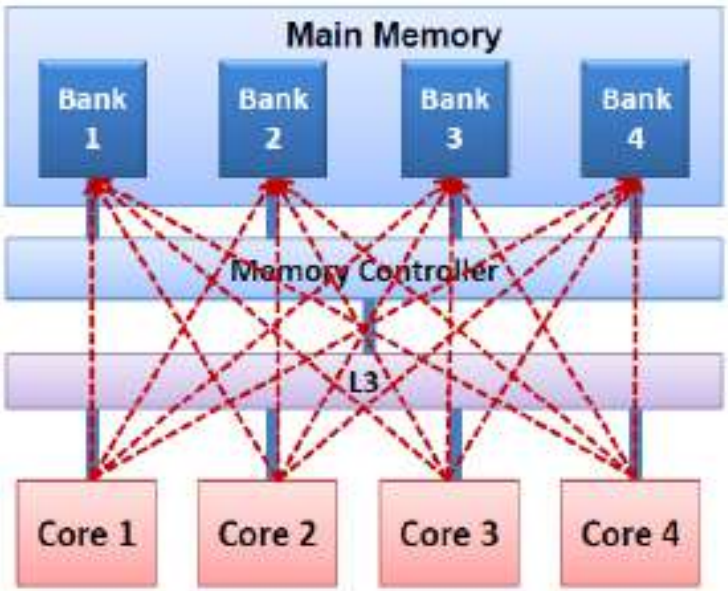
Example: impact on WCET

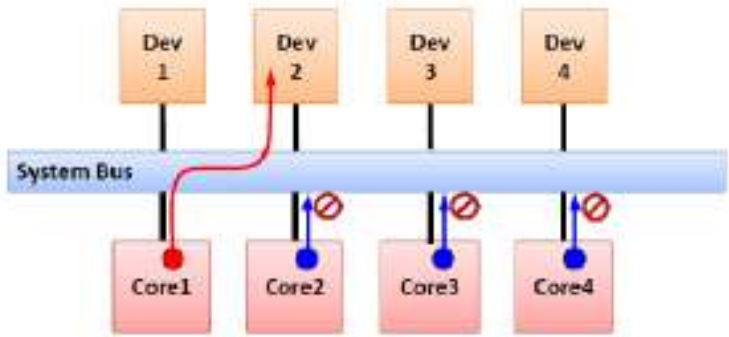
- In multicore systems, **L1 and L2** caches have the same problem seen in single-core systems
- **L3 cache lines can also be evicted by applications running on different cores**
- Possible approaches to attenuate the impact include e.g. partition the last level cache to simulate the cache architecture of a single-core chip. But the size of each partition becomes small...



Other sources of WCET growth

Similar situation with access to main memory, I/O devices, etc.

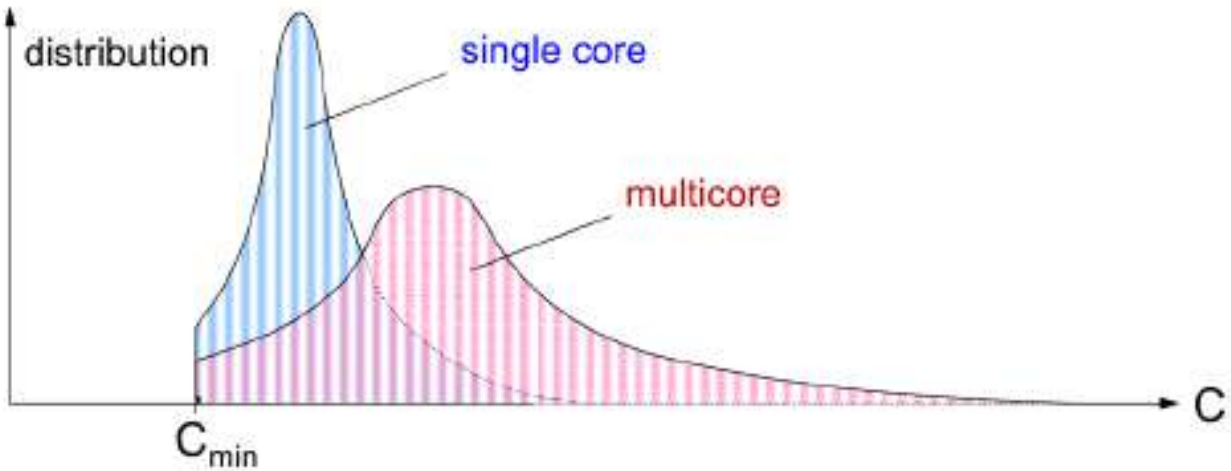




Note

Despite largely out of the control of the programmer, these issues have a significant impact on the scheduling strategies, as we will see!

Impact on WCET distribution



Note

In summary, the WCET uncertainty get much worse in Multicore systems!

Definitions - Processor types

Multiprocessor types usually considered

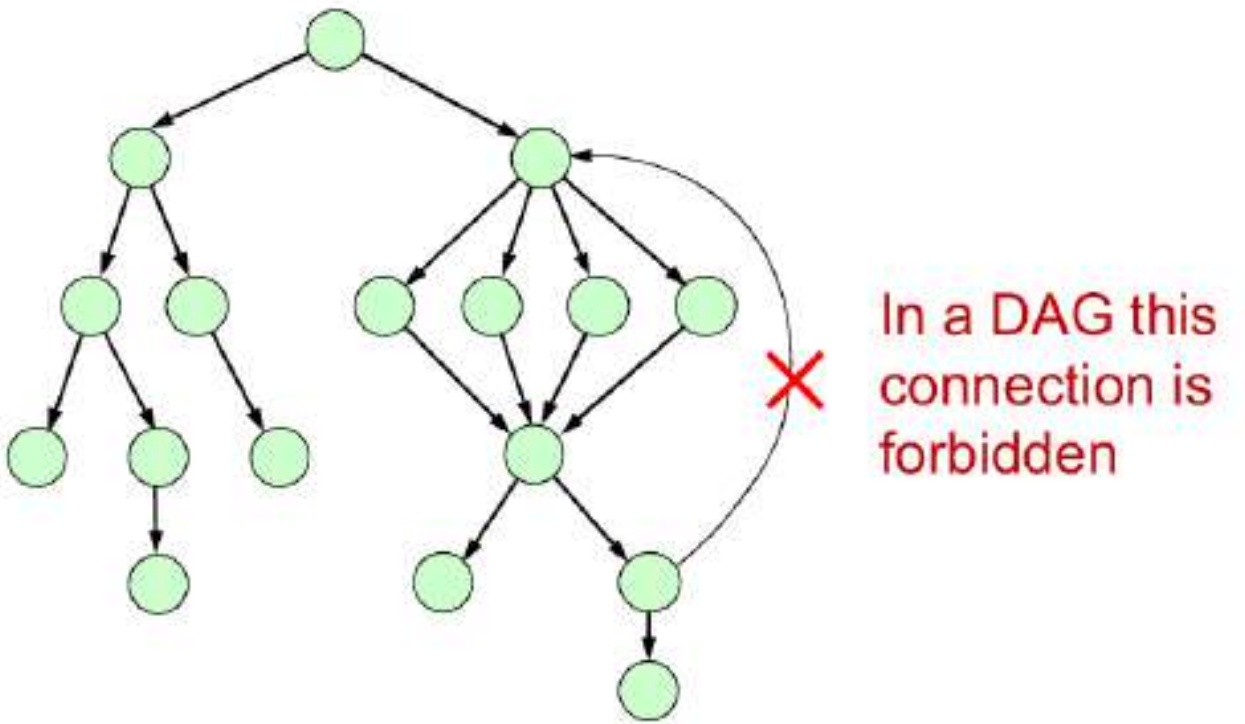
Identical Processors are of the same type and have the same speed. Each task has the same WCET on each processor.

