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(s), 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.

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

- They are usable on hardware with limited resources.

- The scheduling strategy is precisely documented.
- The internal mechanisms (e.g., the longest interval during which interrupts are disabled by the kernel, the implementation of system calls, . . .) are engineered so as to minimize latencies.
- 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.
- Dynamic memory allocation is usually optional.
- 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

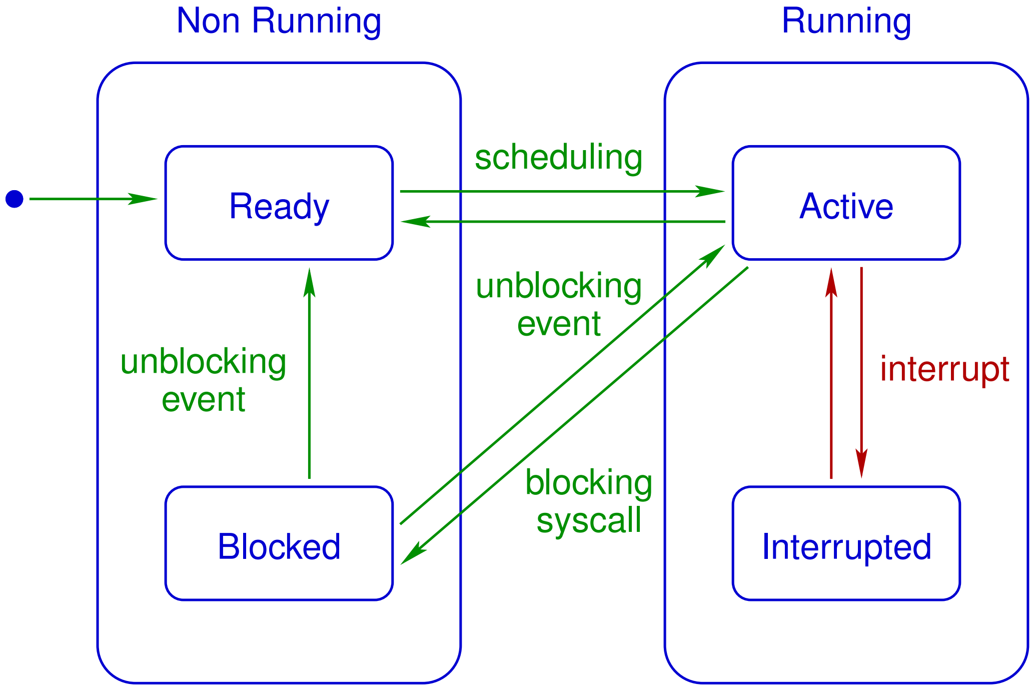
- a kernel operation (possibly located in an interrupt routine),
- an instruction belonging to an interrupt routine programmed by the user, or
- an instruction of a user task.

Process states

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

- Ready: 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, a timeout, 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 processes.

Principles:

- Each task is characterized by a priority (either constant or variable during its execution).
- The scheduler always assigns the processor to one of the running tasks with the highest priority.

If several tasks share the highest priority, then the task that is selected can be chosen 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.

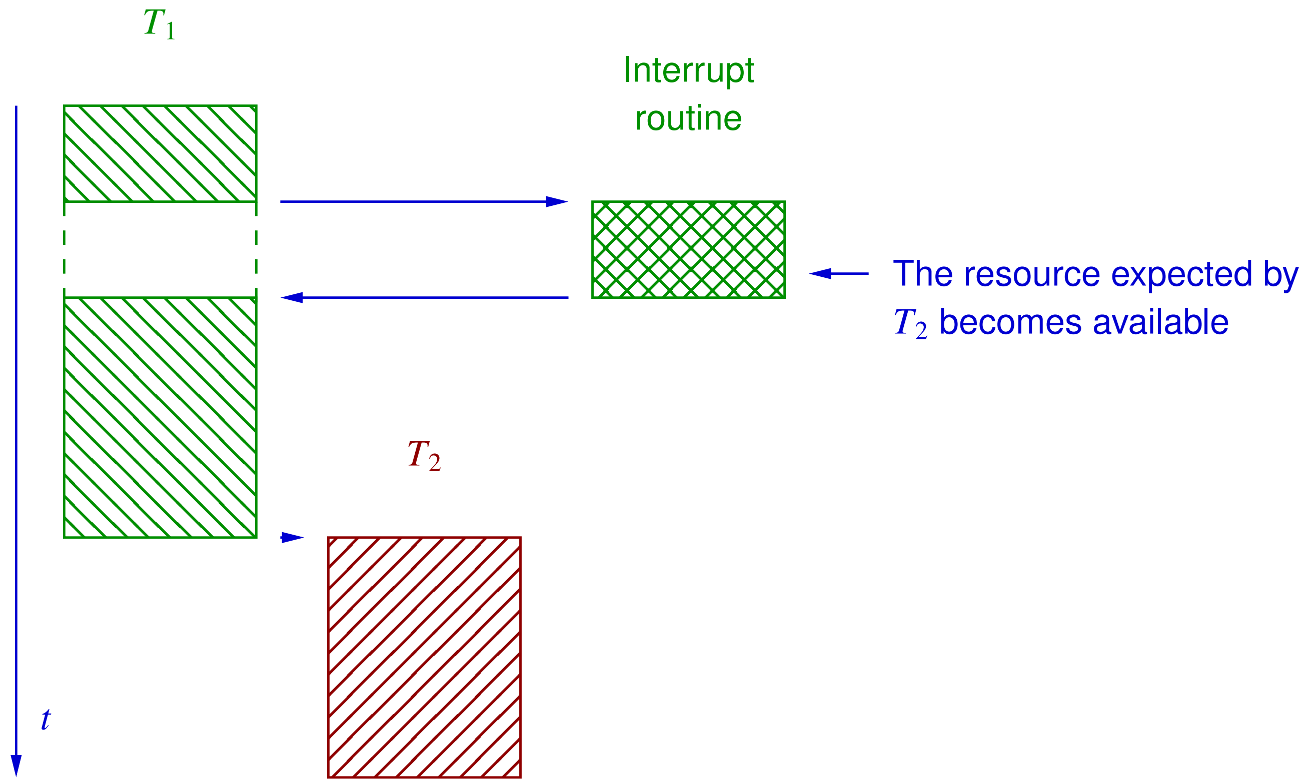
- One can alternatively assign the processor to a task that is chosen arbitrarily.
- Another solution is to forbid different tasks to share the same priority.

