Introduction to Embedded Systems

4. Programming Paradigms

Prof. Dr. Marco Zimmerling

Organization

Join ILIAS course:

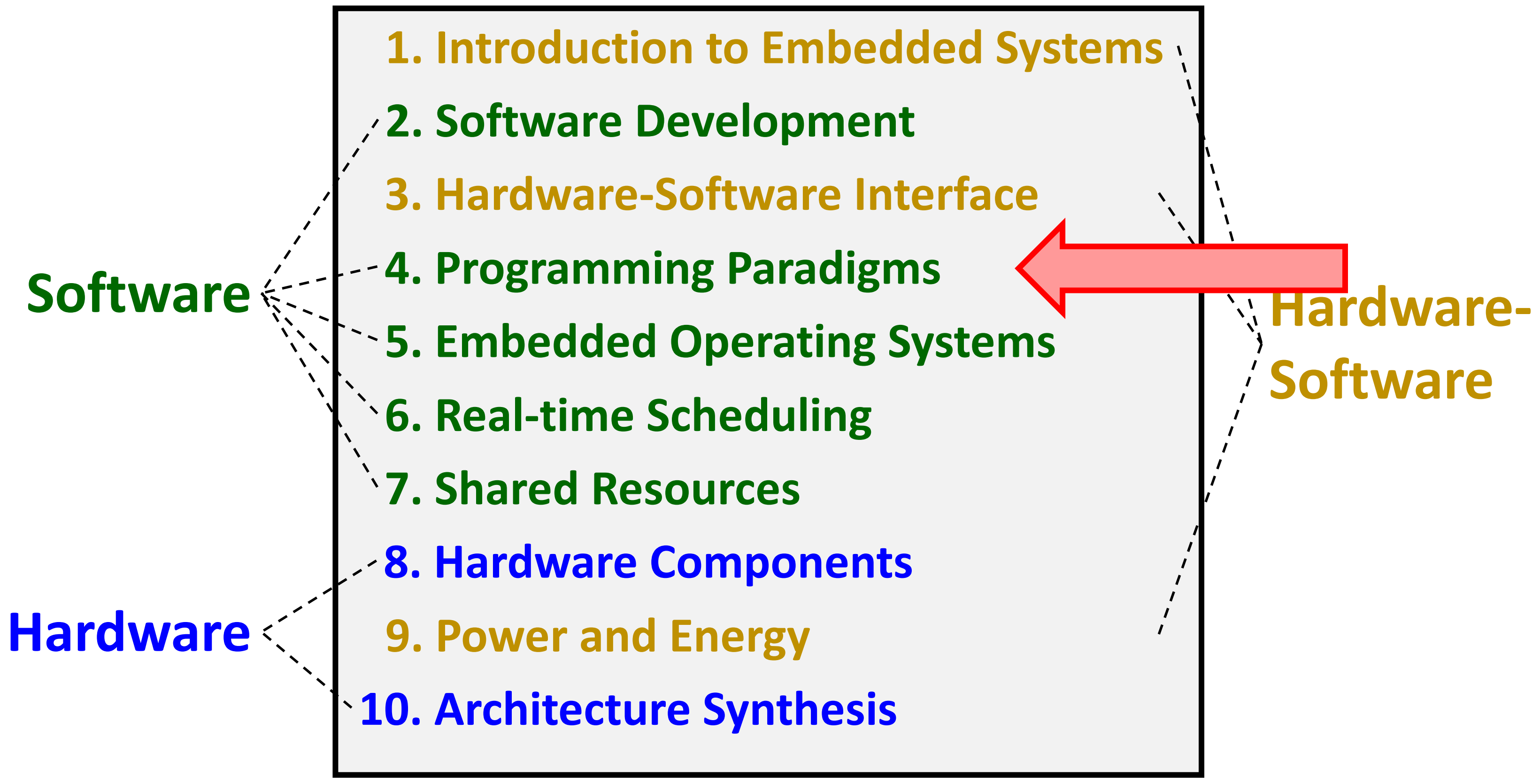
- Login: RZ username + password
- Course password: **es-0x8af**

Exercises:

- From 12:00 to 14:00 in HS 00-036 (English) and SR 02-016/18 (German)
- Today (November 22):
  - Solutions of second exercise sheet presented
- Next week (November 29):
  - Third exercise sheet released + overview of tasks given

Exam: March 4, 2023 from 10:00 to 12:00, five rooms in Georges-Köhler-Allee 101

Where we are …

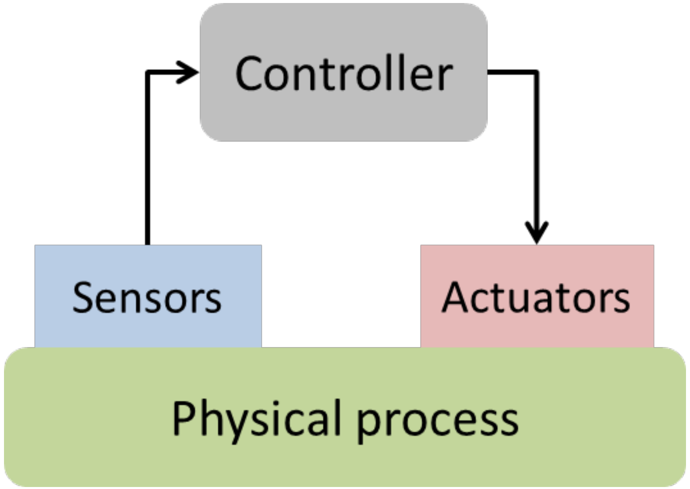


Software
Hardware
1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Hardware-
Software