Uniform Processors are of the same type but may have different speeds. Task WCETs are smaller on faster processors.

Heterogeneous Processors can be of different type. The WCET of a task depends on the processor type and the task itself.

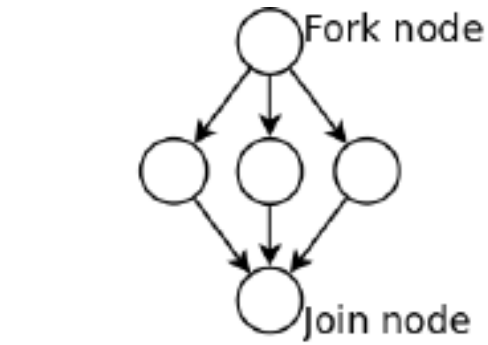
Task Models - DAG

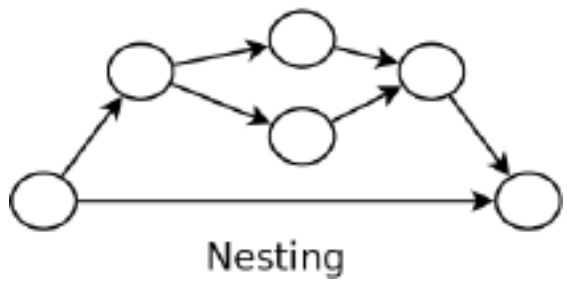
- Representing parallel code requires more complex structures than for sequential tasks.
- A Directed Acyclic Graphs (DAG) is a graph in which links have a direction and there are no cycles



Fork-Join Model Task Model

- Special type of Directed Acyclic Graph
- Nodes represent processing
- After a **Fork** node all successors have to be executed. The order is not relevant (nor defined).
- **Join** nodes only execute after the completion of all its predecessors
- Nesting is allowed

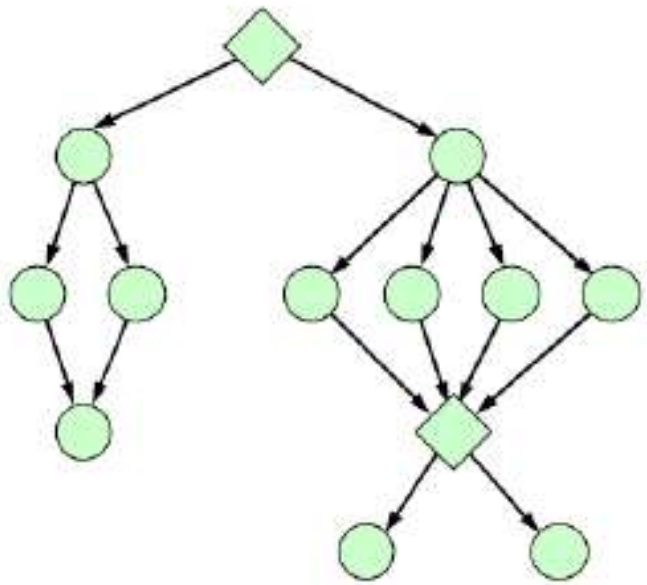




And-Or Graphs

The most general graph representation

- OR nodes represent conditional statements ( )
- AND nodes represent parallel computations ( )

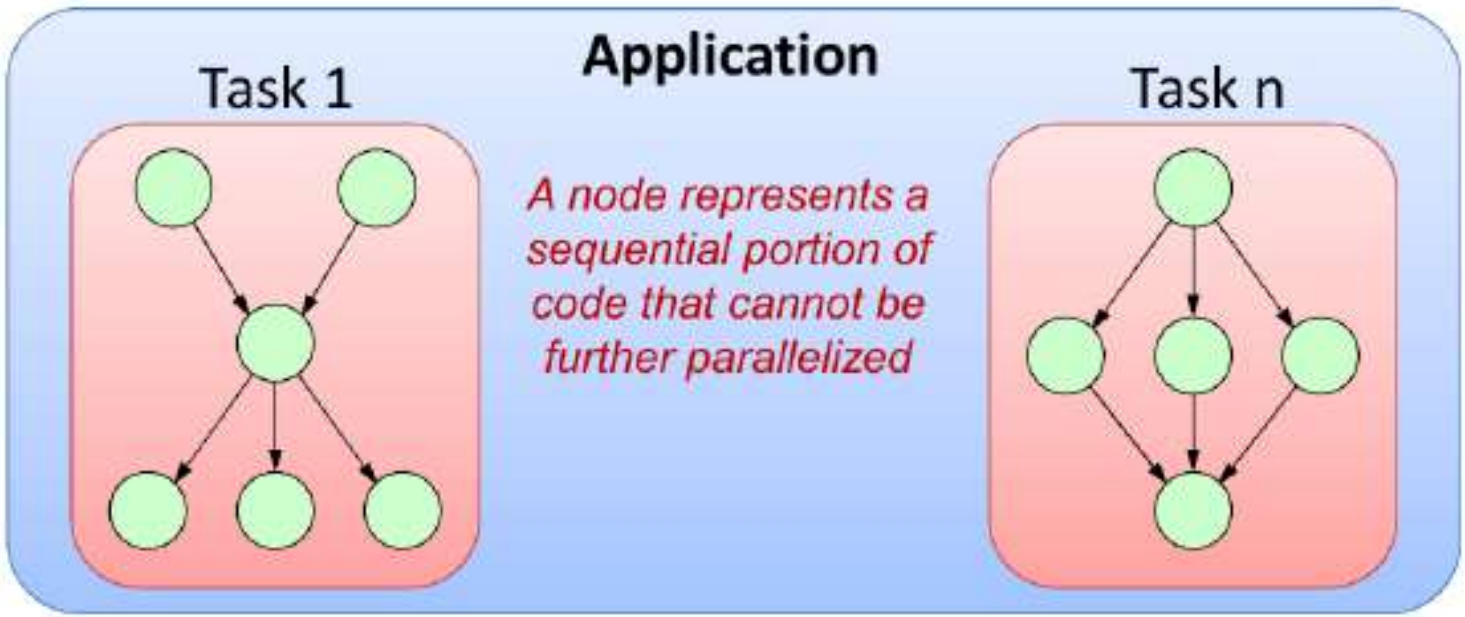


Note

There are other (less common) graph types ...

