Continental

Real Time Systems

What is a real time system?

- A system where correctness depends **not only on the correctness of the logical result** of the computation, but also on the **result delivery time**.

If the timing constraints of the system are not met, system failure is said to have occurred.

- A system that responds in a timely predictable way to unpredictable external stimuli arrivals.

- Hard Real-Time : Systems where it is absolutely that responses occurs within the required deadline: Example - Flight Control, Air-Bag

- Soft Real-Time: System where deadlines are important but which will still function correctly if deadlines are occasionally missed.

Scheduler

- The scheduler is a very important functional component for multitasking and multi process operative systems. It is essential for the real time systems.

- Its job is to share the microcontroller available time between all process available for execution

- Basically a Scheduler keeps the Microcontroller as busy as possible doing different tasks

What is a task

- A task is a container for application code.

- The code of a task is not executed spontaneously: the execution of the task has to be requested.

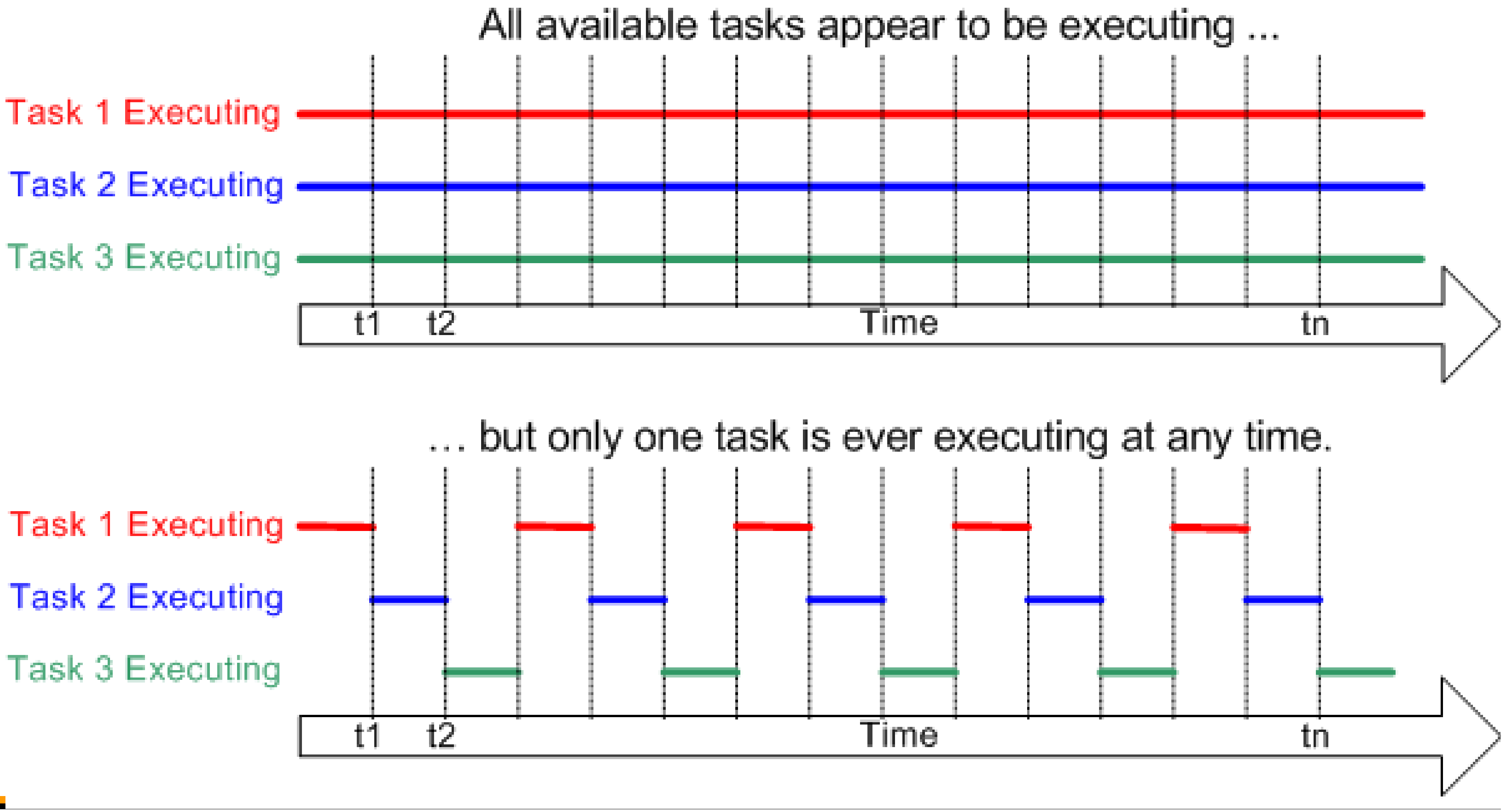
How the Scheduler works?