Note: With the first two strategies, computing the deadline of a task can become difficult.

Preemption

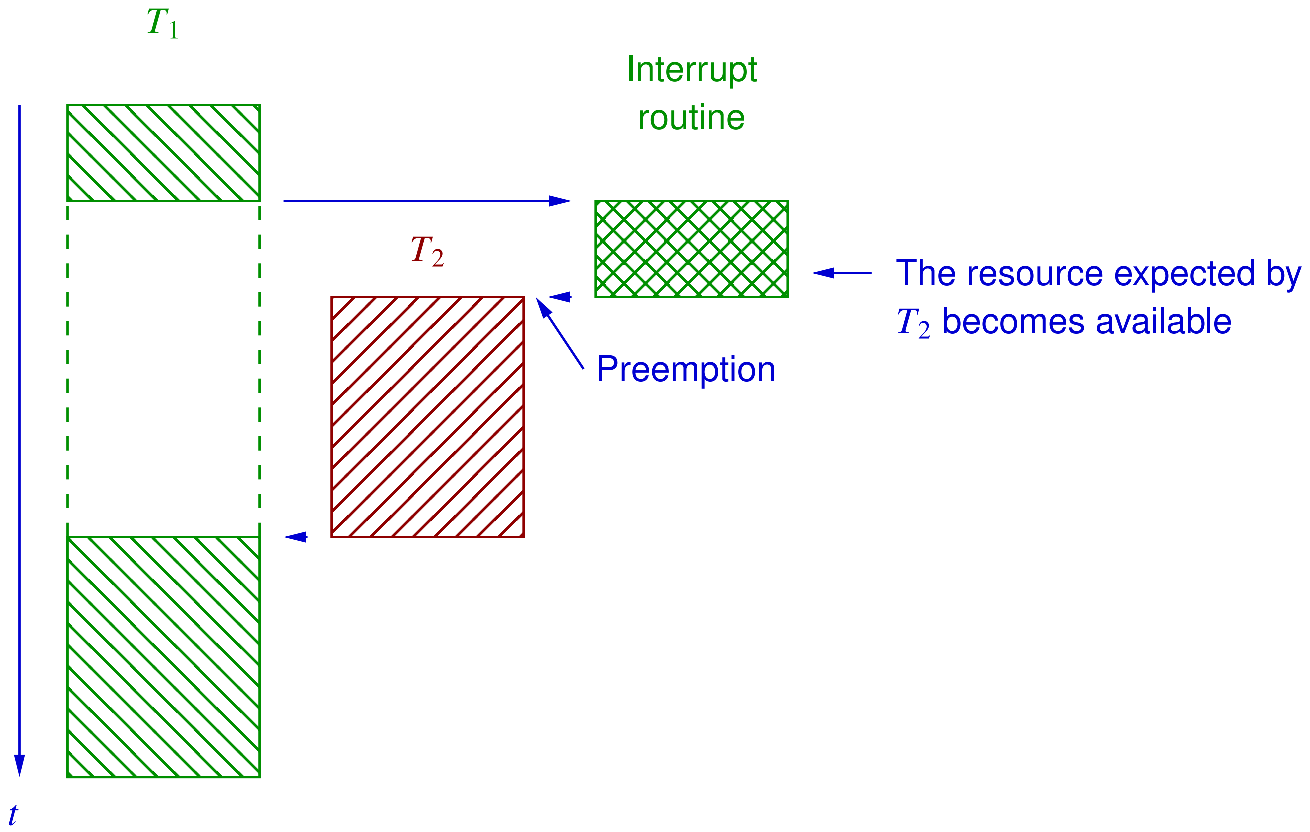
If a task T_2 has a higher priority than the active task T_1 and switches from the blocked to the ready state, then there are two possible scheduling strategies:

- The task T_2 remains non running (in ready 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 ready, and assigns the processor to T_2. The scheduler is said to be preemptive.



Context switching

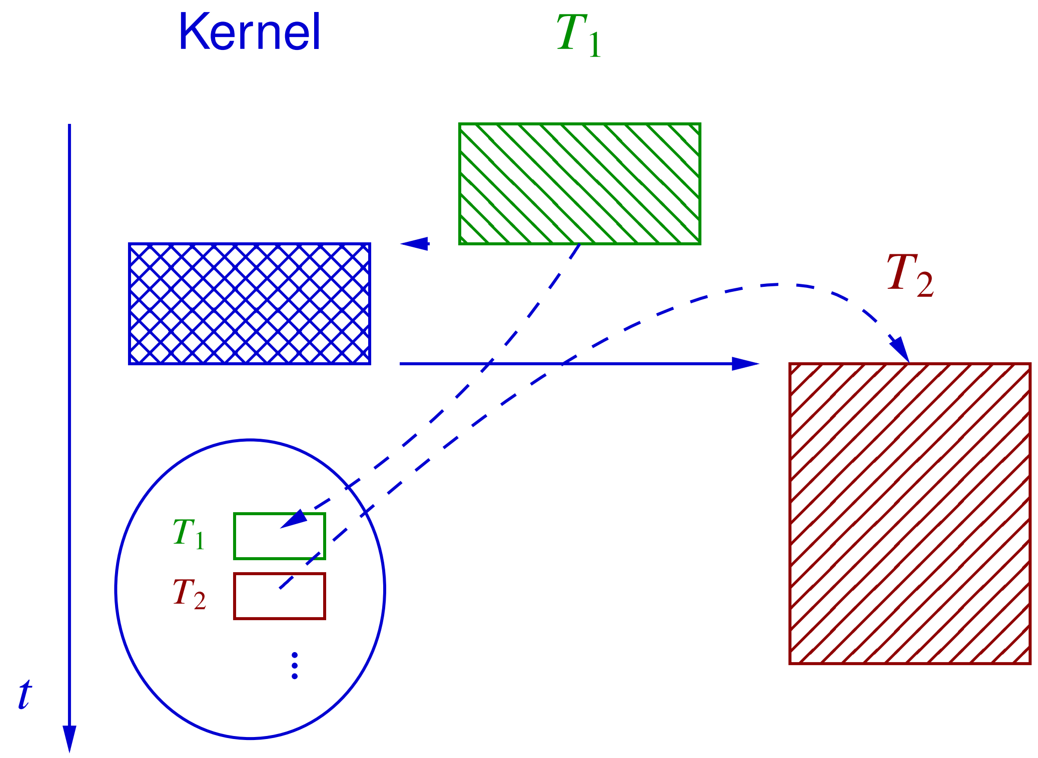
The scheduler performs a context switch when it transfers the processor from a process to another.

