Embedded systems

Bernard Boigelot

E-mail : Bernard.Boigelot@uliege.be
WWW : https://people.montefiore.uliege.be/boigelot/
https://people.montefiore.uliege.be/boigelot/courses/embedded/

References :

- *An Embedded Software Primer*, David E. Simon, Addison-Wesley, 1999.
- *μC/OS-III: The Real-Time Kernel*, Jean J. Labrosse, Micriμm Press, 2010.
- *Real-Time Systems*, Jane W. S. Liu, Prentice Hall, 2000.

Chapter 1

Introduction

Embedded systems

Definition: An embedded system is a computer system used as a component of a more complex entity.

Typical applications:

- multimedia players, radios, televisions, mobile phones, GPS receivers;
- still and video cameras;
- wristwatches, calculators, smart cards, RFID tags, remote controls;
- home appliances;
- computer peripherals;
- measurement equipment, sensors;
- cash dispensers, self-service machines;
- medical devices, implants;

- elevators, intrusion-detection devices, domotic systems;
- telephone switches, network routers;
- automotive systems (ABS, ESP, injection controllers, parking assistance, . . .);
- avionics (fly-by-wire controls, glass cockpits, navigation aids, TCAS, . . .);
- industrial process controllers, robots;
- artificial satellites, spatial probes;
- . . .

Advantages

- Moving some functionalities from hardware to software makes electronic circuits
  - simpler,
  - cheaper to build,
  - more powerful.
- Complex features can be implemented.
- Software components can easily be updated during the lifetime of a product, as well as reused in other projects.

Developing embedded systems: Main difficulties

- Low-performance hardware: Low computing power, small amount of memory ...
- Specificity to a particular application.
- Concurrence: Several tasks operate in parallel.
- Reactivity: The system must constantly be able to answer solicitations.
- Real-time constraints.
- High level of quality expected: Reliability, robustness and efficiency are critical.
- Limited user interface.
- Adverse exploitation environment.
- Energy management is often necessary.

Chapter 2

Hardware

Main components of an embedded system

- One or several processor(s) (CPU):
  - Microcontrollers (MCU): 8051 (Intel), PIC, AVR (Microchip), . . .
  - Digital Signal Processors (DSP): TMS320 (Texas Instruments), SHARC (Analog Devices), MSC81xx (NXP), . . .
  - Microprocessors dedicated to embedded applications: ARM [v7, v8], ColdFire (NXP), ARMADA (Marvell), PowerPC (IBM, NXP, STMicroelectronics), x86, x86-64 (Intel, AMD), . . .
  - Generic microprocessors: Snapdragon (Qualcomm), Xeon, Core, Atom, Celeron (Intel), . . .
  - Special architectures: Java Card, multicore processors, reconfigurable processors, . . .

- Memory:
  - Static or dynamic RAM, ROM (EEPROM, FLASH, . . .).
  - Either internal to the microcontroller, external, or integrated in a System on Chip (SoC).
  - Parallel or serial interface.
  - Possibility of addressing peripherals in memory.
- Internal or external peripherals:
  - Timers,
  - Converters,
  - Communication controllers,
  - . . .
- Communication buses.

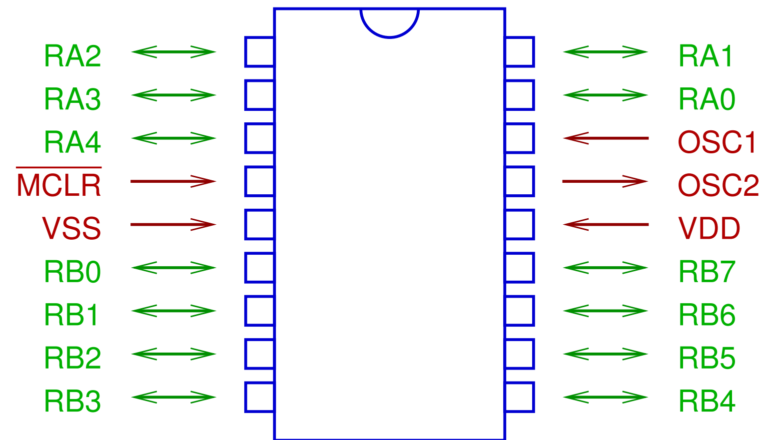
- Interfaces with the circuit environment:
  - Point to point: RS-232, IR, NFC, . . .
  - Buses: I^2C, CAN, SPI, USB, JTAG, . . .
  - Networks: Ethernet, Wi-Fi, ZigBee, Bluetooth,. . .
- Auxiliary components:
  - Power supply,
  - Clock generator,
  - Bus controllers,
  - . . .

Example of embedded microcontroller: Microchip PIC16F716

Main features:

- RISC architecture: Only 35 instructions.
- Harvard memory model: 2048 words of (FLASH) program memory, 128 bytes of (RAM) data memory, both internal.
- Reprogrammable via a serial interface.
- 13 dynamically configurable general-purpose input/output pins.
- Integrated peripherals: 3 timers, PWM controller, analog to digital converter, ...
- Computing power of 5.10^6 instructions per second.
- Low power consumption: About $120\,\mu$A (under 2V) at 1 MHz, $14\,\mu$A at 32 kHz, 100 nA in standby.

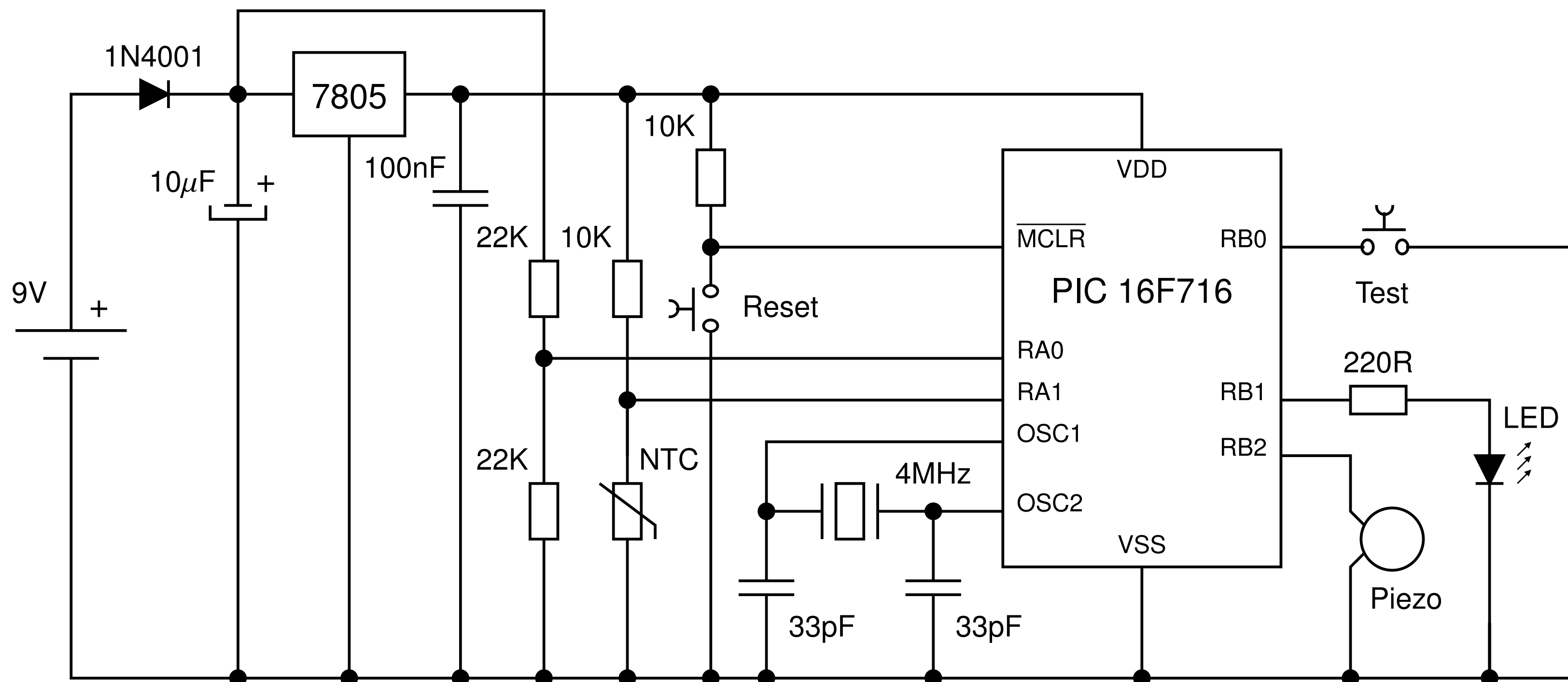
Pinout:



Description:

- VSS, VDD : Power supply (2.0–5.5 V).
- OSC1, OSC2 : Oscillator crystal or external clock source.
- $\overline{\text{MCLR}}$: Operating mode selection (0 V: reset, VDD: program execution, 13 V: programming mode).
- RA0. . . RA4, RB0 . . . RB7 : General-purpose input/output pins (TTL/CMOS), dynamically configurable and multiplexed with some peripherals. RB6 and RB7 alternatively provide a serial interface in programming mode.

Example of application: Temperature alarm



Example of embedded bus

Problem: Managing data transfers between several devices (CPU, memory, sensors, peripherals, . . .) using communication hardware that is as simple as possible.

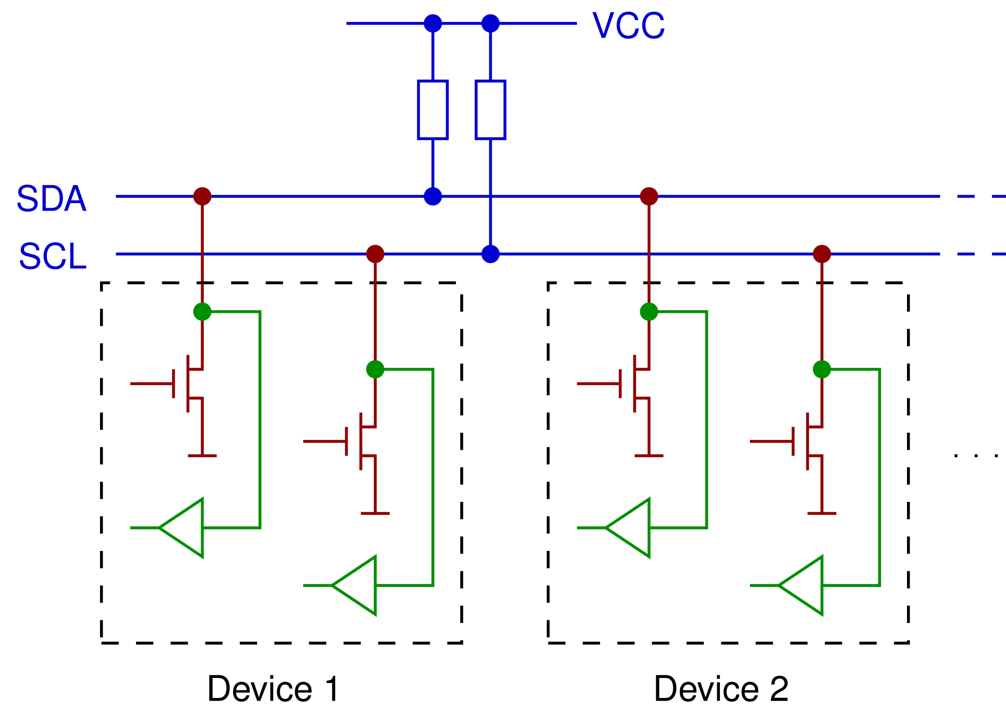
Requirements:

- Bus topology.
- Small number of communication lines.
- Flexible configuration.
- Mechanisms for addressing devices, managing transactions, for performing arbitration and flow control.

Solution: I^{2}C bus

Principles:

- The bus consists of a pair of two-way lines: SDA (Serial DAta) and SCL (Serial Clock).
- The value of each line stays high whenever it is unused.
- Each device connected to the bus can read the value of SDA and SCL, but is only able to force them down, i.e., to write a low value.



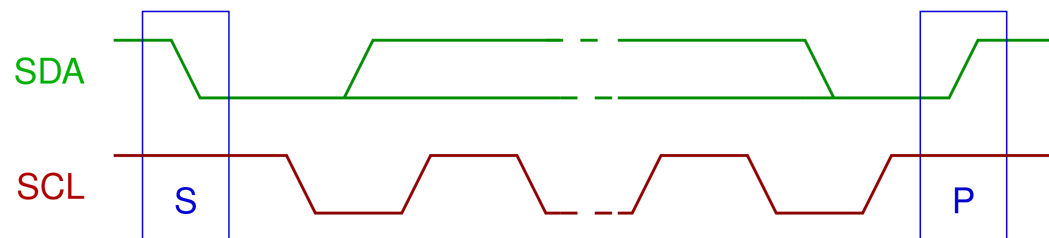
I^2C: Transactions

The master of a transaction is responsible for

- generating a clock signal on SCL during the transaction.
- signaling the beginning (Start, S) and the end (Stop, P) of the transaction. The signals S and P correspond to the two possible transitions of SDA when SCL is high.

When a transaction is in progress (i.e., between S and P), transitions of SDA are only allowed when SCL is low.

Illustration:



I^{2}C: Data transfers

- During a transaction, the sender of data can either be the master or the slave.
- The value of each bit of data sent on the bus corresponds to the value of SDA during a low-to-high transition of SCL.
- Data is exchanged in 8-bit groups, the most significant bit (MSB) being sent first.
- Each group of 8 bits must be followed by an acknowledgment, represented by a low value placed on SDA by the receiver.

  If a group of bits is not acknowledged, then the master immediately aborts the transaction, and the slave stops sending or receiving data.

I^2C: Addressing

When a transaction is initiated, the master has to specify which device is the other participant.

Principles:

- The first 8 bits exchanged in a transaction are always sent by the master.
- The first 7 bits of this group correspond to the address of the intended slave.
- The 8th bit then specifies the direction of the following data transfer:
  - 0 : The master is the sender;
  - 1 : The master is the receiver.

Remark: The first group of 8 bits must thus be acknowledged by the addressed slave, regardless of the data transfer direction.

I^2C: Arbitration

It is possible to have several devices attempting to initiate transactions at the same time, by generating simultaneous Start signals.

For detecting potential conflicts, each master constantly monitors the value of SDA when it sends data. If the observed value differs from the sent one, then the master performing this observation immediately and silently withdraws from the transaction.

Remarks:

- A conflict can only be detected by the device that sends a high value.
- Transmitting simultaneously two exactly identical frames does not lead to a conflict!

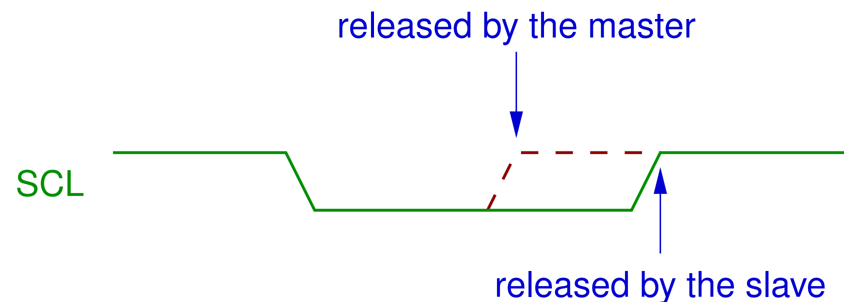
I^2C: Flow control

In some cases, the frequency of the clock signal generated is too high to be followed by the slave.

In such situations, the slave can request the master to permanently or temporarily slow down the clock. This can be done by stretching the low value of SCL until the slave is ready again to send or receive data.

When the master releases SCL while the clock is stretched, it detects that the value of SCL stays low, and pauses its operations until this line is released by the slave.

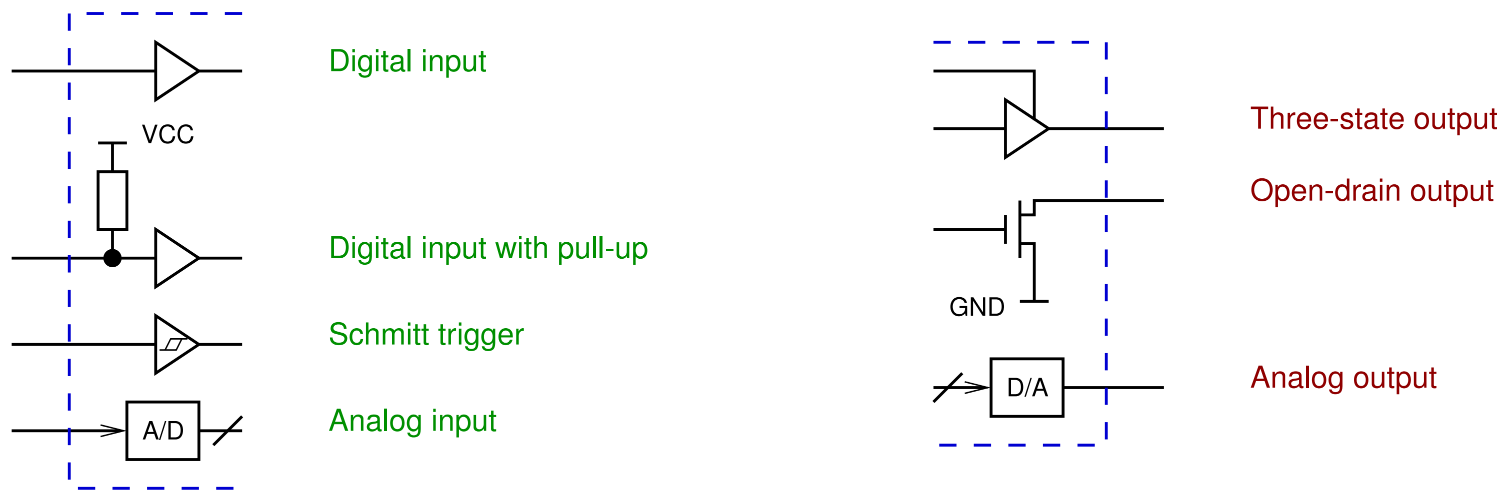
Illustration:



Multiplexed input/output pins

Most microcontrollers allow to dynamically configure input/output pins in software.

Examples of typical configurations:



This feature makes is possible to build simple circuits in which the processor can interact with a large number of peripherals.

Example: Digital multimeter

The problem is to interface a microcontroller offering only 12 dynamically configurable input/output pins with:

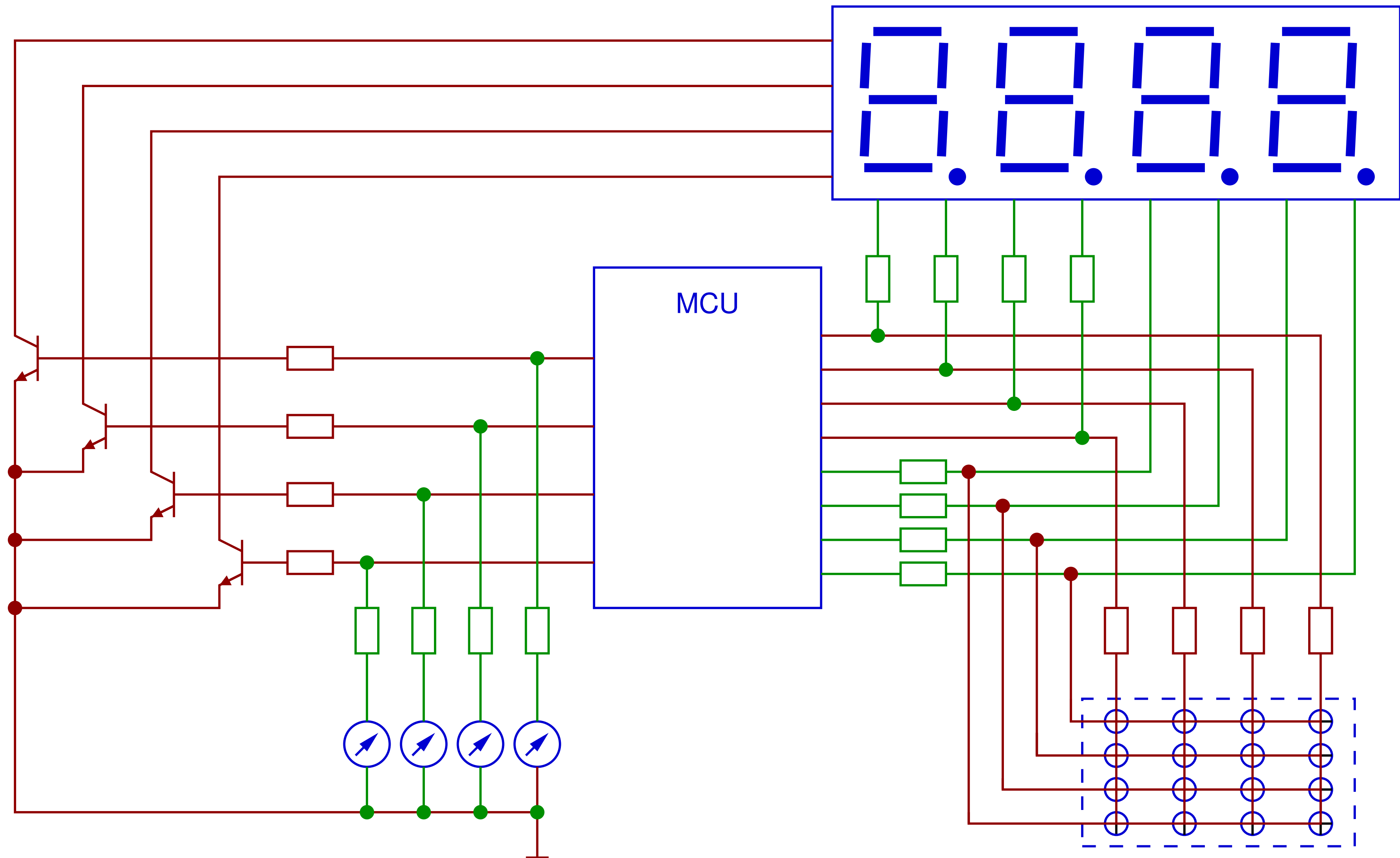
- a screen composed of four 7-segment displays,
- a keyboard organized as a 4×4 matrix,
- 4 analog input channels.

