Embedded Systems

Lecture 6

Real-Time Operating Systems

Michele Magno

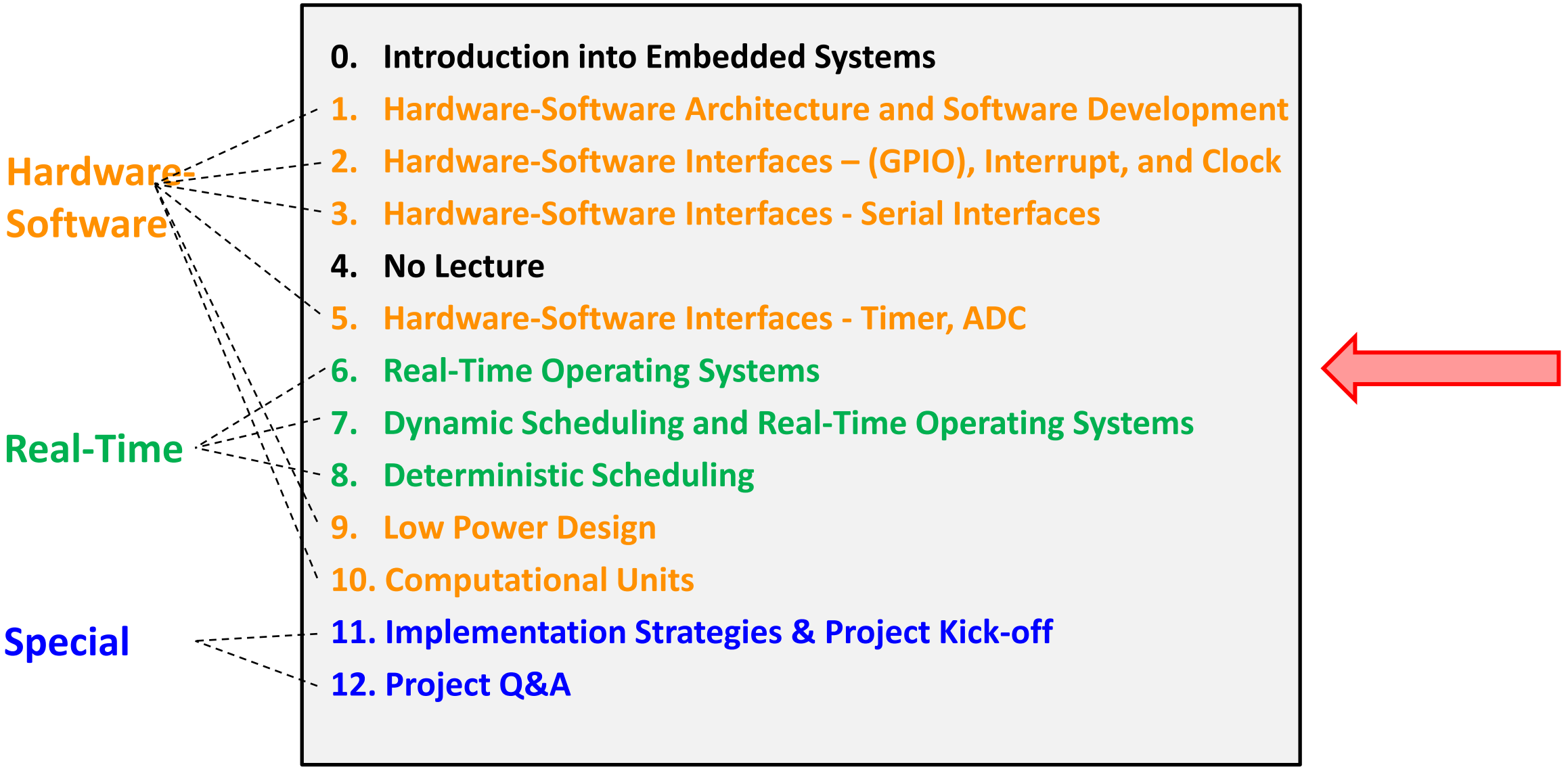
D-ITET Center for Project-Based Learning

Credits: Lothar Thiele

ETH zürich

Slides partially adopted from Prof. Thiele

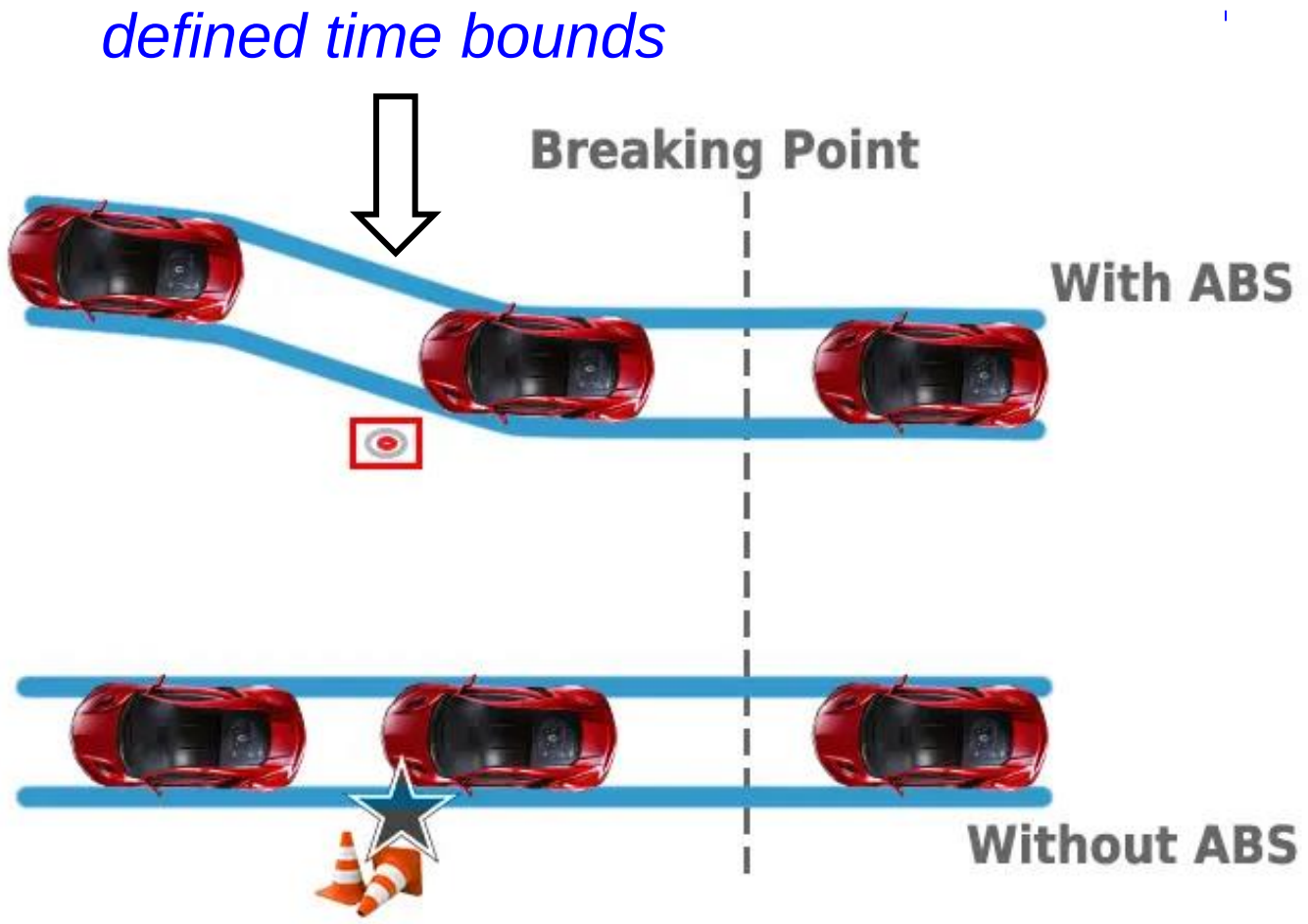
Where We are


Hardware-Software
Real-Time
Special
0. Introduction into Embedded Systems
1. Hardware-Software Architecture and Software Development
2. Hardware-Software Interfaces – (GPIO), Interrupt, and Clock
3. Hardware-Software Interfaces - Serial Interfaces
4. No Lecture
5. Hardware-Software Interfaces - Timer, ADC
6. Real-Time Operating Systems
7. Dynamic Scheduling and Real-Time Operating Systems
8. Deterministic Scheduling
9. Low Power Design
10. Computational Units
11. Implementation Strategies & Project Kick-off
12. Project Q&A

Overview of today's lecture

- What is Real-Time?
- Bare-metal programming Vs Operating systems
- Processes and Threads
- Scheduling
  - Time-Triggered approaches
  - Event-Triggered approaches
- Introduction PC OS vs Embedded OS

Real-Time



What is Real-Time?

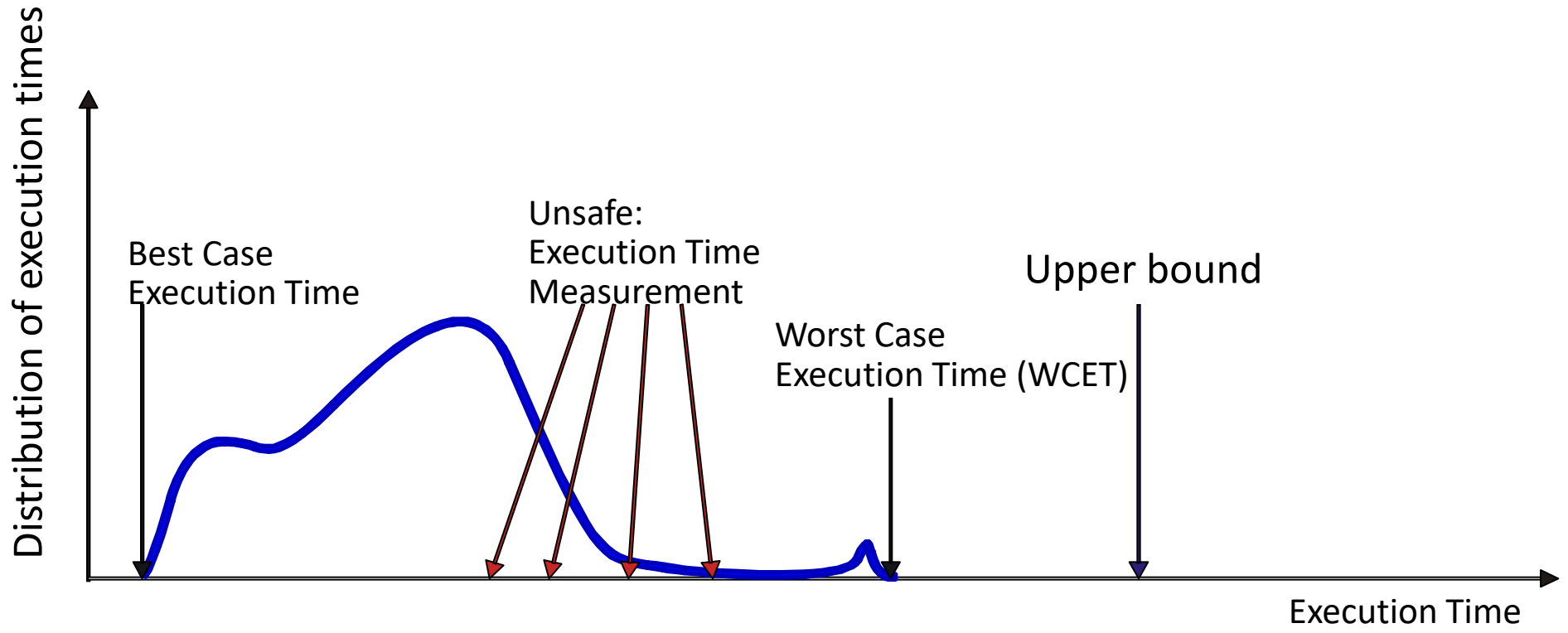
- *Time* indicates that the correctness of the system depends not only on the logical result of the computation but also on the time at which the result is produced. [1]
- *Real* indicates the reaction of the system to an external event must occur during their evolution. [1]

- *Such embedded systems for automatic control* are often expected to *finish the processing* of data and events reliably *within defined time bounds*. Such a processing may involve sequences of computations and communications.

- *Upper bounds on the execution times*.
  - This value is commonly called the *Worst-Case Execution Time* (WCET).
  - Analogously, one can define the lower bound on the execution time, the *Best-Case Execution Time* (BCET).

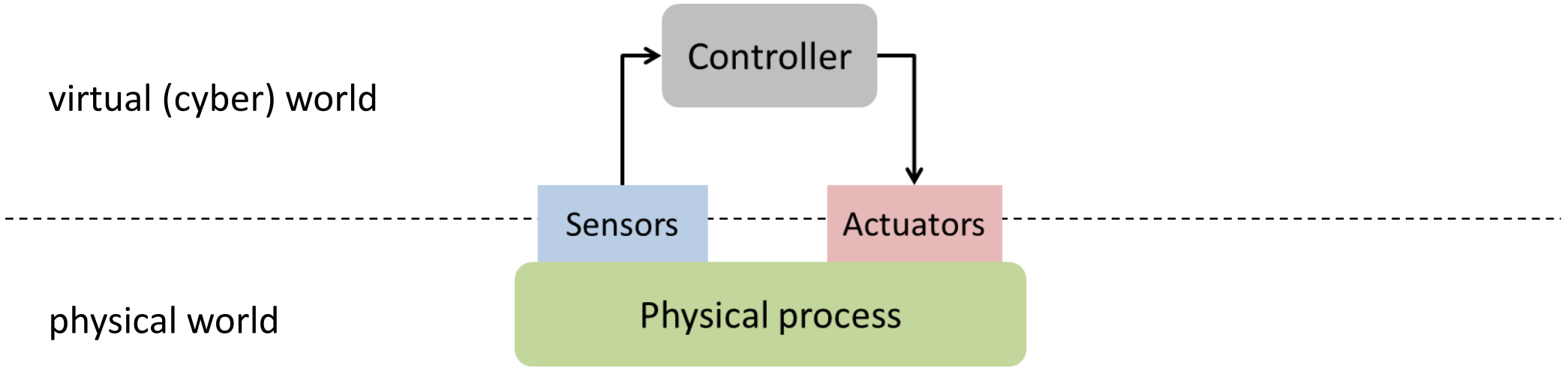
[1] Giorgio C. Butazzo: Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications, 2011 Page 4-5