Application Model

An application can be modeled as a set of tasks, each described by a task graph.



Assumptions

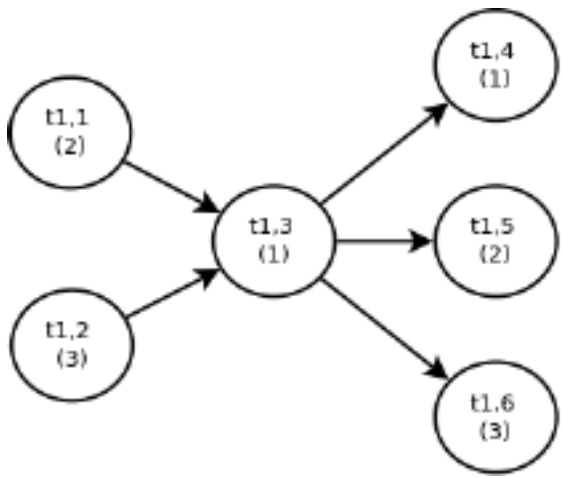
Task and system characteristics

- Arrival pattern
  - Periodic or Sporadic (period or mit is T)
- Preemption
  - Allowed, but eventually with critical sections
- Task migration allowed?
  - May or may not. We will see ...
- Task parameters
  - $\tau_i = \{\{c_{i,1}, c_{i,2}, ...c_{i,m}\}, D_i, T_i\}$
  - c_i: WCET of a segment
  - m: number of segments of τ_i
  - dependencies (in form of list(s))

Example

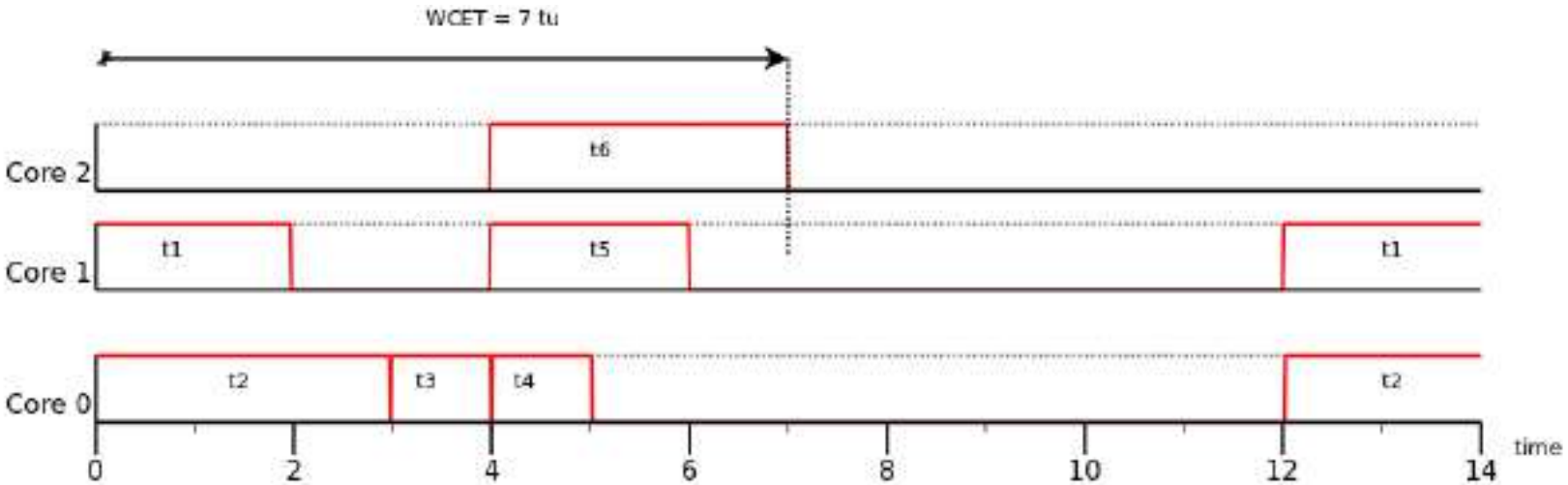
Consider the following example:

- Compute the minimum WCET of task τ_1 in isolation, considering the intra-task (segment) precedence relations.
- Number of cores is not restricted
- Task attributes
  - $\tau_1 = \{\{2, 3, 1, 1, 2, 3\}, 8, 12\}$
  - Dependencies: $\tau_{1,3} \leftarrow \{\tau_{1,1}, \tau_{1,2}\}$ and $\{\tau_{1,4}, \tau_{1,5}, \tau_{1,6}\} \leftarrow \tau_{1,3}$



Example (cont.)

- WCET = 7 tu for a number of cores greater or equal than 3.



Note

More processors are not always helpful ...

Additional definitions and observations

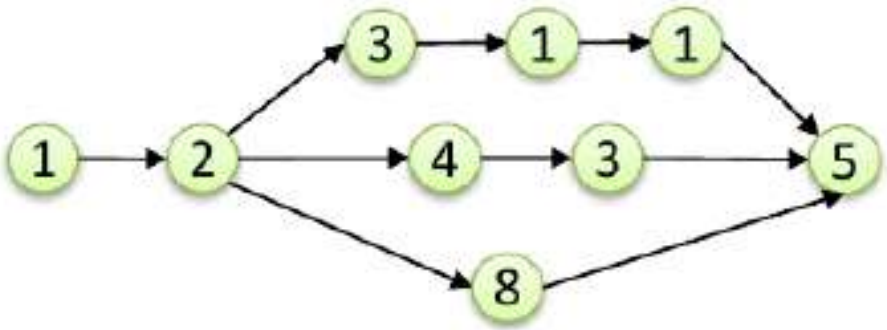
- Sequential Computation time: $C_i^S = \sum_{j=1}^{m_i} c_{i,j}$
- CPU utilization of a task: $U_i = \frac{C_i^S}{T_i}$

Immediate results:

- If $C_i^S \leq D_i$ the task is schedulable in a single core
- If $U_i > K$, where K is the number of cores, then the task **is not schedulable**

Critical path

Critical path length C_i^p length of the longest path in the graph



What is the critical path length?