(*Source: Microchip application note AN557*)

Solution:

- The screen and the keyboard are scanned: At a given time, one can only display a single digit, or read a single column of keys.
- An additional phase is inserted for reading input channels.
- 4 pins are associated to both an input channel and a screen digit. They are alternatively configured as analog inputs (when reading channels) and digital outputs (when displaying a digit or reading the keyboard).
- The 8 remaining pins drive the screen segments during display and channel reading phases (8 digital outputs), and are also able to the scan the keyboard (4 digital outputs + 4 digital inputs with pull-up).

Schematics:



Chapter 3

Interrupts

Introduction

An interrupt is a signal that requests the processor to temporarily suspend program execution, in order to execute an interrupt routine.

Advantages:

- A very short response time to solicitations is achievable.
- Urgent operations can be programmed independently from the main code.

Interrupts can be triggered either by an exterior component:

- Interrupt ReQuest (IRQ), received from dedicated input pins,
- change of logic value at digital input pins,

or by the processor itself:

- timer expiration,
- arithmetic or instruction exception,
- software interrupt request,
- . . .

The interrupt mechanism

Upon receiving and accepting to service an interrupt request, the processor performs the following operations:

1. The execution of the current instruction terminates.
2. A pointer to the next instruction to be executed is stored on the runtime stack.
3. The address of the interrupt routine is read from the appropriate interrupt vector (according to the source of the interrupt request).
4. The interrupt routine is executed.
5. At the end of the interrupt routine, the processor resumes program execution, at the address retrieved from the stack.

Interrupt control

Some critical operations can never be interrupted. It is then necessary to temporarily disable interrupts prior to their execution, and to enable them again afterwards.

Some processors allow to assign specific priorities to interrupts originating from different sources. Such architectures generally provide a mechanism for disabling the interrupts having a priority less than some specified threshold. Interrupt priorities are also used for resolving simultaneous interrupt requests.

Enabling and disabling interrupts is performed by executing specific instructions, or by setting the value of dedicated registers.

Notes:

- At power-on, interrupts are disabled by default, in order to allow correct initialization of the program.

- When an interrupt is triggered, some processors automatically disable all interrupts of less or equal priority. They have to be explicitly reenabled in the interrupt routine if needed.

- When an interrupt request is received, the processor sets interrupt flags, in order to trigger the interrupt as soon as it becomes enabled. Interrupt flags have to be cleared explicitly by the interrupt routine.

- Some architectures provide an interrupt source that cannot be disabled (Non Maskable Interrupt, NMI). Its usage is limited to exceptional situations (e.g., backing up critical data upon detecting an imminent power failure).

Saving and restoring context

The correct operation of a program must not be influenced by interrupts triggered during its execution.

It is thus mandatory for interrupt routines to leave the processor state unchanged: values of registers and flags, interface configuration, status of peripherals, . . . , must not be modified.

This is achieved by saving the context at the beginning of interrupt routines, and restoring it at the end.

Notes:

- The context is either saved on the execution stack or in a specific memory area.
- Some processors automatically save the context (either totally or in part) when an interrupt is triggered.

- Context save and restore operations can sometimes be simplified by using dedicated instructions.

- The processors that automatically disable interrupts when branching to an interrupt routine enable them again as a side effect of context restoration.

Programming interrupts

The compilers aimed at embedded applications provide language extension mechanisms for programming interrupts without going down to assembly language.

- Some functions can be designated as being interrupt routines (e.g., `interrupt` keyword, or `#pragma interrupt` compilation directive in C).

  With some compilers, such mechanisms automatically insert context save and restore instructions to interrupt routines, and take care of setting interrupt vectors.

- Enabling and disabling interrupts is performed with the help of macros or specific compilation directives (i.e., `enable()`/`disable()`, `critical` keyword).

- It is sometimes necessary to inform the compiler than the value of a variable can be modified by interrupt routines, in order to prevent incorrect optimizations (e.g., `volatile` keyword in C).

Communicating with interrupt routines

Interrupt requests are by nature unpredictable. This complicates data exchange operations between interrupt routines and the main code.

Example: Industrial controller. The alarm must sound if two temperature measurements made by an interrupt routine differ.

Wrong solution:

```
static volatile int temp[2];

interrupt void measure(void)
{
  temp[0] = !! first measurement;
  temp[1] = !! second measurement;
}

void controller(void)
{
  int temp0, temp1;
  for (;;)
    {
      temp0 = temp[0];
      temp1 = temp[1];
      if (temp0 != temp1) !! sound the alarm;
    }
}
```

Notes:

- Carrying out the comparison between the two measurements in a single C instruction does not solve the problem:

```
...
void controller(void)
{
  for (;;)
    if (temp[0] != temp[1]) !! sound the alarm;
}
...
```

  (Indeed, such an instruction is generally compiled into several machine instructions.)

- Even in programs written in assembly language, it is possible for the execution of individual instructions to be interrupted before their completion.

  This only happens with specific instructions, often repeatedly performing a simpler operation (e.g., block copy instructions).

- This type of bug can be very difficult to detect and to reproduce!

Correct solution:

The instructions that read the measurements sent by the interrupt routine to the controller form a critical section, the execution of which cannot be interrupted.

```
static volatile int temp[2];

interrupt void measure(void)
{
  temp[0] = !! first measurement;
  temp[1] = !! second measurement;
}

void controller(void)
{
  int temp0, temp1;
  for (;;)
    {
      disable();   /* Disable interrupts */
      temp0 = temp[0];
      temp1 = temp[1];
      enable();    /* Reenable interrupts */

      if (temp0 != temp1) !! sound the alarm;
    }
}
```

Other solution:

```
static volatile int temp_a[2], temp_b[2];
static int controller_uses_b = 0;

interrupt void measure(void)
{
  if (controller_uses_b)
    {
      temp_a[0] = !! first measurement;
      temp_a[1] = !! second measurement;
    }
  else
    {
      temp_b[0] = !! first measurement;
      temp_b[1] = !! second measurement;
    }
}

void controller(void)
{
  for (;; controller_uses_b = !controller_uses_b)
    if (controller_uses_b)
      {
        if (temp_b[0] != temp_b[1]) !! sound the alarm;
      }
    else
      if (temp_a[0] != temp_a[1]) !! sound the alarm;
}
```

Notes:

- This solution does not require to disable interrupts.
- The main code must sometimes perform one useless iteration before sounding the alarm.

Improved solution:

```
#define MAX_FIFO 10     /* Must be even ! */
static volatile int temp_fifo[MAX_FIFO];
static volatile int first = 0;
static int last = 0;

interrupt void measure(void)
{
  /* If the buffer is not saturated */
  if (!((first + 2 == last)
      || (first == MAX_FIFO - 2 && last == 0)))
    {
      temp_fifo[first] = !! first measurement;
      temp_fifo[first + 1] = !! second measurement;
      first += 2;
      if (first == MAX_FIFO)
        first = 0;
    }
  else   !! discard measurements;
}

void controller(void)
{
  int temp0, temp1;

  for (;;)
    if (first != last)    /* If the buffer is not empty */
      {
        temp0 = temp_fifo[last];
        temp1 = temp_fifo[last + 1];
        last += 2;
        if (last == MAX_FIFO)
          last = 0;
        if (temp0 != temp1) !! sound the alarm;
      }
}
```

Note: For this solution to be correct, it is necessary that the instruction `last += 2` executes atomically.

This kind of solution is thus very sensitive to implementation details!

In practice, disabling interrupts during communications with interrupt routines is acceptable in most situations. The more complex solutions are used only when disabling interrupts is impossible or forbidden.

Interrupt latency

The delay between an interrupt request I and the end of execution of urgent operations in an interrupt routine R_I is called the response time, or latency of the interrupt.

This latency is influenced by four parameters:

1. The longest interval during which interrupts of priority larger or equal to I are disabled.
2. The time needed for executing the interrupt routines with a higher priority than R_I.
3. The maximum delay between an interrupt trigger and the branch to the corresponding interrupt routine.
4. The time spent in R_I before having executed the urgent operations.

A good strategy is therefore to

- disable interrupts for the shortest possible time (parameter 1);
- make the interrupt routines quick and efficient (parameters 2 and 4).

Parameter 3 is a feature of the processor, and cannot be influenced by the programmer.

Example

- A system implements the following interrupt routines, sharing the same priority.

Name	Description	Execution time	Period
I_1	Temperature measurement	100 μs	500 μs
I_2	Timer expiration	200 μs	1000 μs
I_3	Network I/0	300 μs	> 1000 μs

- The main program disables interrupts during resp. 200 μs and 250 μs for exchanging data with I_1 and I_2.
- The time needed for triggering I_3 and executing the corresponding urgent operations is equal to 100 μs.

Question: Is the latency of I_3 smaller than 1000 μs ?

Answer:

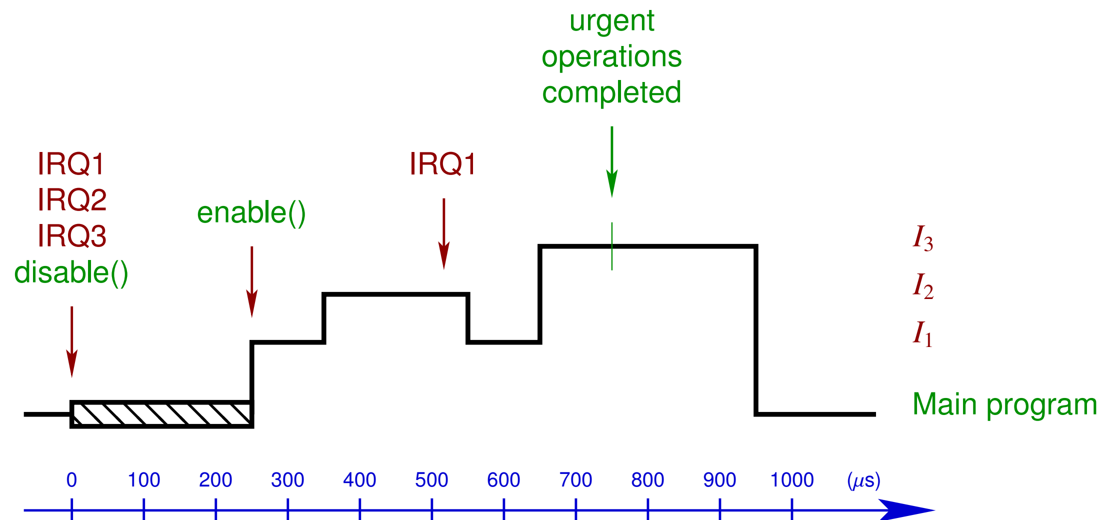
It is sufficient to study the system during an interval of length equal to $1000\ \mu$s. The highest possible latency is obtained with the following delays:

- Interrupts disable time : 250 μs.
- Executing I_1 : $2 \times 100\ \mu$s.
- Executing I_2 : $200\ \mu$s.
- Triggering and executing the urgent operations of I_3 : $100\ \mu$s.
- $\rightarrow$ Total: 750 μs.

Notes:

- Only the largest interval in which interrupts are disabled has to be taken into account!

- Example of scenario in which the maximum latency is reached:



- The execution of I_3 always terminates before $1000\ \mu$s.

Chapter 4

Software architectures

The round-robin architecture

Principles:

- Interrupts are not used.
- Tasks are invoked in turn, and run until their completion.

Illustration:

```
void main(void)
{
   for (;;)
     {
       if (!! task 1 is ready)
         {
           !! operations of task 1;
         }
       if (!! task 2 is ready)
         {
           !! operations of task 2;
         }

       ⋮

       if (!! task n is ready)
         {
           !! operations of task n;
         }
     }
}
```

Advantages:

- Simple solution, but sufficient for some applications.
- Exchanging data between tasks is easy.

Drawbacks:

- The worst-case latency of an external request is equal to the execution time of the entire main loop.
- Implementing additional features can adversely affect the correctness of a system, by increasing latencies beyond acceptable bounds.

Example (multimeter):

```
#include "types.h"
#include "multimeter.h"

static UINT1 phase = 0; /* 0–3: display, 4: keyboard, 5: channels */
static UINT1 display_content[4];
static SINT4 measures[4];

static keyboard_state   keys;
static multimeter_state parameters;

void main(void)
{
  !! initialize global data;

  for (;;)
    {
      switch (phase)
        {
          case 4:
            handle_keyboard();
            if (keys.new_keypress)
              {
                keypress_action();
                keys.new_keypress= 0;
              }
            break;

          case 5:
            handle_channels();
            update_display_content();
            break;

          default:
            handle_display();
        }
      if (++phase > 5)
        phase = 0;
    }
}
```

```
void handle_display(void)

{
  UINT1 digit, segments;

  !! PORTA: 4 digital outputs;
  !! PORTB: 8 digital outputs;

  digit    = !! compute the digit to be displayed, from the
             !! values of display_content and phase;
  segments = !! pattern corresponding to digit;

  out(PORTA, 1 << phase);
  out(PORTB, segments);

  delay(DISPLAY_DELAY);
}

void handle_channels()
{
  !! PORTA: 4 analog inputs;
  !! PORTB: 8 digital outputs;

  out(PORTB, 0);
  delay(CHANNELS_DELAY);

  !! read PORTA, and place the result in measures;
}

void handle_keyboard()
{
  static UINT1 column = 0;
  UINT1 row;

  !! PORTA: 4 digital outputs;
  !! PORTB: 4 digital outputs (low nibble),
  !!        4 digital inputs with pull-ups (high nibble);
```

```
    out(PORTA, 0);
    out(PORTB, 1 << column);
    row = in(PORTB) >> 4;

    !! update keys according to the content of row;

    if (++column >= 4)
      column = 0;
}

void keypress_action()
{
    !! update parameters according to the key that has
    !! been pressed (specified in keys);
}

void update_display_content()
{
    !! update display_content according to the values in
    !! measures and parameters;
}
```

Notes: The parameters `DISPLAY_DELAY` and `CHANNELS_DELAY` must be chosen

- large enough to ensure an accurate conversion of analog samples, and a good illumination of display segments.
- small enough to avoid display flickering, as well as missing key presses.

The round-robin with interrupts architecture

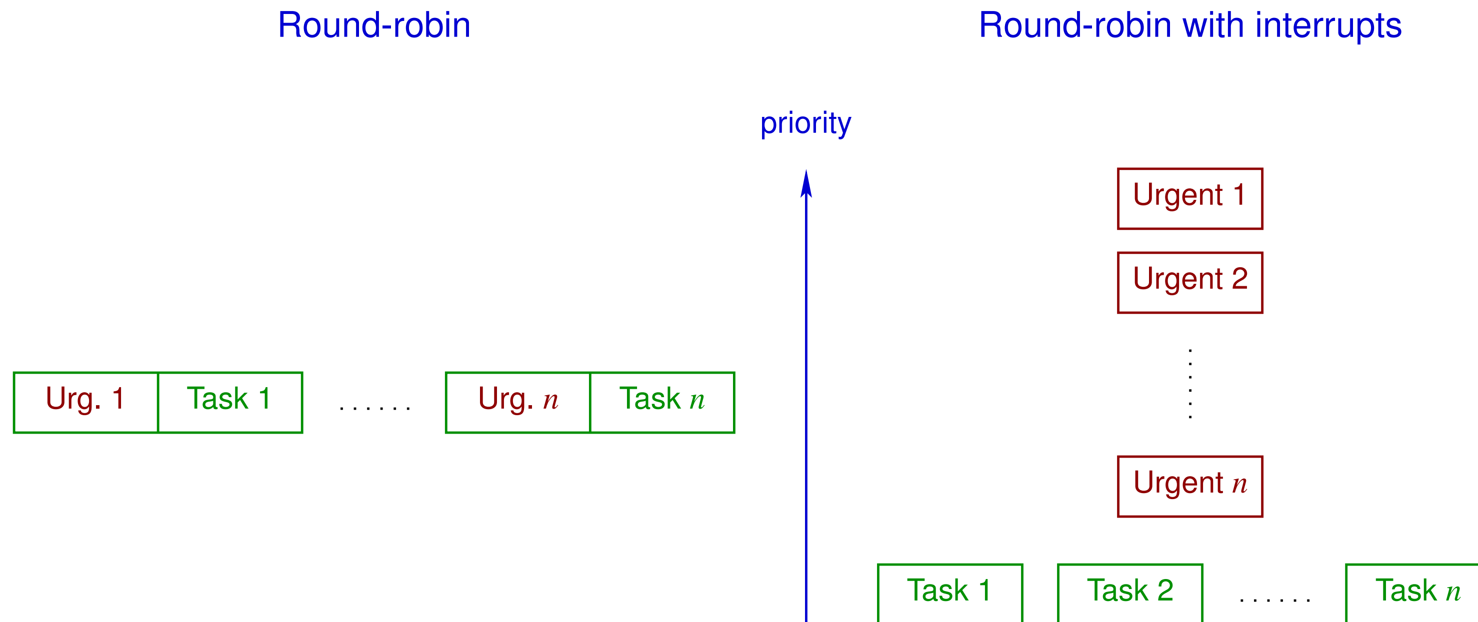
Principles: Tasks are invoked in round-robin fashion, but interrupt routines take care of urgent operations.

Illustration:

```
volatile BOOL ready1 = 0, ready2 = 0, ...,
                 readyn = 0;

interrupt void urgent1(void)
{
   !! urgent operations of task 1;
   ready1 = 1;
}

interrupt void urgent2(void)
{
   !! urgent operations of task 2;
   ready2 = 1;
}

⋮

interrupt void urgentn(void)
{
   !! urgent operations of task n;
   readyn = 1;
}
```

```
void main(void)
{
   for (;;)
      {
         if (ready1)
            {
               !! non-urgent operations of task 1;
               ready1 = 0;
            }
         if (ready2)
            {
               !! non-urgent operations of task 2;
               ready2 = 0;
            }

         ⋮

         if (readyn)
            {
               !! non-urgent operations of task n;
               readyn = 0;
            }
      }
}
```

Advantage: The urgent operations take priority over the non-urgent ones.



Drawbacks:

- The non-urgent tasks share the same effective priority. This yields high latencies when at least one task has a large execution time (e.g., raster generation in laser printers).

> Important note: Moving non-urgent operations from tasks to interrupt routines is not a good solution!

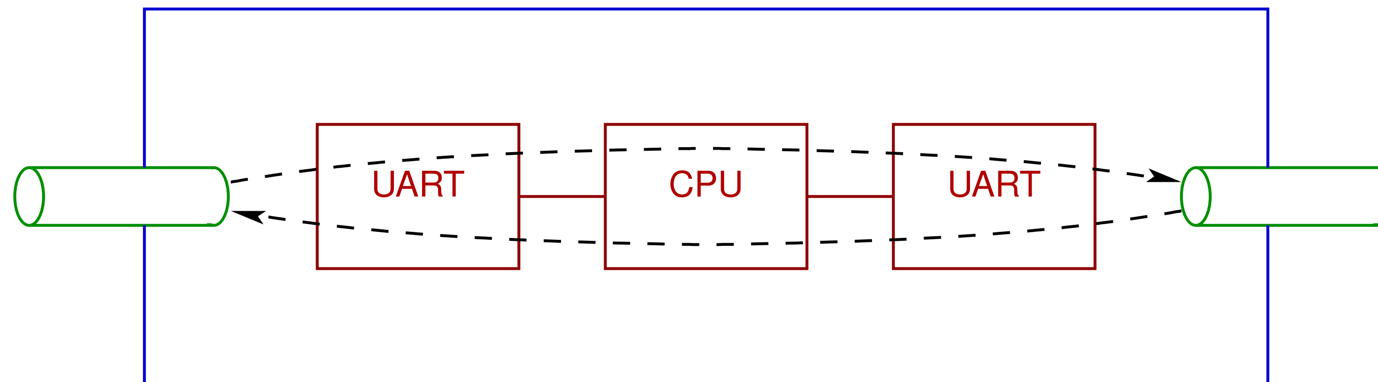
Indeed,

- performing non-urgent operations in an interrupt routine increases the latency of interrupts with a lower or equal priority;
- interrupts do not offer flexible synchronization mechanisms.

- Data exchange operations between interrupt routines and tasks have to be correctly implemented (cf. Chapter 3).

Example: Serial filter

The goal is to develop a two-way filter connecting two serial lines.



Principles:

- Incoming bytes are signaled by interrupt requests, which must be answered as soon as possible (before the next received byte).
- When a UART is ready to send a byte on its output line, it requests an interrupt. The processor is then free to wait for an arbitrarily long time before providing this byte.

Solution:

```
#include "types.h"
#include "fifo.h"
#include "filter.h"

static volatile BOOL uart1_ready, uart2_ready;
static volatile fifo rx1, tx1,
                     rx2, tx2;

interrupt void uart1_rx(void)
{
  char byte;

  byte = !! reception from UART1;
  fifo_put(rx1, byte);
}

interrupt void uart2_rx(void)
{
  char byte;

  byte = !! reception from UART2;
  fifo_put(rx2, byte);
}

interrupt void uart1_ready_to_send(void)
{
  uart1_ready = 1;
}

interrupt void uart2_ready_to_send(void)
{
  uart2_ready = 1;
}
```

```
void main(void)
{
  !! initialize global data;
  !! initialize interrupt vectors;

  enable();

  for (;;)
    {
      if (fifo_content_size(rx1) >= FILTER_THRESHOLD)
        {
          !! remove data from rx1;
          !! filter;
          !! add the result to tx2;
        }

      if (fifo_content_size(rx2) >= FILTER_THRESHOLD)
        {
          !! remove data from rx2;
          !! filter;
          !! add the result to tx1;
        }

     if (uart1_ready && !fifo_is_empty(tx1))
        {
          char byte;

          byte = fifo_get(tx1);
          disable();
          !! send byte to UART1;
          uart1_ready = 0;
          enable();
        }
```

```
        if (uart2_ready && !fifo_is_empty(tx2))
        {
            char byte;

            byte = fifo_get(tx2);
            disable();
            !! send byte to UART2;
            uart2_ready = 0;
            enable();
        }
    }
}
```

Notes:

- Attempting to add data to a saturated FIFO buffer cannot be a blocking operation (i.e., it must instead discard data).

- The functions for handling FIFO buffers must execute correctly both in the interrupt routines and in the main code.

  Example of implementation:

```
void fifo_put(fifo q, char c)
{
  BOOL intr_enabled;

  ...

  intr_enabled = disable();
  !! critical section;
  if (intr_enabled)
    enable();

  ...
}
```

The waiting-queue architecture

Principles:

- In the same way as the round-robin with interrupts architecture, the operations are partitioned into urgent and non-urgent tasks.
- Interrupt routines perform urgent operations, and then place in a waiting queue requests for executing non-urgent tasks.
- The main program retrieves execution requests from the queue and calls the corresponding functions. These requests are not necessarily processed in FIFO order. (For instance, different selection priorities can be assigned to non-urgent tasks.)