Distribution of Execution Times

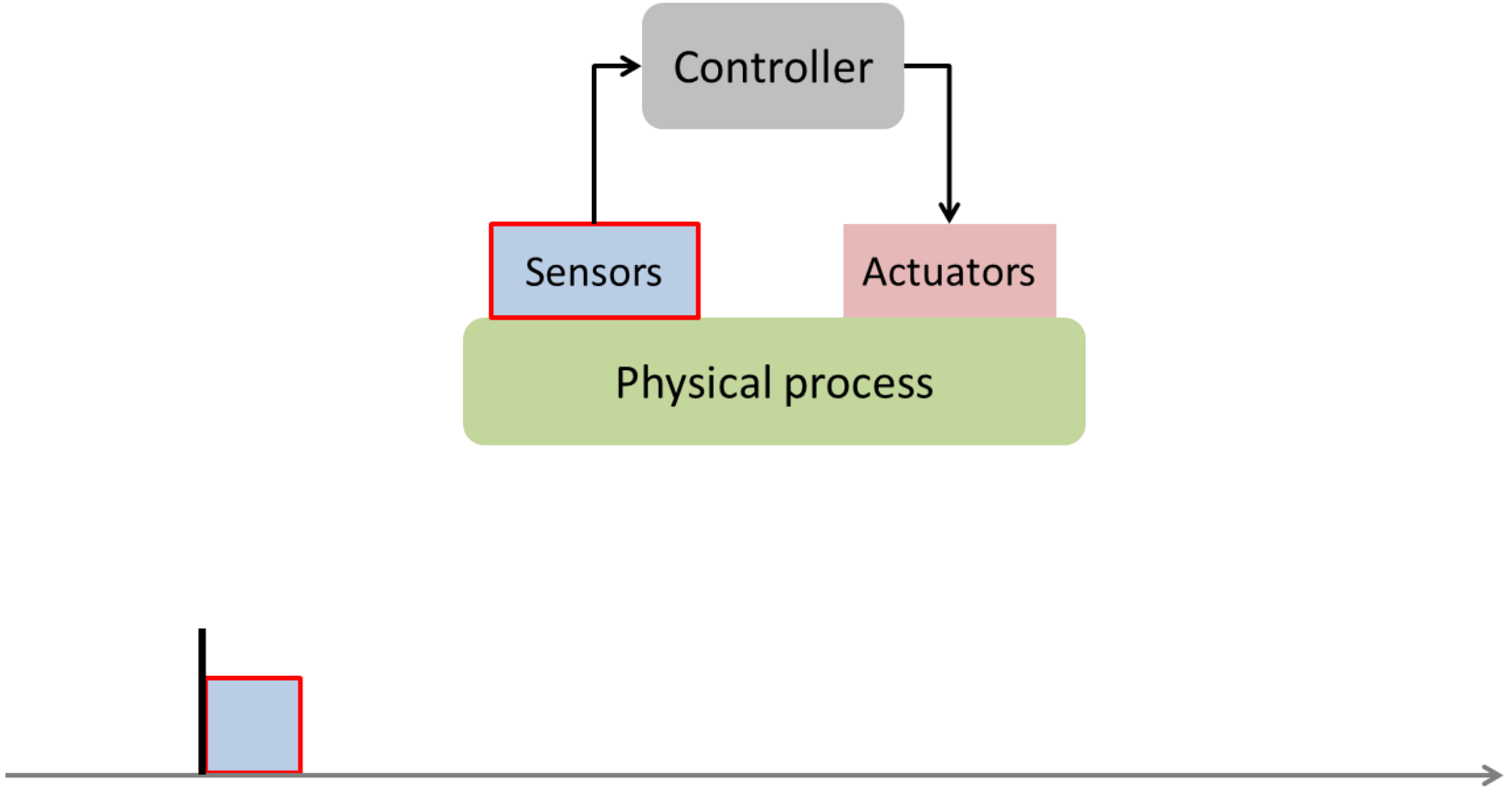


Typical “Cyber-physical” System for Automatic Control

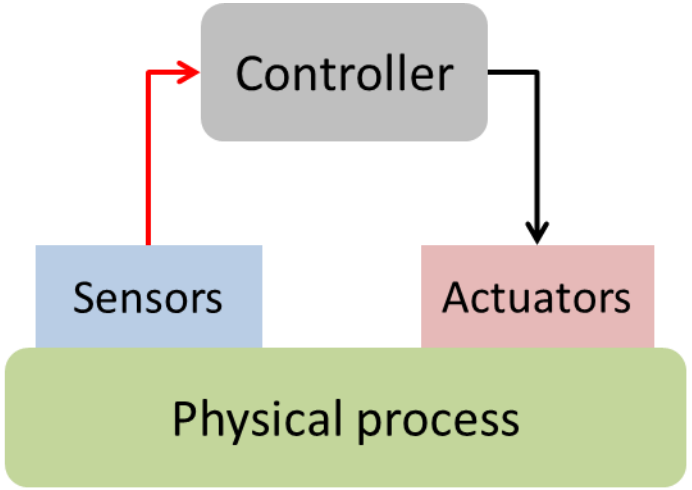
In many *cyber-physical systems (CPSs),* correct timing is a matter of *correctness*, not performance: *an answer arriving too late is consider to be an error.*

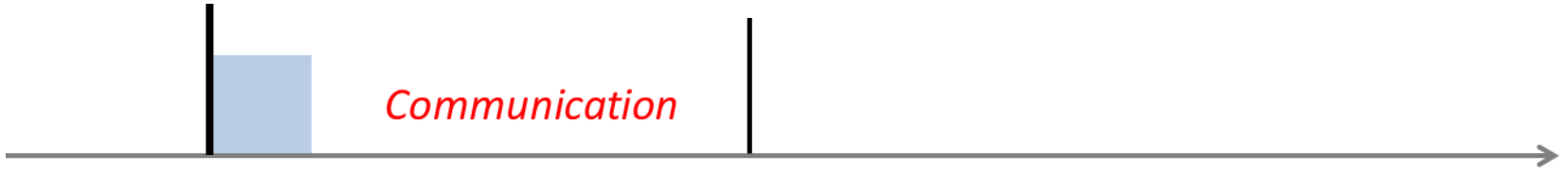


Typical “Cyber-physical” System for Automatic Control

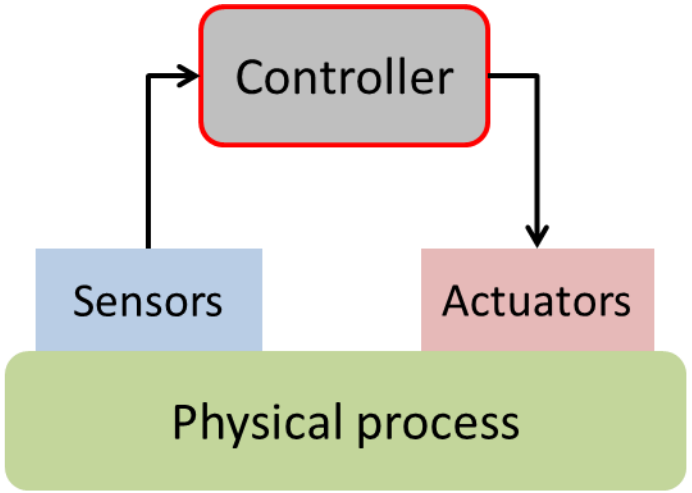


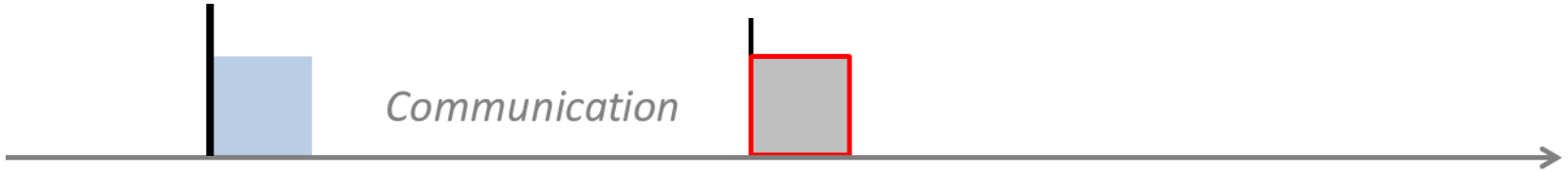
Typical “Cyber-physical” System for Automatic Control



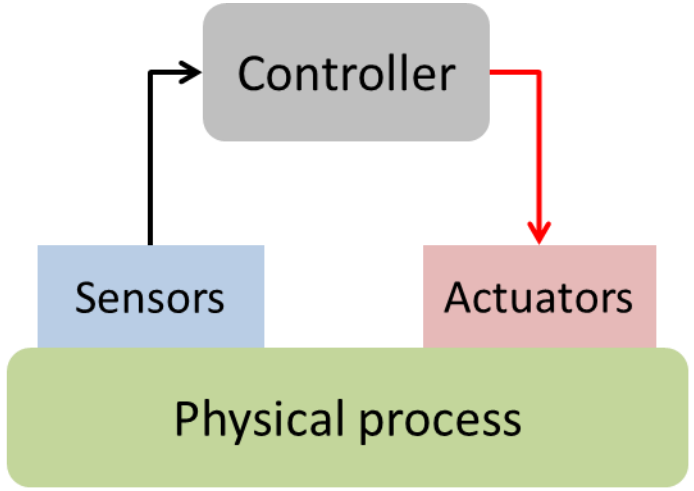


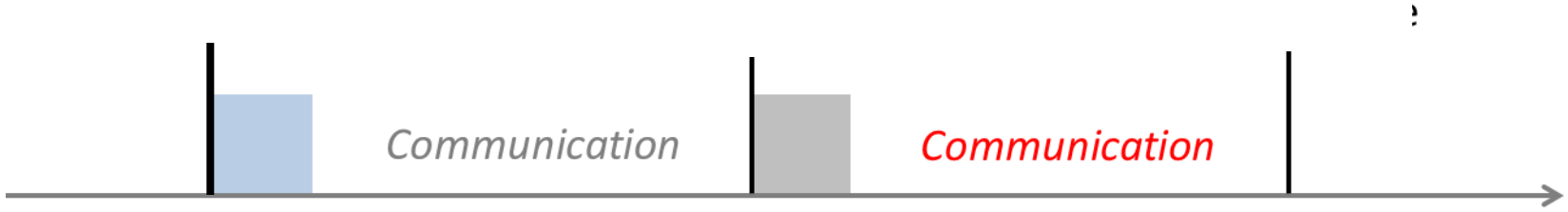
Typical “Cyber-physical” System for Automatic Control



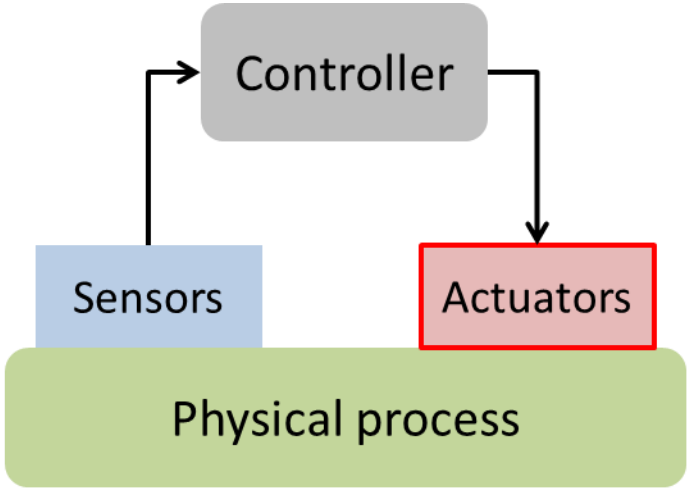


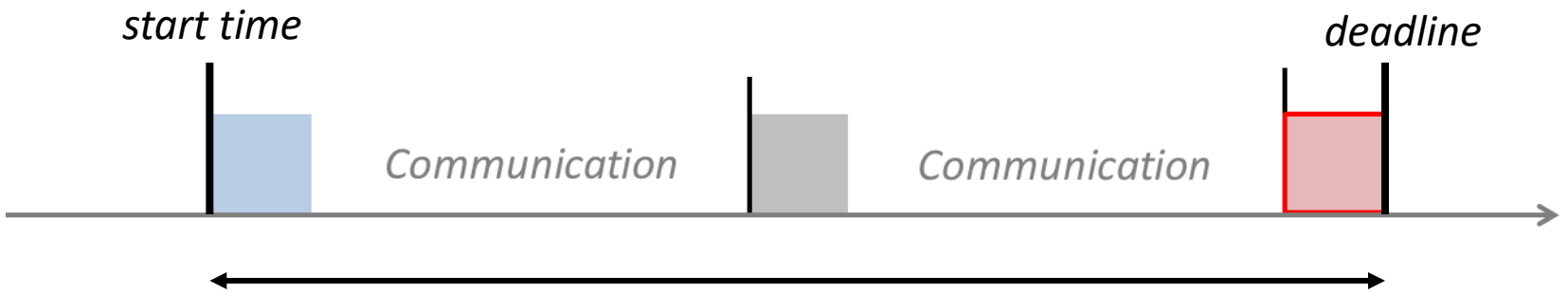
Typical “Cyber-physical” System for Automatic Control





Typical “Cyber-physical” System for Automatic Control





Basic Terms

Real-time systems

- *Hard:* A real-time task is said to be hard, if missing its deadline may cause catastrophic consequences on the environment under control.
  - Examples are sensory data acquisition, detection of critical conditions, actuator servicing.

Side airbag in car,
reaction after event in <10 mSec

- *Soft:* A real-time task is called soft, if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior.
  - Examples are command interpreter of the user interface, displaying messages on the screen.

Modern Hardware Features

- Modern processors *increase the average performance* (execution of tasks) by using *caches, pipelines, branch prediction*, and *speculation* techniques, for example.
- *These features make the computation of the WCET very difficult*: The execution times of single instructions vary widely.
- The microarchitecture has a large *time-varying internal state* that is changed by the execution of instructions and that influences the execution times of instructions.
  - *Best case* - everything goes smoothely: no cache miss, operands ready, needed resources free, branch correctly predicted.
  - *Worst case* - everything goes wrong: all loads miss the cache, resources needed are occupied, operands are not ready.
  - *The span between the best case and worst case may be several hundred cycles.*

Determine the WCET

Complexity of determining the WCET of tasks:

- In the general case, it is even *undecidable* whether a finite bound exists.
- For *restricted classes of programs* it is possible, in principle. Computing accurate bounds is *simple for „old“ architectures*, but very *complex for new architectures* with pipelines, caches, interrupts, and virtual memory, for example.

Analytic (formal) approaches exist for hardware and software.

- In case of software, it requires the *analysis of the program flow* and the *analysis of the hardware* (microarchitecture). Both are combined in a complex analysis flow, see for example www.absint.de.
- *For the rest of the lecture, we assume that reliable bounds on the WCET are available,* for example by means of exhaustive measurements or simulations, or by analytic formal analysis.

How did we program MCUs until now?

“Bare-metal” programming - Recap.