Note

If follows immediately that if for any task $C_i^p > D_i$ then the application is not feasible in any number of cores

Assumptions

- Identical Multicore Systems (all cores are equal)
- WCET (upper bound) is known for any scenario
- Periodic or sporadic tasks

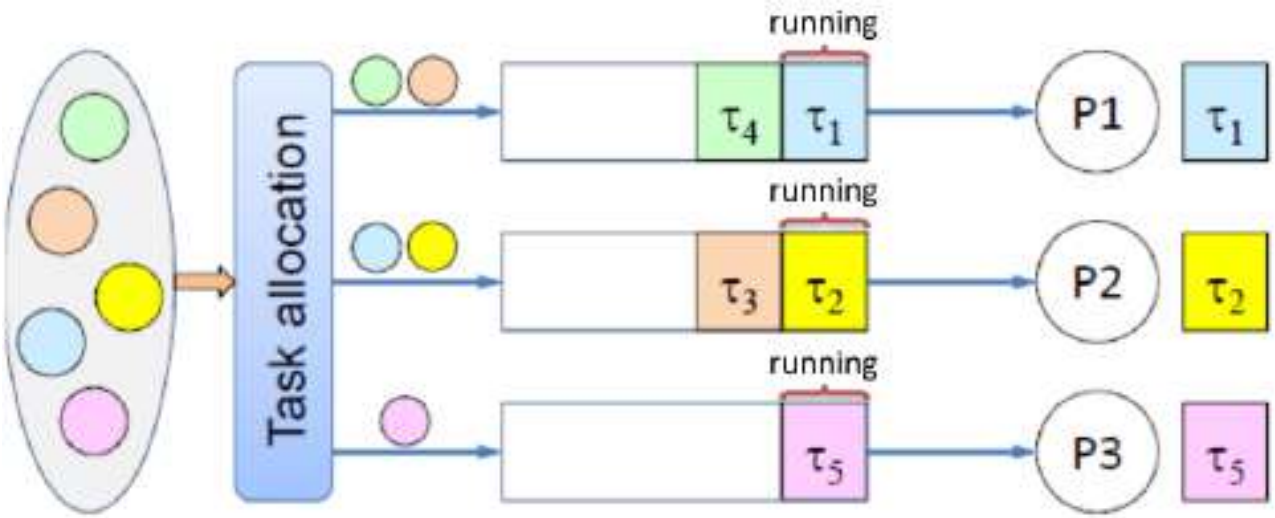
Main approaches:

Partitioned scheduling Tasks are allocated a priori to a given core - individual queues

Global scheduling Tasks are allocated to cores at runtime - single queue

Partitioned scheduling

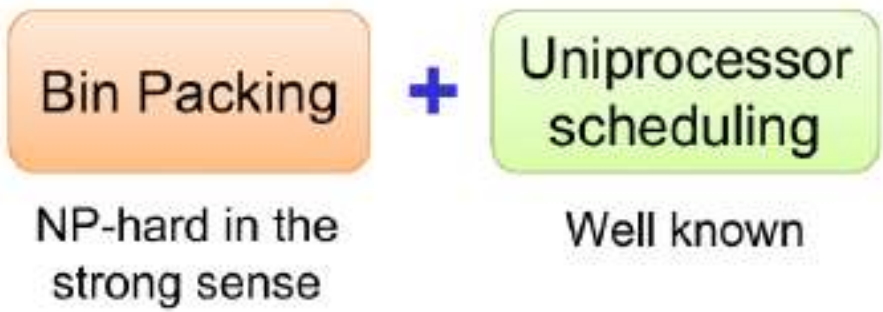
- Each processor has an independent ready queue
- The processor for each task is determined a priori
- Tasks cannot migrate



Partitioned scheduling

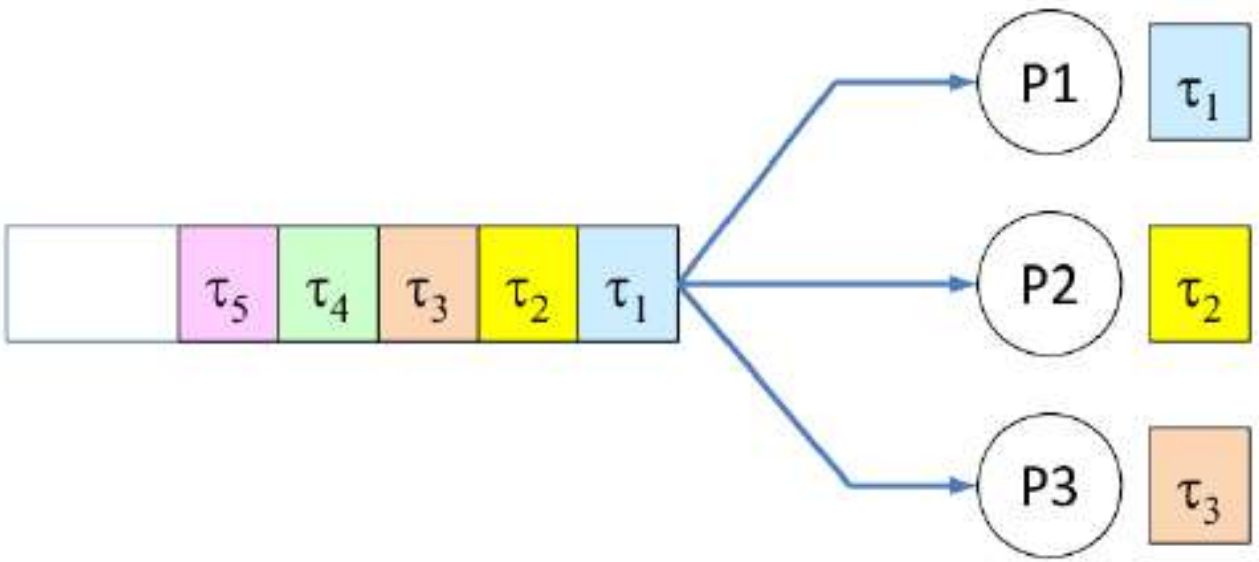
The scheduling problem reduces to:

- Bin-packing problem, for allocating tasks to processors
  - NP-hard in the strong sense.
  - Several heuristics:FirstFit, NextFit, BestFit, etc
- Uniprocessor scheduling problem (already studied and well known)



Global scheduling

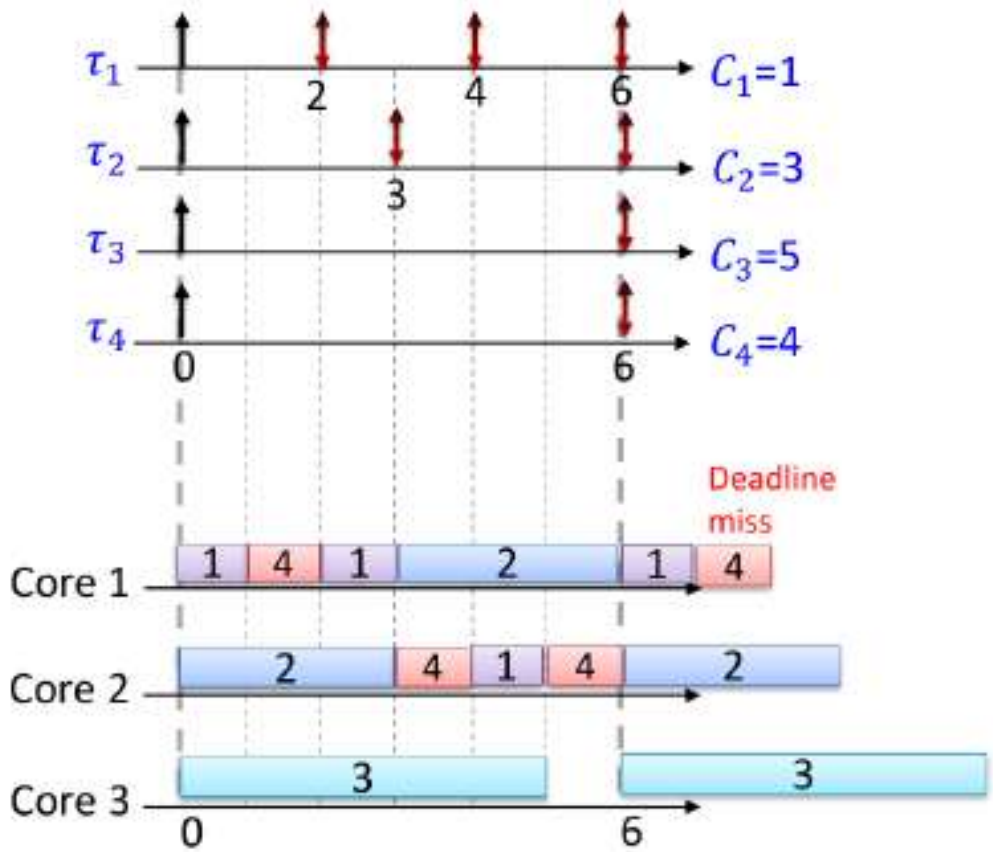
- The system manages a **single queue** of ready tasks
- The processor is determined at run time
- During execution a **task can migrate** to another processor