- Suppose a system with a single microcontroller, this microcontroller can execute only one "program" at a time.

- Besides when a program is running it will never stop by itself . So any "program" can monopolize the microcontroller; and one objective of the Scheduler is to avoid this.

- This is done in several ways, one of this is:
  - When a timer expires. This timer is activated at regular periods of time, and the control of the microcontroller is passed to the scheduler thanks to the hardware that generates an interrupt.

Multi task


All available tasks appear to be executing ...
Task 1 Executing
Task 2 Executing
Task 3 Executing
t1
t2
Time
tn
… but only one task is ever executing at any time.
Task 1 Executing
Task 2 Executing
Task 3 Executing
t1
t2
Time
tn

What is an Interrupt ?

- A hardware signal, managed by the processor itself, that initiates an event
- Upon receipt of an interrupt, the processor:
  - completes the instruction being executed
  - saves the program counter (so as to return to the same execution point as before)
  - loads the program counter with the location of the interrupt handler code
  - executes the interrupt handler (or Interrupt Service Routine).
- Interrupts can be disabled for a short time by the operating system in order to execute critical code section

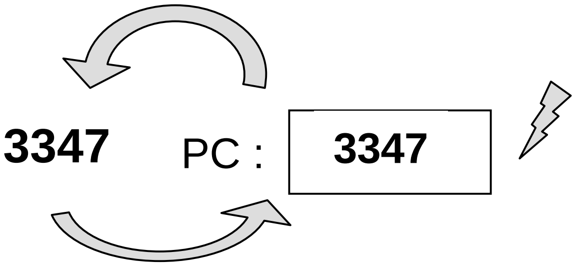
Example

A

```
if (a == TRUE)    0x3345
{                 0x3346

 i++;             0x3347
}
else
{
 i - -;           0x3348
}
```