Generally, whole application with all tasks runs within a single `while` loop.

```c
int main() {

    init(); // Initialize GPIO

    while(1){
        // Toggle the LED pin
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        // Delay 500 ms
        HAL_Delay(500);
    }
}
```

Source code

Compiler

Upload to Flash

```
main:
        push {r3, lr}                   #b508
        bl init                         #f7ff fffe
.L2:
        movs r1, #0                     #2100
        mov r0, r1                      #4608
        bl HAL_GPIO_TogglePin           #f7ff fffe
        mov r0, #500                    #f44f 70fa
        bl HAL_Delay                    #f7ff fffe
        b .L2                           #e7f6
```

Assembly + Machine code

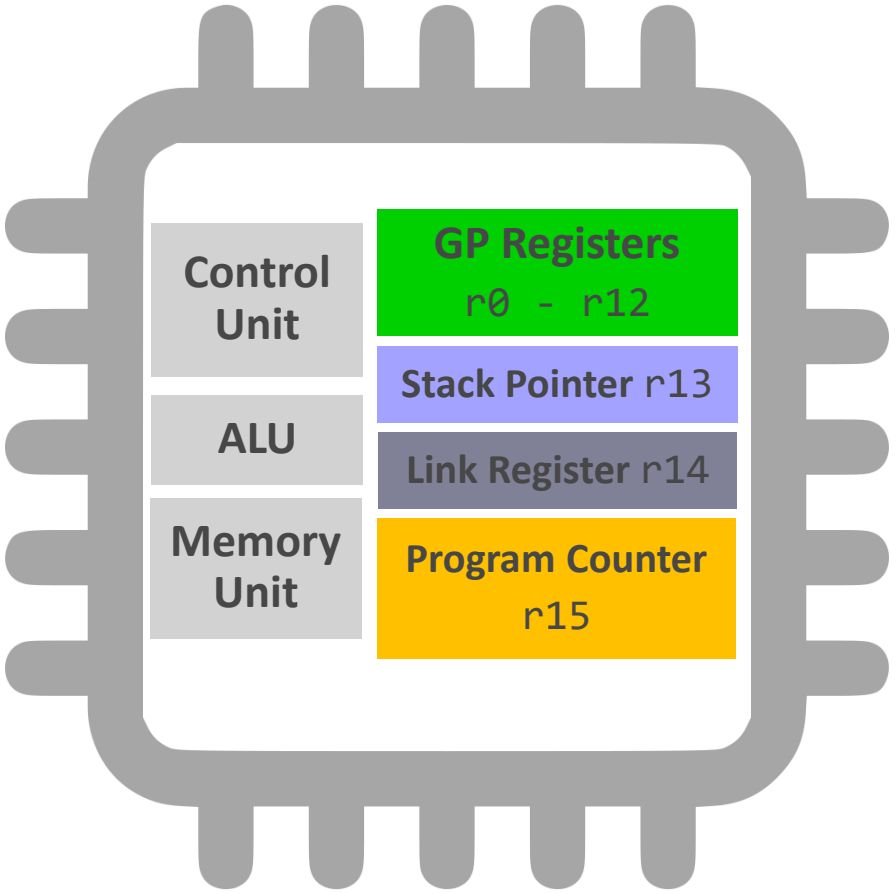
to be run directly on the hardware

“Bare-metal” programming - Recap. Execution

`bl` “Branch with Link” explicitly sets **PC** This jumps to a function but saves the return address to `lr` - **PC** is set back after return.

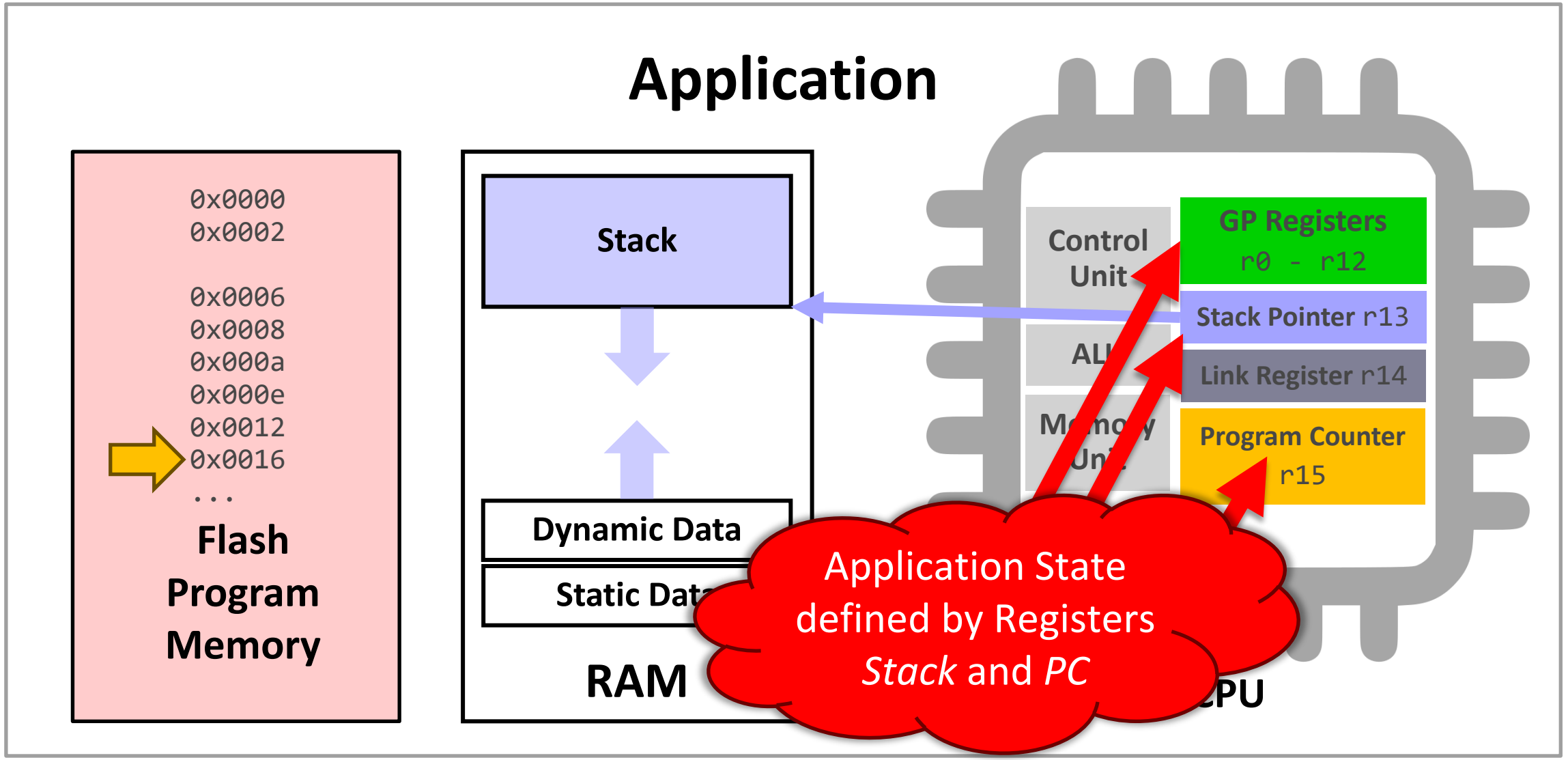
```
0x0000  main:
        push {r3, lr}                   #b508
0x0002  bl init                         #f7ff fffe
        .L2:
0x0006  movs r1, #0                     #2100
0x0008  mov r0, r1                      #4608
0x000a  bl HAL_GPIO_TogglePin           #f7ff fffe
0x000e  mov r0, #500                    #f44f 70fa
0x0012  bl HAL_Delay                    #f7ff fffe
0x0016  b .L2                           #e7f6
...
```

Flash Program Memory



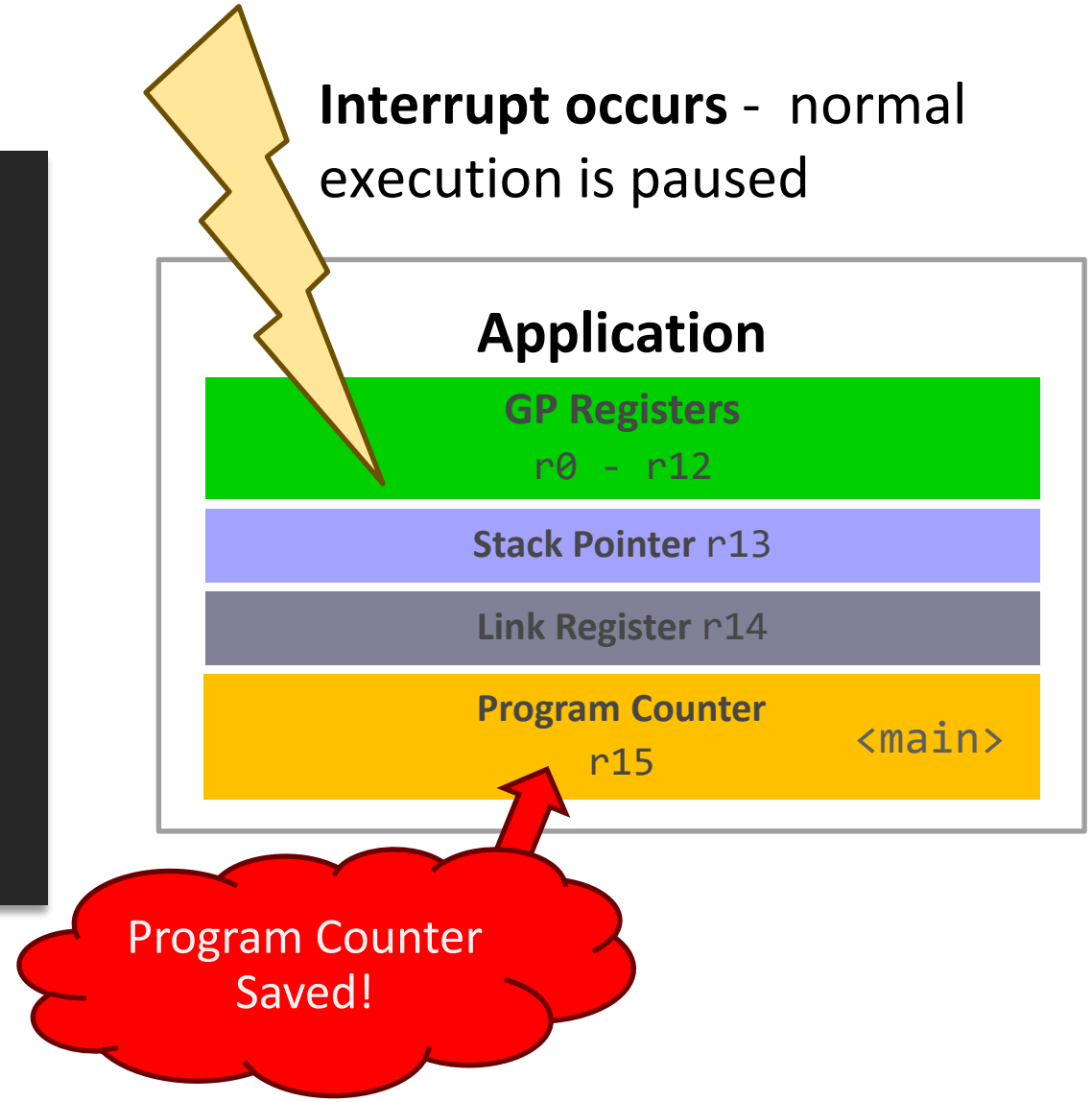
CPU

“Bare-metal” programming - Recap. Execution



“Bare-metal” programming - Recap. Interrupts

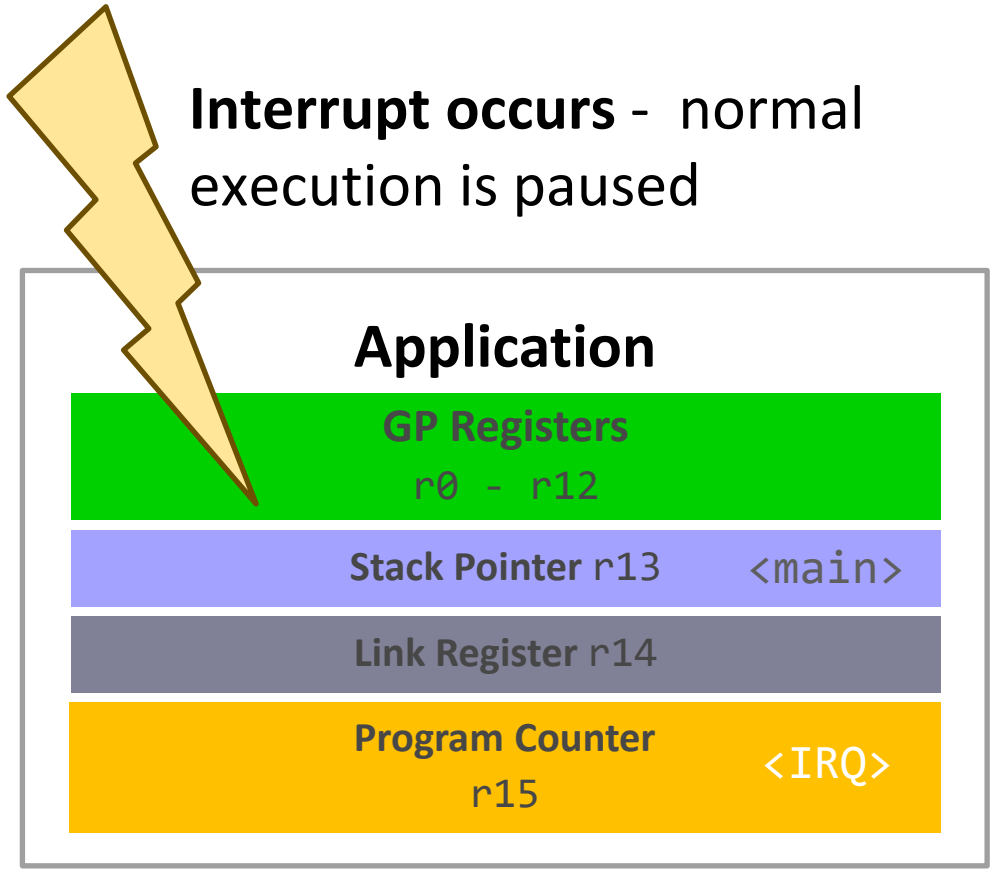
```c
int main() {

    init(); // Initialize GPIO

    while(1){
        // Toggle the LED pin
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        // Delay 500 ms
        HAL_Delay(500);
    }
}
```



“Bare-metal” programming - Recap. Interrupts

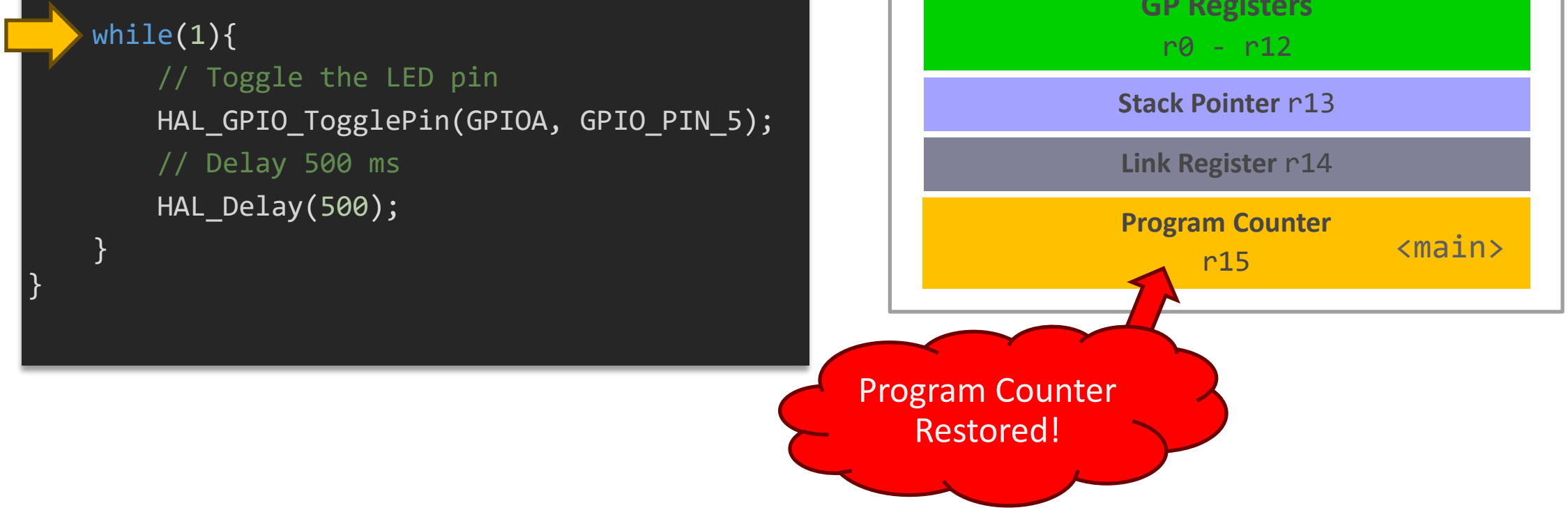
```
int main() {

    init(); // Initialize GPIO

    whi

        PIO_PIN_5);

    }
}
```

```
void IRQ(void) {
    // Handle Interrupt
    // ...
}
```



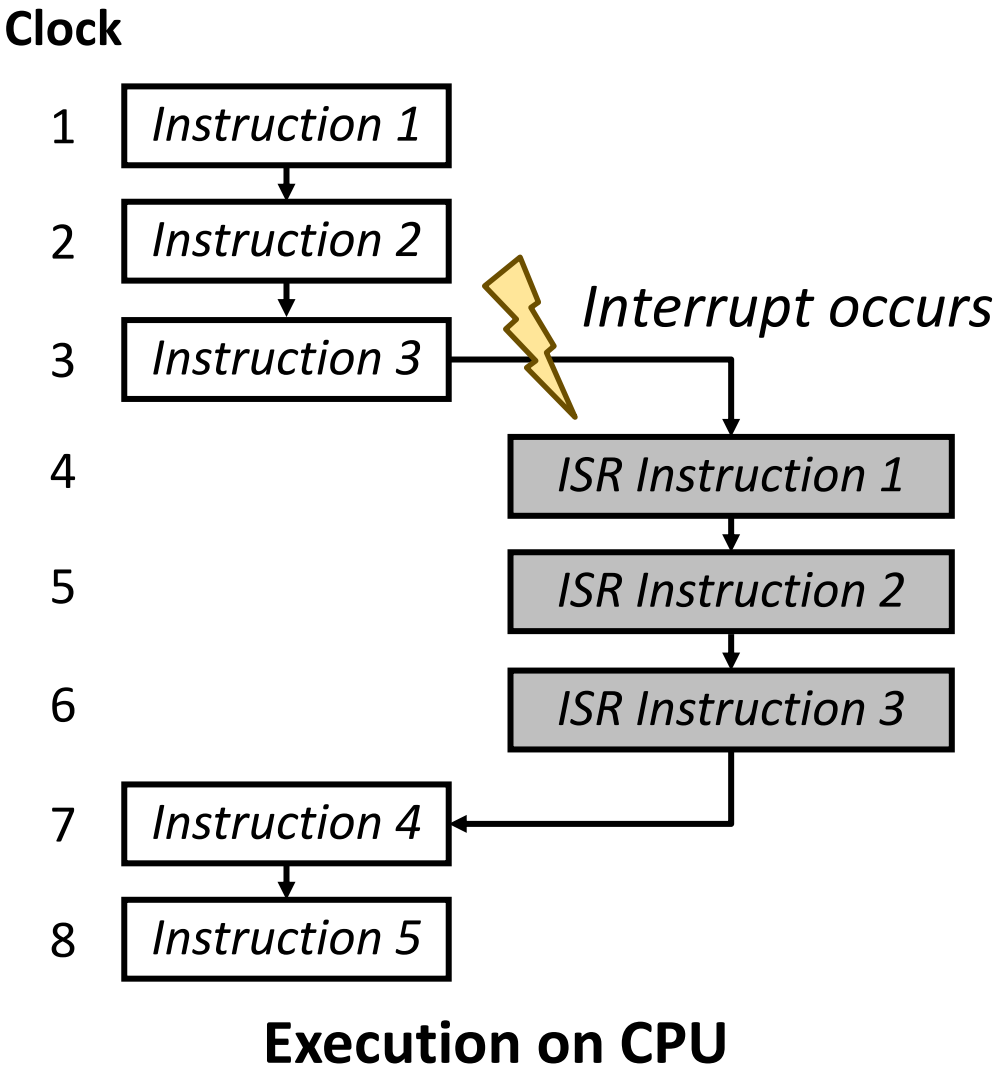
“Bare-metal” programming - Recap. Interrupts

```c
int main() {

    init(); // Initialize GPIO

    while(1){
        // Toggle the LED pin
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        // Delay 500 ms
        HAL_Delay(500);
    }
}
```



“Bare-metal” programming - Recap. Interrupts

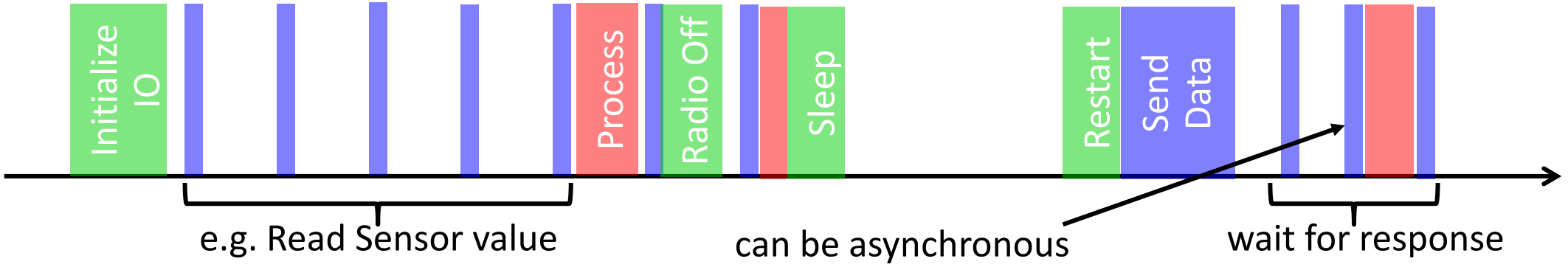
```c
int main() {

    init(); // Initialize GPIO

    while(1){
        // Toggle the LED pin
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        // Delay 500 ms
        HAL_Delay(500);
    }
}
```



Why Multiple Tasks in one Application?

Fundamental problem: ***1 CPU can generally execute 1 set of instructions at a time.***

- Applications generally consist of many different tasks running *concurrently*
  - *IO* – e.g. read sensor data, sending data over SPI…
  - *Processing* – e.g. computing the mean and std. of some data…
  - *System Management* – e.g. entering sleep mode, initializing a peripheral



We need an efficient way to share the CPU resources between all the tasks.

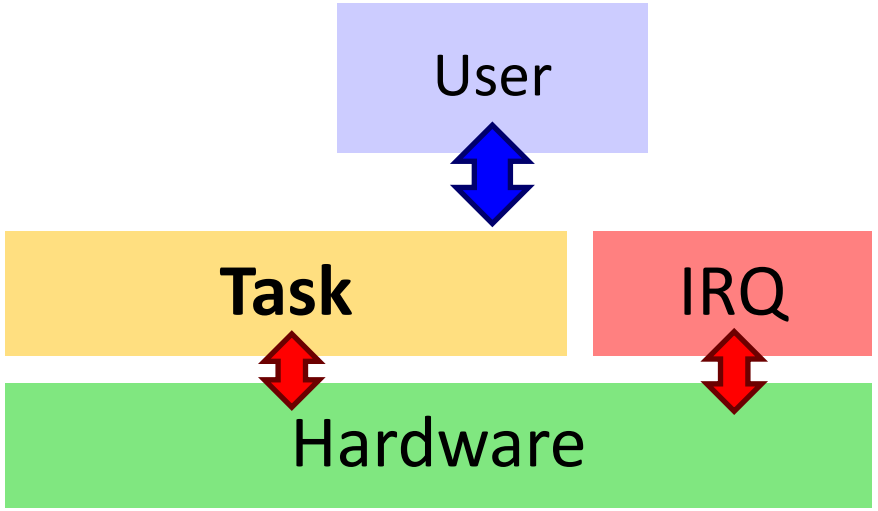
- Can be done on bare metal.
- Becomes exponentially more difficult with growing number of tasks

Summary: “Bare-metal” programming

- ✓ Simplicity
- ✓ Deterministic Execution
- ✓ Simple-multitasking with interrupts
- ✓ Absolute control

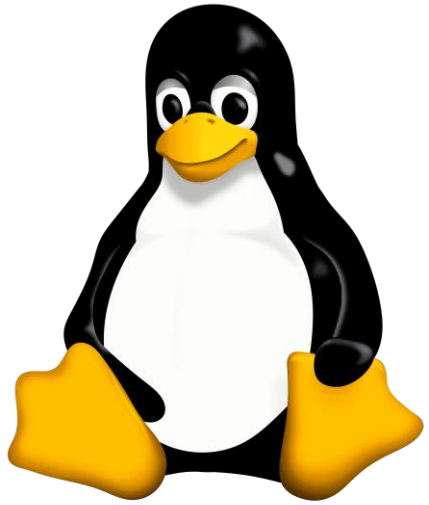
- × Scalability Issues
  - Code becomes complex if trying to handle multiple tasks.
- × Inefficient Resource Utilization
- × Limited Real-time capabilities
- ➢ Use an **Operating System**

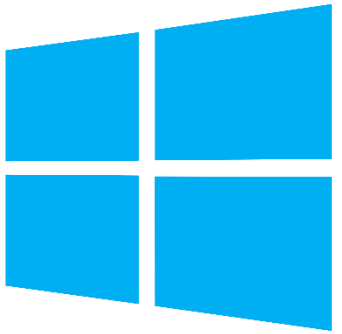
```c
int main() {

    init(); // Initialize GPIO

    while(1){
        // Toggle the LED pin
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        // Delay 500 ms
        HAL_Delay(500);
    }
}
```

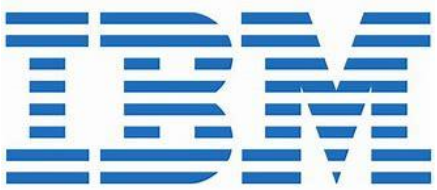


Operating Systems

What is an Operating System?

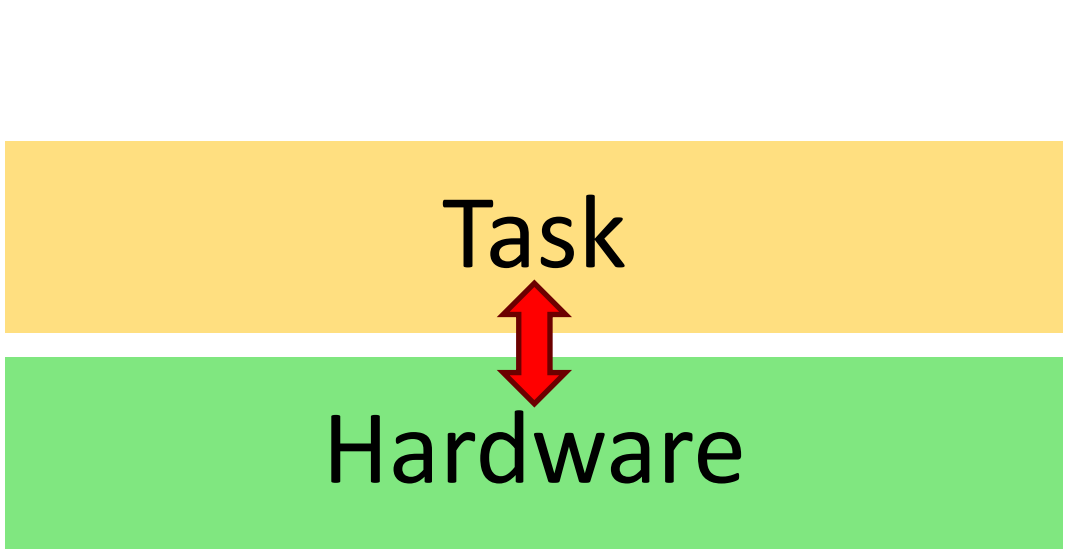




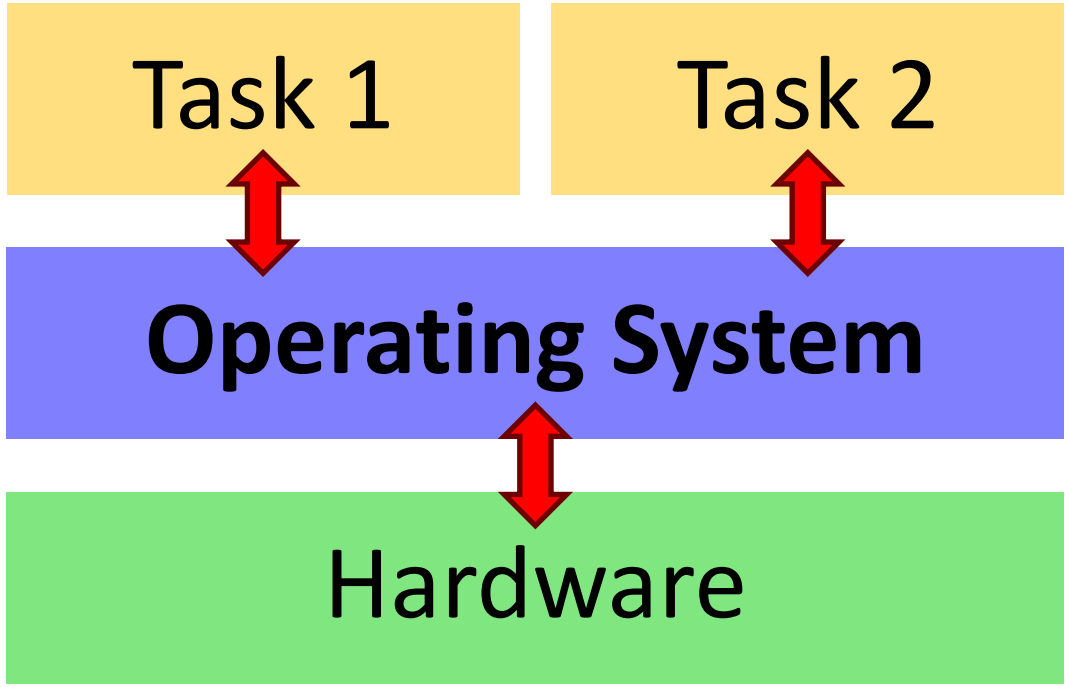


What is an Operating System?

An *Operating System* is a software layer that manages hardware resources and provides a foundation for multitasking, task scheduling, and inter-task communication.



“Bare-metal”



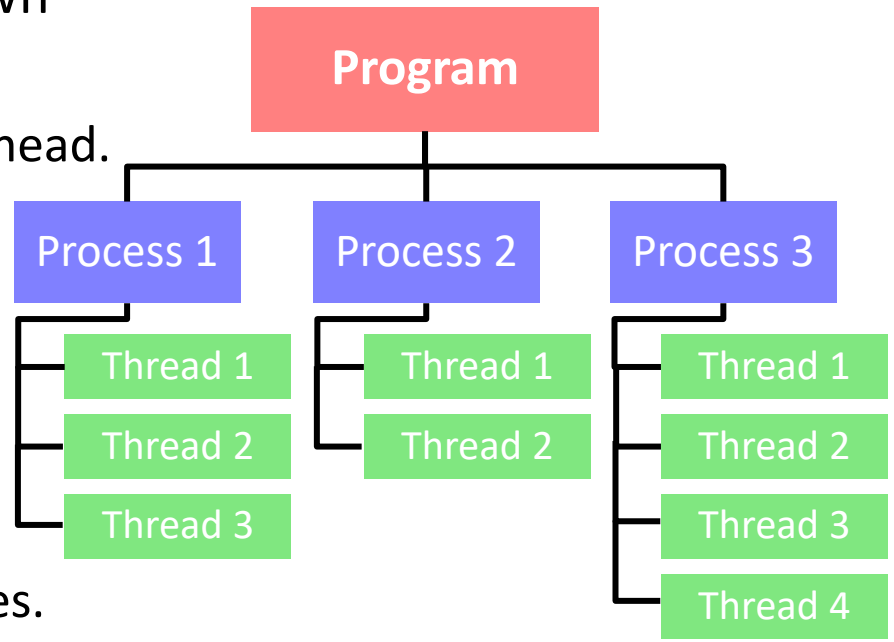
Operating System

Threads & Tasks

Task vs Process vs Thread

The definitions are sometimes unclear, differ by OS and overlap.

- *General Purpose Operating Systems* (Linux, Windows, Mac, etc.)
  - *Process* is an instance of a program execution with own memory space and resources.
    - Its heavy-weight and starting one has significant overhead.
    - It is generally managed by the OS.
  - *Thread* is a lightweight unit of execution within a process that shares memory space with other threads in the same process.
    - It has lower overhead compared to a Process
    - Can be managed either by the OS or user-level libraries.
  - *Task* often synonymous to Process or Thread – a set of instructions to be executed
- *Real-Time Operating Systems* on Microcontrollers
  - Generally, only have *Threads* often called interchangeably *Tasks*.



Hard Real-Time Computing Systems, page 23

Threads

A *Thread* (Task) is the **smallest sequence** of programmed instructions that can be **managed** independently **by a scheduler**; e.g., a thread is a basic unit of CPU utilization.

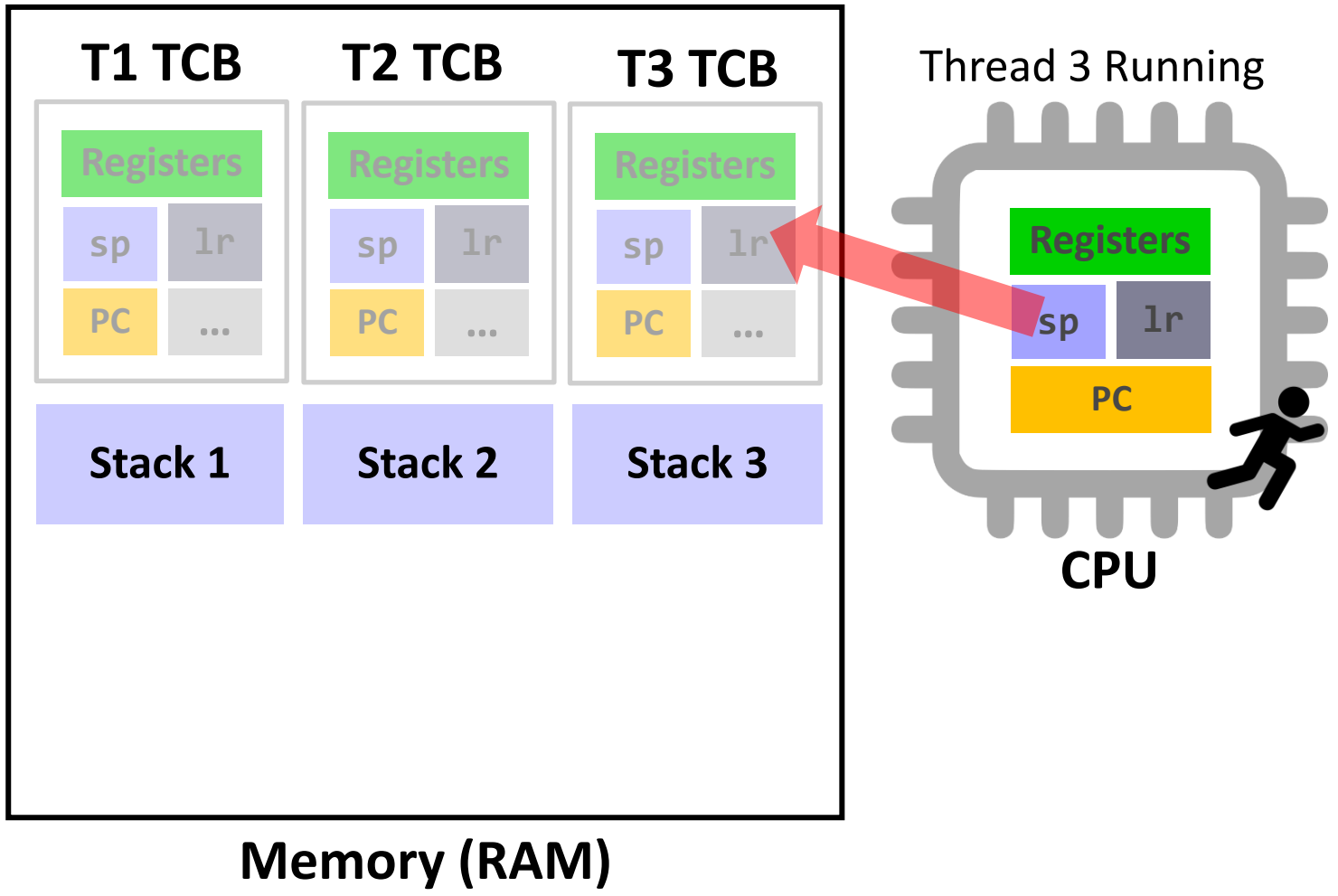
- Typically, an application will have a separate thread for each distinct activity.
  - Threads typically **share the *memory***
  - **Threads have own *registers* and *stack***
- A thread has its own local state called *Activation Record* or *Context*.
  - register values (GP Registers + sp + lr)
  - memory stack in RAM (local variables)
  - program counter (PC)
- *Thread Control Block (TCB)*
  - Data structure stores information needed to manage and schedule a thread
  - This includes the Context.
- Thread advantages
  - Faster to switch between threads - no major intervention by the operating system.

TCB

***Context:* General-purpose (GP) registers, stack pointer (SP), and link register (LR) are specific to each thread.**

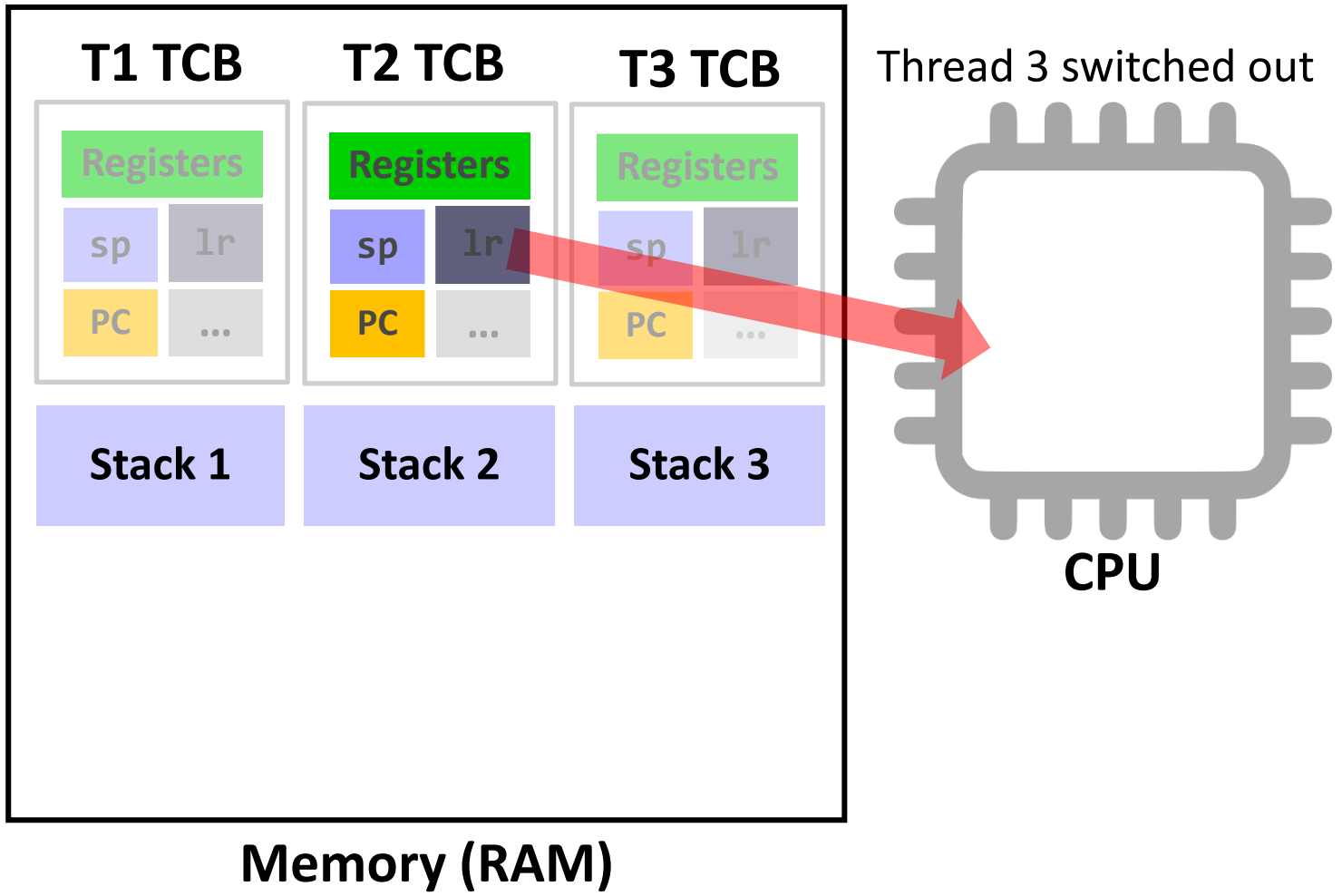
Thread Context Switch

1. Thread 3 is currently running on CPU
2. Context switch starts
   - Thread 3 context is moved out of the CPU and stored into the TCB



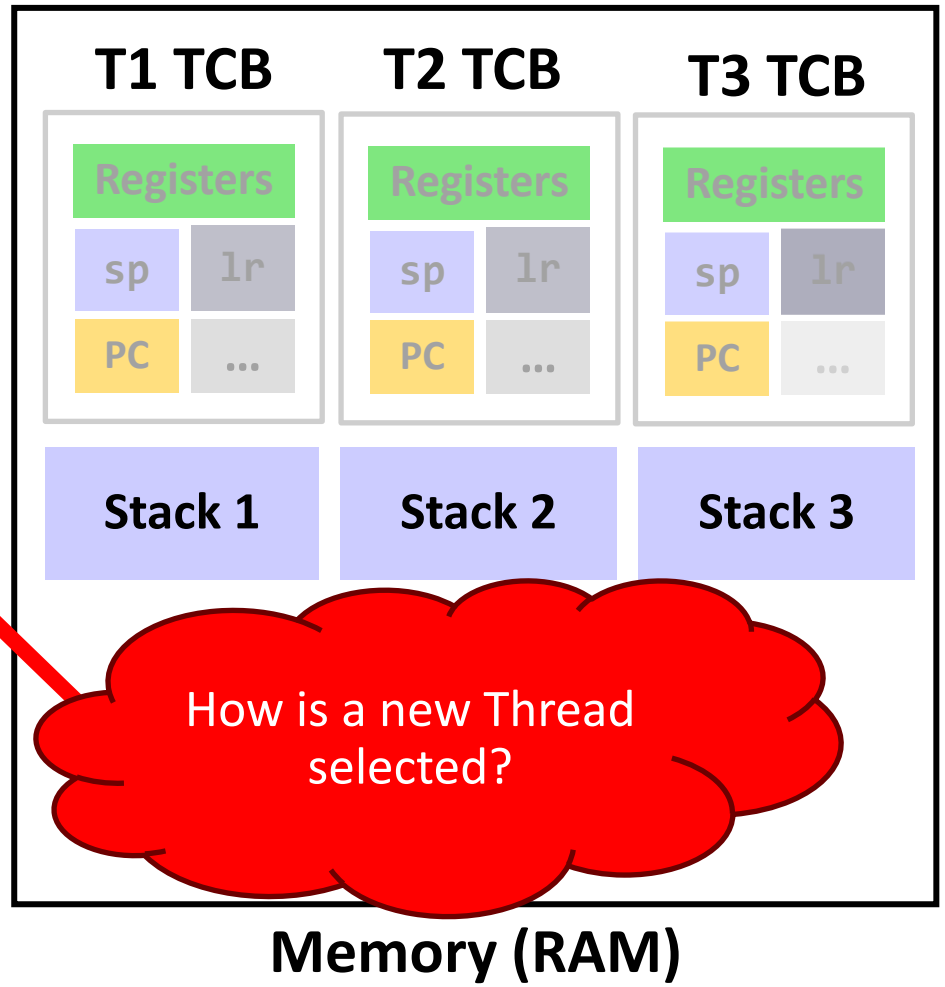
Thread Context Switch

1. Thread 3 is currently running on CPU
2. Context switch starts
   - Thread 3 context is moved out of the CPU and stored into the TCB
3. A new Thread is selected
4. The Stored context of the thread is restored into the CPU registers from the TCB

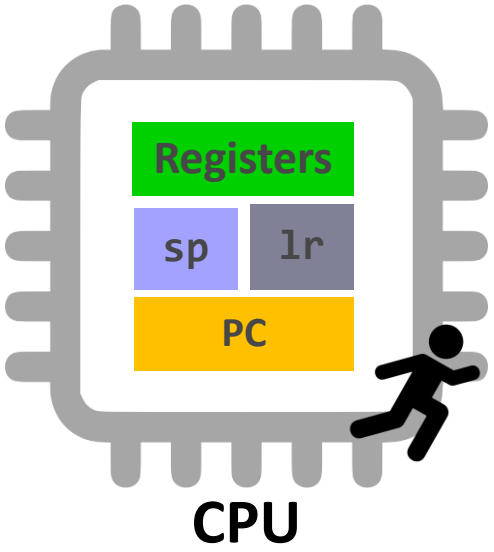


Thread Context Switch

1. Thread 3 is currently running on CPU
2. Context switch starts
   - Thread 3 context is moved out of the CPU and stored into the TCB
3. A new Thread is selected
4. The Stored context of the thread is restored into the CPU registers from the TCB
5. Thread 2 continues running.



Thread 2 switched in and continues running

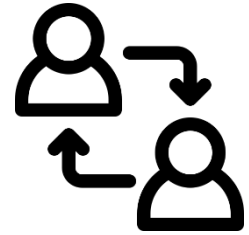


Main Functionality of an Operating System

Task management:

- *Execution of quasi-parallel Tasks* on a processor using *processes* or *threads*
  - maintaining process states, process queuing
  - allowing for preemptive tasks (fast context switching) and quick interrupt handling
- *CPU scheduling* (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
- *Inter-task communication* (buffering)
- *Support of real-time clocks*
- *Task synchronization* (critical sections, semaphores, monitors, mutual exclusion)
  - In classical operating systems, synchronization and mutual exclusion is performed via semaphores and monitors.
  - In **Real-Time OS**, special semaphores and a deep integration of them into scheduling is necessary (for example priority inheritance protocols as described in a later chapter).

Interaction: How to select a next Thread to run?

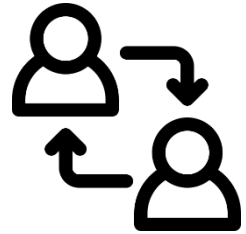


Discuss, how can we best select the next Thread to run?

- What should the choice consider? - What are the constraints?
- How efficient is this strategy?
- What are the advantages and limitations?

Icon by Freepik

Potential Discussed topics

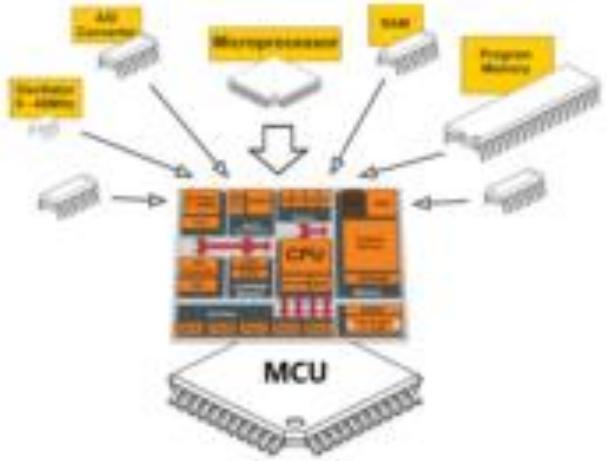


Selection Criteria:
•Prioritize threads based on urgency (priority level), current state (ready, blocked), resource availability, deadline requirements, and energy efficiency.
Constraints:
•Consider real-time constraints, limited CPU and memory resources, and the need for deterministic behavior in critical applications.
Efficiency:
•Assess the scheduling strategy based on context switching overhead, CPU resource utilization, and system response time to events.
Advantages:
•Effective thread scheduling improves responsiveness, resource efficiency, and simplifies inter-thread communication by sharing memory.
Limitations:
•Challenges include complexity in implementation, synchronization issues (e.g., race conditions), and potential problems like priority inversion if not managed properly.

Scheduling : The mechanism to implement the selections of threads

Overview

- *Time triggered approaches*
  - periodic
  - cyclic executive
  - generic time-triggered scheduler
- *Event triggered approaches*
  - non-preemptive
  - preemptive – stack policy
  - preemptive – cooperative scheduling
  - preemptive - multitasking

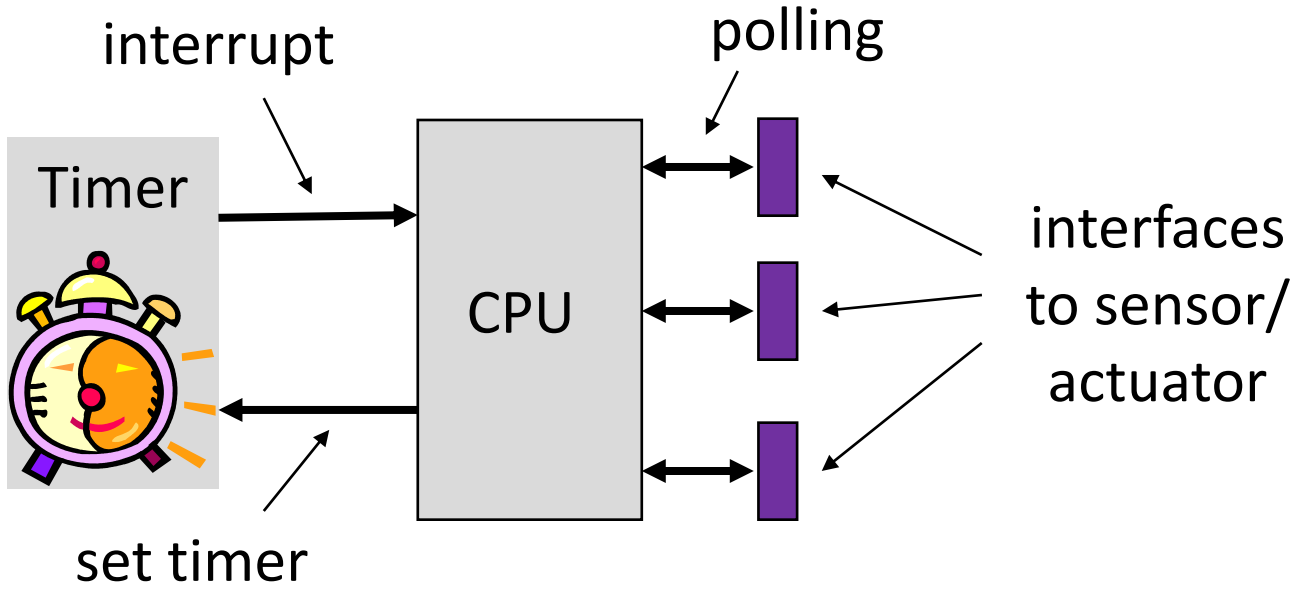


We assume systems contain a single processor

Time-Triggered Systems - Case of single CPU

Pure time-triggered model:

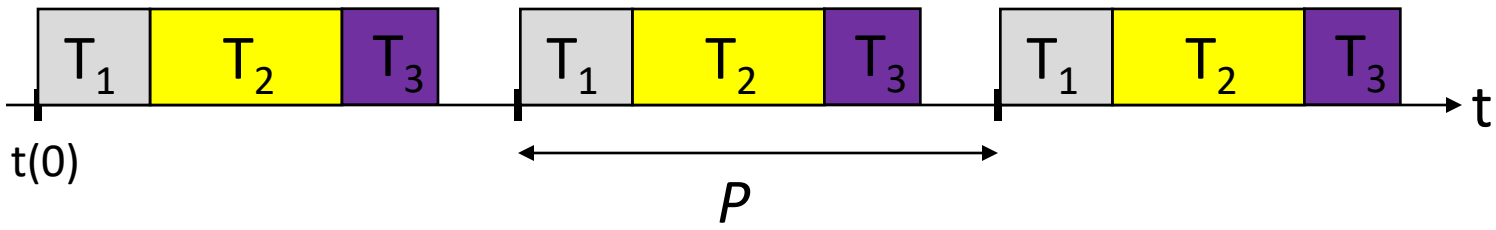
- *no interrupts* are allowed, **except by timers**
- the *schedule* of tasks is *computed off-line* and therefore, complex sophisticated algorithms can be used
- the scheduling at run-time is fixed and therefore, it is *deterministic, crucial for Safety-critical applications.*
- the interaction with environment (sensors, actuators) happens through *polling*



The **predictable timing and high determinism** of such systems are ideal for applications where delays or unexpected behaviors could lead to serious consequences.

Simple Periodic TT Scheduler

- A *timer interrupts regularly* with period *P*.
- All tasks have *same period P*.



- *Properties:*
  - later tasks, for example T_2 and T_3, have unpredictable starting times
  - the communication between tasks or the use of common resources is safe, as there is a static ordering of tasks, for example T_2 starts after finishing T_1
  - as **a necessary precondition**, the sum of WCETs of all tasks within a period is bounded by the period *P*:

$$\sum_{(k)} WCET(T_k) < P$$

Simple Periodic Time-Triggered Scheduler

```
main:
   determine table of tasks (k, T(k)), for k=0,1,…,m-1;
   i=0; set the timer to expire at initial phase t(0);
   while (true) sleep();

Timer Interrupt:
   i=i+1;
   set the timer to expire at i*P + t(0);
   for (k=0,…,m-1){ execute task T(k); }
   return;
```

usually done offline

set CPU to low power mode;
processing starts again after interrupt

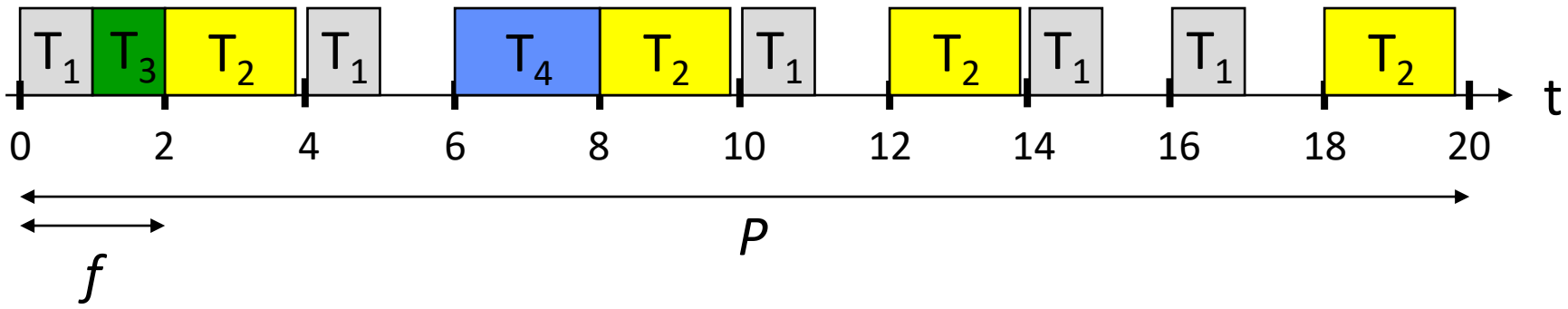
for example using a function pointer in C;
task(= function) returns after finishing.

k	T(k)
0	T_1
1	T_2
2	T_3
3	T_4
4	T_5

m=5

Time-Triggered Cyclic Executive Scheduler

- Suppose now, that *tasks may have different periods*.
- To accommodate this situation, the *period P is partitioned into frames of length f*.



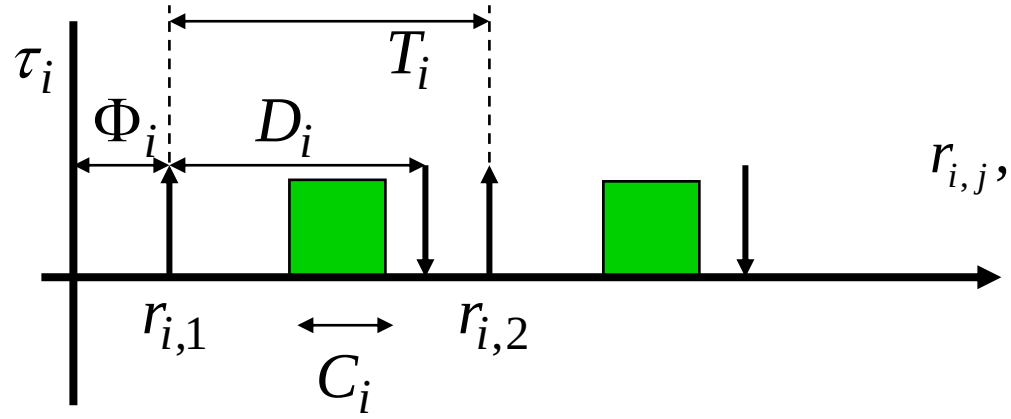
- We have a *problem* to determine a feasible schedule, if there are *tasks with a long execution time*.
  - long tasks could be partitioned into a sequence of short sub-tasks
  - but this is tedious and error-prone process, as the local state of the task must be extracted and stored globally

Time-Triggered Cyclic Executive Scheduling

- *Examples for periodic tasks:* sensory data acquisition, control loops, action planning and system monitoring.
- When a control application consists of several concurrent periodic tasks with individual timing constraints, *the schedule has to guarantee* that each periodic instance is *regularly activated* at its proper rate and is *completed within its deadline*.

Time-Triggered Cyclic Executive Scheduling

- *Example* of a single periodic task τ_i:



Definitions:

Γ : denotes the set of all periodic tasks

τ_i : denotes a periodic task

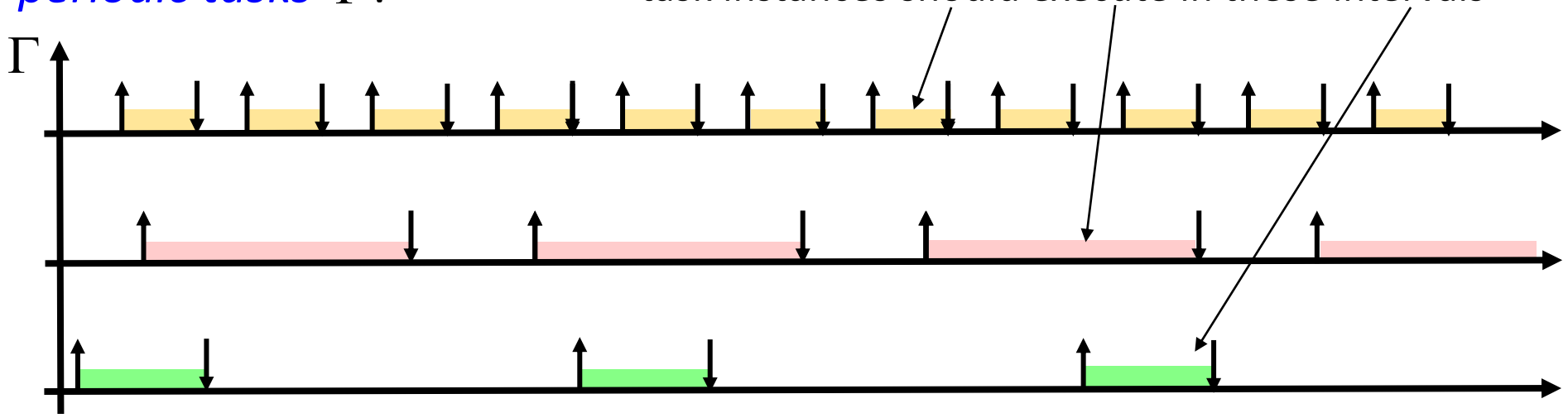
$\tau_{i,j}$: denotes the *j*th instance of task *I*

$r_{i,j}$, $d_{i,j}$: denote the release time *j*'th instance of task ; di, the jth absolute deadline.

Φ_i : phase of task *i* (release time of its first instance)

D_i : relative deadline of task *i*

- *A set of periodic tasks* Γ:



Time-Triggered Cyclic Executive Scheduling

- The following *hypotheses* are assumed on the tasks:
  - *The instances of a periodic task are regularly activated at a constant rate*. The interval T_i between two consecutive activations is called period. The release times satisfy

$$r_{i,j} = \Phi_i + (j-1)T_i$$

  - *All instances have the same worst case execution time* C_i . The worst case execution time is also denoted as *WCET*(*i*) .
  - *All instances of a periodic task have the same relative deadline* D_i. Therefore, the absolute deadlines satisfy

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

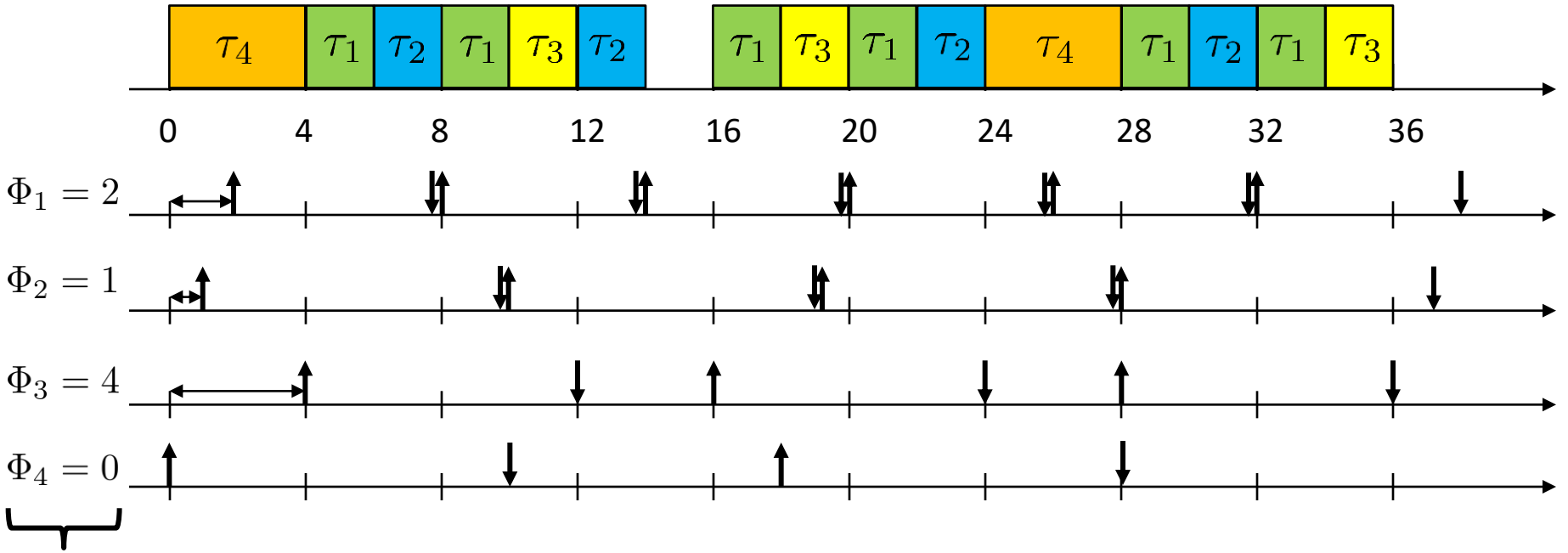
Time-Triggered Cyclic Executive Scheduling

Example with 4 tasks:

- $\tau_1 : T_1 = 6, D_1 = 6, C_1 = 2$ $\quad \tau_2 : T_2 = 9, D_2 = 9, C_2 = 2$
  $\tau_3 : T_3 = 12, D_3 = 8, C_3 = 2$ $\quad \tau_4 : T_4 = 18, D_4 = 10, C_1 = 4$
- $P = 36, f = 4$

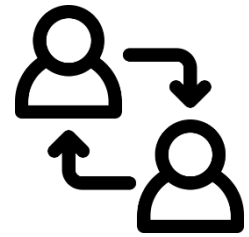
requirement

schedule



not given as part of the requirement

Time-Triggered Cyclic Executive Scheduling



Checking for correctness of schedule:

- f_{ij} denotes the number of the frame in which that instance *j* of task τ_i executes.
- Is *P* a common multiple of all periods T_i ?
- Is *P* a multiple of *f* ?
- Is the frame sufficiently long?

$$\sum_{\{i \mid f_{ij}=k\}} C_i \leq f \qquad \forall 1 \leq k \leq \frac{P}{f}$$

- Determine offsets such that instances of tasks start after their release time:

$$\Phi_i = \min_{1 \leq j \leq P/T_i} \{(f_{ij} - 1)f - (j-1)T_i\} \qquad \forall \text{ tasks } \tau_i$$

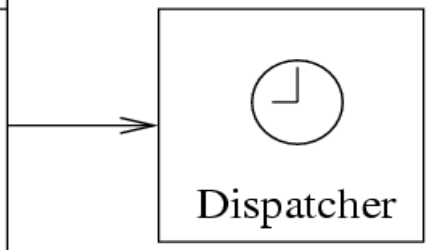
- Are deadlines respected?

$$(j-1)T_i + \Phi_i + D_i \geq f_{ij} f \qquad \forall \text{ tasks } \tau_i \,,\, 1 \leq j \leq P/T_i$$

Generic Time-Triggered Scheduler

- In an *entirely time-triggered system*, the temporal control structure of all tasks is established a priori by off-line support-tools.
- This *temporal control structure is encoded in a Task-Descriptor List (TDL)* that contains the cyclic schedule for all activities of the node.
- This *schedule* considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary.
- *The dispatcher is activated by a synchronized clock tick.* It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].

Time	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T2	20
47	send M3	



Simplified Time-Triggered Scheduler

```
main:
   determine static schedule (t(k), T(k)), for k=0,1,…,n-1;
   determine period of the schedule P;
   set i=k=0 initially; set the timer to expire at t(0);
   while (true) sleep();

Timer Interrupt:
   k_old := k;
   i := i+1; k := i mod n;
   set the timer to expire at ⌊i/n⌋ * P + t(k);
   execute task T(k_old);
   return;
```

usually done offline

set CPU to low power mode;
processing continues after interrupt

for example using a function pointer in C;
task returns after finishing.

k	t(k)	T(k)
0	0	T_1
1	3	T_2
2	7	T_1
3	8	T_3
4	12	T_2

n=5, P = 16

Summary Time-Triggered Scheduler

Properties:

- *deterministic schedule*; conceptually simple (static table); relatively easy to validate, test and certify
- *no problems* in using *shared resources*

- external communication only via *polling*
- *inflexible* as no adaptation to the environment
- serious *problems* if there are *long tasks*

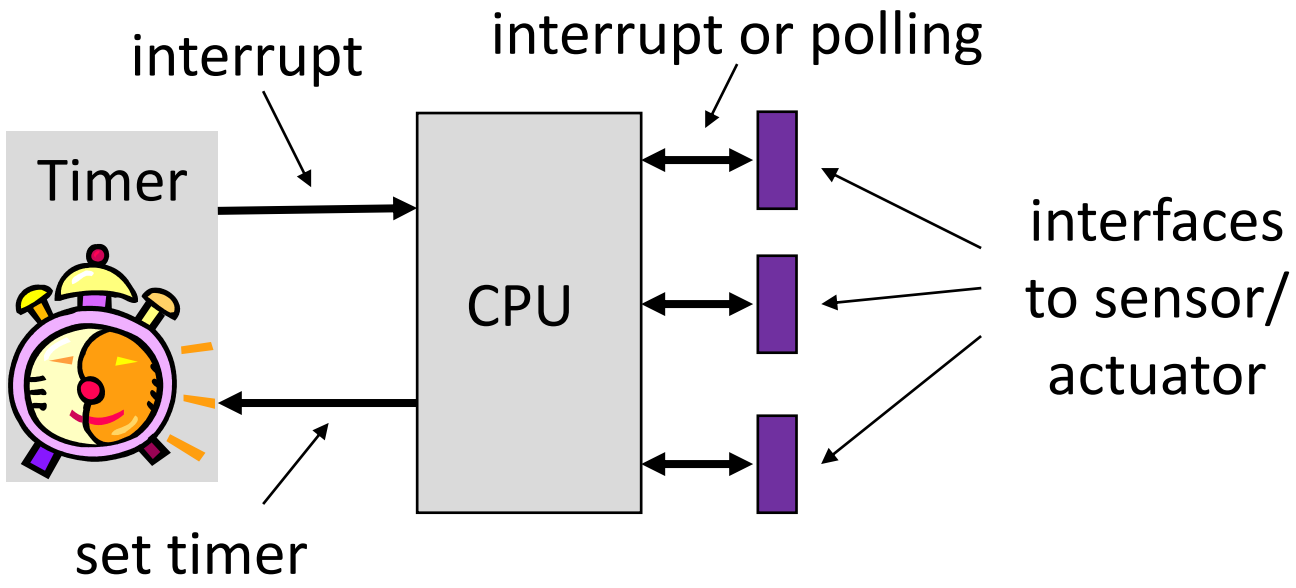
Extensions:

- *allow interrupts* → be careful with shared resources and the WCET of tasks!!
- *allow preemptable* background tasks
- *check for task overruns* (execution time longer than WCET)

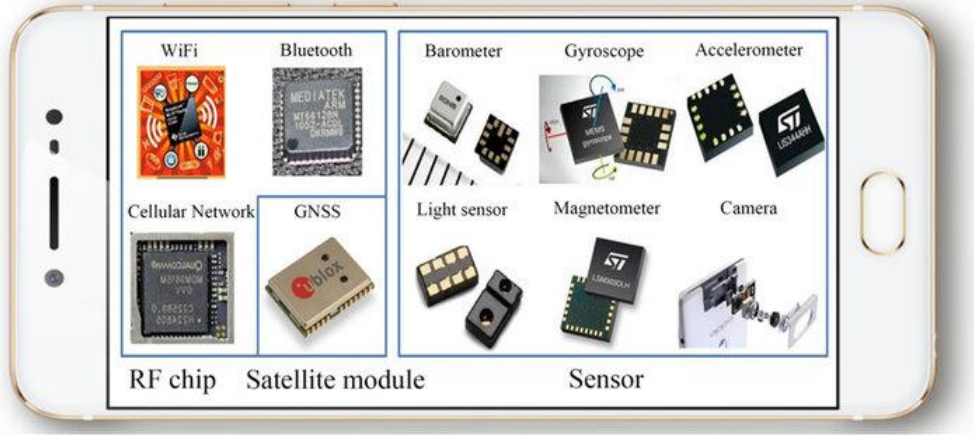
Event Triggered Systems

The schedule of tasks is determined by the occurrence of external or internal events:

- *dynamic and adaptive*: there are possible problems with respect to timing, the use of shared resources and buffer over- or underflow
- *guarantees* can be given either off-line (if bounds on the behavior of the environment are known) or during run-time



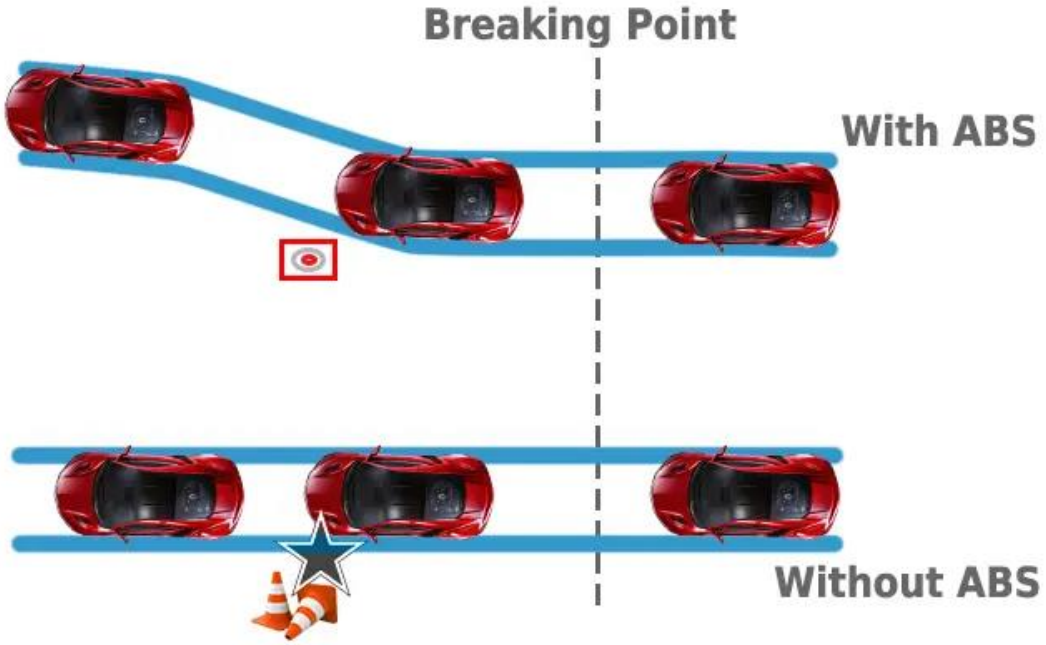
Wen, DanQi. "Research progress of smart phone positioning and navigation sensors." In *Journal of Physics: Conference Series*, vol. 1930, no. 1, p. 012013. IOP Publishing, 2021.



This **flexible, reactive nature** of event-driven scheduling makes it ideal for applications where events occur unpredictably and tasks must respond quickly, such as responding to a user's touch, handling network communications, or processing data from a sensor in an IoT device.

Non-Preemptive Event-Triggered Scheduling

- In an ABS embedded system, sensors and control mechanisms must work closely to monitor and react to wheel slip conditions to prevent skidding. The system architecture and event scheduling follow the Non-Preemptive Event-Triggered Scheduling Principle to ensure critical tasks (such as wheel slip detection and response) are handled in a structured, deterministic manner.



Non-Preemptive Event-Triggered Scheduling

Principle:

- To each event, there is associated a corresponding task that will be executed.
- Events are emitted by (a) external interrupts or (b) by tasks themselves.
- All events are collected in a single queue; depending on the queuing discipline, an event is chosen for execution, i.e., the corresponding task is executed.
- Tasks can **not be preempted**.

Extensions:

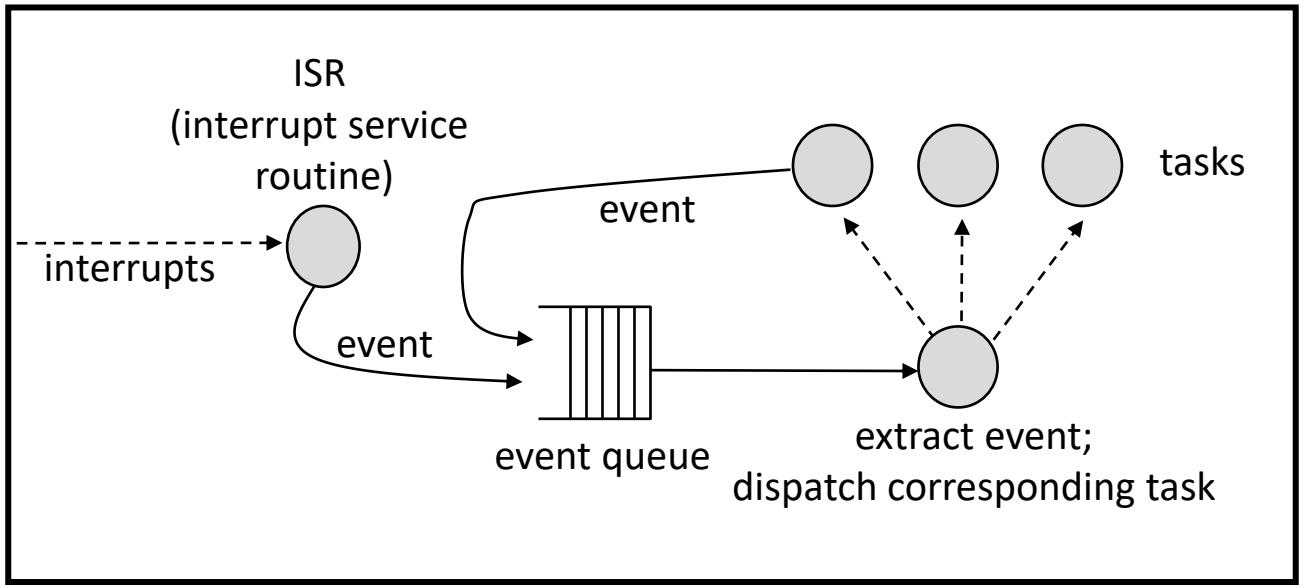
- A *background task* can run if the event queue is empty. It will be preempted by any event processing.
- *Timed events* are ready for execution only after a time interval elapsed. This enables periodic instantiations, for example.