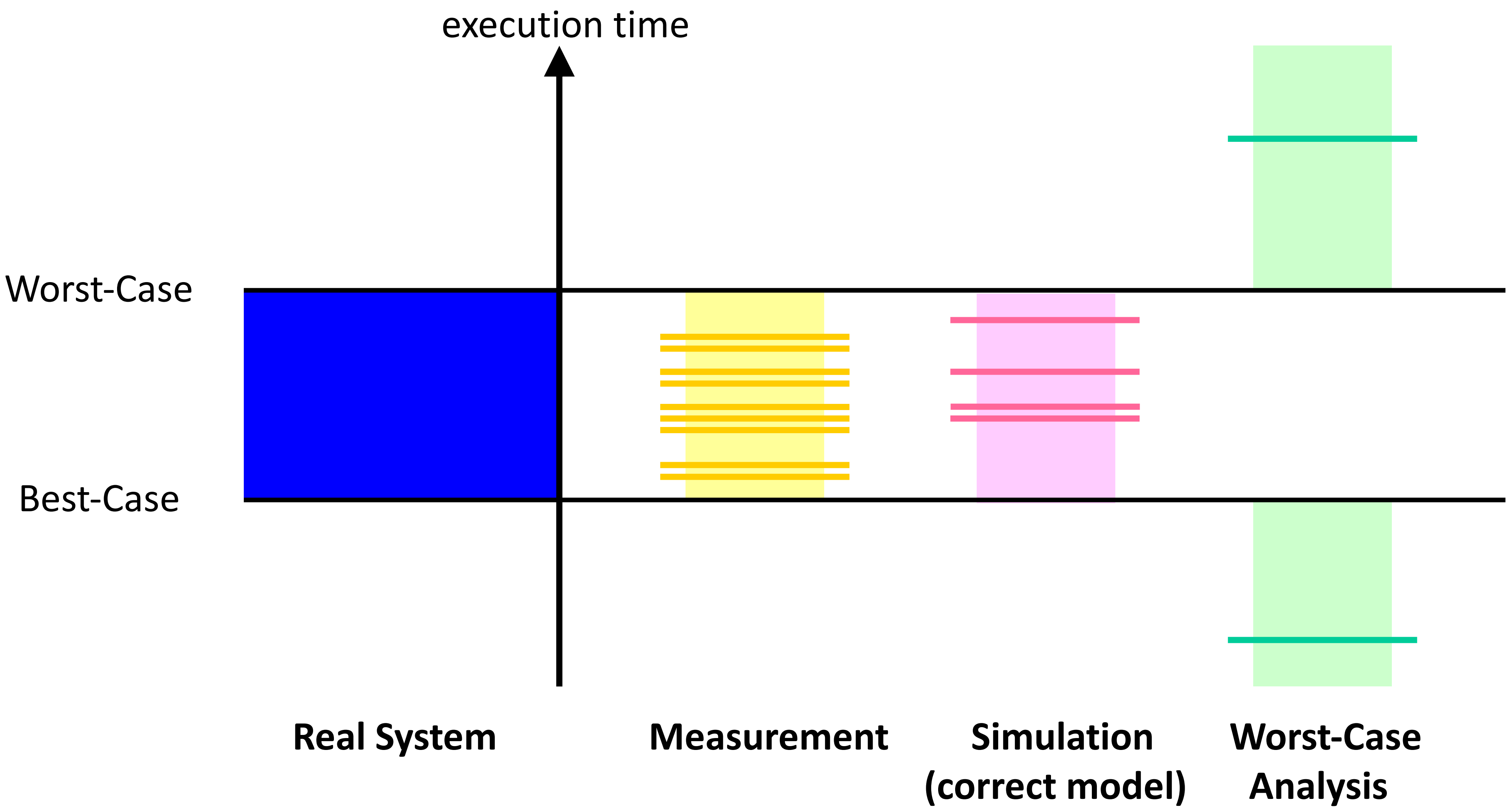
Reactive Systems and Timing

Real-Time Systems

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.*



Methods to Determine the Execution Time of a Task



Different Programming Paradigms

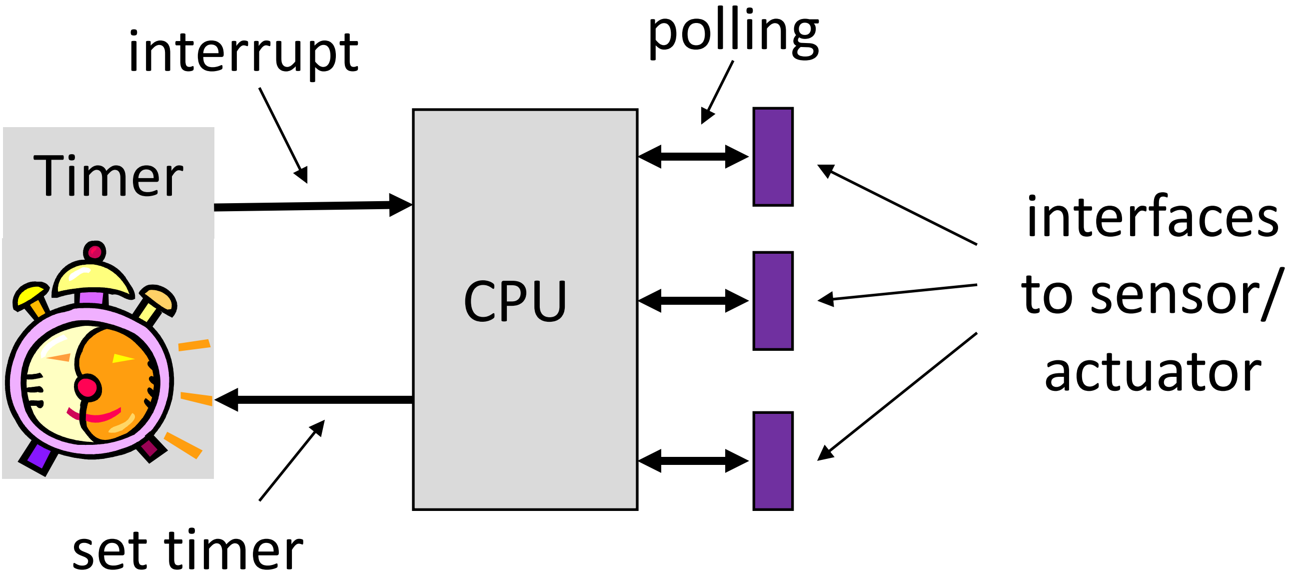
Overview

- There are many *structured ways of programming an embedded system*.
- In this lecture, only the main principles will be covered:
  - *time triggered approaches*
    - periodic
    - cyclic executive
    - generic time-triggered scheduler
  - *event triggered approaches*
    - non-preemptive
    - preemptive – stack policy
    - preemptive – cooperative scheduling
    - preemptive – multitasking