B

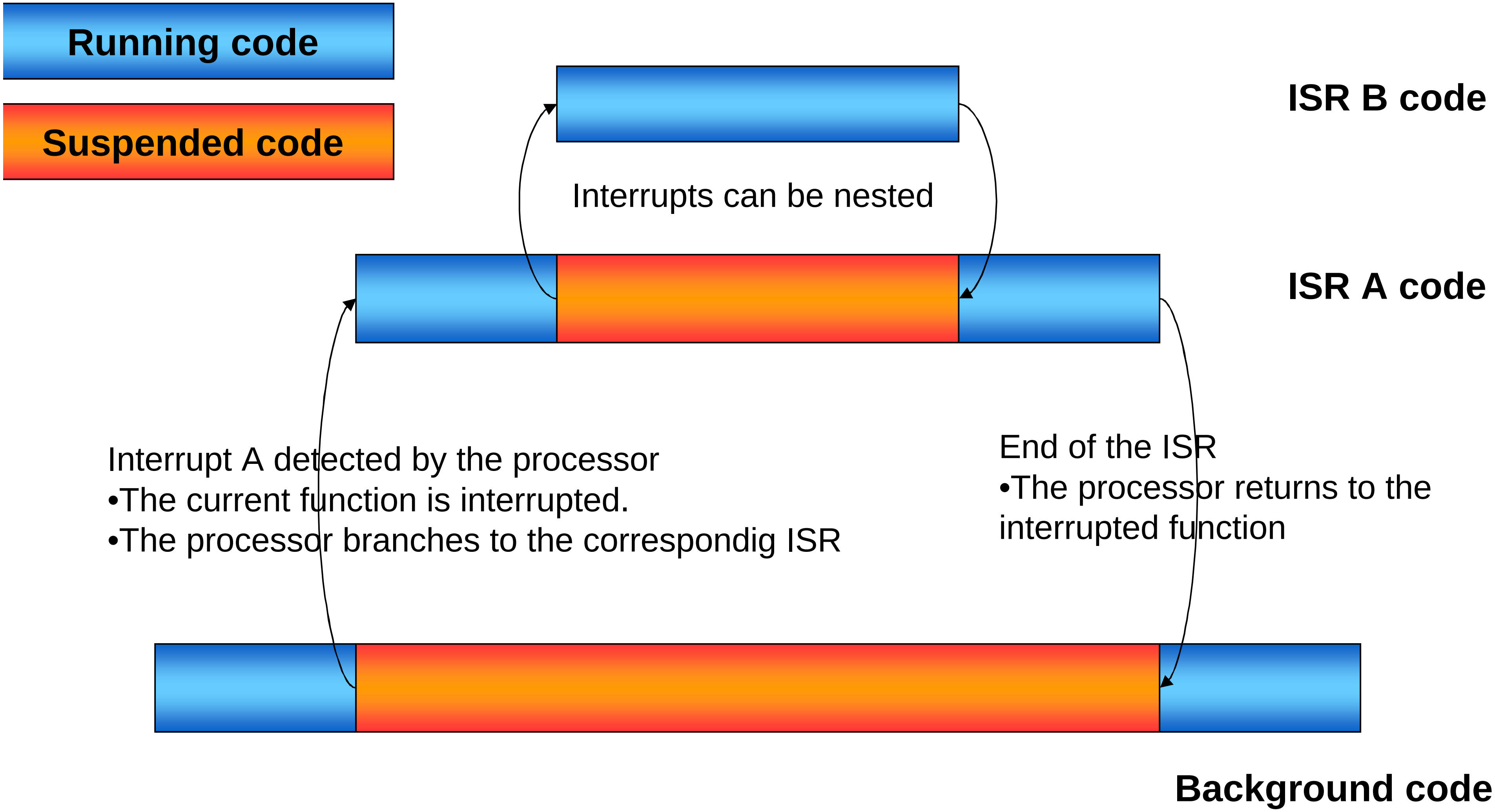
```
if (b  == TRUE)        0x5614
{                      0x5615

i = 45;                0x5616
}
else
{
 i = 0;                 0x5617
}
```



3347

PC : 3347

What is an interrupt?



Running code

Suspended code

ISR B code

Interrupts can be nested

ISR A code

Interrupt A detected by the processor
•The current function is interrupted.
•The processor branches to the correspondig ISR

End of the ISR
•The processor returns to the interrupted function

Background code

OS Concepts

- Critical Section. Code that needs to be treated indivisibly.
- Resource. Any entity used by a task
- Shared Resource. Resource used by more than one task.
- Task. A simple process that thinks it has the CPU all to itself.
- Context Switch or Task Switch. When the multitasking kernel decides to run a different task.
- Kernel. Part of the O.S. responisble for management of tasks and communication between tasks.

Scheduler Policy

- The scheduler is one of the basic parts of an OS. It has the function of switching from one task to another.

- The execution of a task can be requested:
  - cyclically by the operating system
  - by another task
  - by an interrupt service routine.

Only one task can use the CPU at the same time.