Global scheduling - Example

- Consider a Global Scheduler with RM policy and the task set below.
- Is the system schedulable?

τ_i	C_i	T_i
1	1	2
2	3	3
3	5	6
4	4	6

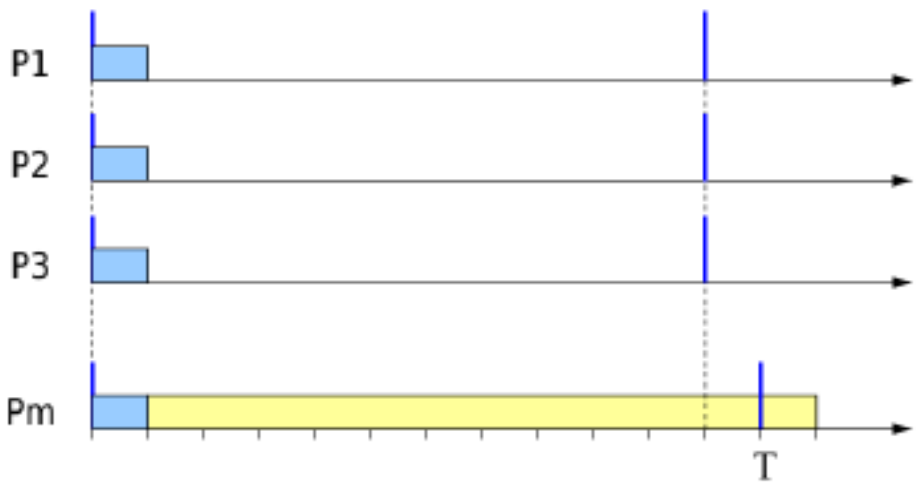


Global Scheduling Evaluation

Dhall's effect: Global RM and Global EDF produce unfeasible schedules for task set utilizations arbitrary close to 1.

- m processors and m+1 tasks

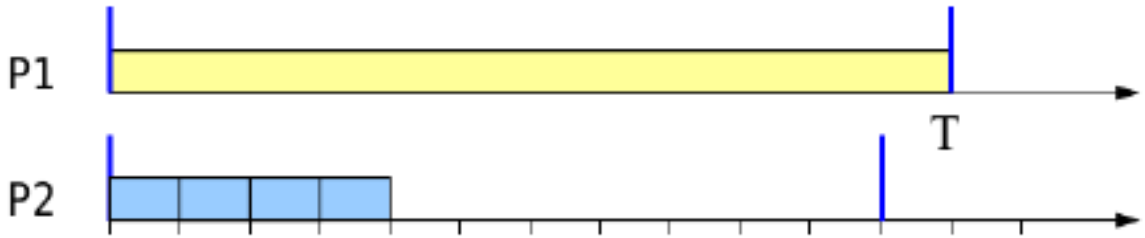
τ_i	C_i	T_i	U_i
τ_1	1	T-1	ϵ
τ_2	1	T-1	ϵ
...	...	...	...
τ_m	1	T-1	ϵ
τ_{m+1}	T	T	1



Global Scheduling Evaluation

But Partitioned Scheduling allows to schedule such system with just 2 processors!

τ_i	C_i	T_i	U_i
τ_1	1	T-1	ϵ
τ_2	1	T-1	ϵ
...	...	...	...
τ_m	1	T-1	ϵ
τ_{m+1}	1	T	1



Note

There are examples of task sets that one of the methods can schedule while the other cannot!

Global vs Partitioned Scheduling

Global

+ Automatic load balancing
+ Lower avg. response time
+ Easier re-scheduling
+ More efficient reclaiming and overload management
+ Lower number of preemptions

- Migration costs
- Inter-core synchronization
- Loss of cache affinity
- Weak scheduling framework

Partitioned

+ Supported by automotive industry (e.g., AUTOSAR)
+ No migrations
+ Isolation between cores
+ Mature scheduling framework
+ Low scheduling overhead (no need to access a global ready queue)

- Cannot exploit unused capacity
- Rescheduling not convenient
- NP-hard allocation problem

Hybrid approaches

- Task/job migration can be costly
  - contex must be moved, impact on cache, etc.
- There are other types of partitioning that consider **restrictions on task migration**

Job migration Tasks are allowed to migrate, but only at jobs boundaries.

Semi-partitioned scheduling Some tasks are statically allocated to processors, others are split into chunks (subtasks) that are allocated to different processors.

Clustered scheduling A task can only migrate within a predefined subset of processors (cluster).

Task allocation

How to partition tasks?

- By information sharing
- By functionality
- Minimizing resources:number of cores, frequency, energy, etc.
- and many others (e.g fault-tolerance)

Note:

These approaches do not aim at schedulability / real-time performance

Task allocation problem

Partitioning problem (bin packing problem):

Given a set of tasks $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$, and a multiprocessor platform with m processors,find an assignment from tasks to processors such that each task is assigned to one and only one processor

Solutions:

- Select a fitness criteria
  - Example: task utilization $U_i = \frac{C_i}{T_i}$ or task density $\alpha_i = \frac{T_i}{D_i}$
- Decide how to sort the tasks (decreasing, increasing, random)
- Decide what is the fitness evaluation method (assignment rejection rule)
- Use a fitting policy to assign tasks to processors:
  - FirstFit, BestFit, WorstFit, NextFit (there are others)

Bin packing heuristics - definitions

The Bin Packing problem

Pack n objects of different size a1, a2, ..., an into the minimum number of bins (containers) of fixed capacity c.