Illustration:

```
#include "queue.h"

static volatile queue waiting_queue;

interrupt void urgent1(void)
{
   !! urgent operations of task 1;
   !! add task1 to waiting_queue;
}

interrupt void urgent2(void)
{
   !! urgent operations of task 2;
   !! add task2 to waiting_queue;
}

⋮

interrupt void urgentn(void)
{
   !! urgent operations of task n;
   !! add taskn to waiting_queue;
}
```

```
void main(void)
{
   !! initialize waiting_queue with an empty content;

   for (;;)
     {
       while (!queue_is_empty(waiting_queue))
         {
           !! extract a function from waiting_queue;
           !! execute this function;
         }
     }
}

void task1(void)
{
   !! non-urgent operations of task 1;
}

void task2(void)
{
   !! non-urgent operations of task 2;
}

⋮

void taskn(void)
{
   !! non-urgent operations of task n;
}
```

Advantage: The latency of a non-urgent high-priority task can become smaller than the execution time of all the non-urgent operations.

Drawbacks:

- The maximum latency of a non-urgent task is still at least as large as the execution time of the slowest task.
- Implementing the waiting-queue data structure can be tricky.

Example of application: A system monitors an industrial process by receiving data from an array of sensors, processing this data, and displaying summarized results.

With the queue architecture, it is possible to ensure that the values produced by critical sensors are always taken into account, even in the case of data saturation caused by a malfunctioning low-priority sensor.

The real-time operating system architecture

Principles:

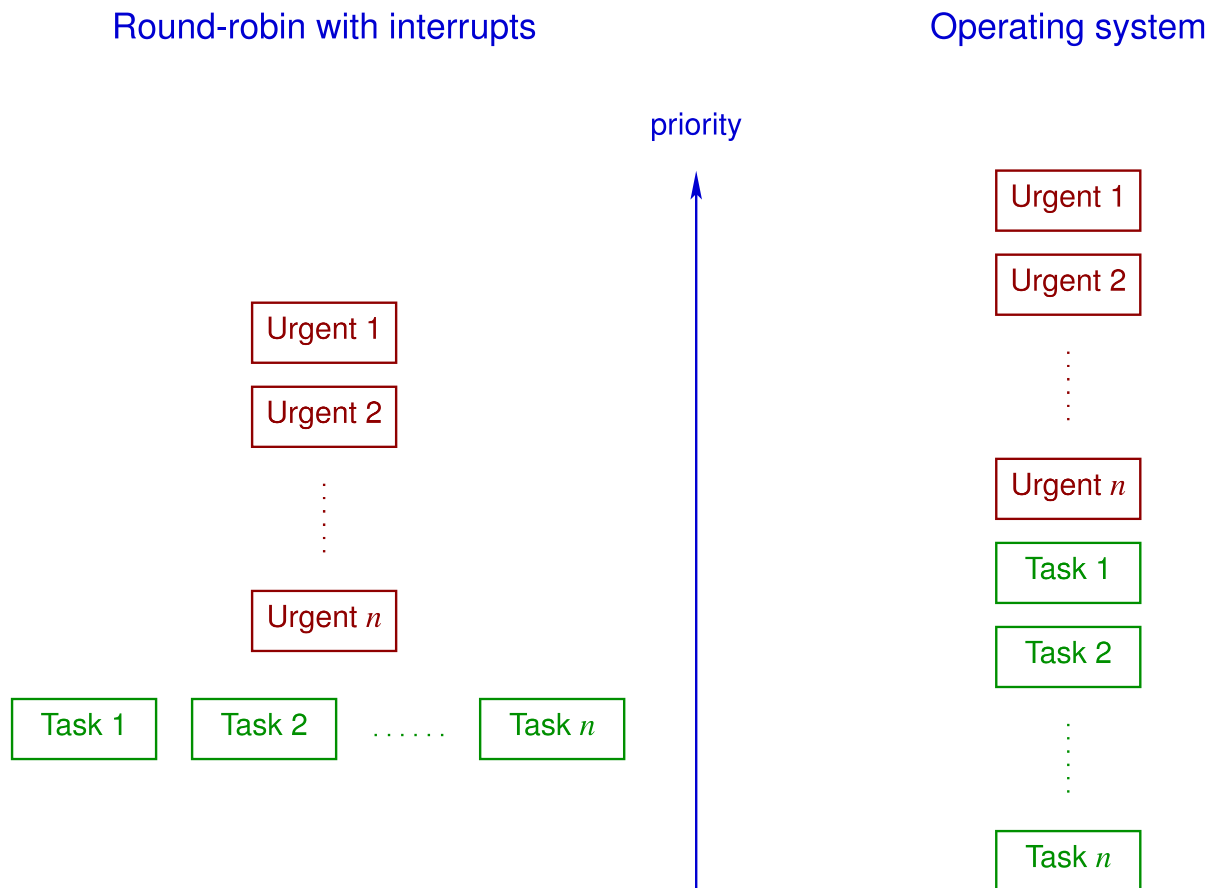
- Urgent operations are performed by interrupt routines. Those are able to signal to other tasks that non-urgent operations are ready to be carried out.
- The non-urgent tasks are invoked dynamically rather than in a predefined order. The responsibility of calling tasks is assigned to the operating system, implemented as an additional software component.
- The operating system is able to suspend the execution of a task before its completion, in order to transfer the processor to another task.
- The signals exchanged between tasks are handled by the operating system, instead of being implemented with shared variables.

Illustration:

```
#include "signal.h"

interrupt void urgent1(void)
{
   !! urgent operations of task 1;
   !! send signal 1;
}

interrupt void urgent2(void)
{
   !! urgent operations of task 2;
   !! send signal 2;
}

⋮

void task1(void)
{
   !! wait for signal 1;
   !! non-urgent operations of task 1;
}

void task2(void)
{
   !! wait for signal 2;
   !! non-urgent operations of task 2;
}

⋮

void main(void)
{
   !! initialize the operating system;
   !! create and enable tasks;
   !! start task sequencing;
}
```

Advantages:

- One can easily combine low-latency operations together with long computations.



- The system is efficient: When a non-urgent task is waiting for a signal, the processor remains available for other computations.
- The structure of the system is robust: New features can easily be added without affecting the latencies of urgent operations or of high-priority tasks.
- Operating systems tailored to embedded applications are commercially available.

Drawbacks:

- The system is complex (but this complexity is mainly located in the operating system, which can be reused over many projects).
- Data exchange operations have to be coordinated between a task and an interrupt routine, but also between tasks.
- The operating system consumes some amount of system resources (a typical figure is 2 to 4 % of the instructions executed by the processor).

Summary

Task priorities and latencies:

Architecture	Available priorities	Maximum latency
round-robin	none	total execution time of all tasks
round-robin with interrupts	interrupt routines; all tasks share the same priority	total execution time of all tasks + interrupt routines
waiting queue	interrupt routines, then tasks	execution time of the longest task + interrupt routines
operating system	interrupt routines, then tasks	execution time of interrupt routines

Robustness and simplicity:

Architecture	Robustness against modifications	Complexity
round-robin	poor	very simple
round-robin with interrupts	good for interrupt routines, poor for the tasks	must handle data exchanges between tasks and interrupt routines
waiting queue	fair	must handle data exchanges, and implement the waiting queue
operating system	very good	quite complex

Chapter 5

Real-time operating systems

Introduction

An operating system (OS) is a software component responsible for coordinating the concurrent execution of several tasks, by

- managing the system resources (processor, memory, access to peripherals, . . .);
- providing services (communication, synchronization, . . .).

An OS is implemented by a kernel (an autonomous program), together with a library of functions for accessing conveniently its services.

The real-time operating systems (RTOS) are operating systems specifically suited for embedded applications:

- They are usable on hardware with limited resources.

- The following features are precisely documented:
  - the scheduling strategy,
  - the maximum execution time of each service,
  - every internal mechanism that can influence the latencies (e.g., the longest interval during which interrupts are disabled by the kernel).
- The user can implement urgent operations as interrupt routines.
- The OS provides time-oriented services: one-shot or periodic timers, periodic execution of tasks, . . .
- Complex protection mechanisms against invalid user code may be absent.
- The kernel configuration can be parameterized in detail by the programmer.

Execution levels

At a given time, the instruction currently executed by the processor can either be

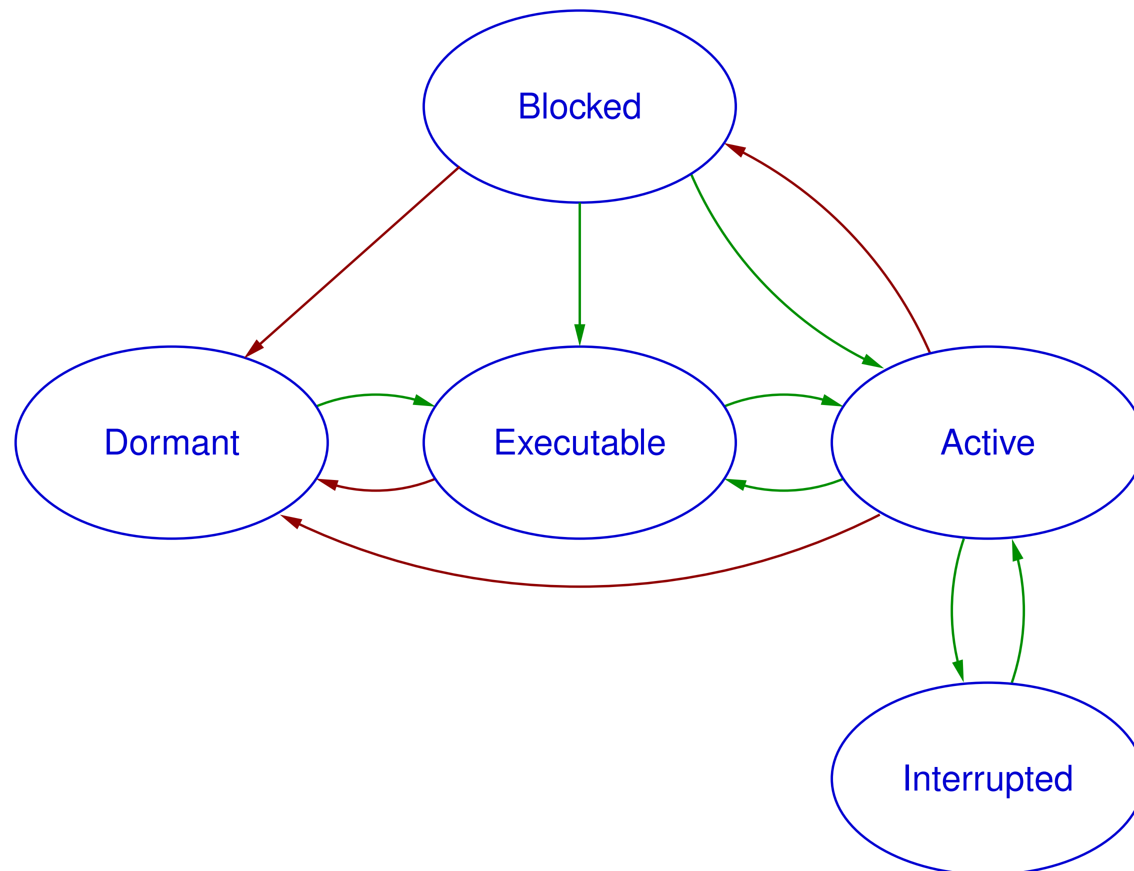
- a kernel operation (possibly located in an interrupt routine),
- an instruction belonging to an interrupt routine programmed by the user, or
- an instruction of a user task.

The processes

Each task managed by an OS is represented by a process. At a given time, a process is in one out of five possible states:

- Dormant: The task is not scheduled (e.g., because it is not yet known to the OS).
- Executable: The task is ready to execute instructions, but is not currently running.
- Active: The instructions of the task are now being executed by the processor.
- Blocked: The execution of the task is suspended while waiting for a signal, or for a resource to become available.
- Interrupted: The task is executing an interrupt routine programmed by the user.

Possible transitions between the states of a process:



The scheduler

The scheduler is the kernel component responsible for managing the state of the processes, i.e., for assigning the processor to the processes.

Principles:

- Each task is characterized by a priority (either constant or variable during its execution).
- The scheduler always assigns the processor to the non-dormant and non-blocked task that has the highest priority.

If several tasks share the highest priority, then the conflict can be solved in several ways:

- The time slicing approach consists in assigning the processor in turn to each of these tasks, in order to execute a bounded sequence of instructions.

- One can alternatively assign the processor to a task that has been arbitrarily chosen.

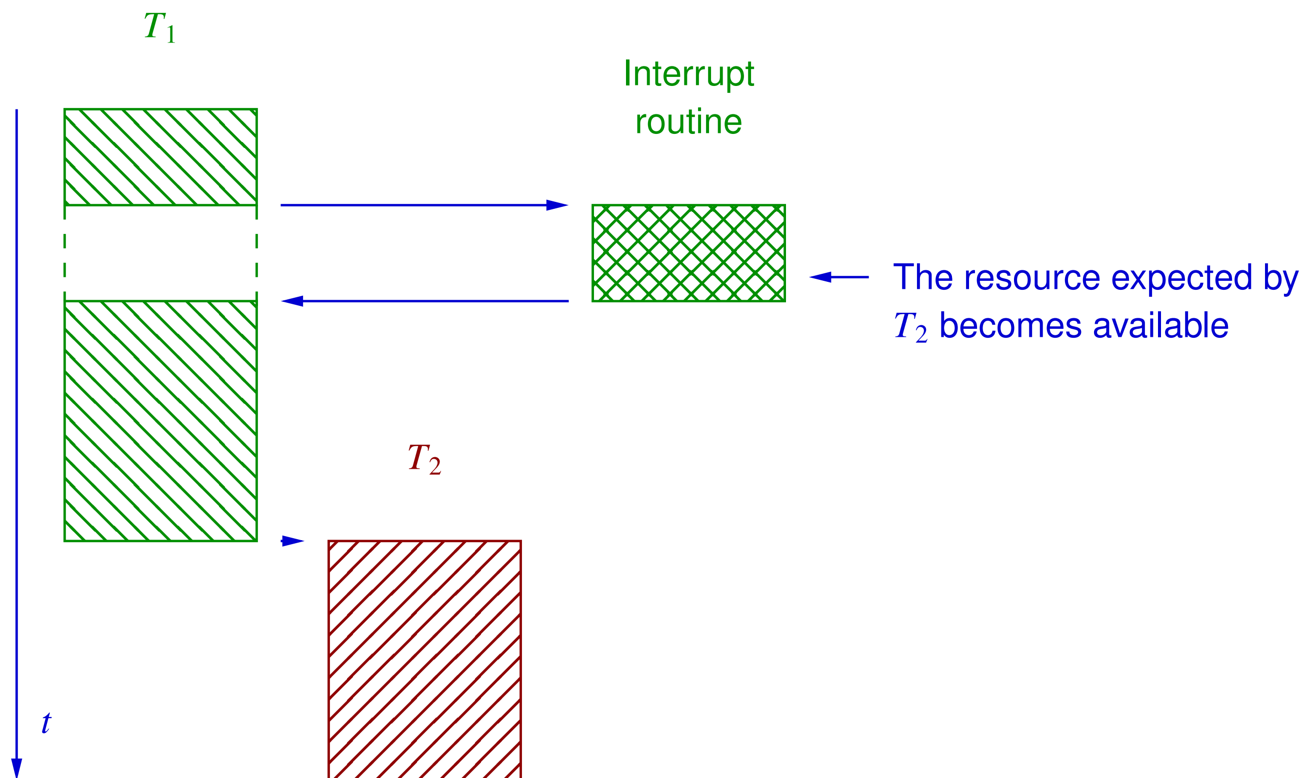
- Another solution is to forbid different tasks to share the same priority.

Note: With the first two strategies, computing the deadline of a task becomes difficult. Most real-time operating systems thus implement the third solution.

Preemption

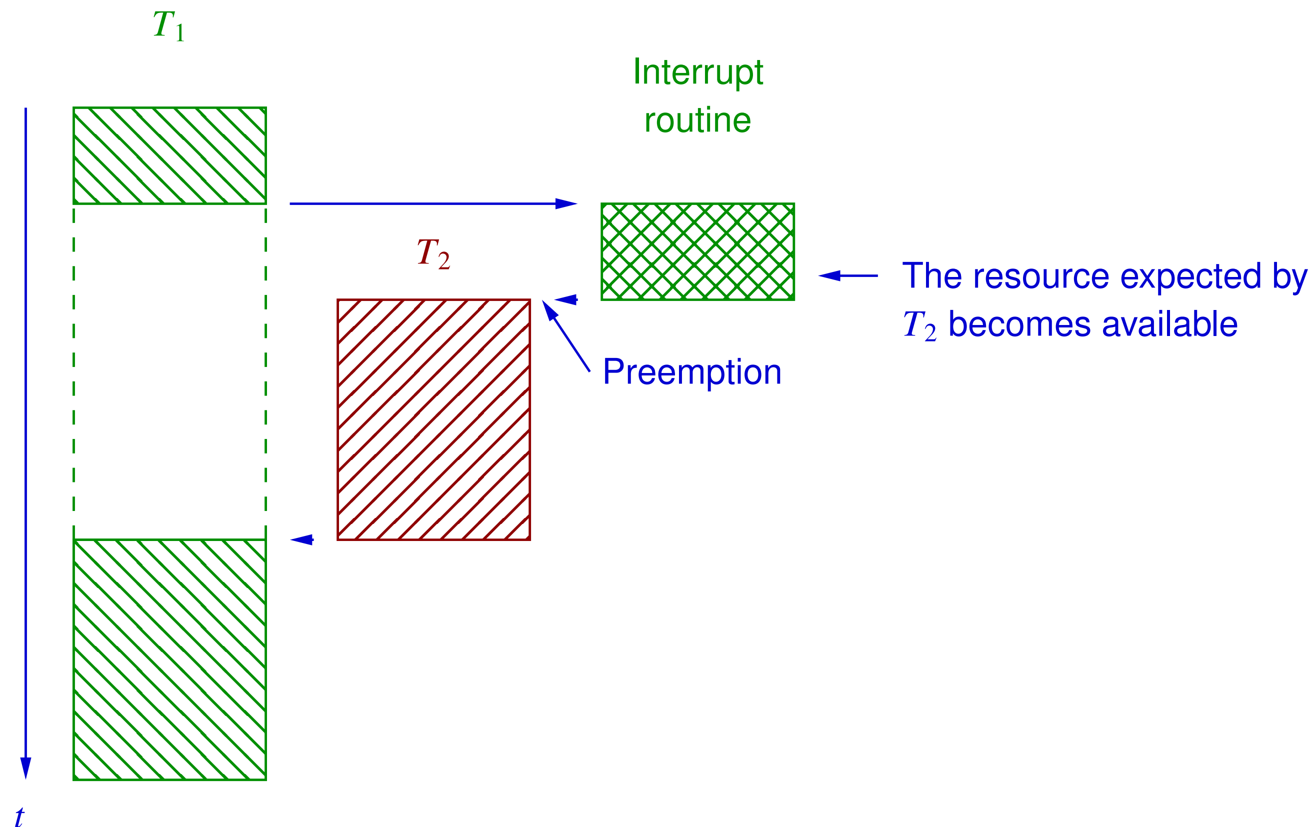
If a task T_2 has a higher priority than the active task T_1 and switches from the blocked to the executable state, then there are two possible scheduling strategies:

- The task T_2 remains suspended (in executable state) until completion of T_1. The scheduler is said to be non-preemptive.



Drawback: The latency of a task is influenced by the behavior of tasks with a lower priority.

- The scheduler turns the task T_1 executable, and assigns the processor to T_2. The scheduler is said to be preemptive.



Context switching

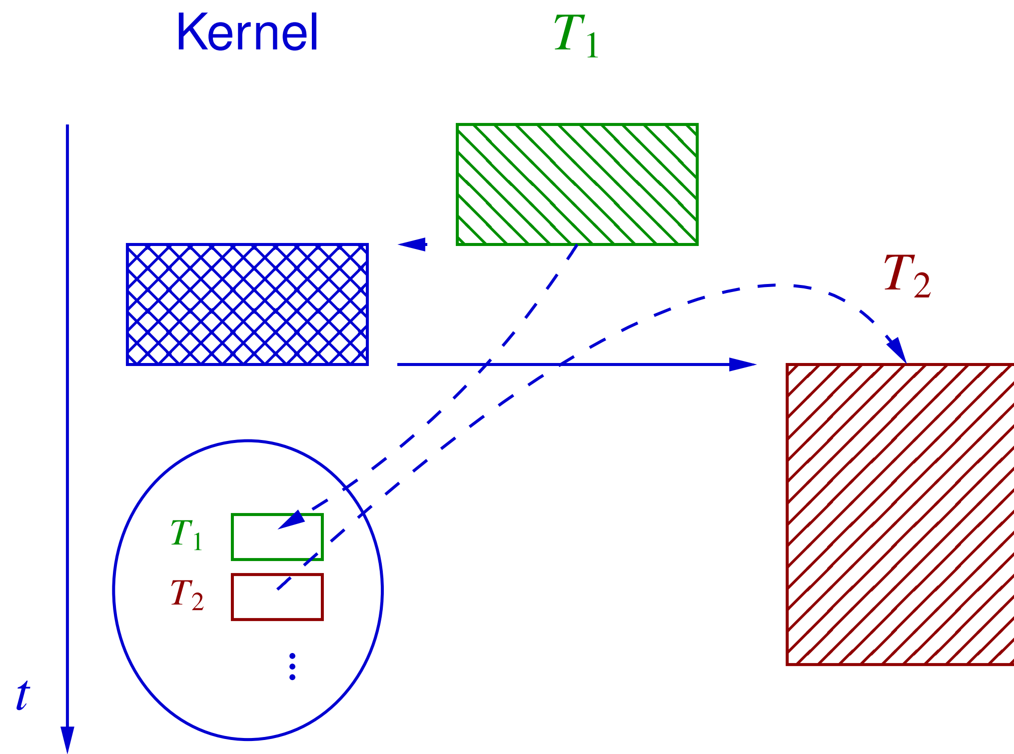
The scheduler performs a context switch when it transfers the processor from a process to another.

Principles:

- The suspended task must be able to resume its execution later. The state of the processor thus has to be saved when the task is suspended.