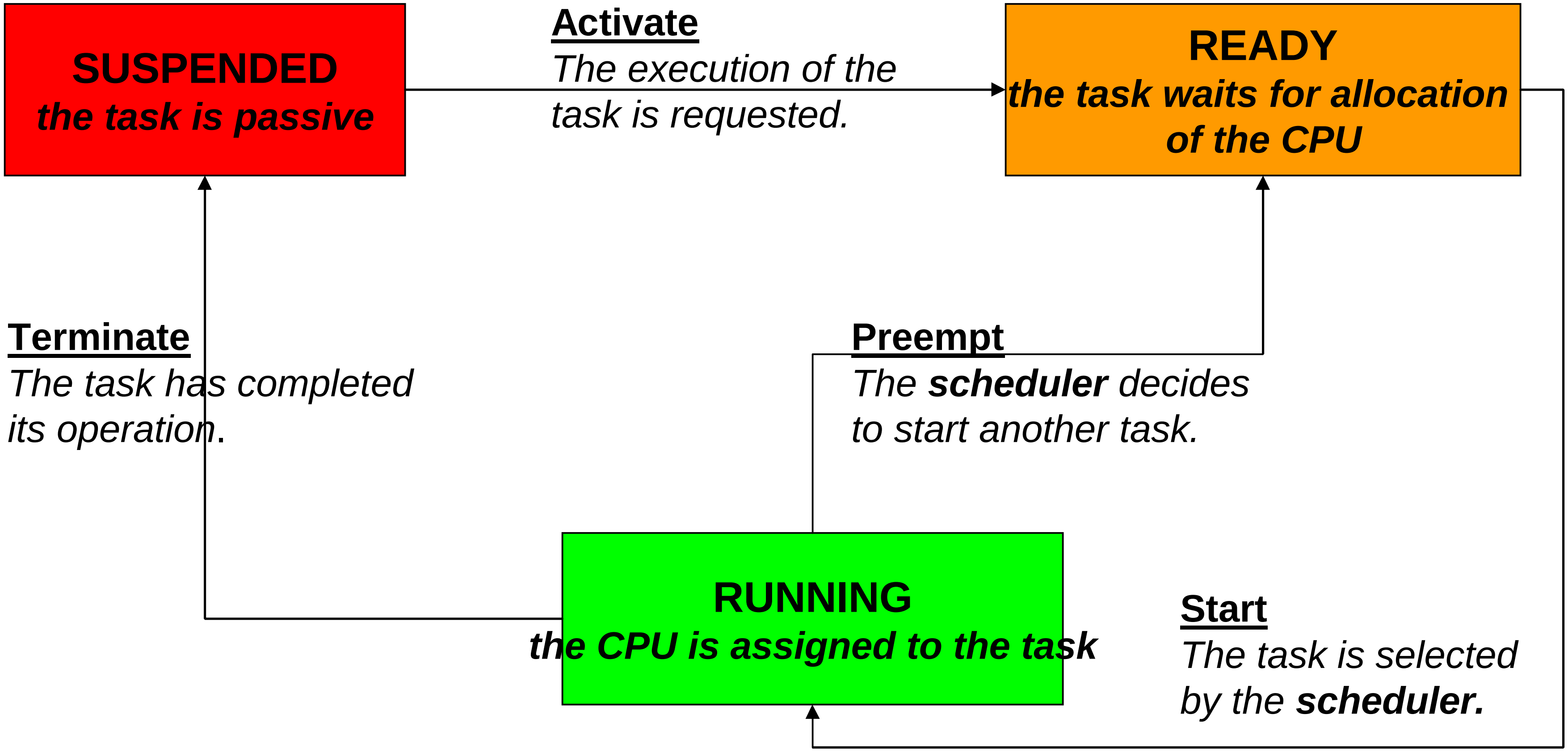
Scheduler Policy

- To understand scheduling, it has to be understood that each task has three different states: running, ready and suspended.

- If there is more than one task in the ready state (waiting for the CPU), you need a decision-making algorithm to determine which task can use the processor first.

- **This algorithm is called the scheduling policy.**

OSEK Task Model



SUSPENDED
the task is passive

<u>Activate</u>
The execution of the task is requested.

READY
the task waits for allocation of the CPU

<u>Terminate</u>
The task has completed its operation.

<u>Preempt</u>
*The **scheduler** decides to start another task.*

RUNNING
the CPU is assigned to the task

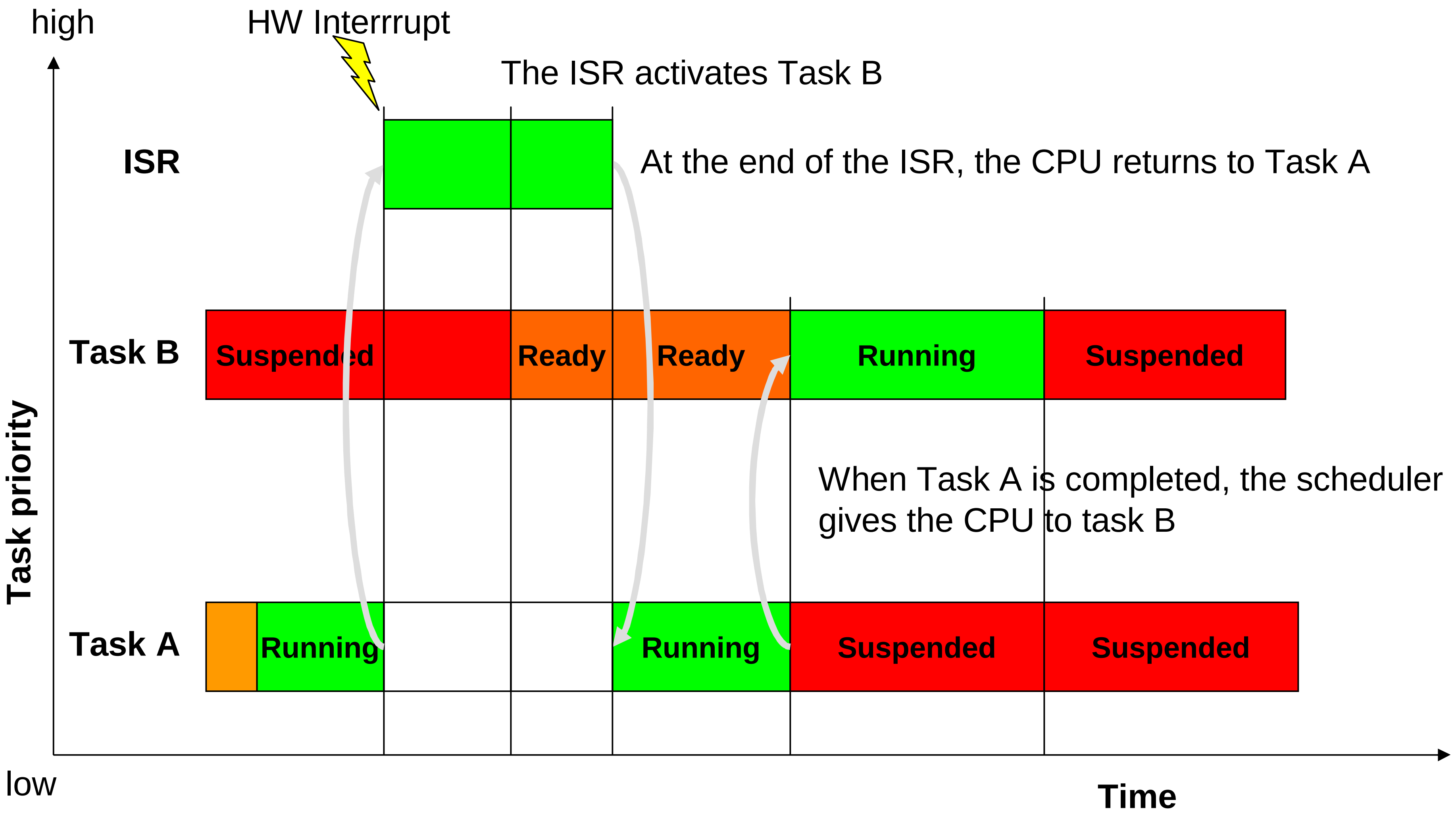
<u>Start</u>
*The task is selected by the **scheduler.***

Scheduling Algorithm

- **Rate-monotonic scheduling**
  - Is a scheduling used in RTOS with a static-priority scheduling class. The static priorities are assigned on the basis of the cycle duration of the job: the shorter the cycle duration is, the higher is the job's priority
- **EDF** (Earliest Deadline First):
  - Assign the higher priority to the task with the shorter time to deliver.
- **Dual Priority**:
  - Assign the lower priority to the task with lower execution time.
- **Round-robin** (RR)
  - Is one of the simplest scheduling algorithms for processes in an operating system, which assigns time slices to each process in equal portions and in circular order, handling all processes without priority

Scheduling Policy
Cooperative scheduling

- Cooperative scheduling relies on the cooperation of your applications to allow a system to function correctly.
- Each task runs until it no longer has anything to process. Then it returns control to the OS, which runs the next task.
- The implication is that no task in the system should take very long to run. If a task needs an excessive amount of time, the programmer should split its operation into steps.
- The scheduler gives the CPU to a task according to its priority.
- The running task can only be interrupted by an ISR (interrupt service routine).



high

HW Interrrupt

The ISR activates Task B

ISR

At the end of the ISR, the CPU returns to Task A

Task B

Suspended

Ready

Ready

Running

Suspended

When Task A is completed, the scheduler gives the CPU to task B

Task A

Running

Running

Suspended

Suspended

Task priority

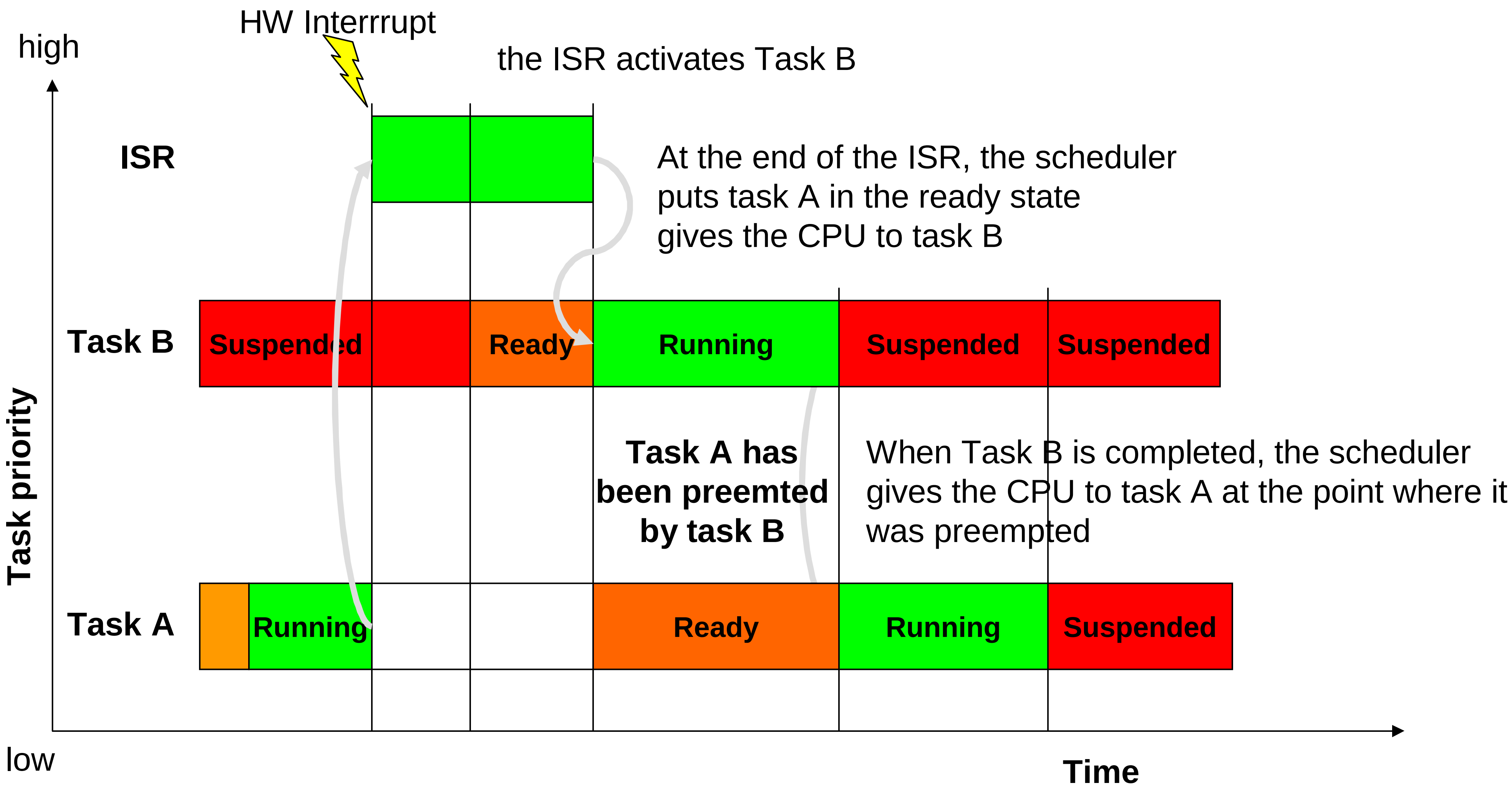
low

Time

Scheduling policy
Preemptive scheduling

- Higher priority tasks may interrupt lower priority tasks.