Definitions

M_A number of bins used by an algorithm A

M_0 minimum number of bins used by the optimal algorithm

M_{lb} (Lower bound) Number of bins required for sure by any algorithm

M_{ub} (Upper bound) Number of bins that cannot be exceeded for sure by any algorithm

Bin packing heuristics - definitions

Simple and immediate bounds

Given a set of n items of volume $V = \sum_i^n a_i$

- No algorithm can use less than $M_{lb} = \lceil \frac{V}{c} \rceil$
- No algorithm can use more than $M_{ub} = \lceil \frac{2 \cdot V}{c} \rceil$

- "c" is the capacity of each bin
- Optimality implies clairvoyance for online sequences $\rightarrow$ heuristics

Bin packing heuristics

Partitioning heuristics (some examples)

First fit (FF) Place each item in the first bin that can contain it.

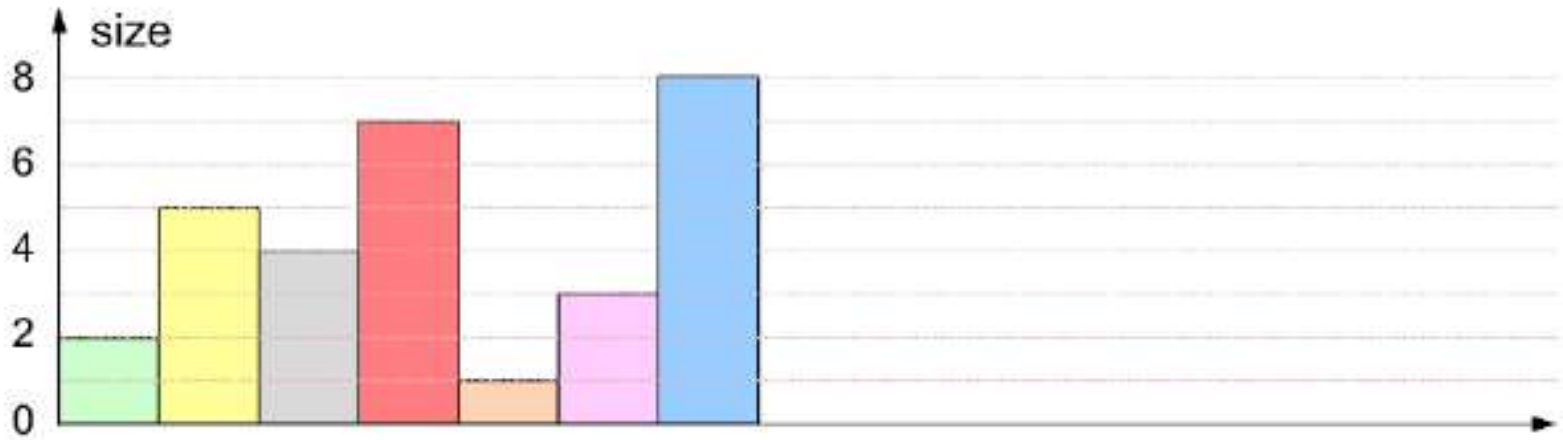
Best fit (BF) Place each item in the bin with the smallest empty space.

Worst fit (WF) Place each item in the used bin with the largest empty space.

Next fit (NF) Place each item in the same bin as the last item.

Bin packing heuristics

First fit : Place each item in the first bin that can contain it.

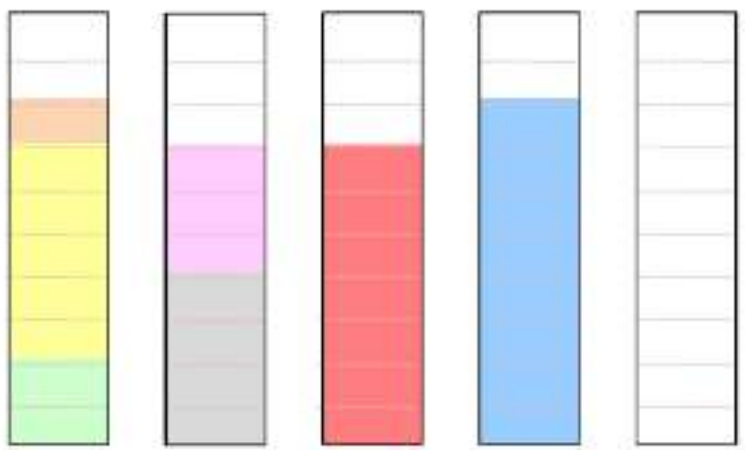


$V = 30$

$c = 10$

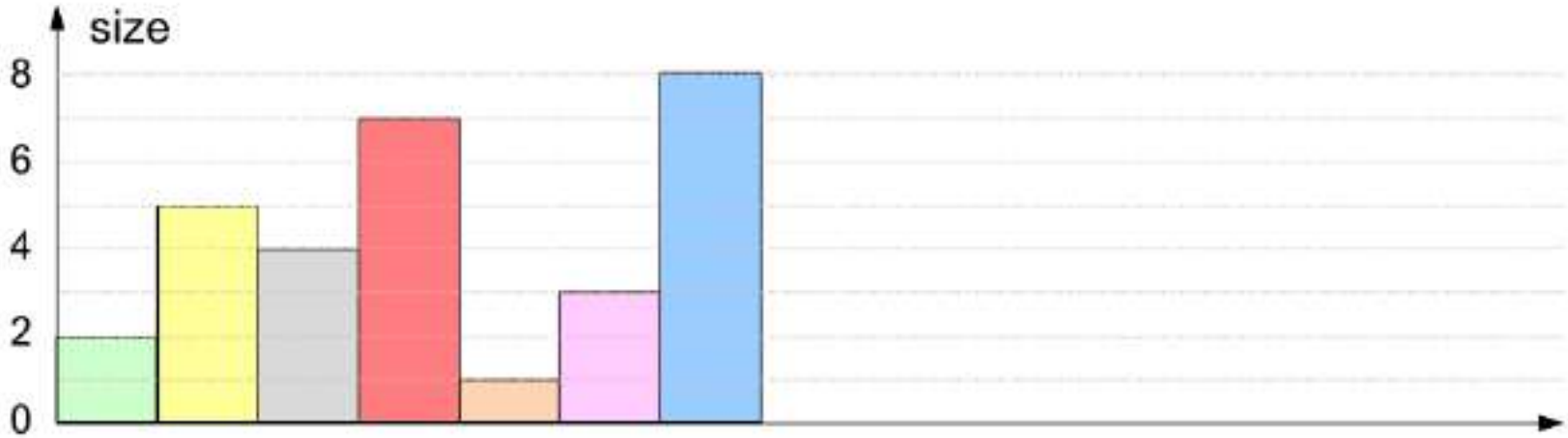
$M_0 = 3$

$M_{FF} = 4$



Bin packing heuristics

Best fit : Place each item in the bin with the smallest empty space.