  The kernel memory maintains for each non-dormant process a context storage area for this purpose.

Illustration:

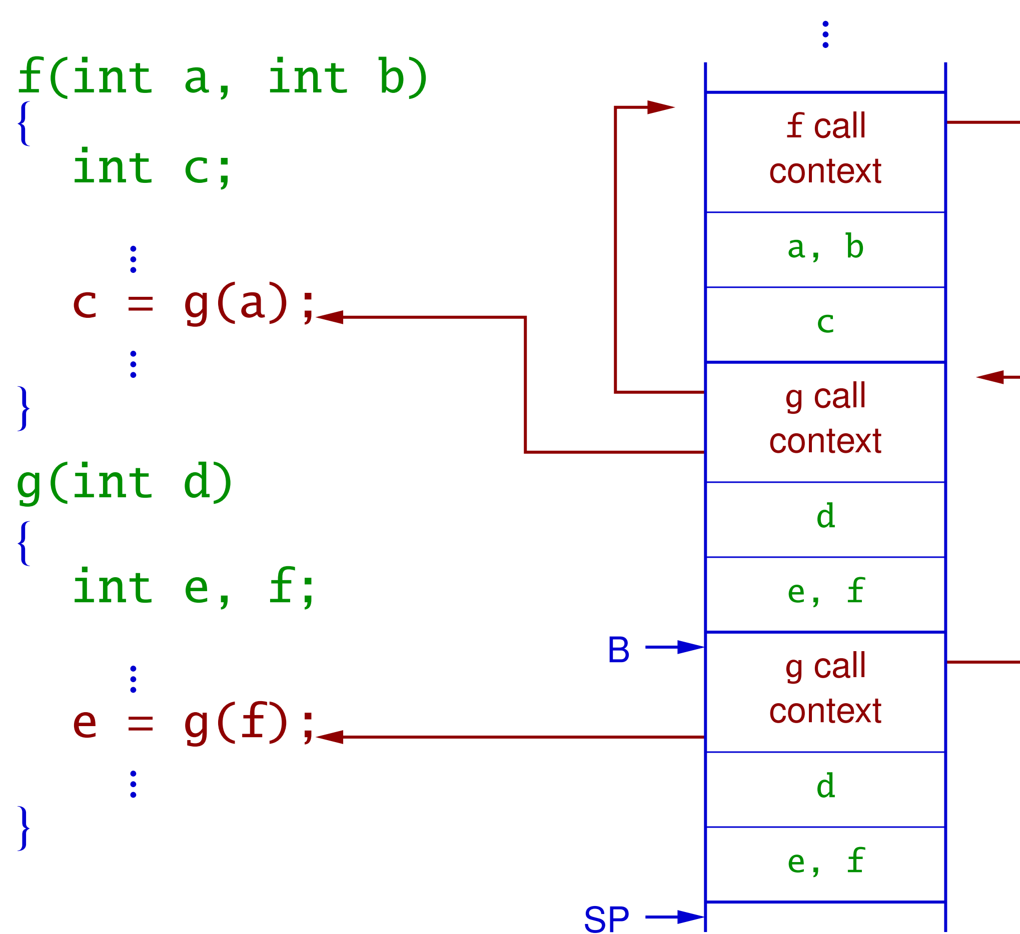


- The working data of the suspended task has to be preserved until its execution can be resumed.

  This data is located on the runtime stack of the task, which contains

  - the context (return address, stack register values) of the active function calls, and
  - the arguments and local variables of these function calls.

Example:



Notes:

– Since a task can be suspended at any time, it is necessary for each process to manage its own stack.

– In general the stack pointers (e.g., top of stack, base of current stack frame) are particular processor registers. Those pointers are therefore saved, together with the other registers, during a context switch.

– The kernel also manages its own stack.

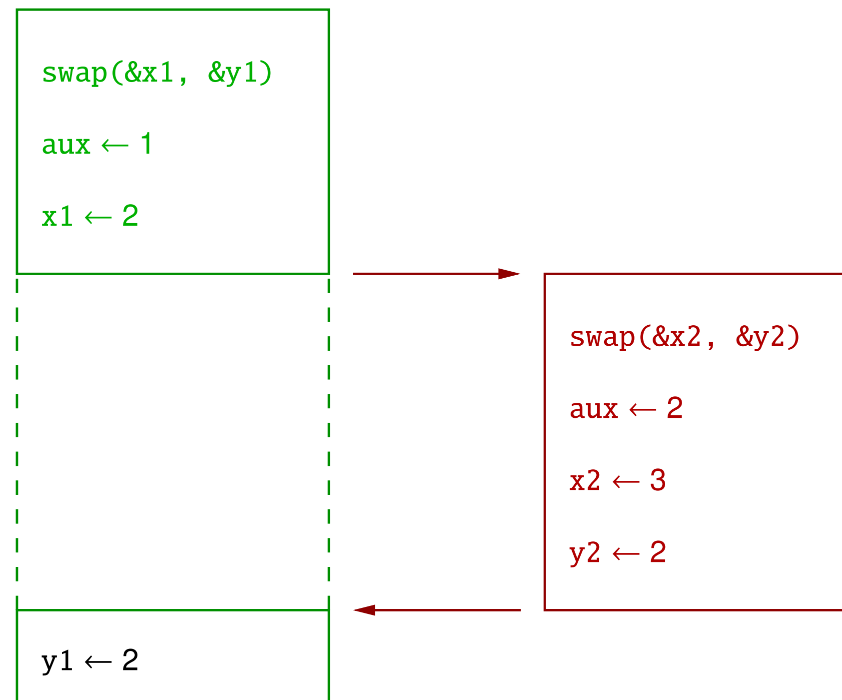
Reentrancy

With a preemptive scheduler, calling the same function in different tasks can be problematic.

Example:

```
static int aux;

void swap(int *p1, int *p2)
{
  aux = *p1;
  *p1 = *p2;
  *p2 = aux;
}
```

aux	
x1	1
y1	2
x2	2
y2	3



Definition: A function is said to be reentrant if it can be simultaneously called by several tasks without possibility of conflict.

Examples:

- Reentrant function:

```
void swap(int *p1, int *p2)
{
  int aux;

  aux = *p1;
  *p1 = *p2;
  *p2 = aux;
}
```

- Non-reentrant function:

```
volatile int is_new;  /* modified by another task */

void display(int v)
{
  if (is_new)
    {
      printf(" %d", v);
      is_new = 0;
    }
  else
    printf(" ---");
}
```

Note: The second function is non-reentrant for three reasons:

- The test and assignment operations over the global variable `is_new` are performed by different instructions.
- The operations involving `is_new` are not necessarily atomic.
- The function `printf` might not be reentrant.

Communication between tasks

Organizing data transfers between processes is more difficult than between tasks and interrupt routines:

- Context switches can occur unpredictably at any time.
- Context switches can only be disabled in software, by modifying the scheduling policy.

Solution: One can use services provided by the kernel, aimed at

- synchronizing the operations of concurrent tasks, and
- coordinating data transfers from a process to another.

Note: Using incorrectly communication or synchronization services can lead to deadlocks, when every task is suspended waiting for resources that can only be provided by other tasks.

The semaphores

A semaphore s is an object that

- has a value $v(s) \geq 0$,
- over which the two following operations can be performed:
  - wait(s):
    - $*$ if $v(s) > 0$, then $v(s) \leftarrow v(s) - 1$;
    - $*$ if $v(s) = 0$, the task is suspended (in blocked state).
  - signal(s):
    - $*$ if at least one task is suspended as the result of an operation *wait(s)*, make one of them become executable;
    - $*$ otherwise, $v(s) \leftarrow v(s) + 1$.

Notes:

- The operations that test and modify the value of a semaphore must be implemented atomically.
- Binary semaphores are semaphores with a value restricted to the set $\{0, 1\}$.
- There are several possible strategies for selecting a task blocked on a semaphore in order to make it executable again: arbitrary choice, FIFO policy, priorities, . . .

In most applications, acquiring a semaphore represents the access right to a resource.

Example: Mutual exclusion between two tasks (binary semaphore s initialized to 1).

```
void task1(void)                void task2(void)
{                               {
  for (;;)                        for (;;)
    {                               {
      wait(s);                        wait(s);
      !! critical section;            !! critical section;
      signal(s);                      signal(s);
      !! other operations;            !! other operations;
    }                               }
}                               }
```

The message queues

A message queue is an object that implements synchronous or asynchronous data transfers between tasks.

Principles:

- The maximum capacity of a queue (i.e., the maximum number of messages that have been written and not yet read) and the size of each message are fixed.
- Send and receive operations are performed atomically.
- A task that is waiting to receive data from a queue is suspended by the scheduler (in blocked state).

Variants:

- Several data access policies are possible: FIFO order, arbitrary selection, priority mechanism.

- Sending data to a saturated message queue can either discard the new message, block the sender, block the sender during a bounded amount of time, . . .
- When a task is blocked waiting for data from an empty queue, a maximum suspension delay (i.e., a timeout) can be specified.
- The maximum capacity of a queue can be reduced to zero (rendez-vous synchronization).

Programming with interrupts

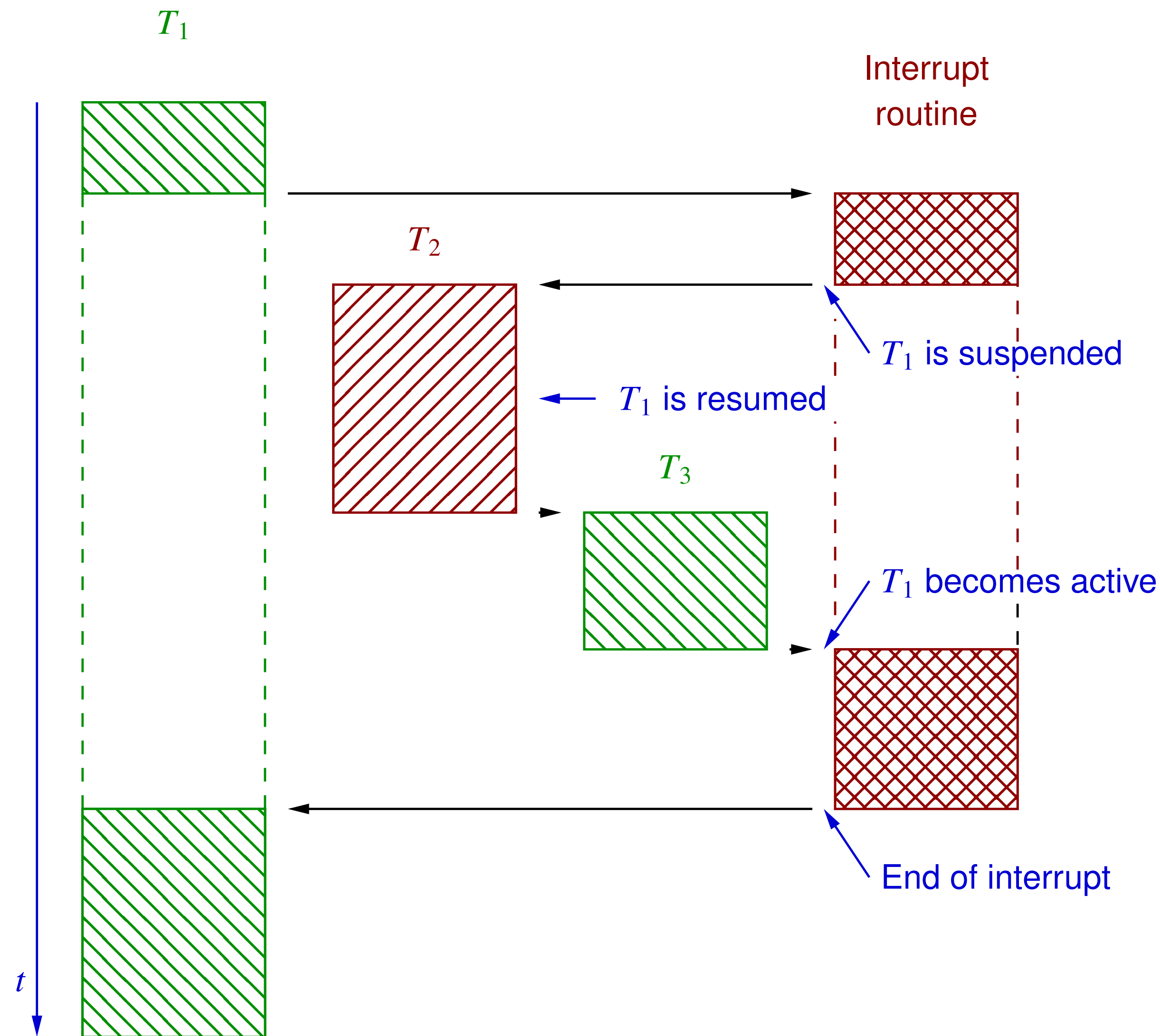
The scheduler and the interrupt mechanism are both able to move the control point from one location in the program code to another. One must take care of avoiding conflicts between those mechanisms.

First rule:

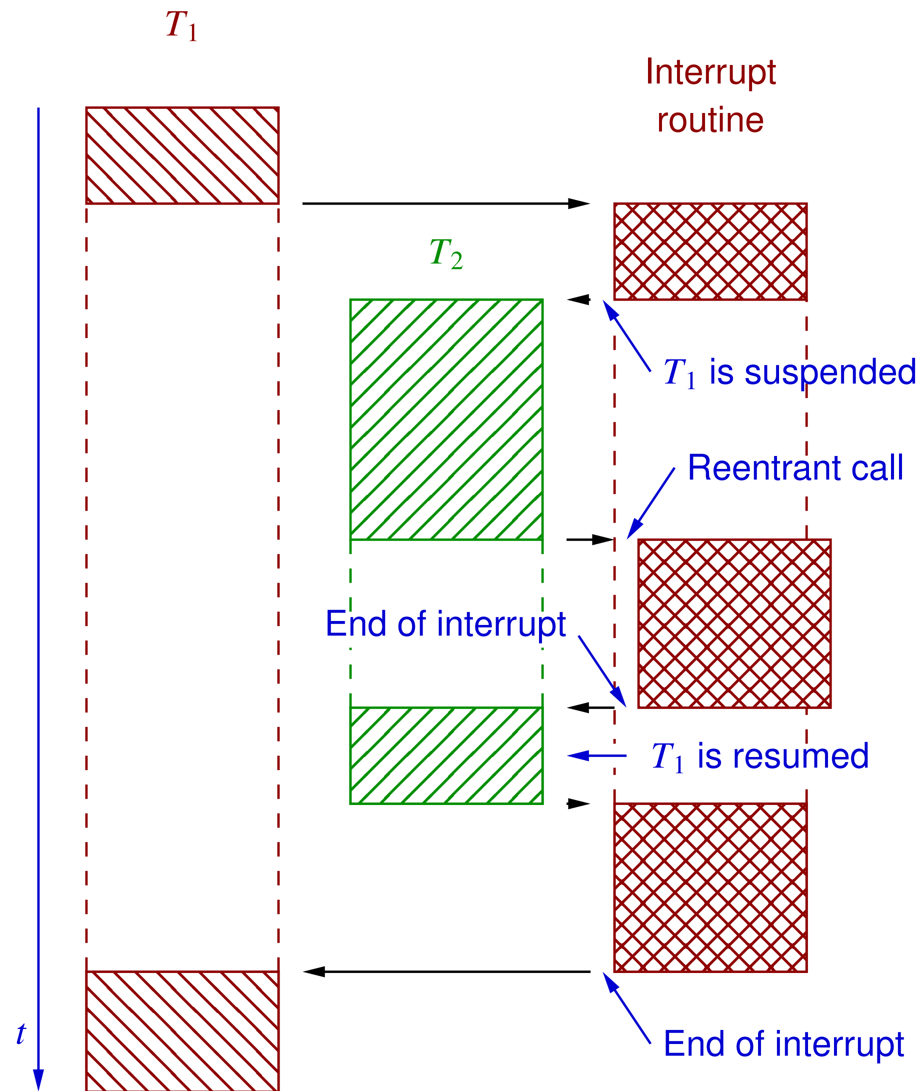
> *An interrupt routine is not allowed to call an OS service if this service can suspend the current task (e.g., acquiring a semaphore (wait), receiving data from a message queue, . . .).*

- Indeed, if this rule is not respected, then an interrupt routine can get suspended, which amounts to assigning to this interrupt routine an effective priority smaller than the one of a task.

Example:



- Moreover, the interrupt routine might get called again before its completion. If this routine is not reentrant, then erroneous behaviors are possible (e.g., overwriting a saved processor context).

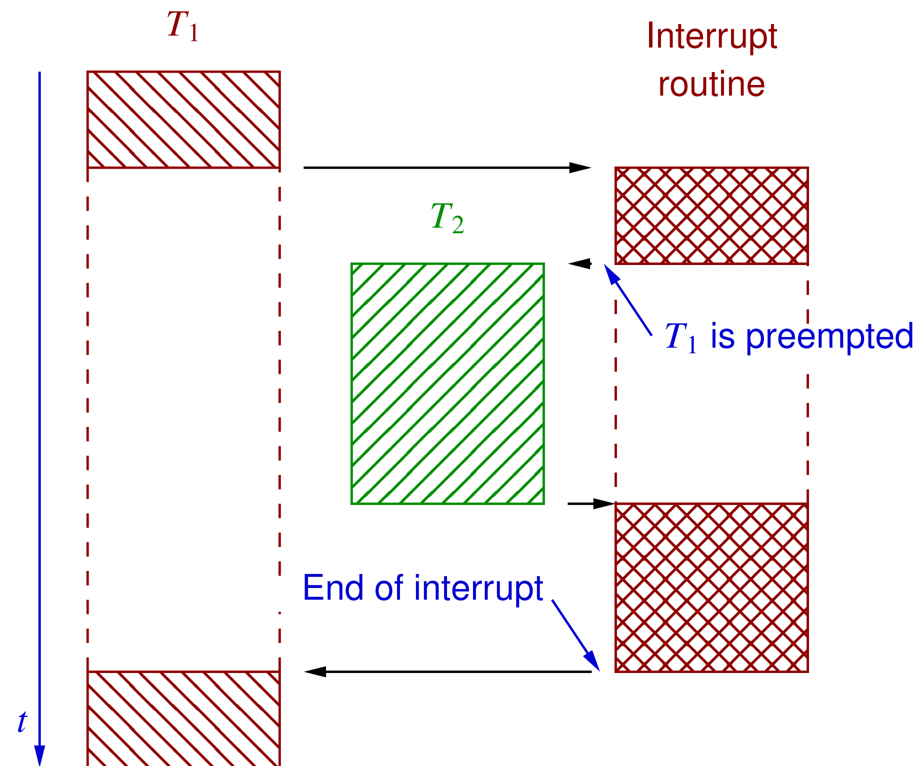


Second rule:

> *If an interrupt routine calls an OS service that can lead to a context switch, then the scheduler must be informed that this service call is performed inside an interrupt routine.*

If this rule is not respected, then the scheduler can suspend the execution of an interrupt routine.

Example:

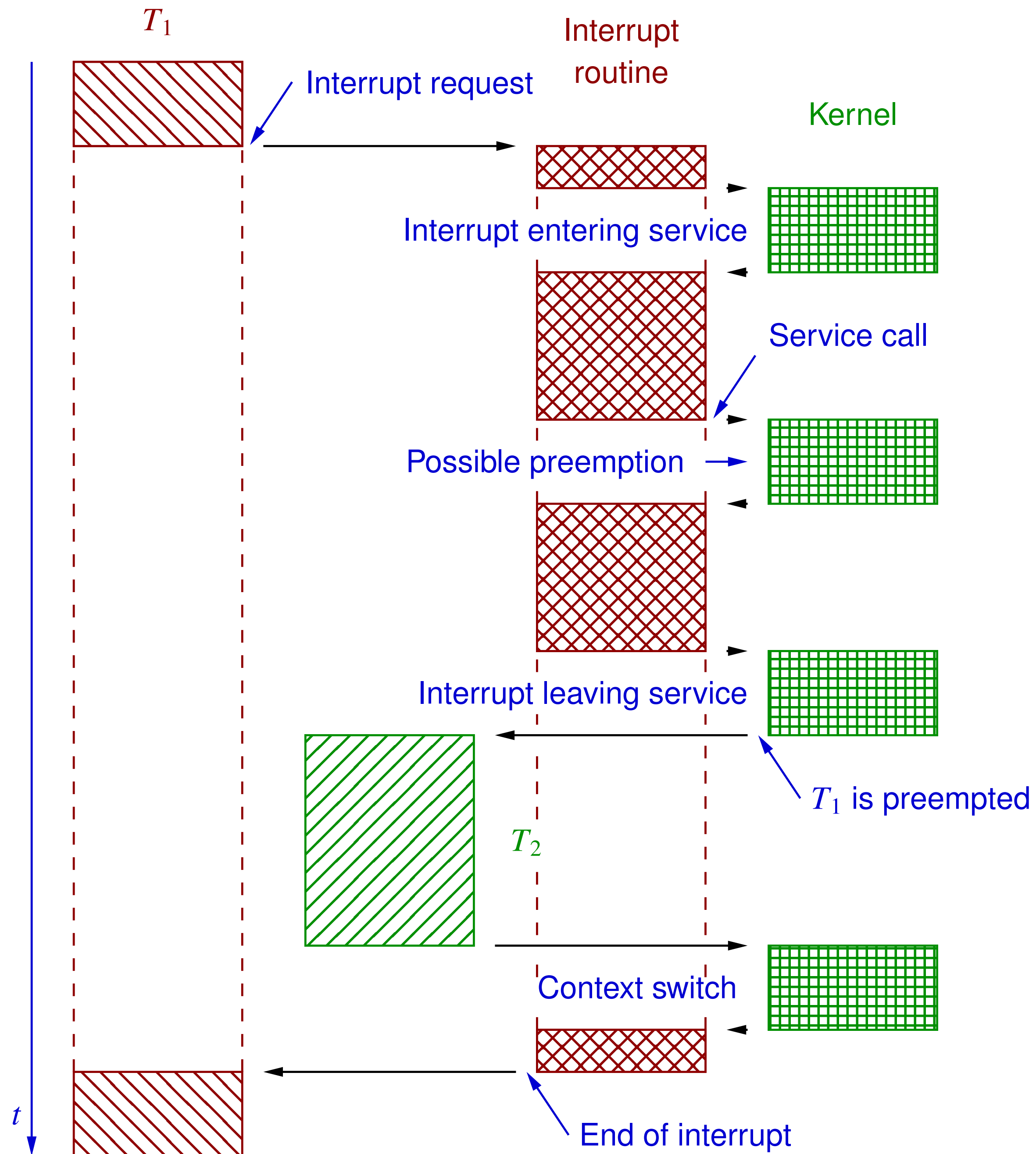


Solution: At the beginning and at the end of each interrupt routine programmed by the user, special OS services must be called in order to inform the kernel that the processor is currently executing an interrupt routine.