Non-Preemptive Event-Triggered Scheduling

```
main:
   while (true) {
        if (event queue is empty) {
               sleep();
        } else {
               extract event from event queue;
               execute task corresponding to event;
        }
   }

Interrupt:
   put event into event queue;
   return;
```

set the CPU to low power mode;
continue processing after interrupt

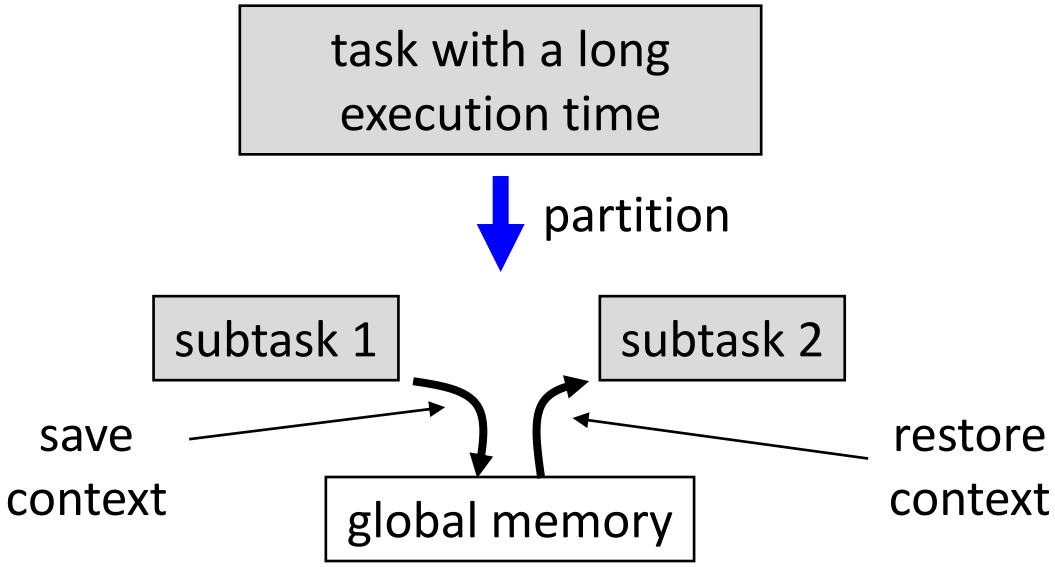
for example using a function pointer in C;
task returns after finishing.



Non-Preemptive Event-Triggered Scheduling

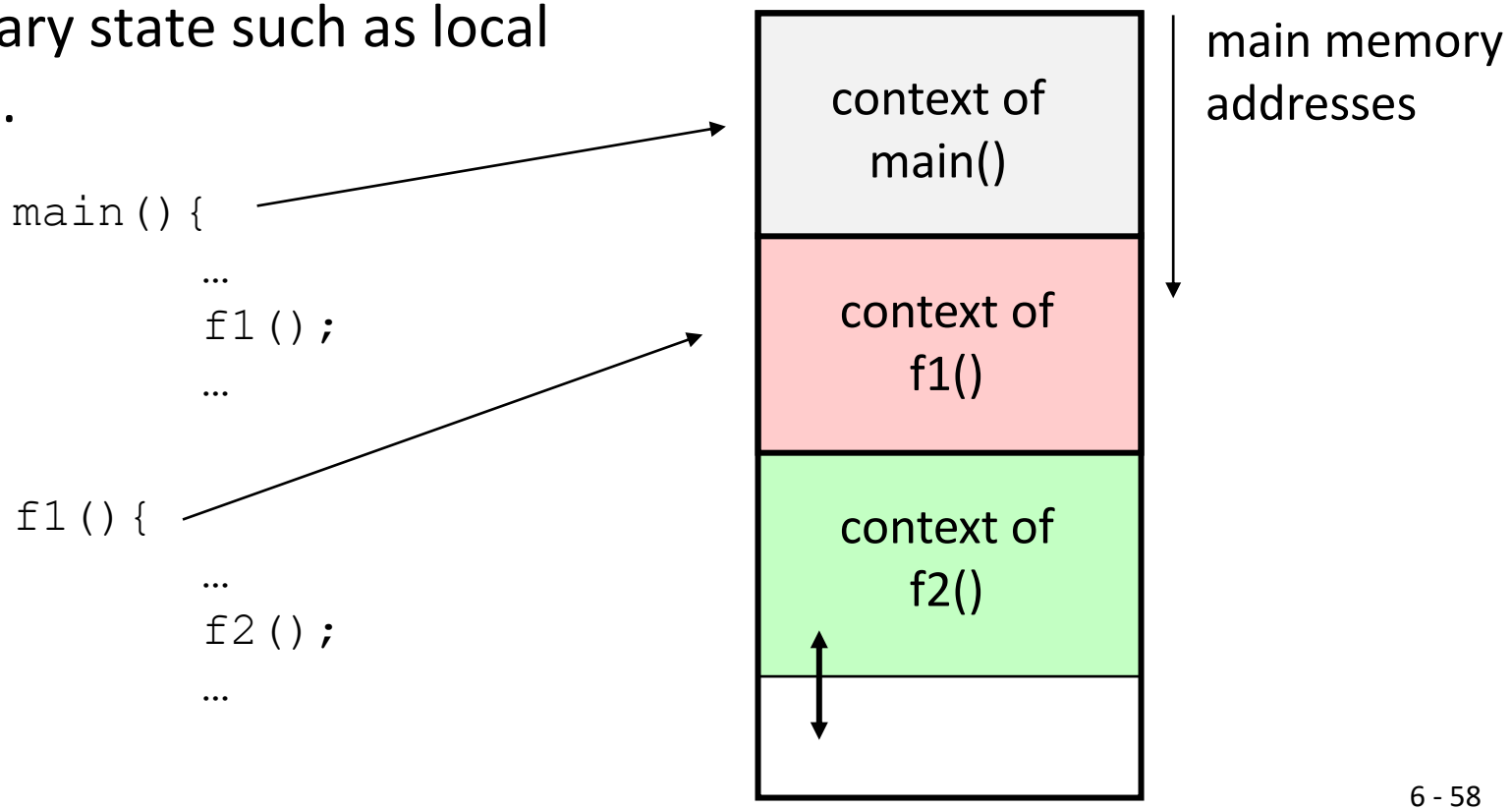
Properties:

- *communication between tasks* does **not lead** to a simultaneous access to shared resources, but interrupts may cause problems as they preempt running tasks
- *buffer overflow* may happen if too many events are generated by the environment or by tasks
- *tasks with a long running time* prevent other tasks from running and may cause buffer overflow as no events are being processed during this time
  - partition tasks into smaller ones
  - but the local context must be stored

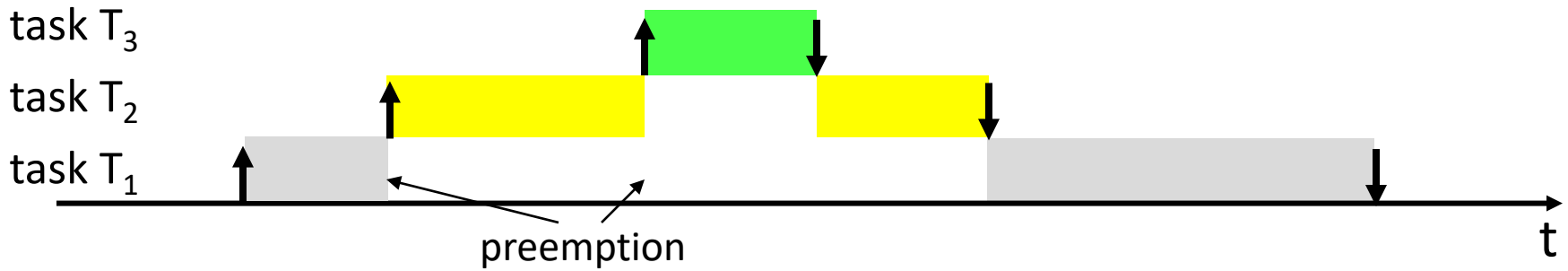


Preemptive Event-Triggered Scheduling – Stack Policy

- This case is similar to non-preemptive case, but *tasks can be preempted by others*; this resolves partly the problem of tasks with a long execution time.
- If *the order of preemption is restricted*, we can use the usual stack-based context mechanism of function calls. The context of a function contains the necessary state such as local variables and saved registers.



Preemptive Event-Triggered Scheduling – Stack Policy



- *Tasks must finish in LIFO (last in first out) order* of their instantiation.
  - this restricts flexibility of the approach
  - it is not useful, if tasks wait some unknown time for external events, i.e., they are blocked
- *Shared resources* (communication between tasks!) *must be protected*, for example by disabling interrupts or by the use of semaphores.

Preemptive Event-Triggered Scheduling – Stack Policy

```
main:
   while (true) {
        if (event queue is empty) {
               sleep();
        } else {
               select event from event queue;
               execute selected task;
               remove selected event from queue;
        }
   }

InsertEvent:
   put new event into event queue;
   select event from event queue;
   if (selected task ≠ running task) {
        execute selected task;
        remove selected event from queue;
   }
   return;
```

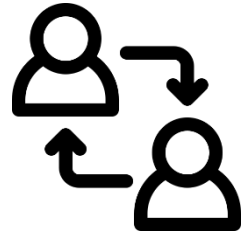
set CPU to low power mode; processing continues after interrupt

for example using a function pointer in C; task returns after finishing.

```
Interrupt:
   InsertEvent(…);
   return;
```

may be called by interrupt service routines (ISR) or tasks

Exercise: Task Management Scenario



Goal: Creating the Schedule with Stack-Based Preemption for the ABS with the following tasks-events

Event ID	Event Type	Task Description	Time to Complete
E1	External Interrupt	Wheel Speed Sensor (Left Wheel)	3 ms
E2	External Interrupt	Wheel Speed Sensor (Right Wheel)	3 ms
E3	Timed Event (10 ms)	Brake Pressure Adjustment	5 ms
E4	Internal Event	System Diagnostics	4 ms
E5	External Interrupt	Wheel Speed Sensor (Front Wheel)	3 ms

Event Arrival Timeline:

0 ms - E1 (Wheel Speed Sensor Left Wheel)

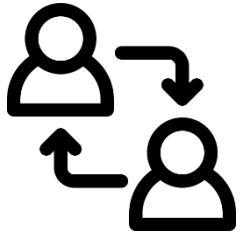
1 ms - E2 (Wheel Speed Sensor Right Wheel)

10 ms - E3 (Brake Pressure Adjustment, Timed Event)

12 ms - E4 (System Diagnostics, Internal Event)

15 ms - E5 (Wheel Speed Sensor Front Wheel)

Solution



Time Interval	Task Executed	Stack State	Event Queue State
0 - 1 ms	E1: Wheel Speed Sensor Left Wheel	[]	[E2]
1 - 2 ms	E2: Wheel Speed Sensor Right Wheel	[E1]	[E3]
10 - 12 ms	E3: Brake Pressure Adjustment	[E1, E2]	[E4]
12 - 16 ms	E4: System Diagnostics	[E1, E2, E3]	[E5]
15 - 18 ms	E5: Wheel Speed Sensor Front Wheel	[E1, E2, E3, E4]	[]
18 - 23 ms	E4: System Diagnostics (Resumed)	[E1, E2, E3]	[]
23 - 28 ms	E3: Brake Pressure Adjustment	[E1, E2]	[]
28 - 30 ms	E2: Wheel Speed Sensor Right Wheel	[E1]	[]
30 - 33 ms	E1: Wheel Speed Sensor Left Wheel	[]	[]

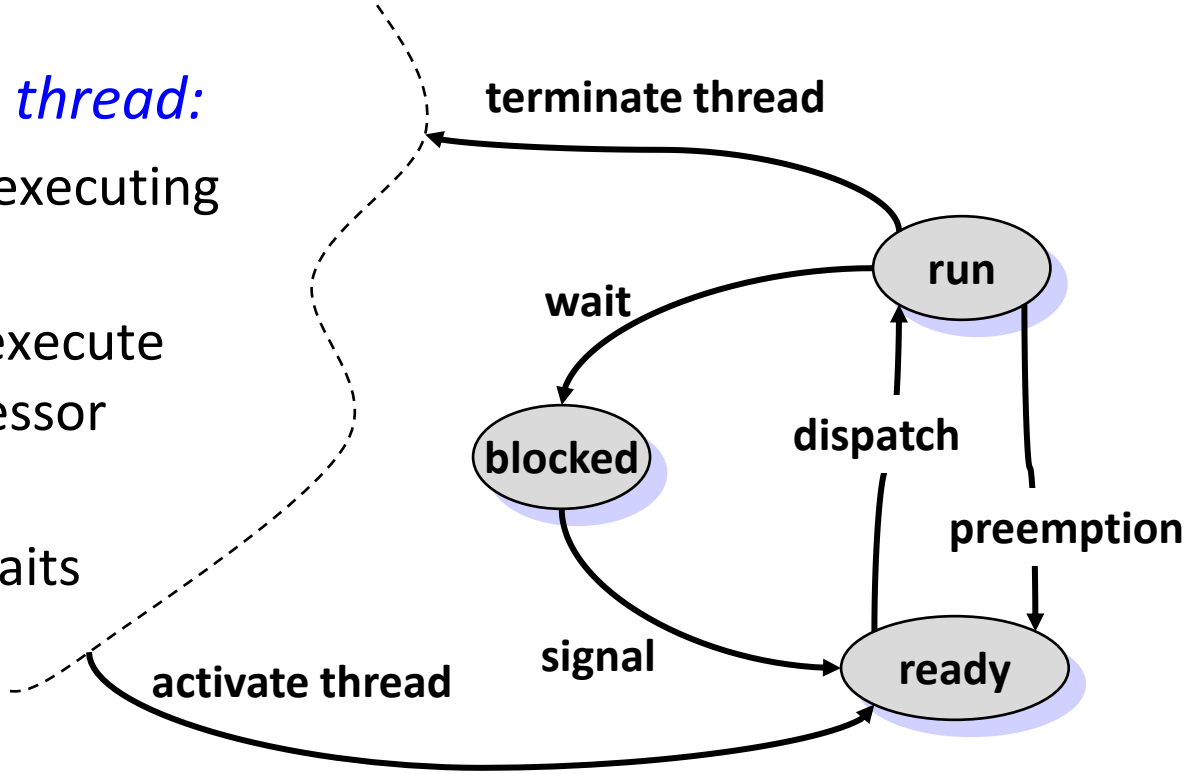
Is this the best scheduling for ABS?

Useful for demonstrating LIFO handling, might not be ideal. An ABS system is a real-time system where certain events, like brake pressure adjustments, need immediate, predictable responses to ensure safety.

Preemptive Multitasking

- *Most general form of multitasking:*
  - The scheduler in the runtime system (operating system) controls when contexts switches take place.
  - The scheduler also determines what thread runs next.

- *State diagram corresponding to each single thread:*
  - *Run:* A thread enters this state as it starts executing on the processor
  - *Ready:* State of threads that are ready to execute but cannot be executed because the processor is assigned to another thread.
  - *Blocked*: A task enters this state when it waits for an event.



What Did You Learn?

- ✓ Real-Time Systems
- ✓ Operating Systems
- ✓ Threads
- ✓ Multitasking
- ✓ Scheduling
- ✓ Embedded Operating Systems