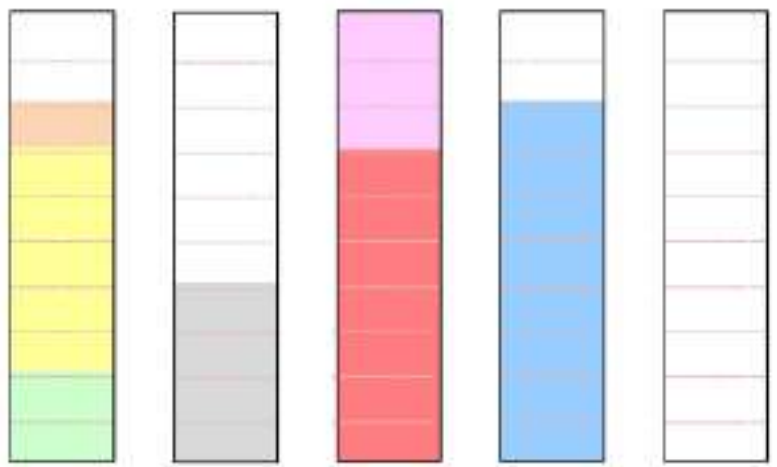


$V = 30$

$c = 10$

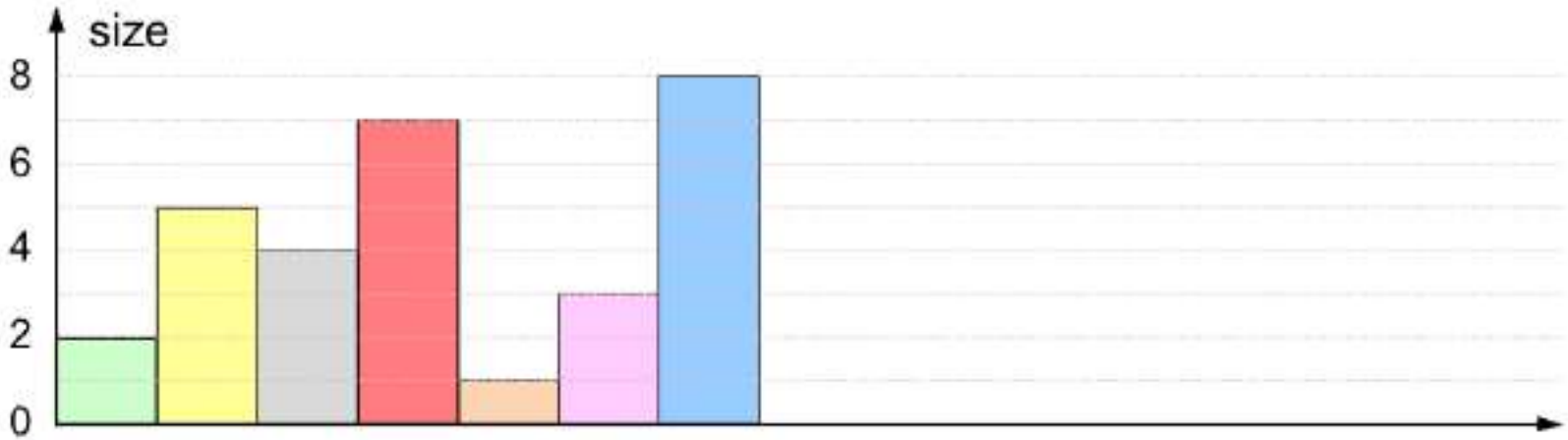
$M_0 = 3$

$M_{BF} = 4$



Bin packing heuristics

Worst fit : Places each item in the used bin with the largest empty space, otherwise start a new bin.

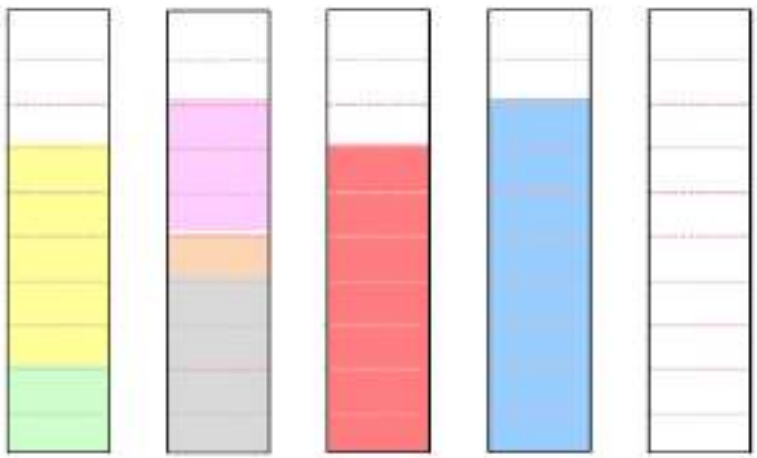


$V = 30$

$c = 10$

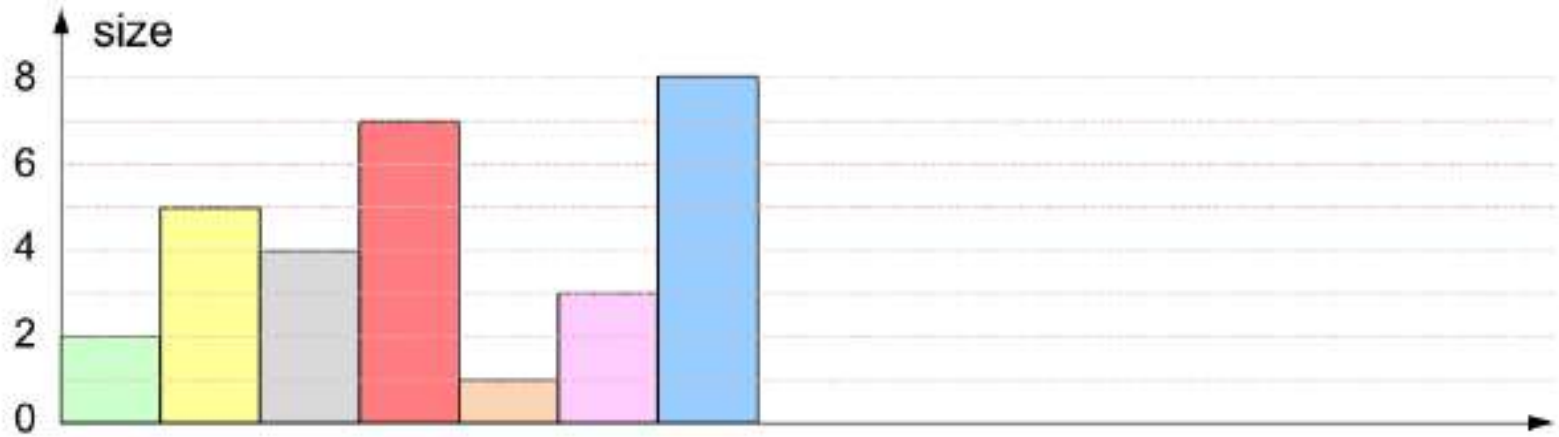
$M_0 = 3$

$M_{WF} = 4$



Bin packing heuristics

Next fit : Place each item in the same bin as the last item. If it does not fit, start a new bin.

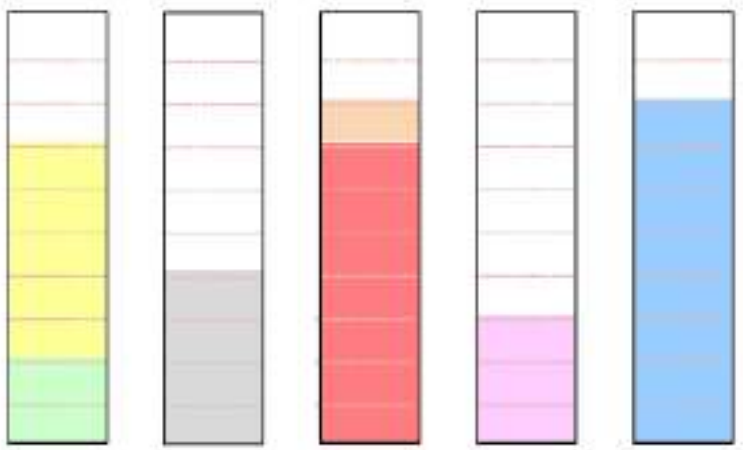


$V = 30$

$c = 10$

$M_0 = 3$

$M_{NF} = 5$



Bin packing heuristics

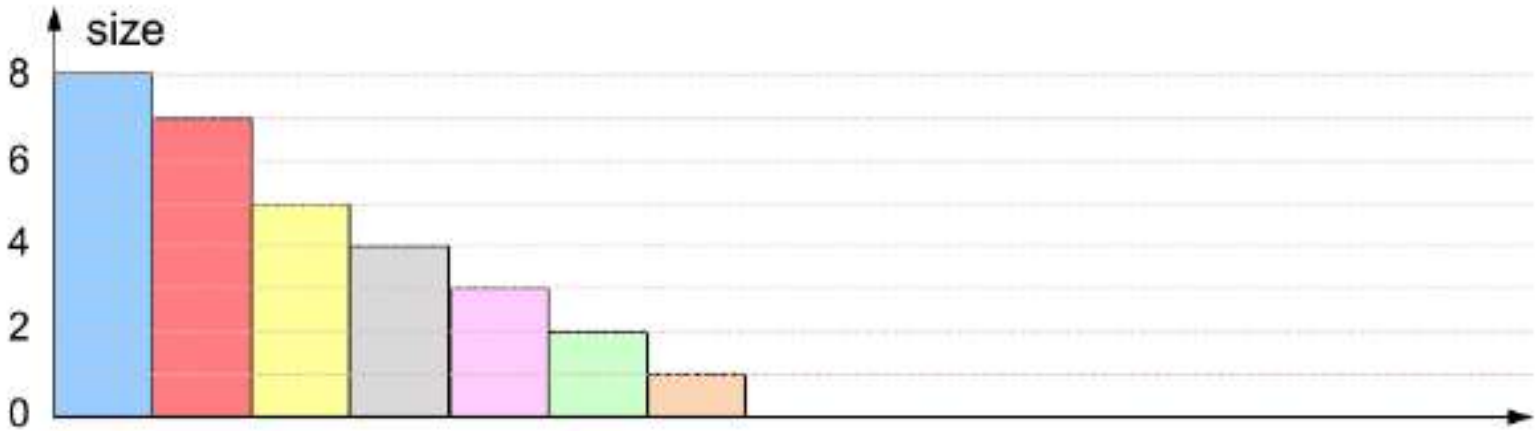
The **performance** of each algorithm strongly **depends on the input sequence** .

However:

- NF has a poor performance since it does not exploit the empty space in the previous bins
- FF improves the performance by exploiting the empty space available in all the used bins.
- BF tends to fill the used bins as much as possible.
- WF tends to balance the load among the used bins.

First Fit Decreasing

In this case sorting tasks by decreasing execution times would allow to minimize the number of bins

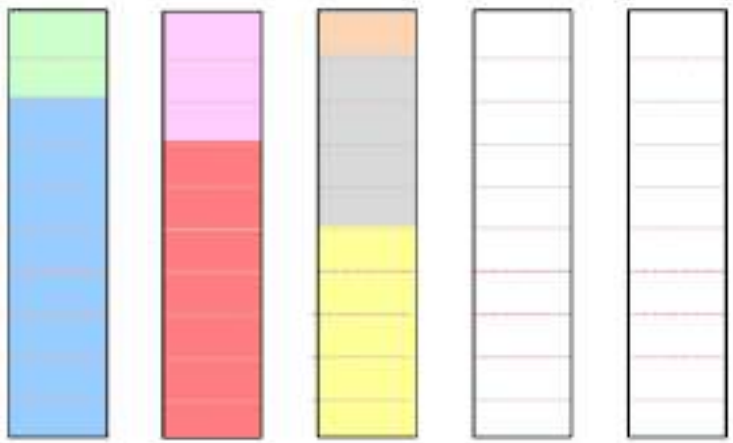


$V = 30$

$c = 10$

$M_0 = 3$

$M_{FFD} = 3$



Some theoretical results

Theoretical results

- Any online algorithm uses at least 4/3 times the optimal number of bins:
  $M_{on} \geq \frac{4}{3} M_o$
- NF and WF never use more than $2 \cdot M0$ bins
- FF and BF never use more than $(1.7 \cdot M0 + 1)$ bins
- FFD never uses more than $(\frac{4}{3} M0 + 1)$ bins
- FFD never uses more than $(\frac{11}{9} M0 + 4)$ bins

Some utilization bounds

- Any task set with total utilization $U \leq \frac{m+1}{2}$ is schedulable in a multiprocessor made up of m processors using FF allocation and EDF scheduling on each processor.(Lopez-Diaz-Garcia)
- Any task set with total utilization $U \leq m(2^{\frac{1}{2}} - 1)$ is schedulable in a multiprocessor made up of m processors using FF allocation and RM scheduling on each processor.(Oh and Baker)
- When each task has utilization $u_i \leq \frac{m}{3m-2}$, the task set is feasible by global RM scheduling if $U \leq \frac{m^2}{3m-2}$. (Andersson, Baruah, Jonsson)

Bibliography

Book

- Multiprocessor Scheduling for Real-Time Systems (Embedded Systems) 2015 Edition, Sanjoy Baruah, Marko Bertogna, Giorgio Buttazzo
  Available at https://www.springer.com/gp/book/9783319086958

Papers

- A survey of hard real-time scheduling for multiprocessor systems,
  By Robert I. Davis and Alan Burns from University of York, U.K.
  Available at https://dl.acm.org/citation.cfm?id=1978814

Slides

- Giorgio Buttazzo
  On multiprocessor systems (part1):
  http://retis.sssup.it/ giorgio/slides/cbsd/mc1-intro-6p.pdf
  On multiprocessor systems (part2):
  http://retis.sssup.it/ giorgio/slides/cbsd/mc2-sched-6p.pdf
  Note: These slides have been largely adapted from Buttazzo's slides