Notes:

- In the case of many levels (i.e., priorities) of interrupts, those services must handle correctly nested interrupt routine calls.
- Some operating systems provide alternate versions of some services, intended to be called from interrupt routines.
- Interrupt latencies are increased by the time needed for executing the notification services.

Example:



Note: An interrupt routine containing explicit instructions for saving the processor state must perform the corresponding restore operations before informing the kernel that the interrupt routine is about to end.

Time-oriented services

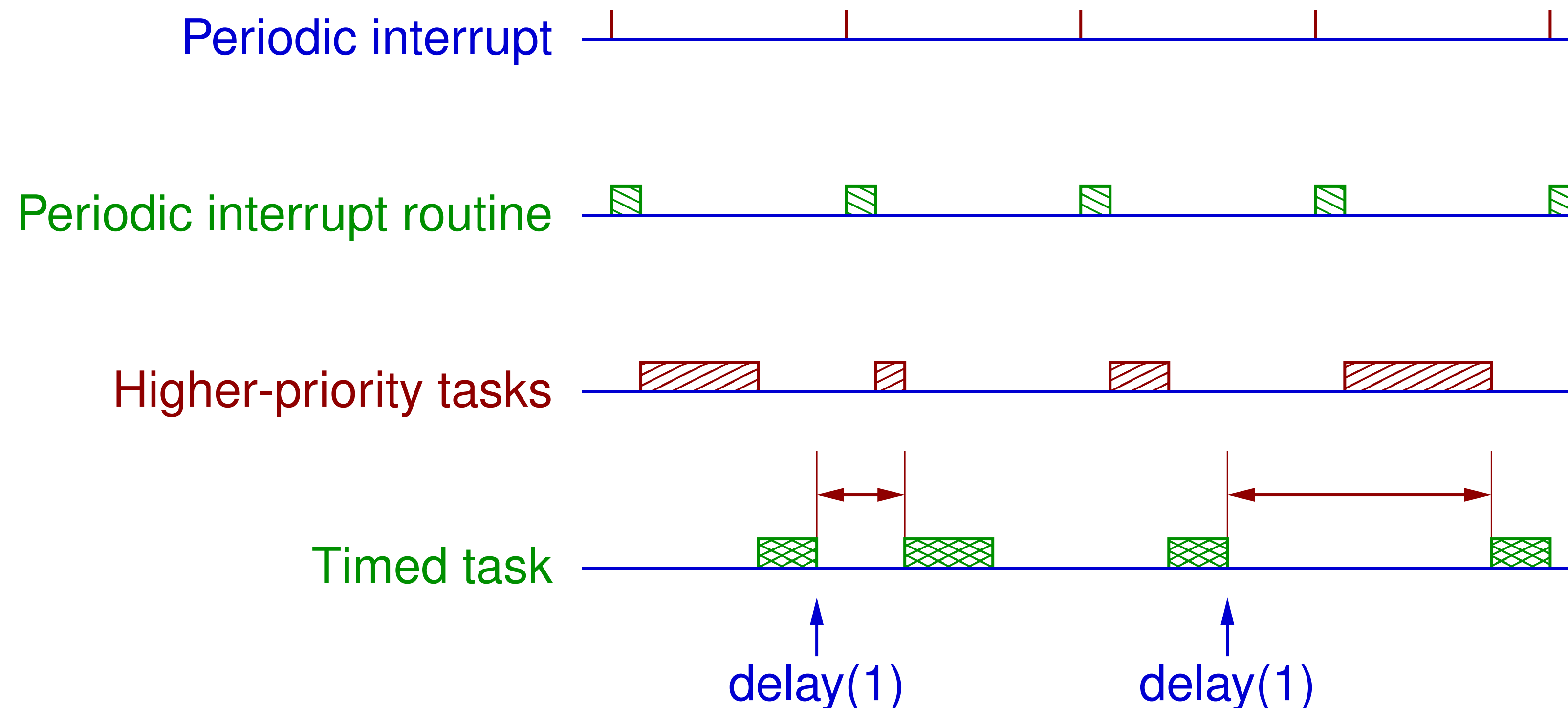
The real-time operating systems offer timed services, for instance for suspending a task for a predefined amount of time.

Principles:

- A dedicated component triggers periodic requests (clock, heartbeat) for an interrupt that has
  - a higher priority than the interrupts programmed by the user, and
  - an interrupt routine implemented by the kernel.
- The delay during which a task is suspended is expressed in the number of occurrences of this interrupt request signal (ticks, beats).

Note: The precision is limited. Asking to suspend the task during k ticks only ensures that the suspension delay is greater or equal to $k - 1$ times the clock period.

Example:



Chapter 6

Real-time operating systems: Implementation issues

Overview of the main problems

- The following operations need to be efficient (i.e., ideally, to have a maximum execution time that is independent from the number of tasks managed by the operating system):
  - Identifying the executable process with the highest priority in order to make it active.
  - Performing a context switch.
  - Selecting the process that has to be unblocked following an operation over a communication object.
- For operations over communication or synchronization objects that can suspend a process, it should be possible to specify a timeout (i.e., a maximum suspension delay).
- One needs to be able to access all processor registers.
- For some applications, the real-time operating system has to share the processor with another operating system.

Task control blocks

A Task Control Block (TCB) is a data structure that represents a non-dormant process inside the kernel memory. This structure contains:

- The current priority of the task.

  Note: When the task priorities are fixed and unique, they can also be used as process identifiers.

- The context of the task, i.e., the state of the processor, saved when the task was last suspended.

  Notes:

  - This context contains, in particular, a pointer to the runtime stack of the process.
  - Some operating systems (e.g., μCOS-III) save the bulk of the context on this stack.

- The current state of the process (executable, active, blocked, or interrupted).

- Information for managing a potential timeout.
- A pointer to a data structure representing a communication object that the task is (possibly) attempting to acquire.
- Pointers linking the TCB to the global data structures of the kernel.
- Auxiliary data aimed at speeding up some operations (e.g., values derived from the task priority).
- Additional information managed by the user (e.g., configuration data for a peripheral controlled by the task).

Global data structures of the kernel

The global information managed by the kernel essentially contains:

- Sets of task control blocks corresponding to
  - the executable (or active) processes,
  - the processes suspended for a given delay,
  - the free blocks.

  Those sets are organized as simply or doubly-linked lists, or by hash tables, in order to be able to manipulate them in constant time.
- A structure for identifying efficiently the executable process with the highest priority.
- An index for accessing directly a task control block from its corresponding process identifier.
- Data structures representing the state of communication objects.

Example: μCOS-III

- The maximum number of process priorities is a compile-time configuration parameter of the kernel (`OS_CFG_PRIO_MAX`).
- The set of executable processes is represented by
  - an array `OSPrioTb[]` of bit fields (with a width suited for the processor architecture). Each set bit corresponds to a priority for which there exists at least one executable (or active) process.
  - an array `OSRdyList[]` associating to each priority level a doubly-linked list of TCB of executable processes.

    Note: μCOS-III allows several processes to share the same priority. Such processes are then scheduled by time slicing.
  - A pointer `OSTCBCurPtr` to the TCB of the currently active task.
- The set of processes suspended for some delay is represented by a hash table `OSCfg_TickWheel[]`, indexed by their deadline.

- Keeping an index of all non-dormant tasks is not necessary, because process identifiers are defined as pointers to the corresponding TCB.
- Managing a list of free TCB is avoided by letting the user code allocate TCB when tasks are created.
- A global counter `OSIntNestingCtr` keeps tack of the current interrupt nesting level.

Illustration (32-bit processor)

OSPrioTb[]

31																															0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1

OSTCBCurPtr

OSRdyList[]

0
1
2
14
42

OS_TCB
(task 14-1)

OS_TCB
(task 14-2)

OS_TCB
(task 42)

14
14
42

stack of task 14-1

stack of task 14-2

stack of task 42

Note: Identifying quickly the executable process with the highest priority is achieved by

- exploiting specific processor instructions (e.g., *Count Leading Zeros*, CLZ), or
- tabulating the possible values of bit fields.

The scheduler

The scheduler is implemented by a kernel function called after each operation that can potentially influence the state of processes:

- Creating or destroying a task, or modifying a task priority.
- Acquiring or releasing a communication object.
- Servicing the clock interrupt.

This function must be kept simple and efficient, and only performs the following operations:

1. Checking whether the scheduler is allowed to preempt tasks.
2. Identifying the executable task with the highest priority.
3. Performing a context switch in order to assign the processor to this task.

Note: The possibility of enabling or disabling preemption is offered because

- preempting the current task should be prevented inside interrupt routines (cf. Chapter 5).
- it provides a simple mechanism for manipulating atomically shared variables or communication objects (*preemption locks*). However, this mechanism
  - increases the latency of the tasks (by the duration of the longest interval during which preemption is disabled), and
  - affects all the tasks of the system (not only those that need to be coordinated).

Context switching

The main issue for implementing context switching is to be able to save and restore all the processor registers.

For many processors, these operations are automatically performed (totally or in part) during interrupts:

- When an interrupt routine is called: The current value of the registers is saved on the runtime stack of the interrupted task.
- When an interrupt routine returns: The values extracted from the current stack are loaded into the processor registers.

A simple solution (when allowed by the processor architecture) consists of performing context switching in an interrupt routine, the corresponding interrupt request being triggered by the scheduler.

The operations performed by this interrupt routine thus amount to

1. Saving the stack pointer of the suspended task into its associated TCB.
2. Loading the stack pointer of the task that becomes active from its associated TCB.
3. Executing a return from interrupt instruction.

Notes:

- This approach avoids the need to store the entire state of the processor into a TCB.
- Preserving the state of the processor between a kernel service call and the subsequent context switch can be tricky.
- The user can sometimes define a hook function that will be called at every context switch (a typical application is to put a peripheral in sleep mode).

Task creation and destruction

Creating or destroying a process essentially amounts to updating the data structures managed by the kernel, and to then call the scheduler.

Notes:

- The runtime stack of a new process is allocated by the task that asks this process to be created.
- The initial processor context of a new task, including its entry point, is built when its stack is initialized.
- A parameter can generally be passed to a newly created task, in order to make it possible for several tasks sharing the same code to behave differently.
- One must take care of removing references to a task that is destroyed from the structures managing communication objects.

The idle task(s)

Some operating systems systematically create one or many internal tasks, with a lower priority than the processes instantiated by the user.

There are many advantages to this approach:

- The scheduler can be more efficient, since it does not have to check whether there exists at least one executable task.
- Such tasks make it possible to measure the processor utilization.

  Example: μCOS-III defines two idle tasks:

  - `OS_IdleTask()`, executing an infinite loop incrementing global counters.
  - `OS_StatTask()` (optional), computing at regular intervals the processor utilization from the value of the counters.

- In the case of a mobile system, an idle task can put the processor and some peripherals in sleep mode in order to conserve energy.

Time management

Quantitative time management is performed by the clock interrupt routine.

Principles: At each clock tick:

- One computes the set of suspended tasks that must become executable again.
- One increments a counter aimed at measuring elapsed time.

Notes:

- The maximum execution time of the clock interrupt routine depends on the mechanism used for waking up tasks.

  With the help of suitable data structures, this time can become equal to the number of tasks becoming executable.

- It is often necessary to configure and calibrate the clock interrupt source during initialization.

Example: μCOS-III

- The time management operations are not directly performed in the clock interrupt routine, but in an internal task `OS_TickTask()` woken up by this routine.

  Advantage: Some user-defined tasks can have a higher priority than the time management operations.

- The deadline of the tasks suspended to a timeout is expressed with respect to a global clock counter.

- A hash table `OSCfg_TickWheel[]` stores pointers to the TCB of those tasks, indexed by their deadline. Tasks sharing the same table entries are sorted in increasing deadline order.

Communication and synchronization objects

In the kernel memory, a communication or synchronization object is represented by a structure containing:

- The type of the object (semaphore, message queue, . . .).
- Data representing the state of the object (e.g., for a semaphore, an integer counter).
- A set of suspended tasks, waiting to acquire the object.

  In the case of a priority-based selection policy, such a set can be represented by a doubly-linked list of TCB, sorted in decreasing priority order.

If necessary, the kernel also maintains a table of allocated objects.

Finally, the TCB of each task waiting for an object contains a pointer to the structure managing this object.

Note: Implementing kernel services that update the state of objects does not require specific instructions such as *test and set*, since it is sufficient to disable interrupts during non-atomic operations.

Combining a real-time and a non-real-time operating systems

It is possible to combine in a single application a real-time operating system (*RTOS*) with another operating system (*host OS*). There are two possibilities:

- The operations of the host operating system are suspended when the real-time OS is started, and get the processor back when the RTOS terminates (e.g., μCOS-III).
- The host operating system is seen as special task that has a lower priority than all the tasks managed by the real-time OS (e.g., RTAI).

  For implementing this approach, it is necessary to ensure that the host OS can never disable the interrupts managed by the kernel or the real-time tasks.

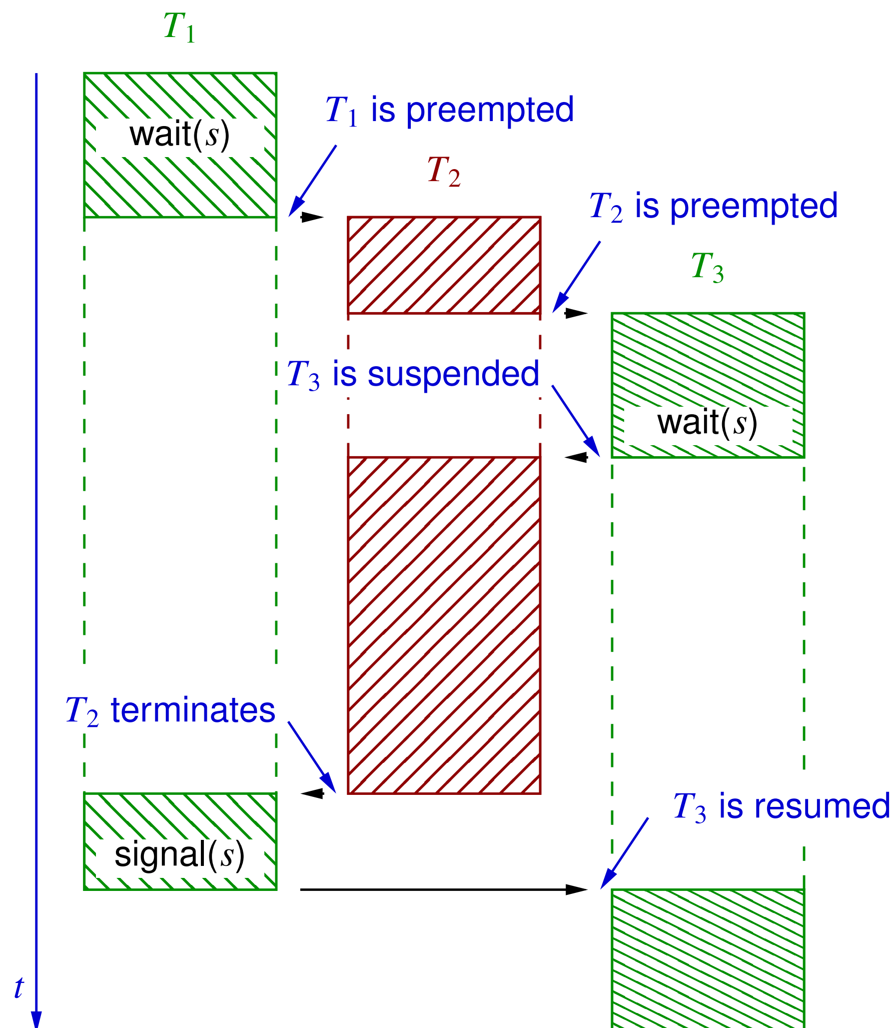
Chapter 7

Scheduling problems

Priority inversion

Priority inversion happens when a task is suspended waiting for a resource controlled by another task with a lower priority.

Example:



Problem:

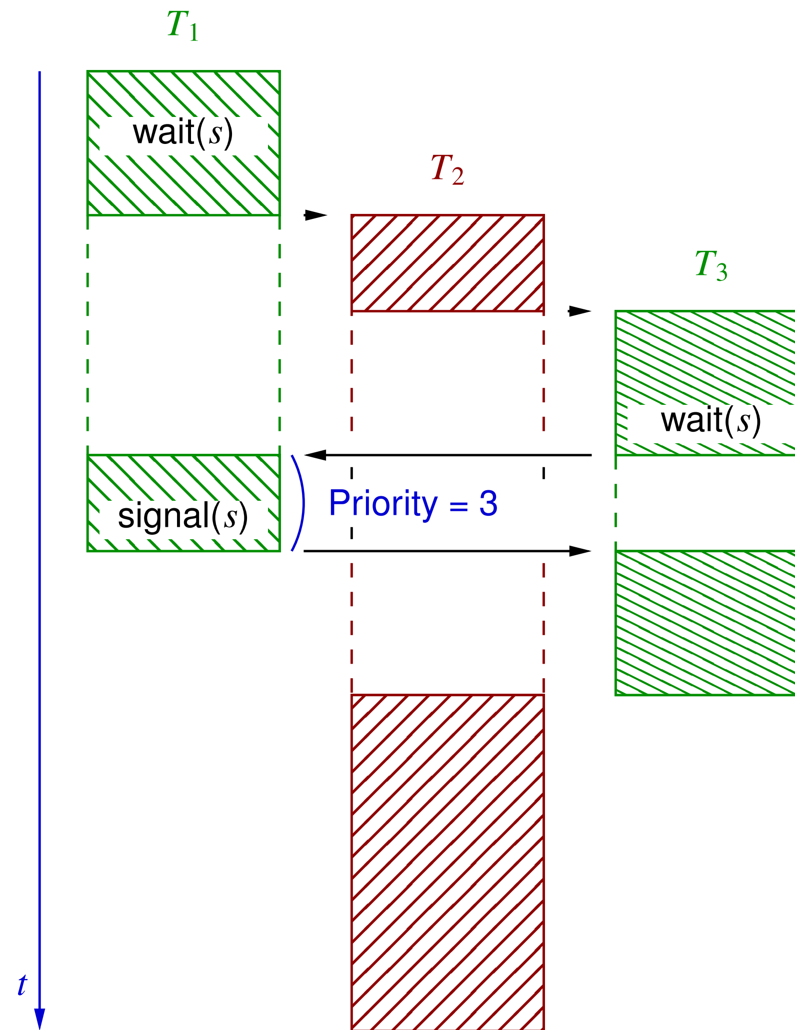
In such a situation, the effective priority of T_3 becomes equal to the one of T_1.

Solution:

The priority of T_1 can be momentarily increased (becoming equal to that of T_3) during all the time that T_3 is suspended waiting for the semaphore acquired by T_1.

This priority inheritance mechanism is automatically applied by some operating systems.

Illustration:



Periodic tasks scheduling

We consider a simplified programming environment satisfying the following hypotheses:

- The number of tasks to be executed is fixed.
- Each task is characterized by a distinct and constant priority.
- The execution requests for each task occur periodically, i.e., with a constant delay between two successive requests.

  In particular, the timing of execution requests for a task cannot depend on operations performed by other tasks.
- The execution time of each task is constant.

- The following real-time constraint must be satisfied:

  Each execution of a task must finish before or at the same time as the next request for executing this task.

- Context switches are instantaneous and preemptive.

Critical instants and critical zones

In addition to its priority, each task τ_i is characterized by

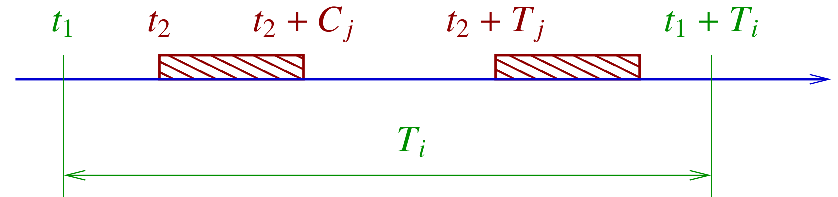
- its period T_i, and
- its execution time (for each period) C_i.

Definitions:

- The response time of an execution request for τ_i is the delay between this request and the end of the corresponding execution of this task.
- A critical instant for the task τ_i is an occurrence of an execution request for τ_i that leads to the largest possible response time for this task.
- A critical zone for τ_i is an interval of duration T_i that starts at a critical instant (for τ_i).

Theorem 1: A critical instant for τ_i occurs when an execution request for this task coincides with requests for executing all the tasks that have a higher priority than τ_i.

Proof: Assume that an execution request for τ_i occurs at $t = t_1$, and that an execution request for a higher-priority task τ_j is received at $t = t_2$.



Advancing the request for τ_j from t_2 to t_1 can never decrease the response time of τ_i

The same reasoning can be applied to all the tasks that have a higher priority than τ_i.

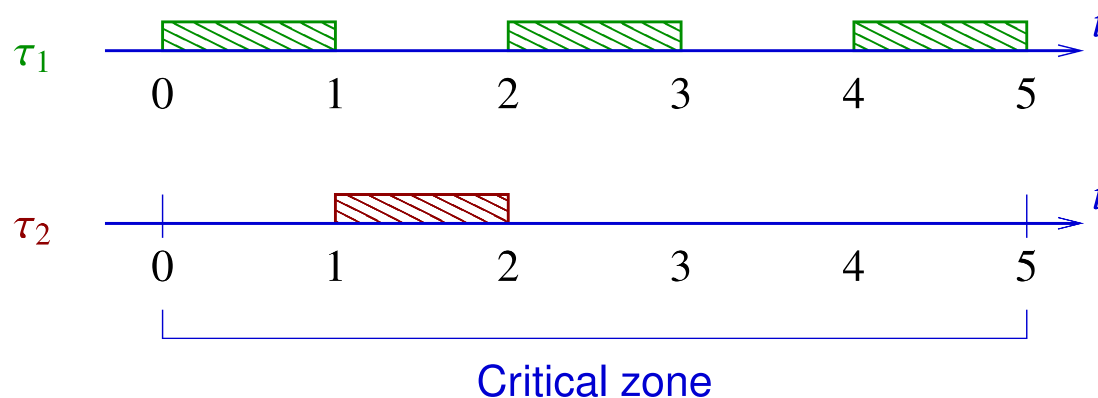
Schedulable tasks

Definition: A set of tasks is schedulable (with respect to a given assignment of priorities) if the response time of each task τ_i is always less than or equal to its period T_i.

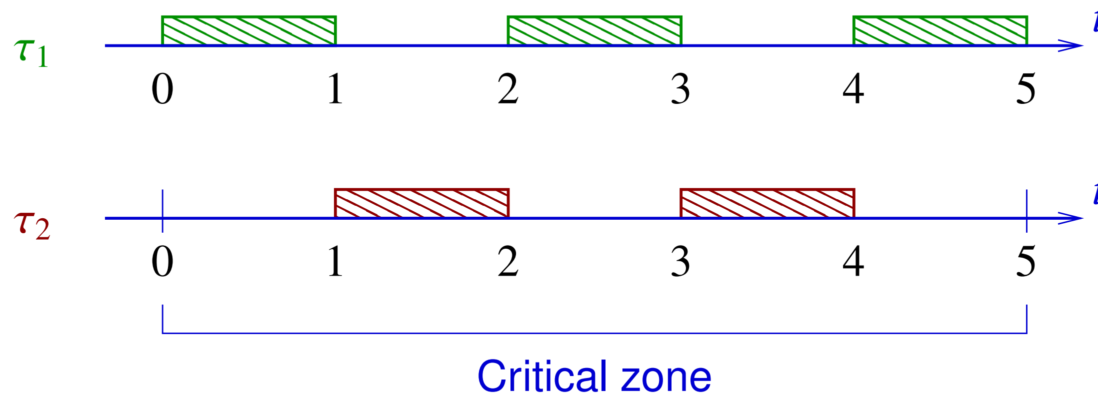
Thanks to Theorem 1, checking whether a given set of tasks is schedulable reduces to simulating the scheduling strategy in the particular case of simultaneous execution requests for all tasks at $t = 0$.

Examples: Consider two tasks τ_1 and τ_2, with $T_1 = 2$, $T_2 = 5$, $C_1 = 1$ and $C_2 = 1$.

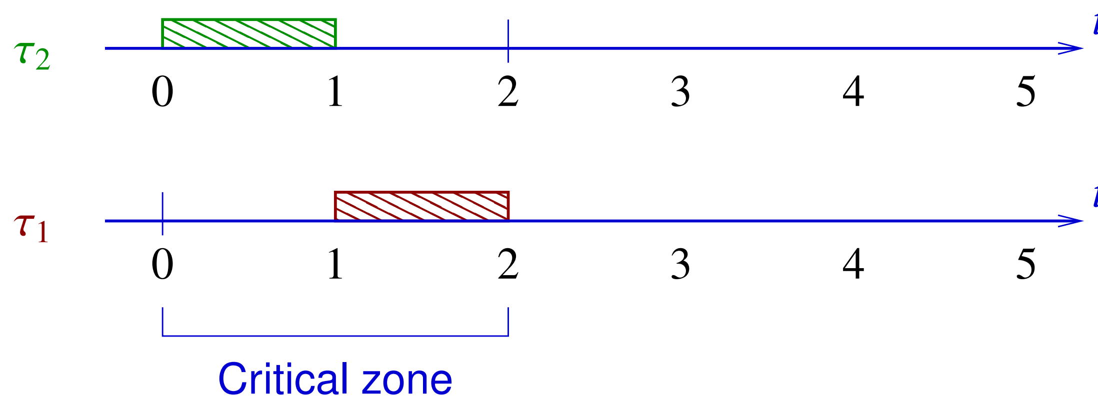
- If τ_1 has a higher priority than τ_2.



The tasks are schedulable, and remain schedulable even if the execution time of τ_2 is increased by one time unit ($C_2 = 2$):



- If τ_2 has a higher priority than τ_1.



The tasks are schedulable.

Note: In this case, the execution time of τ_1 and τ_2 cannot be increased anymore.

Rate-Monotonic Scheduling

In the previous example, the best strategy was to assign the highest priority to the task that has the smallest period.

Definition: Given a set of tasks $\tau_1, \tau_2, \ldots, \tau_n$ with respective periods $T_1, T_2, \ldots, T_n$, the Rate-Monotonic Scheduling (RMS) strategy consists in assigning distinct priorities $P_1, P_2, \ldots P_n$ to the tasks, such that for all i, j:

$$T_i < T_j \;\Rightarrow\; P_i > P_j.$$

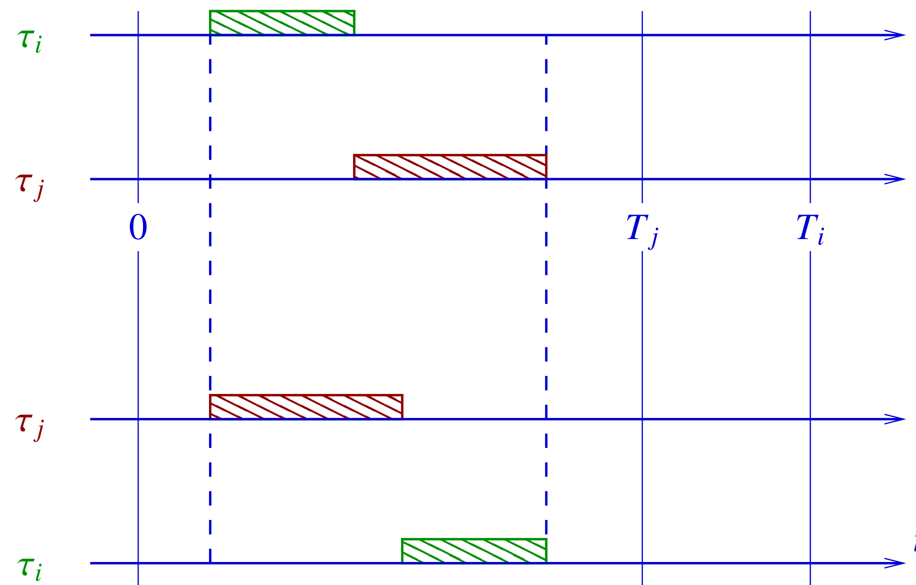
The following result establishes that the RMS strategy is optimal:

Theorem 2: If a set of tasks is schedulable with respect to some priorities assignment, then it is schedulable as well with respect to priorities defined by the RMS strategy.

Proof: Consider a set of tasks $\tau_1, \tau_2, \ldots, \tau_n$ for which there exists a priorities assignment $P_1, P_2, \ldots P_n$ that makes them schedulable.

Let τ_i and τ_j two tasks with adjacent priorities P_i and P_j, such that $P_i > P_j$.

If $T_i > T_j$, then the priorities of τ_i and τ_j can be swapped:



The resulting set of tasks remains schedulable.

By performing repeatedly this operation, one eventually obtains a priorities assignment corresponding to the RMS strategy.

The processor load factor

Consider a set of tasks $\tau_1, \tau_2, \ldots, \tau_n$ with respective periods and execution times $T_1, T_2, \ldots, T_n$ and $C_1, C_2, \ldots, C_n$.

The processor load factor U corresponding to this set of tasks represents the relative amount of CPU time needed for executing them:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

Definition: A set of tasks fully uses the processor if

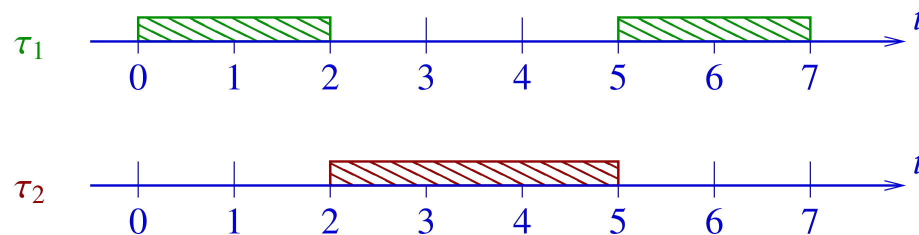
- this set of tasks is schedulable, and
- any increase of the execution time of a task (and hence of the processor load factor) yields a set of tasks that is not schedulable anymore.

Notes:

- Thanks to Theorem 2, checking whether a set of tasks is schedulable or not can be done by assigning RMS priorities to those tasks.
- A set of tasks that has a processor load factor less than 1 is not necessarily schedulable:

  Example:

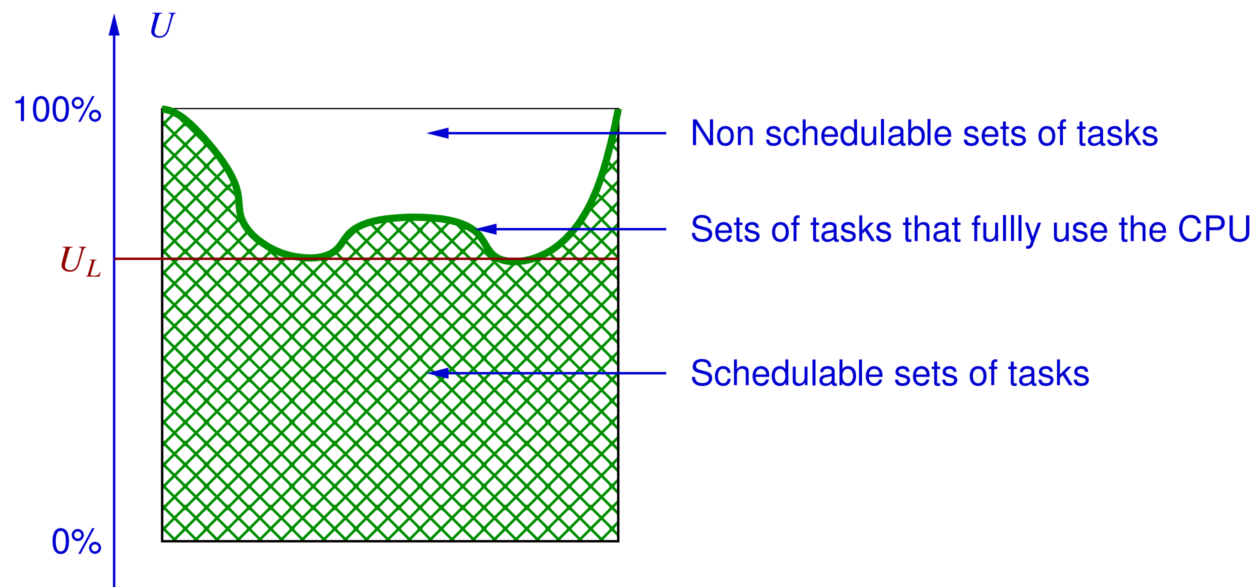
$$\left.\begin{array}{ll}\tau_1: & T_1 = 5,\ C_1 = 2 \\ \tau_2: & T_2 = 7,\ C_2 = 4\end{array}\right\} U = \frac{2}{5} + \frac{4}{7} \approx 97\%$$



Classifying sets of tasks

The set of sets of tasks can be partitioned into three classes:

- The non schedulable sets of tasks.
- The sets of tasks that fully use the processor.
- The schedulable sets of tasks that do not fully use the processor.



The best lower bound U_L on the processor load factor of the sets of tasks that fully use the processor is such that:

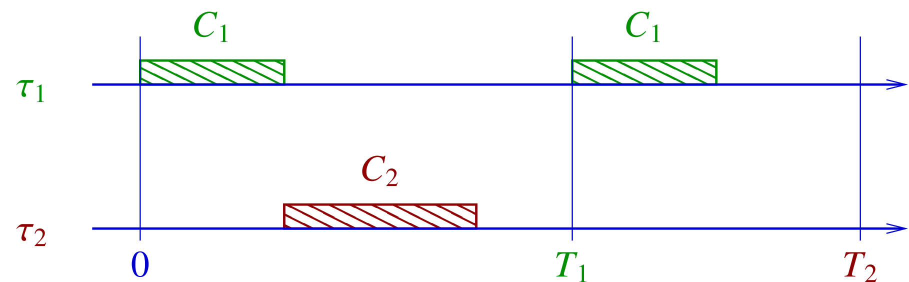
- If the processor load factor of a set of tasks is less than or equal to U_L, then this set of tasks is schedulable (regardless of the periods and execution times of the tasks!).
- If the processor load factor of a set of tasks is greater than U_L, then this set of tasks may or may not be schedulable, depending on the details of the tasks.

U_L: Case of two tasks

Let τ_1 and τ_2 be two tasks with respective periods and execution times T_1, T_2 and C_1, C_2. We assume $T_1 < T_2$. According to the RMS strategy, we assign a higher priority to τ_1.

During a critical zone of τ_2, the number of execution requests for τ_1 is equal to $\left\lceil \frac{T_2}{T_1} \right\rceil$.

- If all the executions of τ_1 in the interval $[0, T_2]$ terminate earlier than or at $t = T_2$.



The following condition is satisfied:

$$C_1 \leq T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor .$$

For a given value of C_1, the largest possible value of C_2 is given by

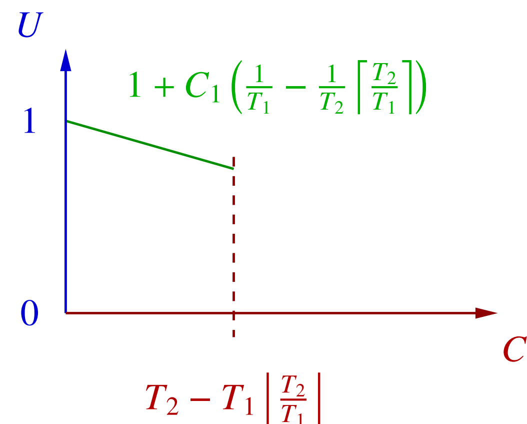
$$C_2 = T_2 - C_1 \left\lceil \frac{T_2}{T_1} \right\rceil.$$

It follows that the highest possible processor load factor is equal to

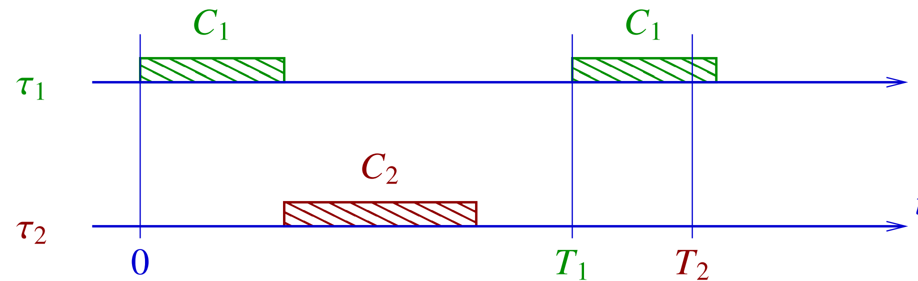
$$\begin{aligned} U &= \frac{C_1}{T_1} + \frac{C_2}{T_2} \\ &= 1 + C_1 \left(\frac{1}{T_1} - \frac{1}{T_2} \left\lceil \frac{T_2}{T_1} \right\rceil \right). \end{aligned}$$

Note that we have $\frac{1}{T_1} - \frac{1}{T_2} \left\lceil \frac{T_2}{T_1} \right\rceil \leq 0$.

Therefore, for given values of T_1 and T_2, the maximum processor load factor decreases with C_1.



- If an execution of τ_1 is still unfinished at $t = T_2$.



The following condition is satisfied:

$$C_1 > T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor .$$

For a given value of C_1, the largest possible value of C_2 is given by

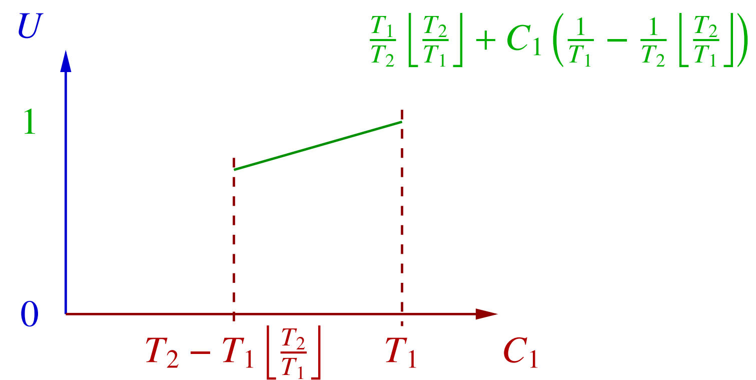
$$C_2 = (T_1 - C_1) \left\lfloor \frac{T_2}{T_1} \right\rfloor .$$

Hence, the highest possible processor load factor is equal to

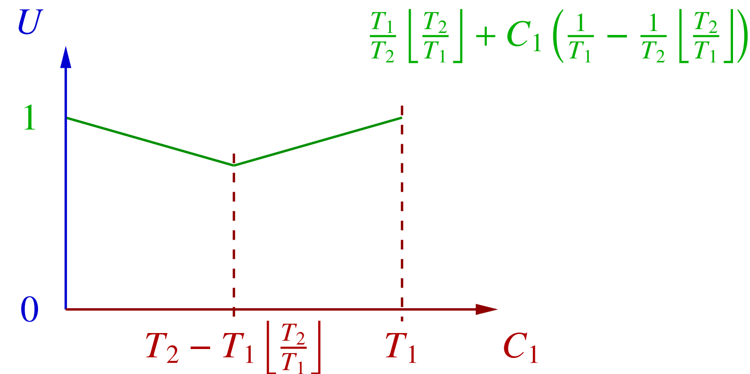
$$U = \frac{T_1}{T_2} \left\lfloor \frac{T_2}{T_1} \right\rfloor + C_1 \left(\frac{1}{T_1} - \frac{1}{T_2} \left\lfloor \frac{T_2}{T_1} \right\rfloor \right) .$$

For given values of T_1 and T_2, this expression increases with C_1, since

$$\frac{1}{T_1} - \frac{1}{T_2}\left\lfloor\frac{T_2}{T_1}\right\rfloor \geq 0.$$



Summary:



The smallest value of U corresponds to the boundary between the two cases, where we have

$$C_1 = T_2 - T_1\left\lfloor\frac{T_2}{T_1}\right\rfloor.$$

By introducing this value in the expression of U, one obtains

$$\begin{aligned} U &= \frac{T_1}{T_2}\left\lfloor\frac{T_2}{T_1}\right\rfloor + \left(T_2 - T_1\left\lfloor\frac{T_2}{T_1}\right\rfloor\right)\left(\frac{1}{T_1} - \frac{1}{T_2}\left\lfloor\frac{T_2}{T_1}\right\rfloor\right) \\ &= \frac{T_1}{T_2}\left\lfloor\frac{T_2}{T_1}\right\rfloor + \frac{T_2}{T_1} - 2\left\lfloor\frac{T_2}{T_1}\right\rfloor + \frac{T_1}{T_2}\left\lfloor\frac{T_2}{T_1}\right\rfloor^2. \end{aligned}$$

Let us define $I = \left\lfloor \frac{T_2}{T_1} \right\rfloor$ and $f = \frac{T_2}{T_1} - \left\lfloor \frac{T_2}{T_1} \right\rfloor$.

The previous expression becomes

$$\begin{aligned} U &= \frac{I}{I+f} + (I+f) - 2I + \frac{I^2}{I+f} \\ &= 1 - f\frac{1-f}{I+f}. \end{aligned}$$

The smallest possible value of U is obtained with $I = 1$. We then have

$$U = 1 - f\frac{1-f}{1+f},$$

and

$$\frac{dU}{df} = \frac{f^2 + 2f - 1}{(1+f)^2}.$$

The best lower bound U_L on U is thus obtained with $I = 1$ and $f = -1 + \sqrt{2}$:

$$U_L = 1 - (\sqrt{2} - 1)\left(\frac{2 - \sqrt{2}}{\sqrt{2}}\right) = 2(\sqrt{2} - 1) \approx 0.83.$$

Case of two tasks: Conclusions

Theorem 3: If a set of two periodic tasks has a processor load factor that is less than or equal to $2(\sqrt{2} - 1)$, then this set of tasks is schedulable.

Notes:

- This sufficient criterion is independent from the periods and execution times of the tasks.
- In the particular case where T_2 is an integer multiple of T_1, one has $f = 0$, hence

$$U_L = 1.$$

All pairs of tasks satisfying this condition (and such that $\frac{C_1}{T_1} + \frac{C_2}{T_2} \leq 1$!) are thus schedulable.

U_L: Case of n tasks

The goal is now to compute the value of U_L

- for a given number n of tasks, and
- for any number of tasks.

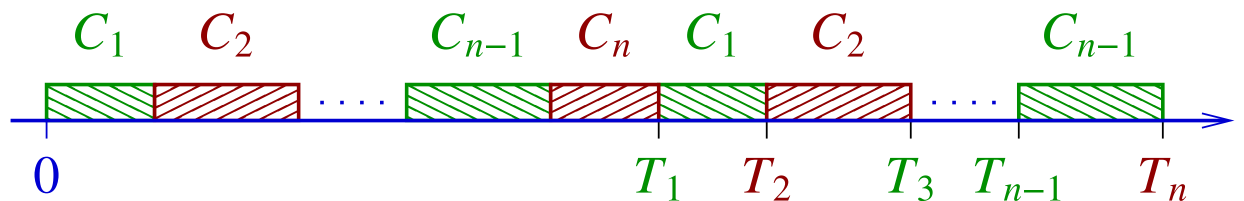
The first step is to establish an intermediate result:

Lemma 1: Let $\tau_1, \tau_2, \ldots, \tau_n$ be periodic tasks with the respective periods and execution times $T_1, T_2, \ldots, T_n$ and $C_1, C_2, \ldots, C_n$, such that

- This set of tasks fully uses the processor,
- $0 < T_1 < T_2 < \cdots < T_{n-1} < T_n < 2T_1$,
- The processor load factor of this set of tasks is minimum among all sets of tasks that fully use the processor.