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 becomes non running.

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

Illustration:



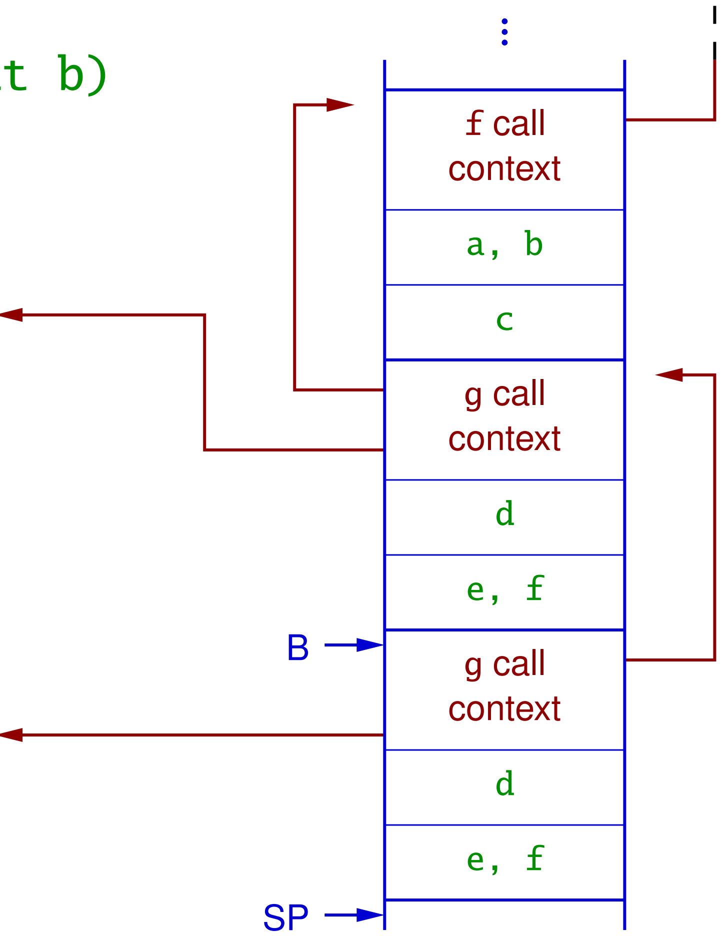
- The working data of the suspended task has to be preserved until its execution can be resumed.

  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:

```
f(int a, int b)
{
   int c;
      ⋮
   c = g(a);
      ⋮
}
g(int d)
{
   int e, f;
      ⋮
   e = g(f);
      ⋮
}
```



Notes:

- Since a task can become non running 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.

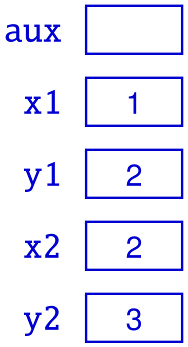
Reentrancy

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

Example:

```
static int aux;

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



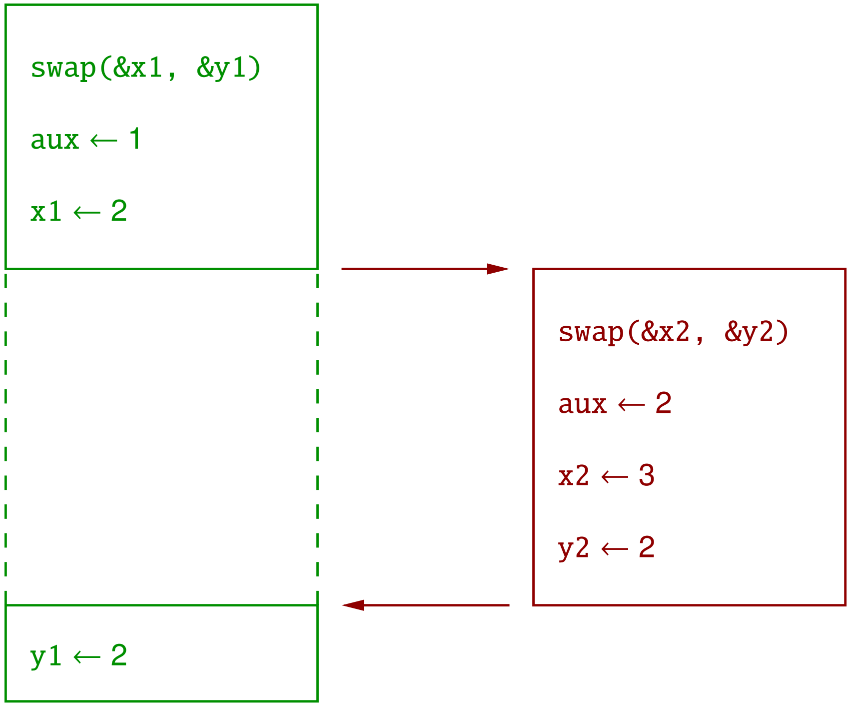
aux

x1 1

y1 2

x2 2

y2 3



`swap(&x1, &y1)`

aux ← 1

x1 ← 2

`swap(&x2, &y2)`

aux ← 2

x2 ← 3

y2 ← 2

y1 ← 2

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 blocked waiting for resources that can only be provided by other tasks.

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 blocked as the result of an operation *wait(s)*, unblock one of them (making it ready or active);
    - 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 unblock it: arbitrary choice, FIFO policy, highest priority, . . .

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;
     }                                   }
}                                   }
```

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 sent to and not yet received from the queue) 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, highest priority.