- Each task runs until it no longer has anything to process or until the scheduler decides to run a higher priority task.
- The implication is that a task can be interrupted (preempted) at any point by another task. When a task is preempted (i.e. interrupted by another task), its local context (variables, instruction pointer) have to be saved.
- The scheduler gives the CPU to a task according to its priority.
- The running task can be interrupted by an ISR (interrupt service routine) or by a higher priority task

Scheduling Policy : example of preemtive scheduling



Scheduling Policy
Mixed preemptive/cooperative scheduling

- Combines cooperative scheduling
- Groups of tasks are defined
- Between tasks belonging to the same group, the cooperative scheduling policy is applied.
- Between tasks belonging to different groups, the preemptive scheduling policy is applied

	Cooperative	Preemptive	Mixed Cooprerative/preemtive
Advantages	ü Simple ü Uses few resources (stack, RAM, ROM) ü No data sharing problem between tasks	ü Reaction time ü Task duration is not limited.	ü Use preemtion only when really needed. ü Good compromize reaction time/resource consumption
Drawbacks	ü Reaction time. ü Task duration must be limited.	ü Complexity ü Run time behaviour difficult to predict. ü Uses resources (stack, RAM, ROM) ü Data sharing between tasks.	ü Data sharing between groups. ü Task duration must be limited within the same group

Real Time Problems

Synchronization

- In common use, “synchronization” means making two things happen at the same time. In computer systems, synchronization is a little more general; it refers to relationships among events—any number of events, and any kind of relationship (before, during, after).

- Computer programmers are often concerned with synchronization constraints, which are requirements pertaining to the order of events. Examples include:

- **Serialization: Event A must happen before Event B.**

- **Mutual exclusion: Events A and B must not happen at the same time**

.Execution model

- In the simplest model, computers execute one instruction after another in sequence. In this model, synchronization is trivial; we can tell the order of events by looking at the program. If Statement A comes before Statement B, it will be executed first.
- There are two ways things get more complicated.
  - One possibility is that the computer is parallel, meaning that it has multiple processors running at the same time.
  - Two is that a single processor is running multiple threads of execution. A thread is a sequence of instructions that execute sequentially. If there are multiple threads, then the processor can work on one for a while, then switch to another, and so on
  - In general the programmer has no control over when each thread runs; the operating system (specifically, the scheduler) makes those decisions. As a result, again, the programmer can’t tell when statements in different threads will be executed.

The Little Book of Semaphores
Allen B. Downey

- For purposes of synchronization, there is no difference between the parallel model and the multithread model. The issue is the same.
- Within one processor (or one thread) we know the order of execution, but between processors (or threads) it is impossible to tell
- A real world example might make this clearer
- Imagine that you and a live in different cities, and one day, around dinner time, you start to wonder who ate lunch first that day, you or Bob. How would you find out? Obviously you could call him and ask what time he ate lunch. But what if you started lunch at 12:59 by your clock and Bob started lunch at 13:00 by his clock? Can you be sure who started first? Unless you are both very careful to keep accurate clocks, you can’t.
- Concurrent program are often non-deterministcs wich meand is not possible to tell what will happen when it executes. A program might work correctly 1000 time and then crash on the 1001st run.

The Little Book of Semaphores
Allen B. Downey

Serialization with messages

- One solution is to instruct your friend not to eat lunch until you call. Then, make sure you don’t call until after lunch

You

a1 Desayunas
a2 Trabajas
a3 Comes
a4 Llamas a tu amigo

Your Friend

b1 Desayunas
b2 Esperas la llama
b3 Comes

- The first colume is a list of action your perform in other words your thread of execution. the secon column is your friend's thread of execution

- a1 < a2 < a3 < a4 In this case lunch is sequential and breakfast concurrent

  b1 < b2 < b3

The Little Book of Semaphores
Allen B. Downey

Shared variables

- Most of the time, most variables in most threads are local, meaning that they belong to a single thread and no other threads can access them. As long as that’s true, there tend to be few synchronization problems, because threads just don’t interact.

- But usually some variables are shared among two or more threads; this is one of the ways threads interact with each other. For example, one way to communicate information between threads is for one thread to read a value written by another thread. If the threads are unsynchronized, then we cannot tell by looking at the program whether the reader will see the value the writer writes or an old value that was already there.

- Other ways that threads interact are concurrent writes (two or more writers) and concurrent updates (two or more threads performing a read followed by a write)

The Little Book of Semaphores
Allen B. Downey

Concurrent writes

- In the following example X is a shared variable accessed by two writer.
- If this two task are part of a preemptive scheduler what are the possible results of x

- 1) a1, b1,a2 x = 7/print a 7
- 2) a1,a2, b1 x = 7/ print a 5
- 3) b1, a1, a2 x = 5 / print a 5
- 4) Is it possible that x is 5 and print 7?

a1 x = 5
a2 Print x

b1 x = 7

The Little Book of Semaphores
Allen B. Downey

Concurrent updates

A

```
a1  x = x + 1;
```

B

```
b1 x = x + 1;
```

- At first glance, it is not obvious that there is a synchronization problem here. There are only two execution paths, and they yield the same result.
- The problem is that these operations are translated into machine language before execution, and in machine language the update takes two steps, a read and a write. The problem is more obvious if we rewrite the code with a temporary variable, temp.

A

```
a1 temp = x;
a2 x = temp + 1;
```

B

```
b1 temp = x;
b2 x = temp + 1;
```

- This kind of problem is subtle because it is not always possible to tell, looking at a high-level program, which operations are performed in a single step and which can be interrupted.
- **An operation that cannot be interrupted is said to be atomic**.

The Little Book of Semaphores
Allen B. Downey