In this case, one has

$$
\begin{aligned}
C_1 &= T_2 - T_1, \\
C_2 &= T_3 - T_2, \\
&\vdots \\
C_{n-1} &= T_n - T_{n-1}, \\
C_n &= T_n - 2(C_1 + C_2 + \cdots C_{n-1}) \\
&= 2T_1 - T_n.
\end{aligned}
$$

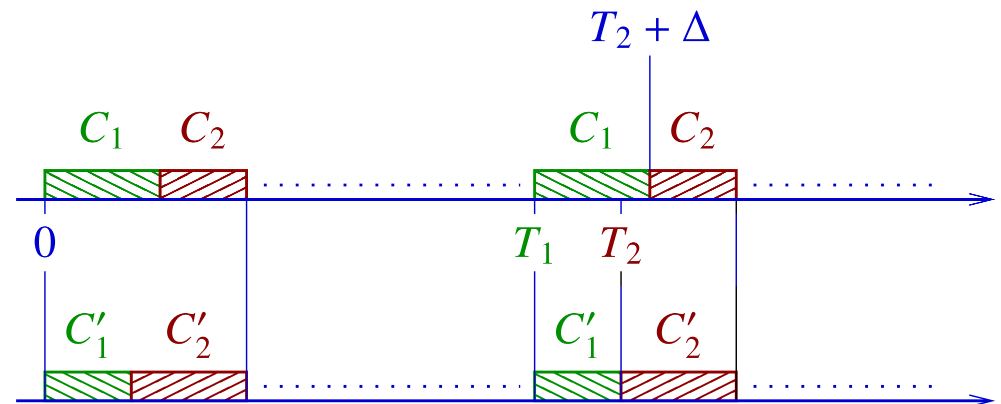


Proof: By contradiction, let us show that we must have $C_1 = T_2 - T_1$.

- If $C_1 = T_2 - T_1 + \Delta$, with $\Delta > 0$.

  We modify the execution time of tasks in the following way:

$$
\begin{aligned}
C'_1 &= C_1 - \Delta, \\
C'_2 &= C_2 + \Delta, \\
C'_3 &= C_3, \\
&\vdots \\
C'_{n-1} &= C_{n-1}, \\
C'_n &= C_n.
\end{aligned}
$$



After the modification, the new set of tasks still fully uses the processor. However, the processor load factor now becomes

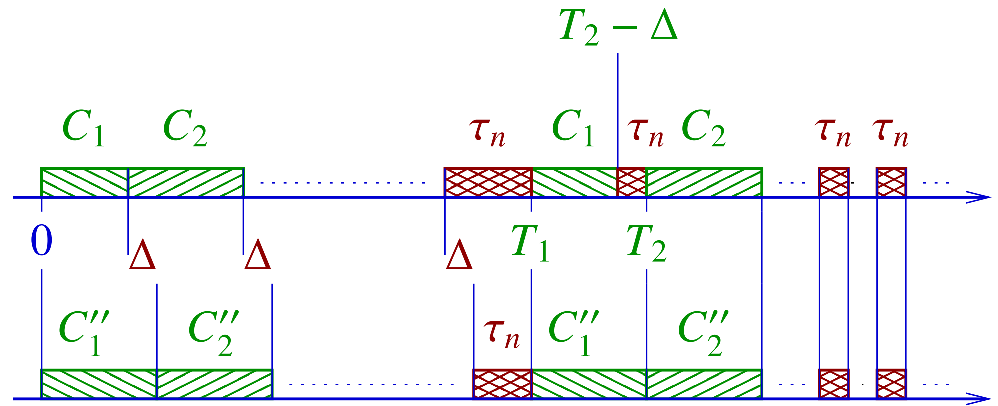
$$U' = U - \frac{\Delta}{T_1} + \frac{\Delta}{T_2} < U,$$

which contradicts the hypothesis that U is minimum.

- If $C_1 = T_2 - T_1 - \Delta$, with $\Delta > 0$.

  We now modify the execution time of tasks as follows:

$$\begin{aligned}
C_1'' &= C_1 + \Delta, \\
C_2'' &= C_2, \\
C_3'' &= C_3, \\
&\vdots \\
C_{n-1}'' &= C_{n-1}, \\
C_n'' &= C_n - 2\Delta.
\end{aligned}$$



The resulting set of tasks fully uses the processor. The processor load factor becomes

$$U' = U + \frac{\Delta}{T_1} - \frac{2\Delta}{T_n}.$$

Since we have by hypothesis $T_n < 2T_1$, this property contradicts $U' < U$.

By similar reasoning, one obtains successively

$$\begin{aligned} C_2 &= T_3 - T_2, \\ C_3 &= T_4 - T_3, \\ &\vdots \\ C_{n-1} &= T_n - T_{n-1}. \end{aligned}$$

Since the processor is fully used, one finally gets

$$C_n = T_n - 2(C_1 + C_2 + \cdots C_{n-1}).$$

Corollary: For each set of tasks that satisfies the hypotheses of Lemma 1, the processor load factor is equal to

$$\begin{aligned} U &= \frac{T_2 - T_1}{T_1} + \frac{T_3 - T_2}{T_2} + \cdots + \frac{T_n - T_{n-1}}{T_{n-1}} \\ &\quad + \frac{2T_1 - T_n}{T_n} \\ &= \frac{T_2}{T_1} + \frac{T_3}{T_2} + \cdots + \frac{T_n}{T_{n-1}} + 2\frac{T_1}{T_n} - n. \end{aligned}$$

For each $i = 1, 2, \ldots, n-1$, let us define $q_i = \dfrac{T_{i+1}}{T_i}$. We then have

$$U = q_1 + q_2 + \cdots + q_{n-1} + \frac{2}{q_1 q_2 \cdots q_{n-1}} - n,$$

and thus for each i,

$$\frac{\partial U}{\partial q_i} = 1 - 2\frac{q_1 q_2 \cdots q_{i-1} q_{i+1} \cdots q_{n-1}}{(q_1 q_2 \cdots q_{n-1})^2}.$$

The best lower bound U_L of U therefore corresponds to

$$\begin{aligned} \frac{\partial U}{\partial q_i} &= 0 \\ 1 - \frac{1}{q_i}.\frac{2}{q_1 q_2 \cdots q_{n-1}} &= 0. \end{aligned}$$

For each i, one has

$$q_i = \frac{2}{q_1 q_2 \cdots q_{n-1}},$$

hence

$$q_1 = q_2 = \cdots = q_{n-1} = 2^{\frac{1}{n}}.$$

By introducing these values in the expression of U, one obtains

$$\begin{aligned} U_L &= (n-1)2^{\frac{1}{n}} + \frac{2}{2^{\frac{n-1}{n}}} - n \\ &= (n-1)2^{\frac{1}{n}} + 2^{\frac{1}{n}} - n \\ &= n(2^{\frac{1}{n}} - 1). \end{aligned}$$

We thus have the following result:

Theorem 4: If the periods $T_1, T_2, \ldots, T_n$ of a set of n tasks are such that

$$0 < T_1 < T_2 < \cdots < T_{n-1} < T_n < 2T_1,$$

with a processor load factor that is less than or equal to $n(2^{\frac{1}{n}} - 1)$, then this set of tasks is schedulable.

In the hypotheses of Theorem 4, the constraint over the task periods is actually not necessary:

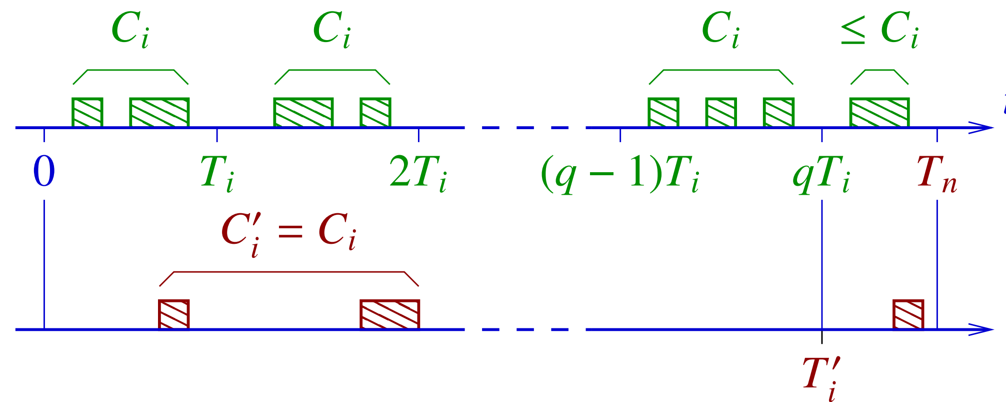
Theorem 5: If a set of n periodic tasks has a processor load factor that is less than or equal to $n(2^{\frac{1}{n}} - 1)$, then this set of tasks is schedulable.

Proof: Let $\tau_1, \tau_2, \ldots, \tau_n$ be tasks with respective periods and execution times $T_1, T_2, \ldots, T_n$ and $C_1, C_2, \ldots, C_n$. We assume that this set of tasks fully uses the processor.

If there exists $i \in \{1, 2, \ldots, n-1\}$ such that $2T_i \leq T_n$, then we define $q = \left\lfloor \frac{T_n}{T_i} \right\rfloor$ and $r = T_n - qT_i$ (we thus have $q > 1$ and $r \geq 0$).

We modify the set of tasks in the following way:

- We replace τ_i by τ'_i with the period $T'_i = qT_i$ and the execution time $C'_i = C_i$.
- We replace τ_n by τ'_n, with the period $T'_n = T_n$ and an execution time C'_n chosen so as to fully use the processor.



In the critical zone of τ_n, the amount of execution time used by τ_i and leaved unused by τ'_i is at most equal to $(q-1)C_i$. Therefore, one has

$$C'_n - C_n \leq (q-1)C_i.$$

After modifying the set of tasks, the processor load factor U' becomes equal to

$$U' \leq U + \frac{C'_i}{T'_i} - \frac{C_i}{T_i} + \frac{(q-1)C_i}{T_n}$$

where U is the processor load factor of the initial set of tasks.

One then obtains

$$U' \leq U + C_i\left(\frac{1}{qT_i} - \frac{1}{T_i} + \frac{q-1}{T_n}\right).$$

Since we have $qT_i \leq T_n$, this leads to

$$\begin{aligned} \frac{1}{qT_i} - \frac{1}{T_i} + \frac{q-1}{T_n} &\leq \frac{1}{qT_i} - \frac{1}{T_i} + \frac{q-1}{qT_i} \\ &\leq 0. \end{aligned}$$

As a consequence, we have $U' \leq U$. This implies that our modification of the set of tasks did not increase the processor load factor.

By repeatedly performing such a modification, one eventually obtains a set of tasks to which Theorem 4 can be applied.

The limit processor load factor

The value of U_L decreases with the number n of tasks. Indeed,

$$\begin{aligned}\frac{dU_L}{dn} &= \left(1 - \frac{\ln 2}{n}\right) 2^{\frac{1}{n}} - 1 \\ &= (1 - x)e^x - 1,\end{aligned}$$

by defining $x = \frac{\ln 2}{n}$. Let us show that we have

$$(1 - x)e^x < 1$$

for all $x > 0$ (which implies $\frac{dU_L}{dn} < 0$ for all $n > 0$).

For all $x > 0$, we have $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$, hence

$$\begin{aligned}(1 - x)e^x &= (1 - x) + (1 - x)x + (1 - x)\frac{x^2}{2!} + (1 - x)\frac{x^3}{3!} + \cdots \\ &= 1 - \left(1 - \frac{1}{2!}\right) x^2 - \left(\frac{1}{2!} - \frac{1}{3!}\right) x^3 - \left(\frac{1}{3!} - \frac{1}{4!}\right) x^4 + \cdots \\ &< 1.\end{aligned}$$

For an asymptotically large number of tasks, we obtain

$$\begin{aligned}
\lim_{n\to\infty} U_L(n) &= \lim_{n\to\infty} n(2^{\frac{1}{n}} - 1) \\
&= \lim_{n\to\infty} \frac{2^{\frac{1}{n}} - 1}{\frac{1}{n}} \\
&= \lim_{n\to\infty} \frac{\frac{\ln 2}{n^2} 2^{\frac{1}{n}}}{\frac{1}{n^2}} \\
&= \ln 2 \\
&\approx 0.69
\end{aligned}$$

In summary, we have the following result:

Theorem 6: If a set of periodic tasks has a processor load factor that is less than or equal to ln 2, then this set of tasks is schedulable.

Conclusion: The following algorithm can be used for checking efficiently whether a set of n periodic tasks with a processor load factor equal to U is schedulable or not:

1. If $U > 100\%$, then the set of tasks is not schedulable;
2. If $U \leq 69\%$, then the set of tasks is schedulable;
3. If $U \leq n(2^{\frac{1}{n}} - 1)$, then the set of tasks is schedulable;
4. Otherwise, one performs an exact scheduling simulation, based on a RMS priorities assignment.

Notes

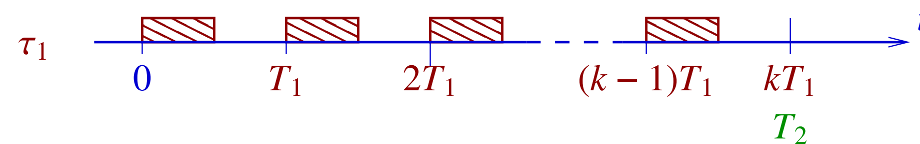
- In situations where $U \leq 69\%$ for the periodic tasks, the processor does not have to remain unused during 31% of the time! One can instead run low-priority tasks that are not bound by real-time constraints.
- For some specific class of sets of tasks, one can obtain $U_L = 100\%$, which guarantees that every set of tasks for which $U \leq 100\%$ is schedulable.

  Example: Let $\tau_1, \tau_2, \ldots, \tau_n$ be a set of tasks with respective periods and execution times $T_1, T_2, \ldots, T_n$ and $C_1, C_2, \ldots, C_n$, such that

  - $0 < T_1 \leq T_2 \leq \cdots \leq T_n$,
  - $\forall i, j \ : i < j \ \Rightarrow T_j$ is an integer multiple of T_i,
  - $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$.

Let us show that this set of tasks is schedulable.

The critical zone of τ_2 contains $\dfrac{T_2}{T_1}$ complete executions of τ_1:



Similarly, for each $j \in \{2, 3, \ldots, n\}$, the critical zone of τ_j contains

$\dfrac{T_j}{T_1}$ complete executions of τ_1,

$\vdots$

$\dfrac{T_j}{T_{j-1}}$ complete executions of τ_{j-1}.

The condition that must be satisfied in order to finish the execution of τ_j before the end of its critical zone is thus

$$C_j \leq T_j - \frac{T_j}{T_1}C_1 - \frac{T_j}{T_2}C_2 - \cdots - \frac{T_j}{T_{j-1}}C_{j-1}$$

After simplification, this condition becomes

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_j}{T_j} \leq 1,$$

which immediately follows from the hypothesis $U \leq 1$.

Chapter 8

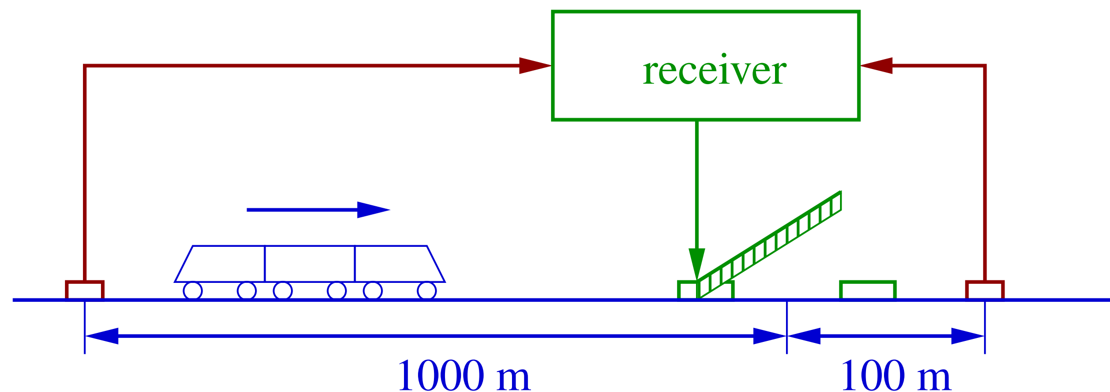
Complex timed systems

Introduction

In order to analyze the properties of a complex system, it is not always sufficient to study the individual behavior of its components.

Example: An embedded system controlling a railroad crossing is composed of the following elements:

- Two sensors located on the tracks 1000 meters before and 100 meters after the crossing, aimed at detecting (respectively) that a train approaches or has passed the crossing.
- A receiver that processes the signals emitted by the sensors, and sends orders to open or close the gate.



The following information is known:

- The speed of the approaching trains is between 48 and 52 m/s. Then, after reaching the first sensor, their speed is reduced to a value between 40 and 52 m/s.
- After it receives a signal from a sensor, the receiver waits for at most 5 seconds before sending an order to close or to open the gate. During this delay, the receiver ignores incoming signals.
- The gate is closed (resp. open) when its angle is equal to 0 (resp. 90) deg. The gate is able to move at the rate of 20 deg/s.
- Two successive trains are always separated by at least 1600 m.

Question: Is the gate always closed when a train passes the crossing?

Modeling a system

In order to analyze the properties of a system, the first step consists in building a model, i.e., an abstract representation of the system that describes its relevant properties without any ambiguity.

For embedded applications, the modeling formalism must be able to express

- operations on integer variables (used as counters, sequence numbers, identifiers, . . .), as well as on real variables (for representing positions, speeds, delays, . . .).
- discrete state transitions (e.g., incrementing a counter) as well as continuous evolution laws (e.g., constant-speed movement).
- composition of elementary systems into a more complex entity.

Hybrid systems

Hybrid systems are a modeling formalism that meets those requirements.

Syntax:

A hybrid system is composed of:

- a finite number p of processes $P_1, P_2, \ldots, P_p$,
- a finite number n of variables $x_1, x_2, \ldots, x_n$, grouped together into a vector $\vec{x} \in \mathbf{R}^n$,
- a finite set L of synchronization labels.

Each process P_i is represented by a graph (V_i, E_i), where

- V_i est a finite set of control locations,
- $E_i \subseteq V_i \times V_i$ is a finite set of transitions.

Each control location $v \in V_i$ is associated with:

- An activity *dif*(v), expressed as a conjunction of linear constraints over the variables $x_1, x_2, \ldots, x_n$ and their first temporal derivative $\dot{x}_1, \dot{x}_2, \ldots, \dot{x}_n$.
- An invariant *inv*(v), expressed as a conjunction of linear constraints over the variables $x_1, x_2, \ldots, x_n$.

Each transition $e \in E_i$ is associated with:

- A guard *guard*(e), that represents a condition that must be satisfied in order to enable this transition.
- An action *act*(e), composed of constraints that specify how the values of the variables are modified when this transition is followed.

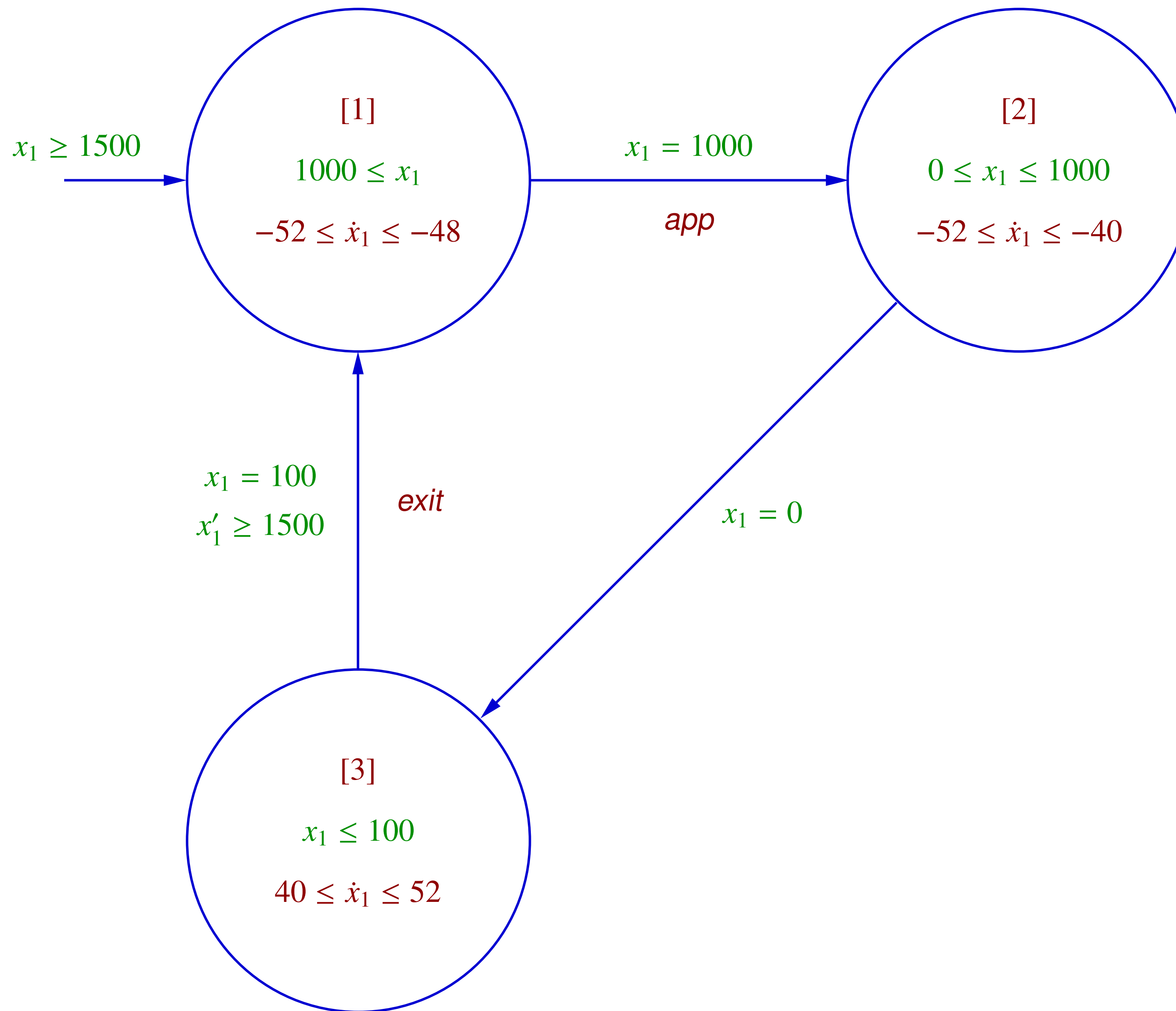
  In practice, the guard and the action can be combined into a conjunction of constraints over the values of the variables before $(x_1, x_2, x_3, \ldots)$ and after $(x'_1, x'_2, x'_3, \ldots)$ following the transition.

- An optional label *sync*$(e) \in L$ that makes it possible to synchronize this transition with a transition belonging to another process.

Finally, one defines an initial control location for each process, and assigns a set of possible initial values for each variable, specified as a conjunction of linear constraints.