- 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 timeout (i.e., a maximum suspension delay) can be specified.
- The maximum capacity of a queue can be reduced to zero (rendez-vous synchronization).
- Queues of capacity one are sometimes called mailboxes.

Programming with interrupts

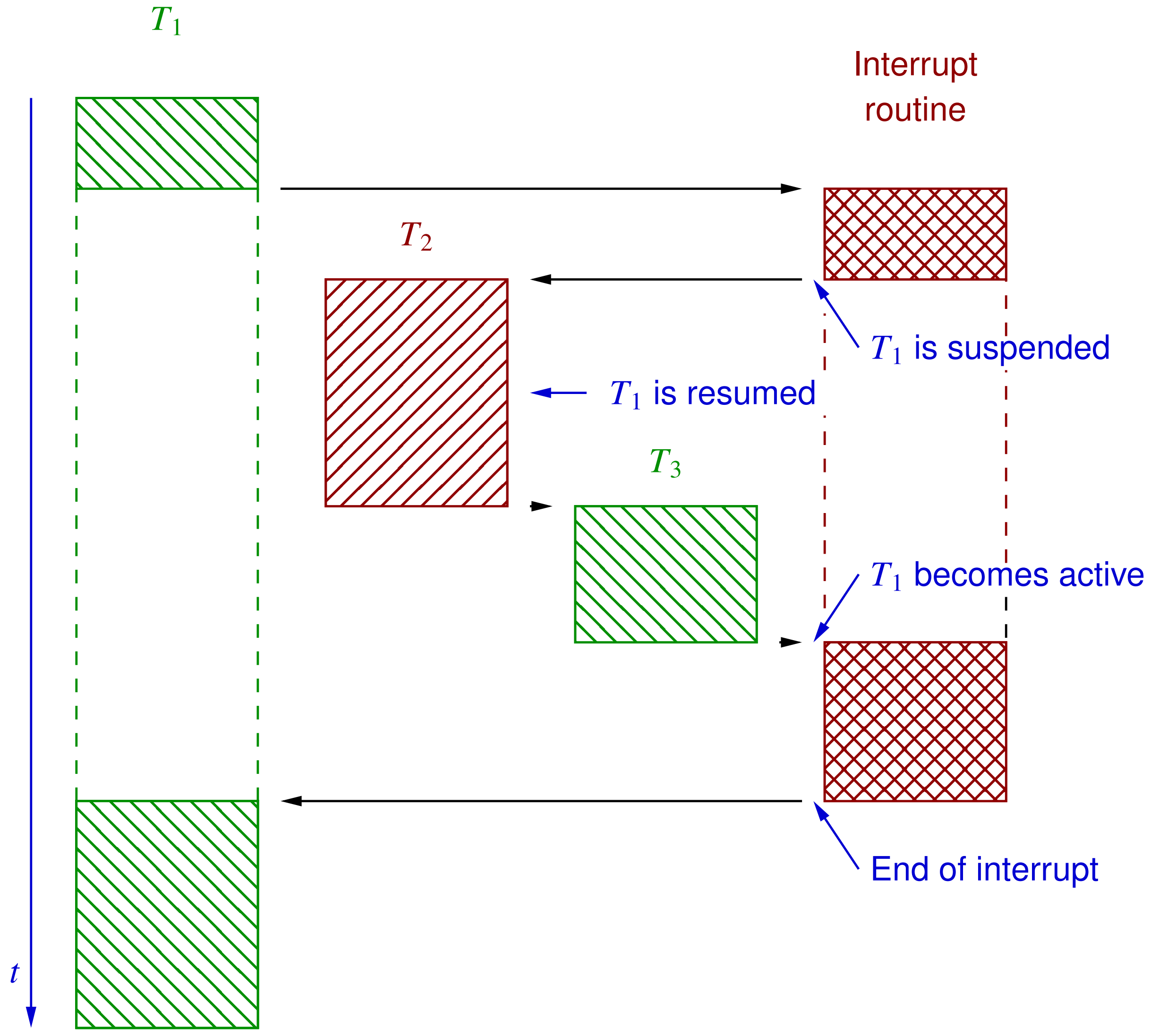
The scheduler and the interrupt mechanism are both able to move the control point from one location in the program code to another. One must take care of avoiding conflicts between those mechanisms.

First rule:

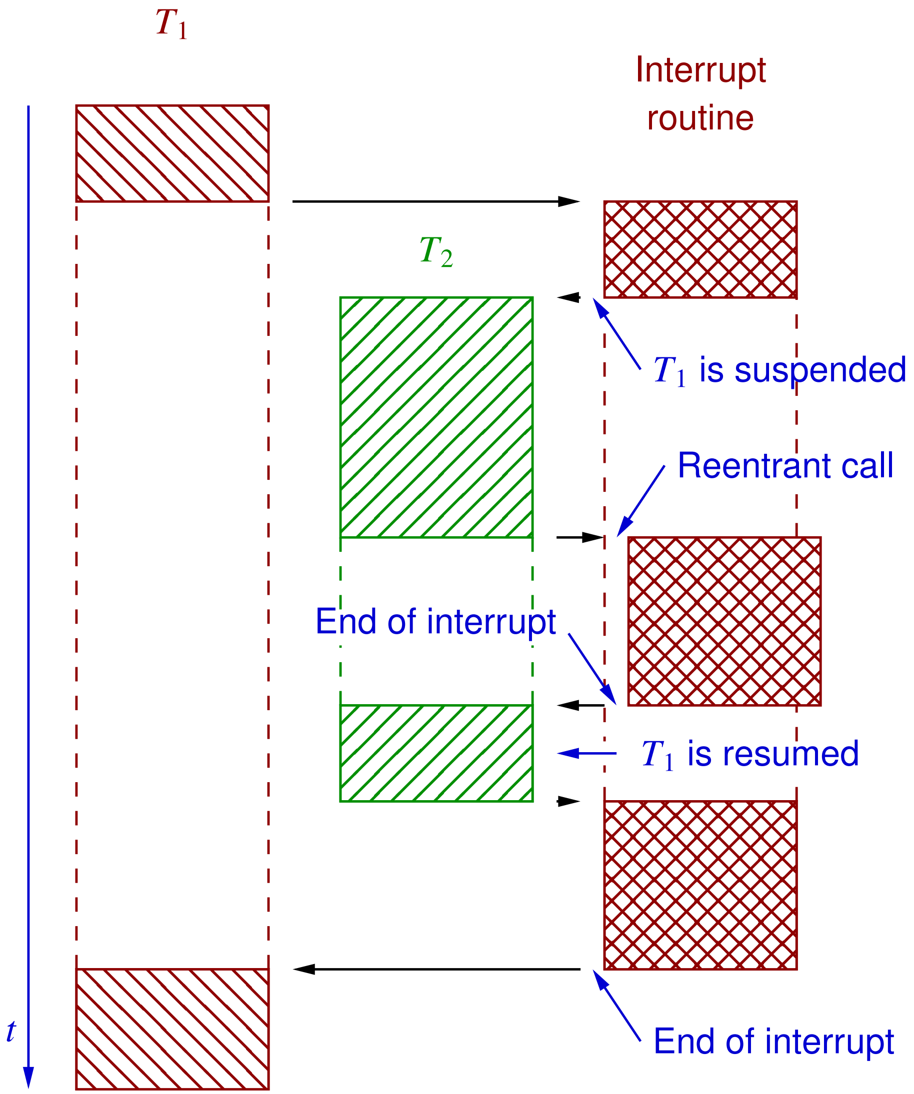
> *An interrupt routine is not allowed to call an OS service if this service can block the current task (e.g., acquiring a semaphore (wait), receiving data from a message queue, waiting for some amount of time, . . .).*

- Indeed, if this rule is not respected, then an interrupt routine can get blocked, 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).

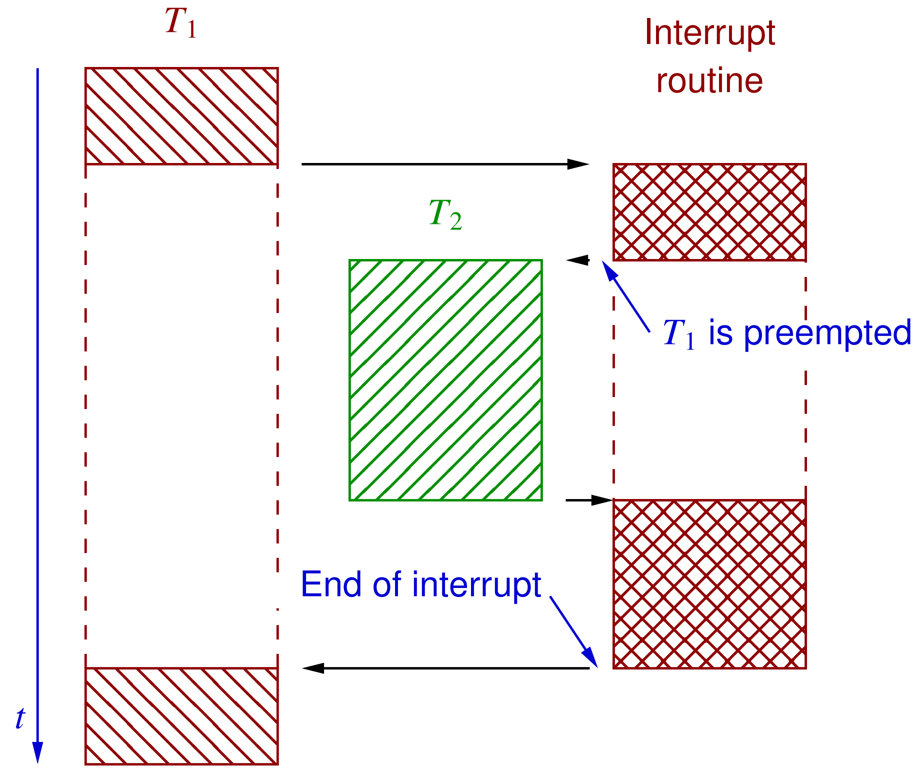


Second rule:

> *If an interrupt routine calls an OS service that can lead to a context switch, then the scheduler must be informed that this system 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:

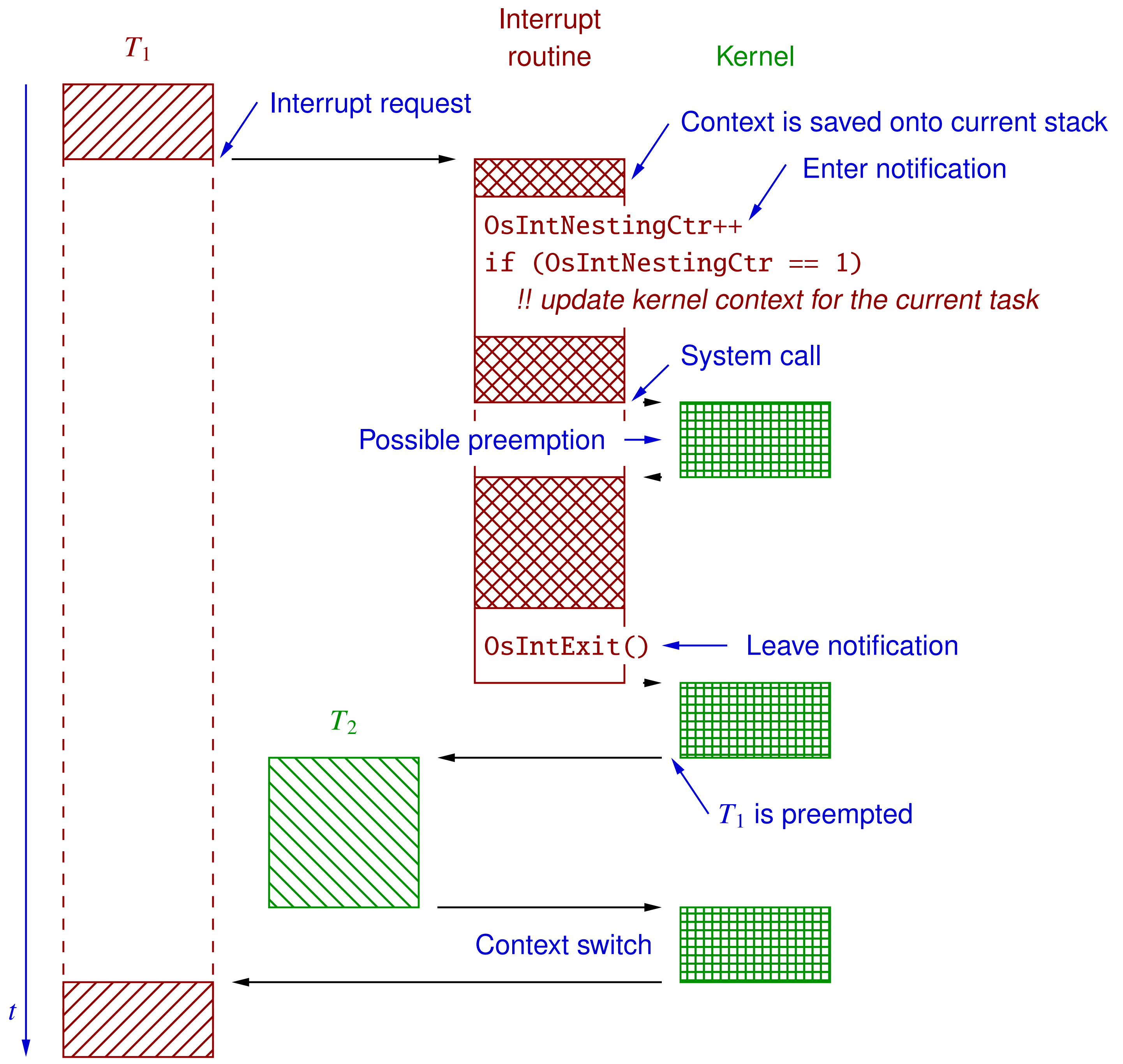


Solution 1: Call dedicated OS services at the beginning and the end of interrupt routines, informing the kernel that the processor is currently running an interrupt routine. Between those calls, preemption of the current task is inhibited.

Notes:

- This approach increases interrupt latency.
- In the case of multiple interrupt priorities, nested interrupt routine calls must be correctly handled.
- At the end of an interrupt routine, context has to be switched to the appropriate task.

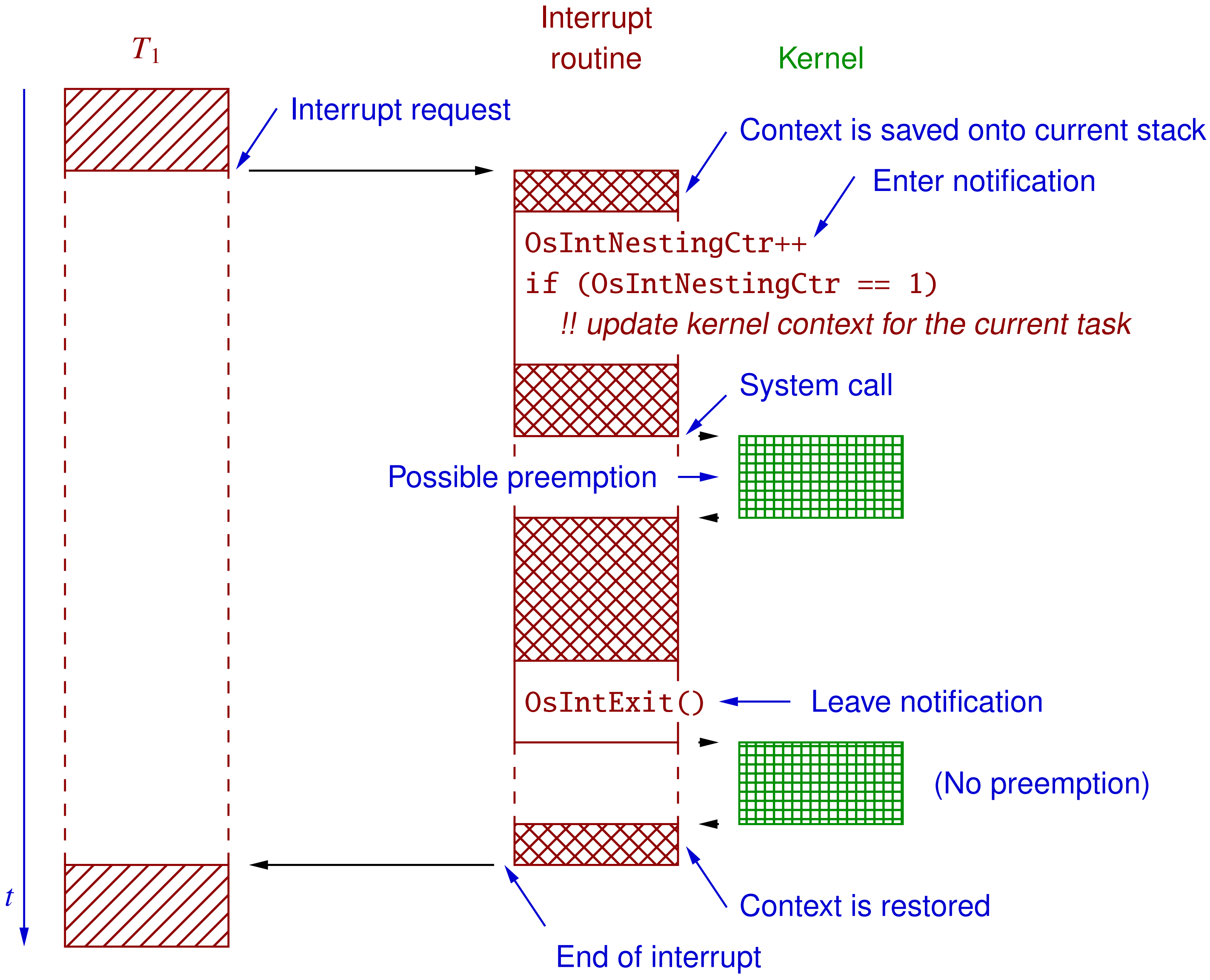
Example (μCOS-III):



```
OsIntNestingCtr++
if (OsIntNestingCtr == 1)
    !! update kernel context for the current task
```

```
OsIntExit()
```