Time-Triggered Systems

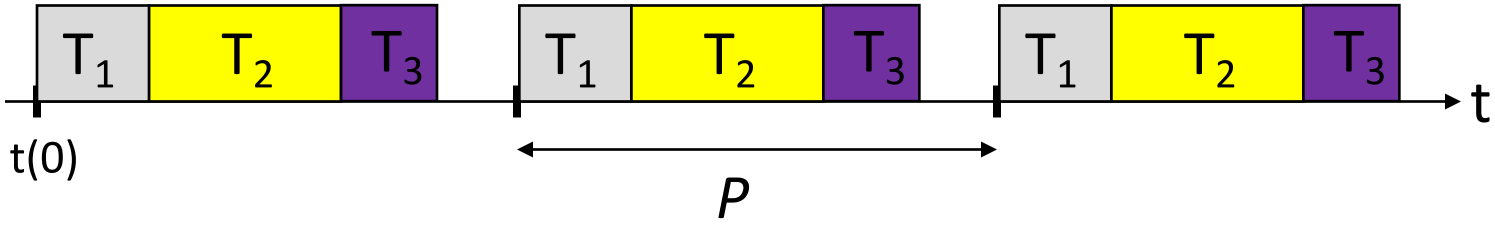
Pure time-triggered model:

- *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*
- the interaction with environment happens through *polling*



Simple Periodic TT Scheduler

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

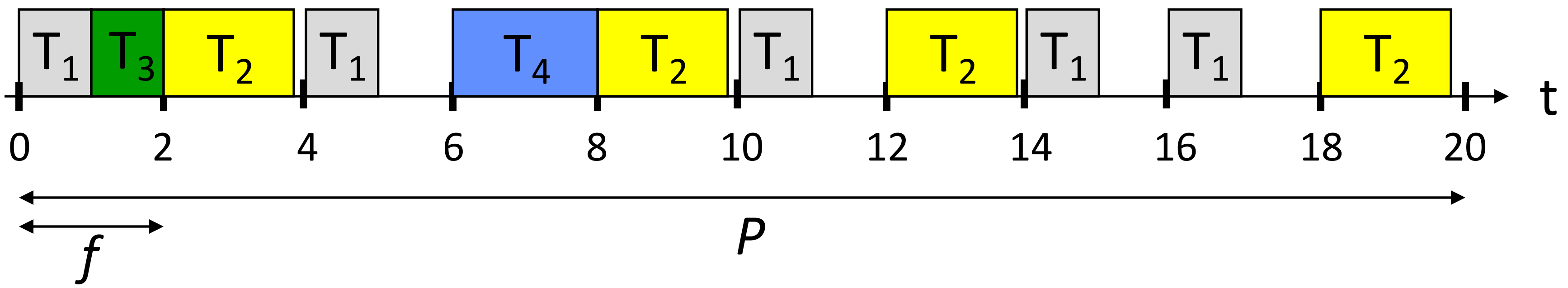


- *Properties:*
  - later tasks, for example T_2 and T_3, have unpredictable starting times
  - the communication between tasks or the use of common resources is safe, as there is a static ordering of tasks, for example T_2 starts after finishing T_1
  - as a necessary precondition, the sum of WCETs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