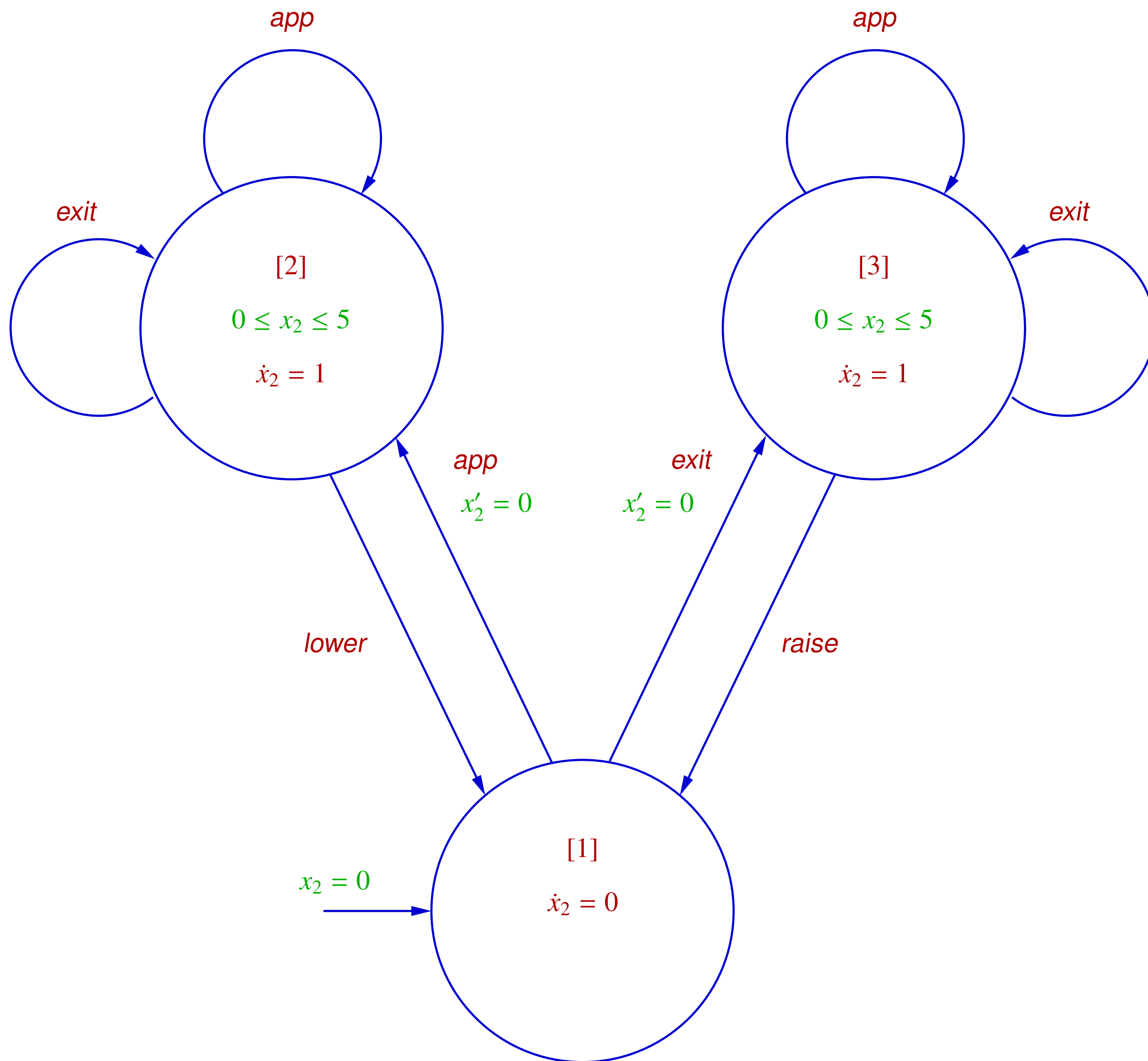
Example: Process modeling the behavior of a train and the two sensors.

- The distance between the train and the crossing is represented by a variable x_1.
- The signals emitted by the sensors are modeled by two synchronization labels *app* and *exit*.


$x_1 \geq 1500$
[1]
$1000 \leq x_1$
$-52 \leq \dot{x}_1 \leq -48$
$x_1 = 1000$
app
[2]
$0 \leq x_1 \leq 1000$
$-52 \leq \dot{x}_1 \leq -40$
$x_1 = 100$
$x_1' \geq 1500$
exit
$x_1 = 0$
[3]
$x_1 \leq 100$
$40 \leq \dot{x}_1 \leq 52$

Process modeling the receiver:

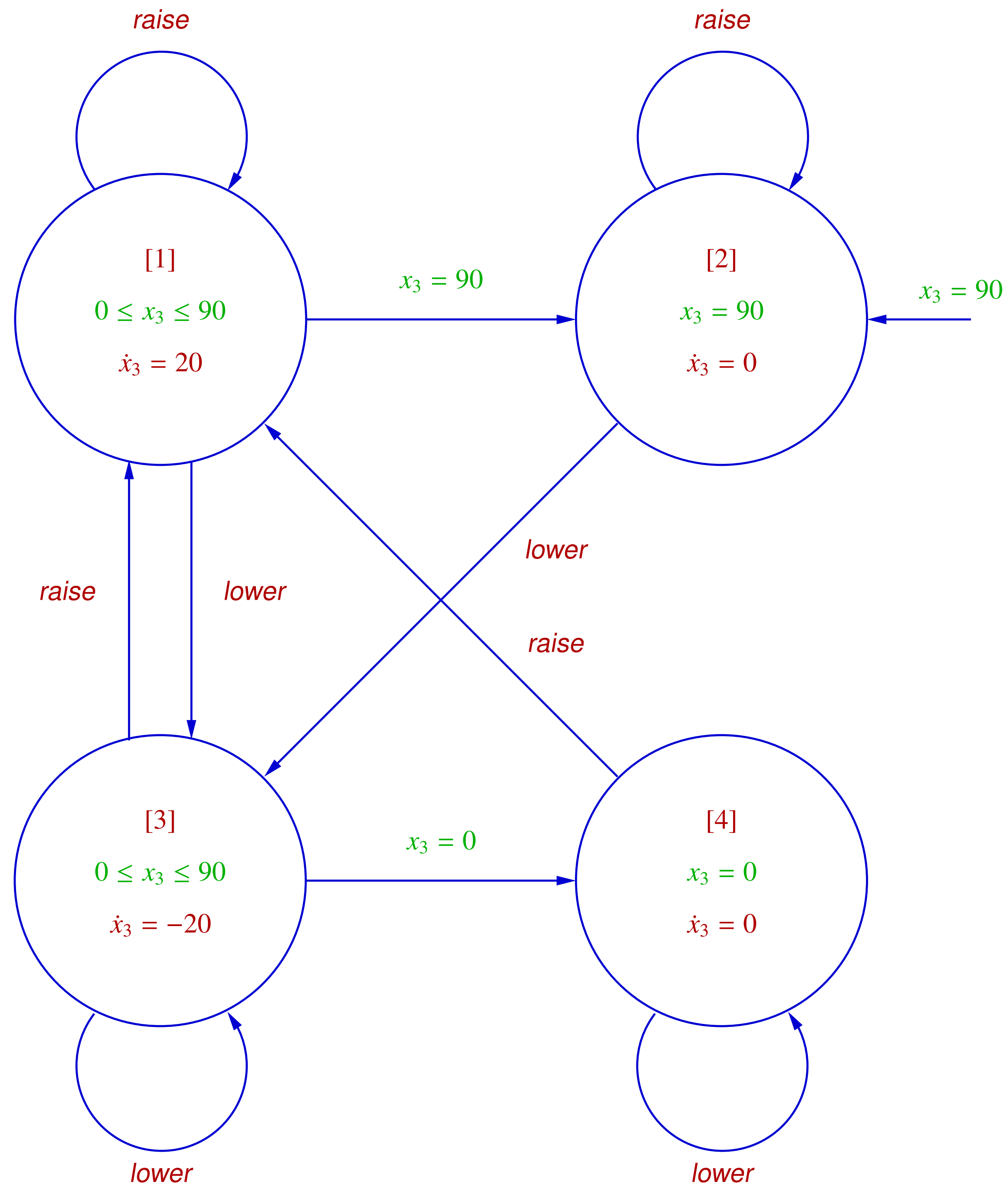
- The delay between receiving a sensor signal and sending an order to the gate is represented by a variable x_2.
- The labels *raise* and *lower* model the orders sent to the gate.



app
app
exit
exit
[2]
$0 \leq x_2 \leq 5$
$\dot{x}_2 = 1$
[3]
$0 \leq x_2 \leq 5$
$\dot{x}_2 = 1$
app
$x_2' = 0$
exit
$x_2' = 0$
lower
raise
$x_2 = 0$
[1]
$\dot{x}_2 = 0$

Process modeling the gate:

- The variable x_3 represents the angular position of the gate.
- The labels *raise* and *lower* correspond to the orders received.


raise
raise
[1]
$0 \le x_3 \le 90$
$\dot{x}_3 = 20$
$x_3 = 90$
[2]
$x_3 = 90$
$\dot{x}_3 = 0$
$x_3 = 90$
lower
raise
lower
raise
[3]
$0 \le x_3 \le 90$
$\dot{x}_3 = -20$
$x_3 = 0$
[4]
$x_3 = 0$
$\dot{x}_3 = 0$
lower
lower

Semantics:

At any given time, the current state of a hybrid system is characterized by

- a control location for each process, and
- a value for each variable.

The state of a system can evolve in two ways:

- By letting time elapse (*time steps*). The control locations of processes stay unchanged, and the values of the variables evolve according to the invariants and activities associated to these locations.
- By following transitions (*transition steps*). One can either
  - follow a single unlabeled transition, or
  - follow a pair of transitions belonging to different processes and sharing the same synchronization label.

In both cases, a transition can only be followed provided that its guard is satisfied by the current variable values.

When a transition is followed, the variable values are modified according to the action associated to the transition. The invariant of the destination location must be satisfied by the new variable values (otherwise, the transition cannot be followed).

A state s_2 is reachable from a state s_1 if there exists a finite sequence of time steps and transition steps that lead from s_1 to s_2.

A state s is reachable if it is reachable from an initial state.

Example: The state $([2],[2],[2],800,4,90)$ of the railroad crossing controller model corresponds to

- the control location $[2]$ for each process.
- the respective values 800, 4 and 90 for the variables x_1, x_2 and x_3.

This state is reachable. Indeed, one has

$$
\begin{aligned}
& ([1],[1],[2],1500,0,90) \\
\stackrel{10}{\Longrightarrow}\ & ([1],[1],[2],1000,0,90) \\
\stackrel{\textit{app}}{\longrightarrow}\ & ([2],[2],[2],1000,0,90) \\
\stackrel{4}{\Longrightarrow}\ & ([2],[2],[2],800,4,90),
\end{aligned}
$$

where

- "$\stackrel{\lambda}{\Longrightarrow}$" denotes a time step with a delay equal to λ,
- "$\stackrel{\ell}{\longrightarrow}$" corresponds to following a pair of transitions sharing the synchronization label ℓ.

Executions of a hybrid system

An execution of a hybrid system is an infinite sequence $s_1, s_2, s_3, \ldots$ of states such that:

- s_1 is an initial state of the system.
- For each i, the state s_{i+1} is reachable from the state s_i in a time $\delta_i \geq 0$.

Note: A hybrid system generally admits several different executions (*non-determinism*). Indeed,

- The time spent at a control location may not be precisely constrained by the invariant.
- A control location can have several outgoing transitions enabled at a given time.

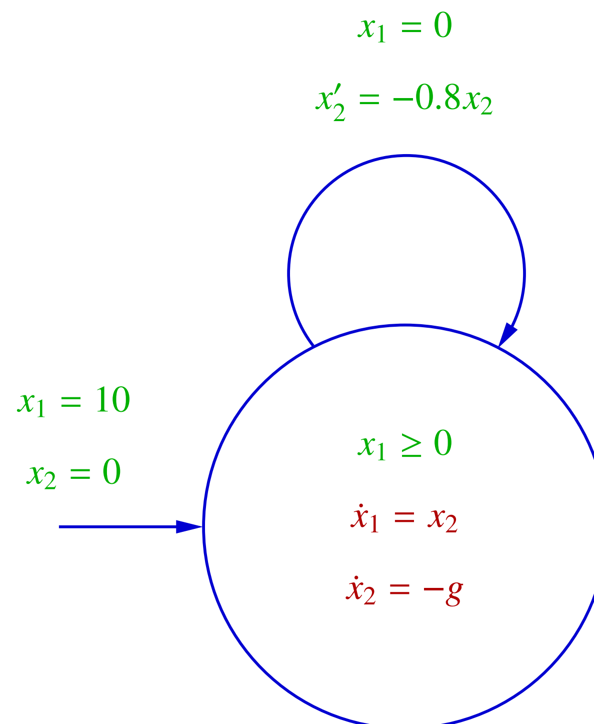
An execution $s_1, s_2, s_3, \ldots$ beginning at time $t = 0$ is said to be divergent if for every $T > 0$, there exists i such that the state s_i is reached later than time $t = T$.

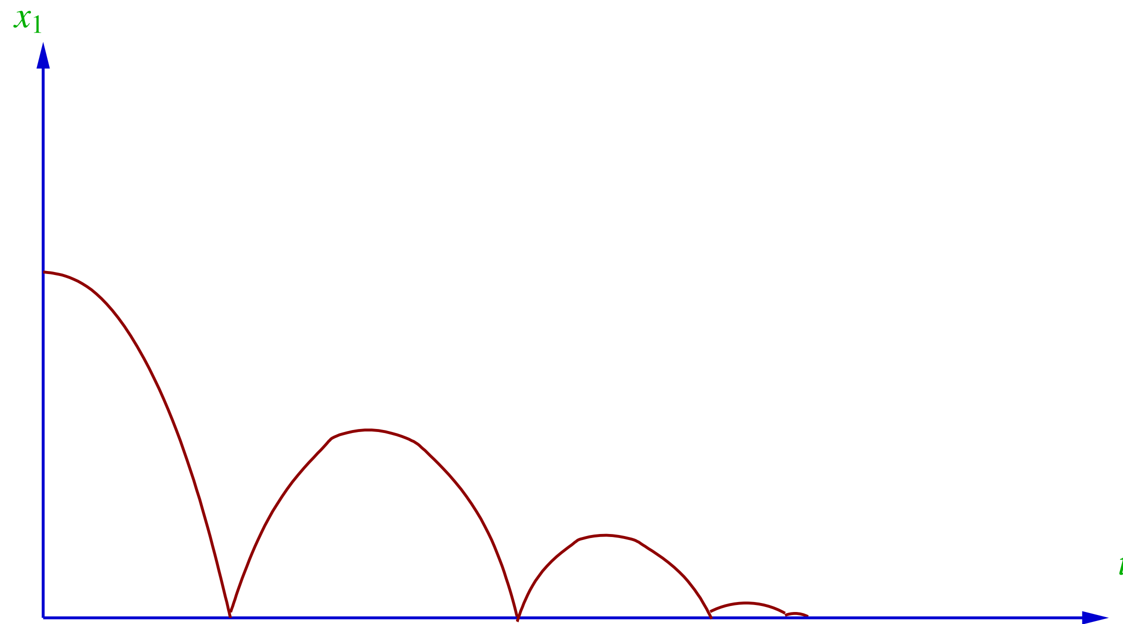
Zeno hybrid systems

A hybrid system is said to have the Zeno property if it admits an execution in which at least one finite prefix is not a prefix of a divergent execution.

In other words, in a Zeno hybrid system, there exists a reachable state from which no execution is able to get past some time bound.

Example: Hybrid system modeling a bouncing ball.





Remarks:

- Such models are inconsistent with physical reality and must be avoided!
- For some restricted classes of hybrid systems, automatic methods have been developed for transforming any given model into another one that does not have the Zeno property, and admits the same divergent executions.

State-space exploration

A large number of interesting properties of a hybrid system can be checked by computing its reachable states.

This computation can be carried out by building, from every initial state, a tree in which each node q represents a reachable state $s(q)$, and the children of q correspond to the states that are reachable from $s(q)$ by

- a time step, or
- a transition step.

Problems:

- The system may have infinitely many initial states.
- The time spent at a control location may take an infinite number of possible values, which leads to trees of infinite degree.
- Since executions are infinite, the trees also have an infinite depth.

Solutions:

- Sets of states sharing the same control locations and differing only in the elapsed time in those locations can be grouped into regions. A tree can be built in which the nodes are associated with regions instead of individual states.
- At each exploration step, a first operation saturates the current region by letting time elapse during all possible delays. Then, the enabled transitions are individually followed, creating new branches.
- The branches of the exploration tree that only contain already visited states can be pruned.

Notes:

- Several exploration strategies are possible: depth-first search (DFS), breadth-first search (BFS), . . .

- For general hybrid systems, the region tree can still be infinite. It is however possible to define restricted classes of models, for which a finite region tree can always be computed.

  Example: Timed automata are hybrid systems in which

  - the activities are of the form $\dot{x}_i = 1$,
  - all invariants, guards and actions are conjunctions of constraints of the form $x_i \# c$ or $x_i - x_j \# c$, where c is an integer number, and $\# \in \{<, \leq, =, \geq, >\}$.

- Some tools are available for exploring automatically the state space of hybrid systems (e.g., *HyTech*) or timed automata (e.g., *Uppaal*).

  Notes: These tools

  - represent and handle regions with the help of dedicated data structures, based on logic formulas, convex polyhedra, difference matrices, . . .
  - are able to check properties that go beyond simple reachability.

Example: Railroad crossing

$$([1],[1],[2]) : x_1 \geq 1500,\ x_2 = 0,\ x_3 = 90.$$

$$\begin{array}{llll}
\Longrightarrow & ([1],[1],[2]) & : & x_1 \geq 1000, \\
 & & & x_2 = 0,\ x_3 = 90. \\
\xrightarrow{app} & ([2],[2],[2]) & : & x_1 = 1000, \\
 & & & x_2 = 0,\ x_3 = 90. \\
\overset{\leq 5}{\Longrightarrow} & ([2],[2],[2]) & : & x_1 \geq 1000 - 52\lambda, \\
 & & & x_1 \leq 1000 - 40\lambda, \\
 & & & x_2 = \lambda,\ x_3 = 90, \\
 & & & 0 \leq \lambda \leq 5. \\
\xrightarrow{lower} & ([2],[1],[3]) & : & x_1 \geq 1000 - 52\lambda, \\
 & & & x_1 \leq 1000 - 40\lambda, \\
 & & & x_2 = \lambda,\ x_3 = 90, \\
 & & & 0 \leq \lambda \leq 5. \\
\overset{\leq 9/2}{\Longrightarrow} & ([2],[1],[3]) & : & x_1 \geq 1000 - 52(\lambda + \mu), \\
 & & & x_1 \leq 1000 - 40(\lambda + \mu), \\
 & & & x_2 = \lambda,\ x_3 = 90 - 20\mu, \\
 & & & 0 \leq \lambda \leq 5,\ 0 \leq \mu \leq 9/2.
\end{array}$$

$$
\begin{array}{llcl}
\xrightarrow{x_3=0} & ([2],[1],[4]) & : & x_1 \geq 766 - 52\lambda, \\
 & & & x_1 \leq 820 - 40\lambda, \\
 & & & x_2 = \lambda,\ x_3 = 0, \\
 & & & 0 \leq \lambda \leq 5. \\
\Longrightarrow & ([2],[1],[4]) & : & 0 \leq x_1 \leq 820 - 40\lambda, \\
 & & & x_2 = \lambda,\ x_3 = 0, \\
 & & & 0 \leq \lambda \leq 5. \\
\xrightarrow{x_1=0} & ([3],[1],[4]) & : & x_1 = 0,\ x_2 = \lambda, \\
 & & & x_3 = 0,\ 0 \leq \lambda \leq 5. \\
\overset{\leq 5/2}{\Longrightarrow} & ([3],[1],[4]) & : & 0 \leq x_1 \leq 100, \\
 & & & x_2 = \lambda,\ x_3 = 0, \\
 & & & 0 \leq \lambda \leq 5. \\
\xrightarrow{\textit{exit}} & ([1],[3],[4]) & : & x_1 \geq 1500,\ x_2 = 0, \\
 & & & x_3 = 0. \\
\overset{\leq 5}{\Longrightarrow} & ([1],[3],[4]) & : & x_1 \geq 1500 - 52\lambda, \\
 & & & x_2 = \lambda,\ x_3 = 0, \\
 & & & 0 \leq \lambda \leq 5. \\
\xrightarrow{\textit{raise}} & ([1],[1],[1]) & : & x_1 \geq 1500 - 52\lambda, \\
 & & & x_2 = \lambda,\ x_3 = 0, \\
 & & & 0 \leq \lambda \leq 5.
\end{array}
$$

$$\overset{\leq 9/2}{\Longrightarrow} \quad ([1],[1],[1]) \quad : \quad \begin{array}{l} x_1 \geq 1500 - 52(\lambda + \mu), \\ x_2 = \lambda,\ x_3 = 20\mu, \\ 0 \leq \lambda \leq 5,\ 0 \leq \mu \leq 9/2. \end{array}$$

$$\overset{x_3=90}{\longrightarrow} \quad ([1],[1],[2]) \quad : \quad \begin{array}{l} x_1 \geq 1266 - 52\lambda, \\ x_2 = \lambda,\ x_3 = 90, \\ 0 \leq \lambda \leq 5. \end{array}$$

$$\Longrightarrow \quad ([1],[1],[2]) \quad : \quad \begin{array}{l} x_1 \geq 1000, \\ x_2 = \lambda,\ x_3 = 90, \\ 0 \leq \lambda \leq 5. \end{array}$$

$$\overset{\textit{app}}{\longrightarrow} \quad ([2],[2],[2]) \quad : \quad \begin{array}{l} x_1 = 1000, \\ x_2 = 0,\ x_3 = 90 \\ \text{(already obtained).} \end{array}$$

Notes:

- In this example, the regions correspond to the sets of states obtained after each time-step operation (denoted by "$\Longrightarrow$").
- Checking whether the gate is always closed when a train reaches the crossing amounts to verifying that in each reachable region, $x_1 = 0$ implies $x_3 = 0$.
- This particular system shows a very deterministic behavior: In each reachable state, there is at most one transition (or a pair of synchronized transitions) that is enabled.

  (This is generally not the case!)