If T_1 is not preempted:

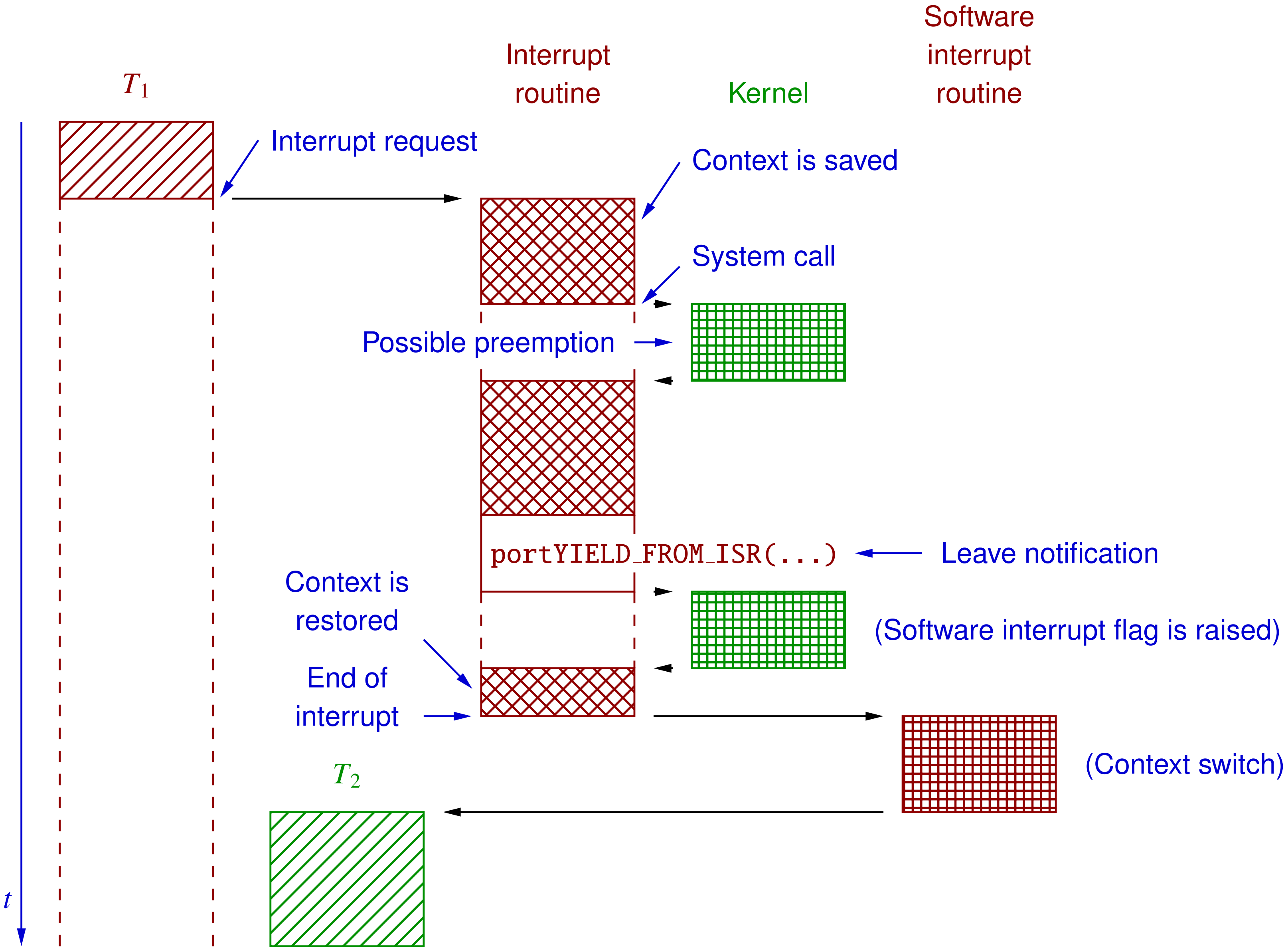


```
OsIntNestingCtr++
if (OsIntNestingCtr == 1)
    !! update kernel context for the current task
OsIntExit()
```

Solution 2: Use alternate versions of OS services in interrupt routines, that do not induce preemption.

Note: There must exist a mechanism for switching the current task immediately after returning from an interrupt routine, if this task must be preempted.

Example (FreeRTOS):



T_1

Interrupt routine

Kernel

Software interrupt routine

Interrupt request

Context is saved

System call

Possible preemption

`portYIELD_FROM_ISR(...)`

Leave notification

Context is restored

(Software interrupt flag is raised)

End of interrupt

(Context switch)