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

Time-Triggered Cyclic Executive Scheduler

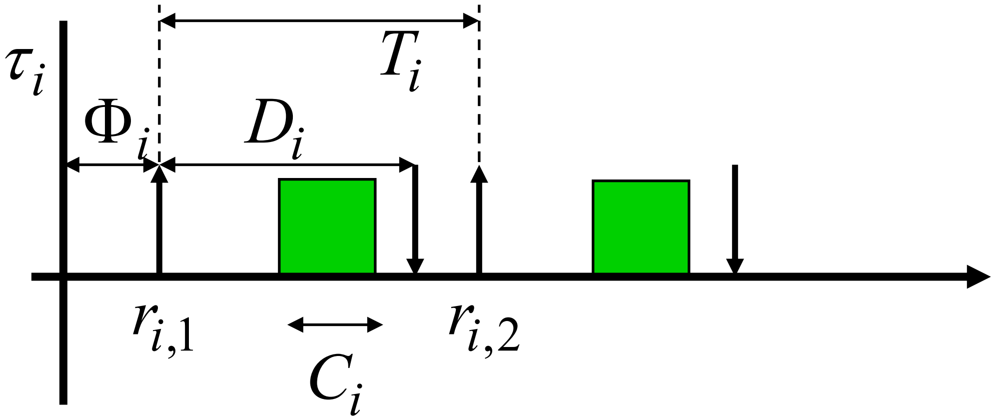
- Suppose now that *tasks may have different periods*.
- To accommodate this situation, the *period P is partitioned into frames of length f*.



- 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 a tedious and error-prone process, as the local state of the task must be extracted and stored globally

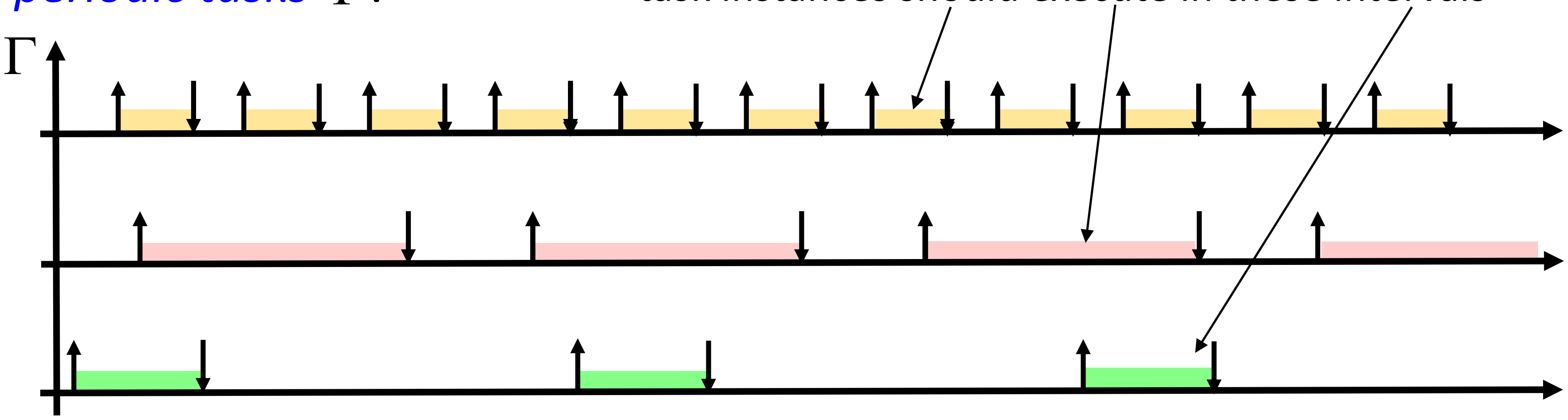
Time-Triggered Cyclic Executive Scheduling

- *Example* of a single periodic task τ_i:



- *A set of periodic tasks* Γ:

task instances should execute in these intervals



Time-Triggered Cyclic Executive Scheduling

Some conditions for period P and frame length f:

- A task executes at most once within a frame:

$$f \leq T_i \ \forall \text{ tasks } \tau_i$$

period of task

- *P* is a multiple of *f*.
- Period *P* is least common multiple of all periods T_k .
- Tasks start and complete within a single frame:

$$f \geq C_i \ \forall \text{ tasks } \tau_i$$

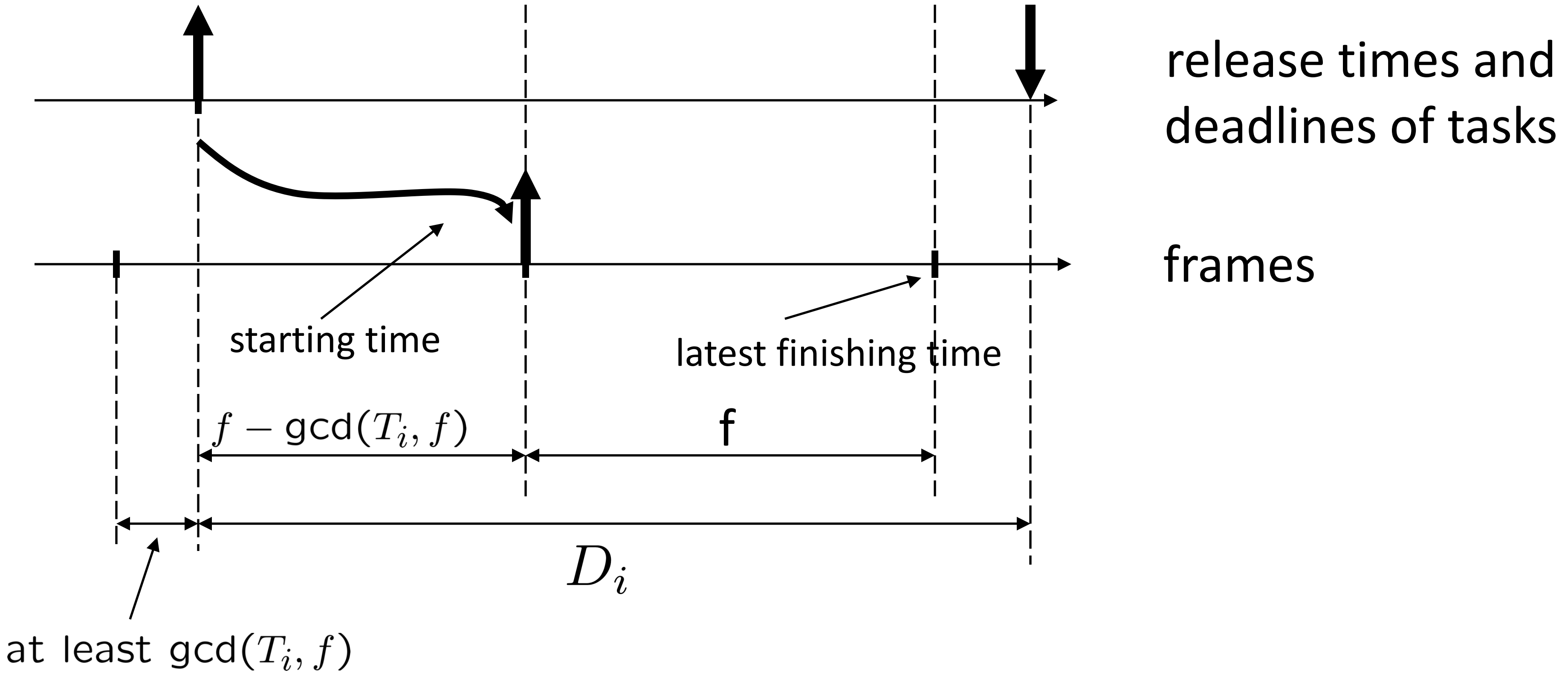
worst case execution time of task

- Between release time and deadline of every task there is at least one full frame:

$$2f - \text{gcd}(T_i, f) \leq D_i \ \ \forall \text{ tasks } \tau_i$$

relative deadline of task

Sketch of Proof for Last Condition



Example: Cyclic Executive Scheduling

Conditions:

$$f \leq \min\{4, 5, 20\} = 4$$

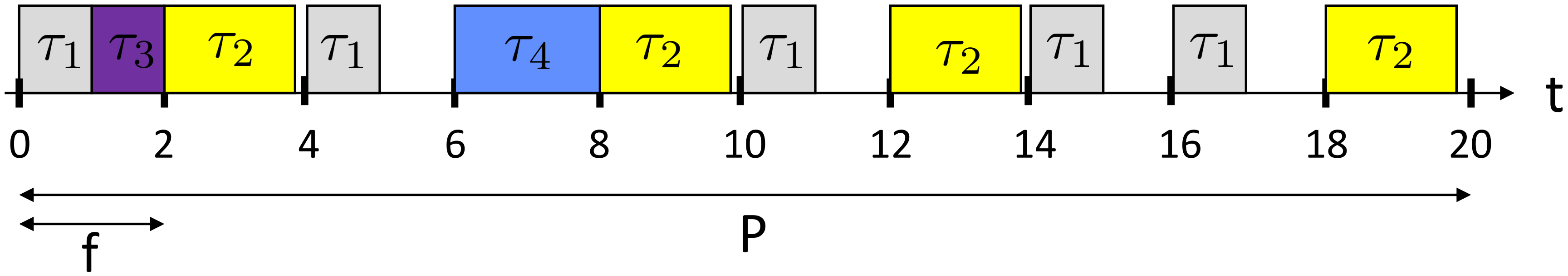
$$f \geq \max\{1.0, 1.0, 1.8, 2.0\} = 2.0$$

$$2f - \gcd(T_i, f) \leq D_i \ \ \forall \text{ tasks } \tau_i$$

possible solution: f = 2

Γ	T_i	D_i	C_i
τ_1	4	4	1.0
τ_2	5	5	1.8
τ_3	20	20	1.0
τ_4	20	20	2.0

Feasible solution (f=2):



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 initial phases such that instances of tasks start after their release time:

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

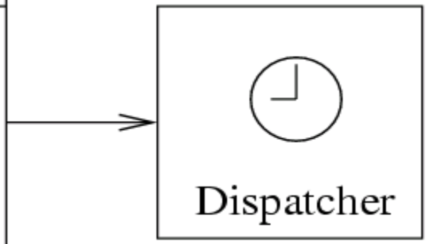
- Are deadlines respected?

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

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	



start times can be arbitrarily chosen within the period

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

Advantages:

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

Disadvantages:

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

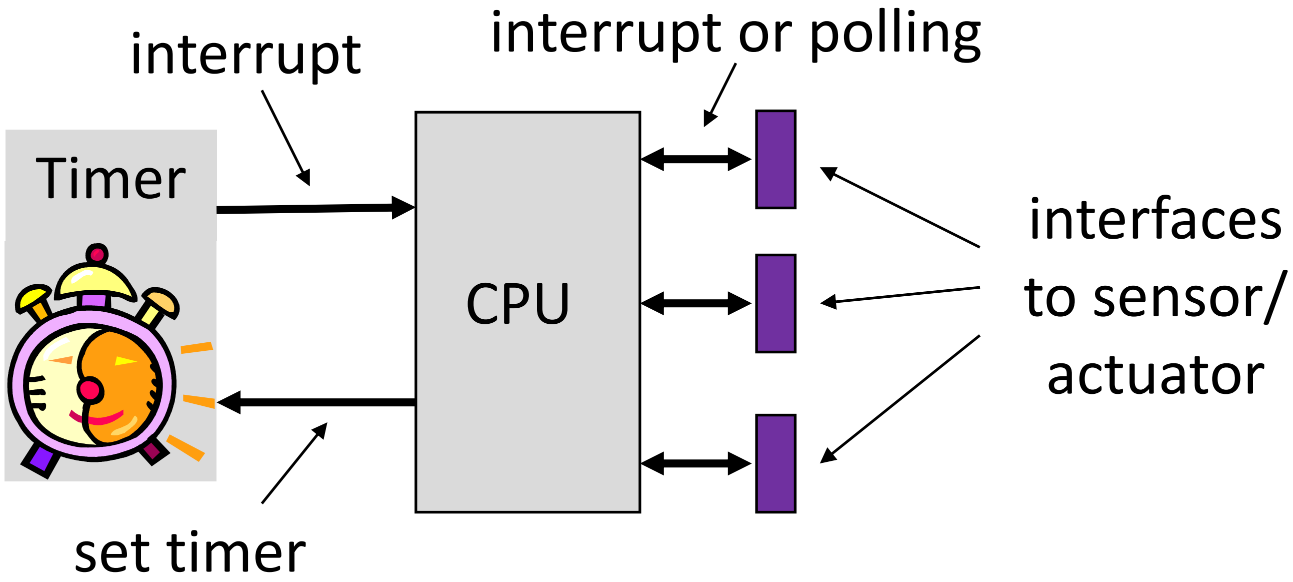
Extensions:

- *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) using a watchdog timer

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



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 (e.g., first come first serve), an event is chosen for execution, that is, the corresponding task is executed.
- A running task *cannot be preempted* by another task. It can only be preempted by an interrupt that registers an event and puts it into the queue.

Extensions:

- A *background task*, which has the lowest priority, 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 certain time interval has elapsed. This enables, for example, periodic task instantiations.

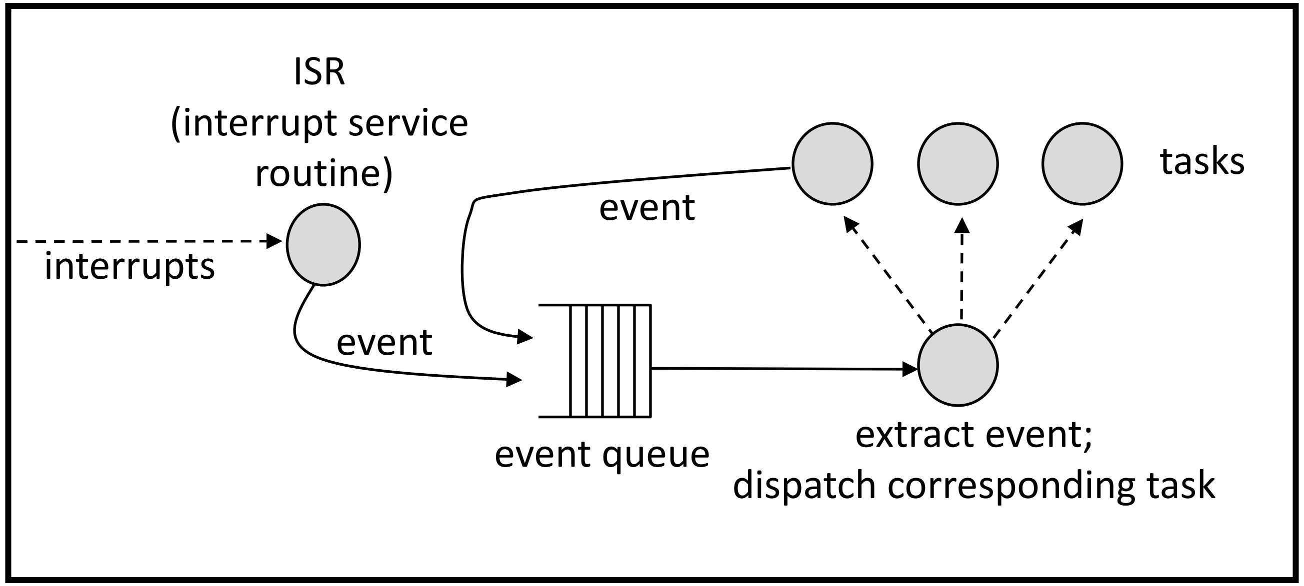
Non-Preemptive Event-Triggered Scheduling

```
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

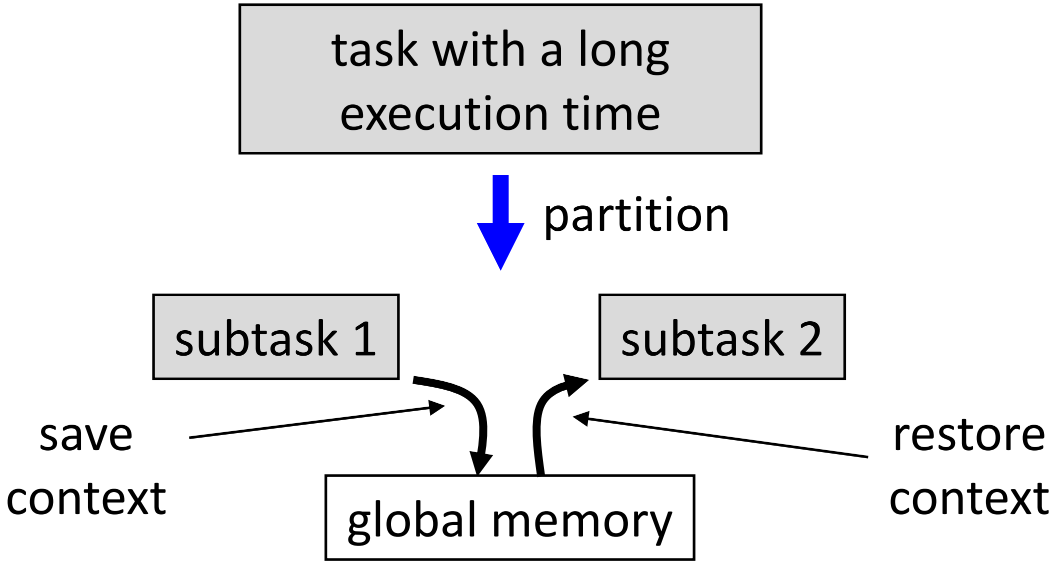
for example using a function pointer in C;
task returns after finishing.



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* of the event queue may happen if too many events are generated by the environment or by tasks (guarantee requires bounded behavior of environment)
- *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

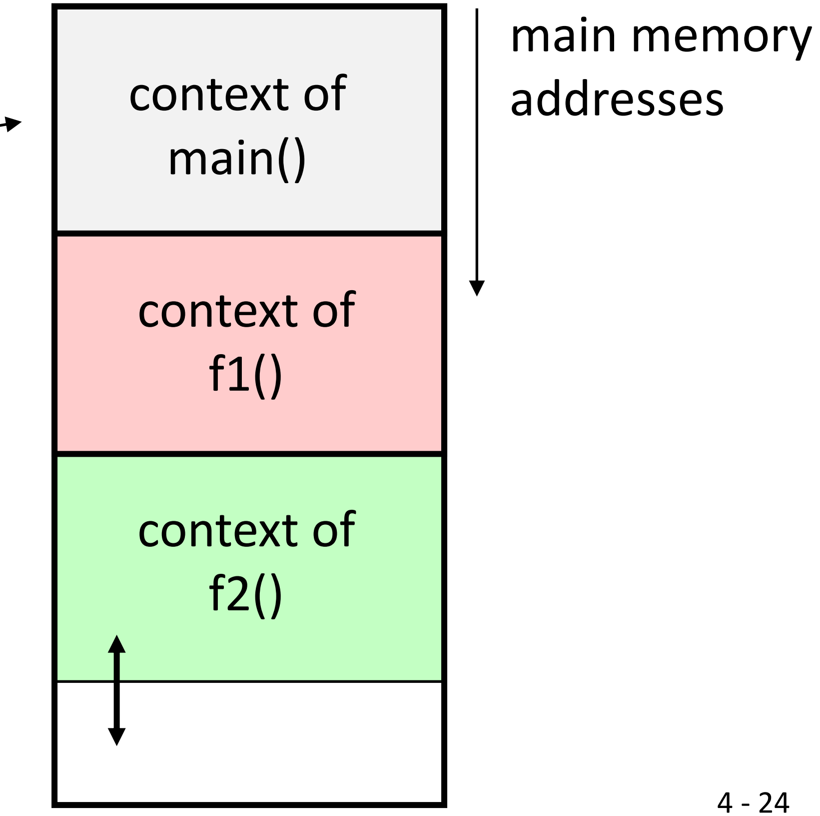


Preemptive Event-Triggered Scheduling – Stack Policy

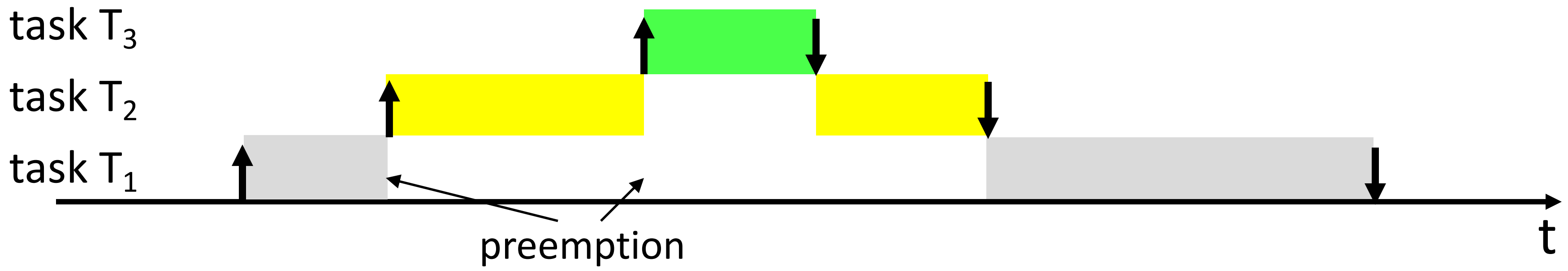
- Each event/task has a *fixed priority*. Tasks with *higher priority can preempt* tasks with lower priority. This partly solves the problem of long-running tasks.
- If *the order of preemption is restricted*, we can use the usual stack-based context mechanism of function calls. The context of a function contains the necessary state such as local variables and saved registers.

```
main(){
        …
        f1();
        …

f1(){
        …
        f2();
        …
```



Preemptive Event-Triggered Scheduling – Stack Policy



- *Tasks must finish in LIFO (last in first out) order* of their instantiation, that is, the preempting task must finish before the preempted task can continue.
  - this restricts flexibility of the approach
  - 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;
        }
   }
```

set CPU to low power mode;
processing continues after interrupt

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

```
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;
```

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

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

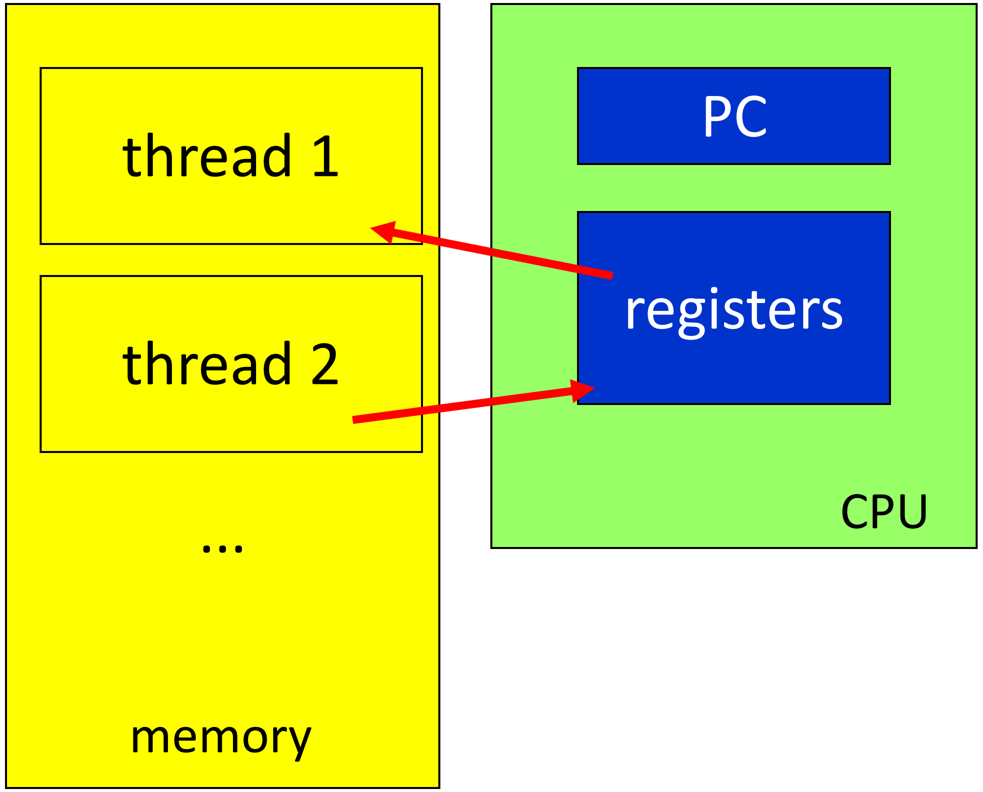
Thread

- *A thread is a unique execution of a piece of code.*
  - Several copies of such a “program” may run simultaneously or at different times.
  - Threads share the same processor and its peripherals.
- *A thread has its own local state.* This state consists mainly of:
  - register values;
  - memory stack (local variables);
  - program counter;
- *Several threads may have a shared state* consisting of global variables.

Threads and Memory Organization

- *Activation record*, or *thread context*, contains the thread-local state, including registers and local data structures.
- Each thread has a *fixed memory region* for storing its context.

- *Context switch:*
  - current CPU context (program counter, registers) goes out
  - new CPU context goes in



Co-operative Multitasking

- *Each thread allows a context switch to another thread* at a call to the `cswitch()` function.
  - This function is part of the underlying runtime system (operating system).
  - A *scheduler* within this runtime system chooses which thread will run next. This could be a different thread, or the same one in case no other thread is ready to run.

- ***Advantages***:
  - predictable, where context switches can occur (programmer has full control)
  - less errors with use of shared resources if the switch locations are chosen carefully

- ***Disadvantages***:
  - programming errors (e.g., if the `cswitch()` function is never called) can keep other threads out as the running thread may never give up the CPU
  - real-time behavior may be at risk if a thread runs for too long before the next context switch is allowed

Example: Co-operative Multitasking

Thread 1

```
if (x > 2)
    sub1(y);
else
    sub2(y);
cswitch();
proca(a,b,c);
```

Thread 2

```
procdata(r,s,t);
cswitch();
if (val1 == 3)
    abc(val2);
rst(val3);
```

Scheduler

```
save_state(current);
p = choose_process();
load_and_go(p);
```

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. Only one thread can be in this state.
  - *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.