T_2

t

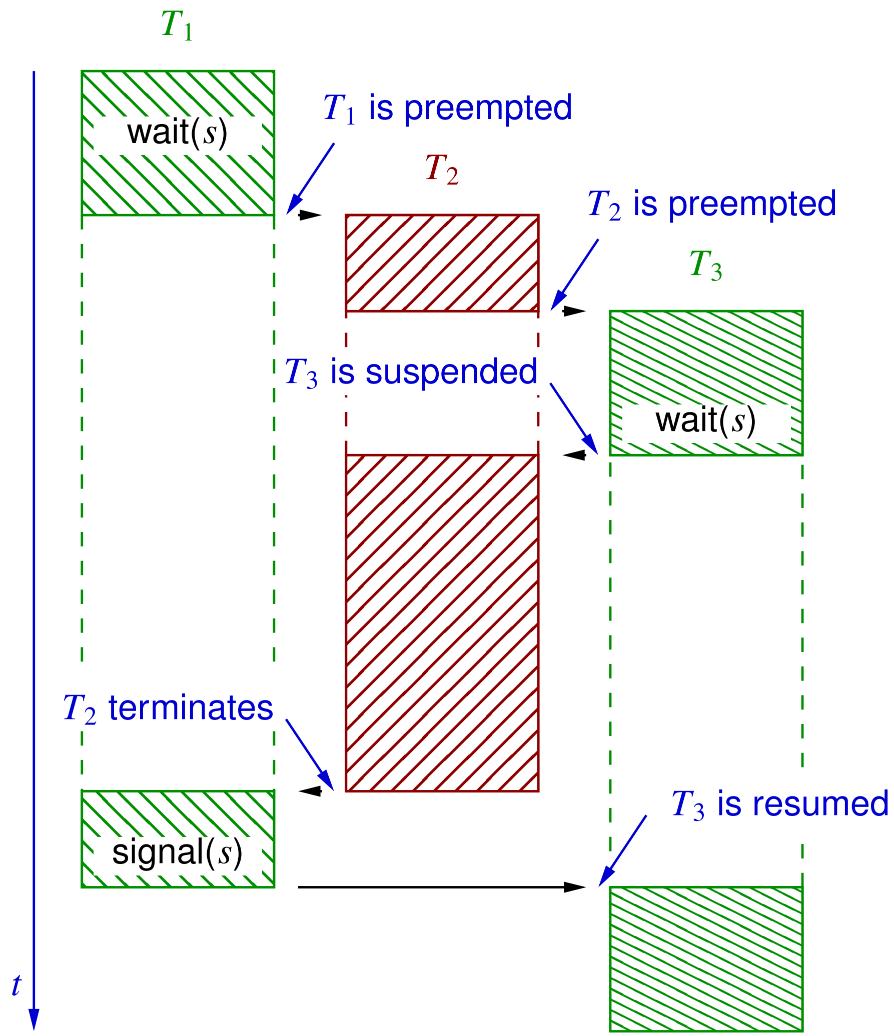
Notes:

- The software interrupt routine implementing the context switch is part of the kernel, and has the lowest interrupt priority.
- If the interrupt leave service is not called, preemption will only occur the next time that the scheduler is called.

Priority inversion

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

Example:



Problem:

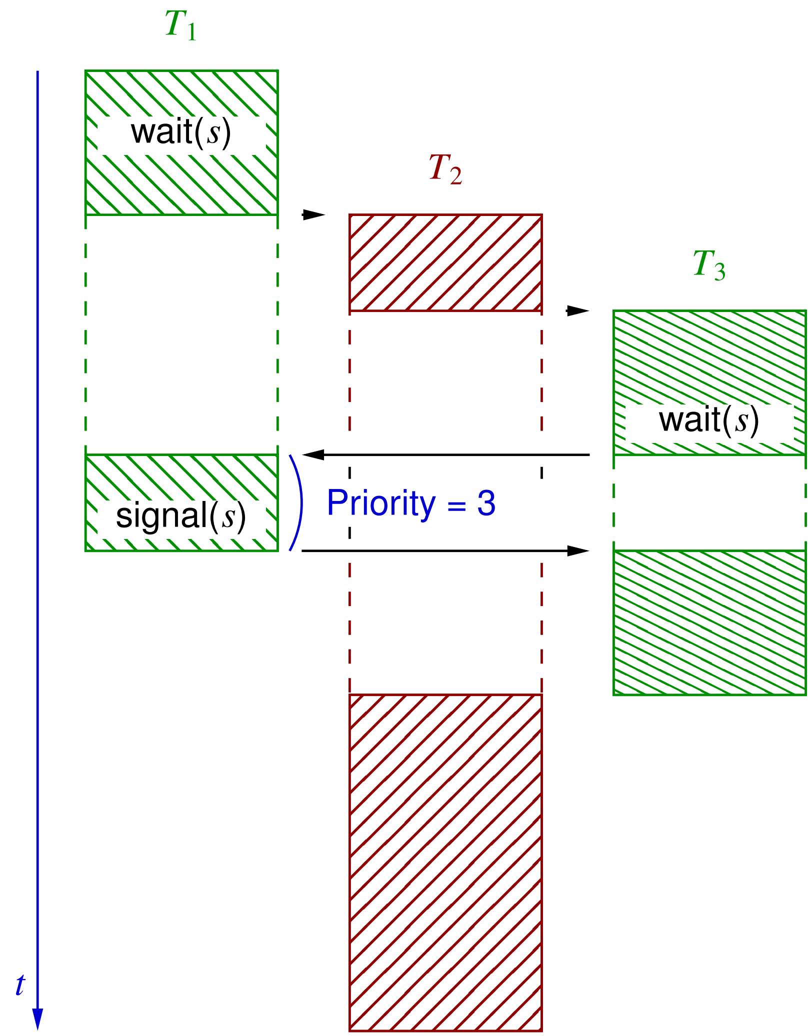
In such a situation, the effective priority of T_3 becomes equal to the one of T_1.

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 (e.g, FreeRTOS, with a special form of semaphores called mutexes).

Illustration:



Time-oriented services

The real-time operating systems offer timed services, for instance for suspending a task for a predefined amount of time, or specifying a timeout for services that can block the current task.

Principles:

- A dedicated component triggers periodic requests for an interrupt that
  - has a low priority, and
  - is serviced by a routine implemented by the kernel. This routine
    - updates the state of the tasks that need to be woken up (or delegates this operation to a kernel task),
    - manages time slicing, and
    - invokes the scheduler, possibly triggering a context switch.
- The delay during which a task is suspended is expressed in the number of occurrences (ticks) of this interrupt request signal.

Note: The precision is limited. Asking to suspend the task during k ticks only ensures that the suspension delay is greater than or equal to the interval between $k - 1$ invocations of the tick interrupt routine.

Example: