Lecture 6 – Programming Paradigms

CSE 456: Embedded Systems

Timing Guarantees

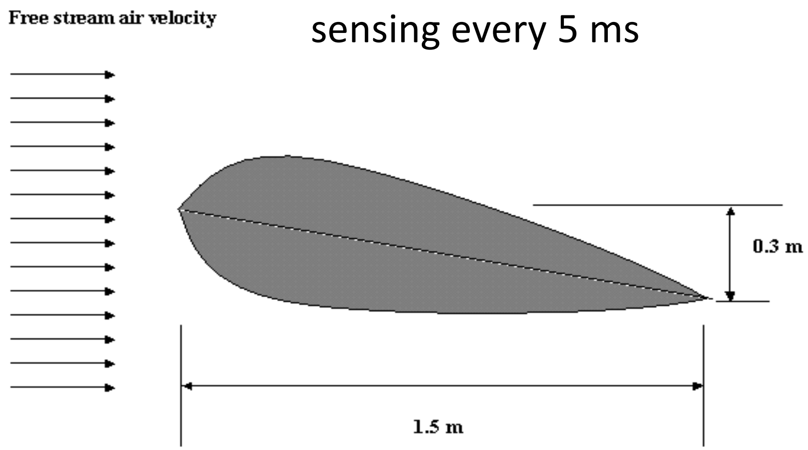
- *Hard real-time systems* can be often found in *safety-critical applications*. They need to provide the result of a computation within a fixed time bound.
- *Typical application domains:*
  - avionics, automotive, train systems, automatic control including robotics, manufacturing, media content production

sideairbag in car,

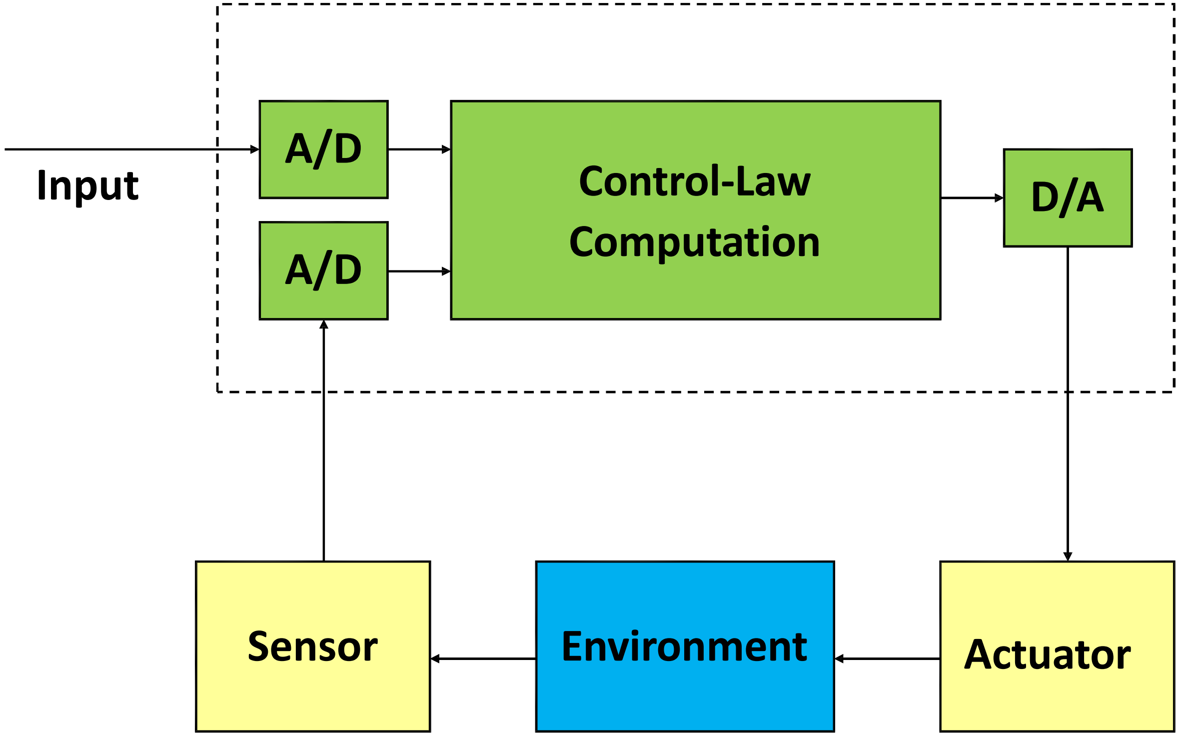
reaction after event in <10 mSec

wing vibration of airplane,

sensing every 5 ms

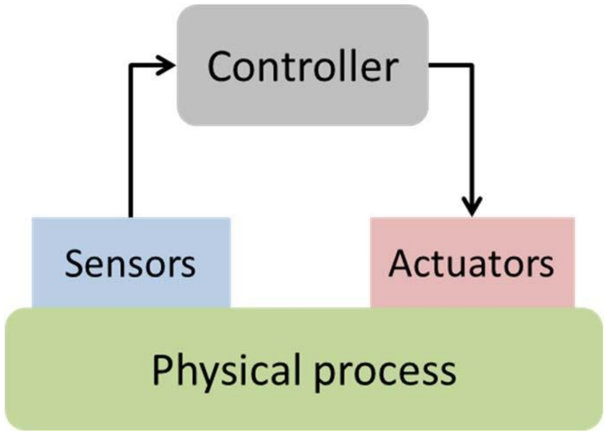


Simple Real-Time Control System

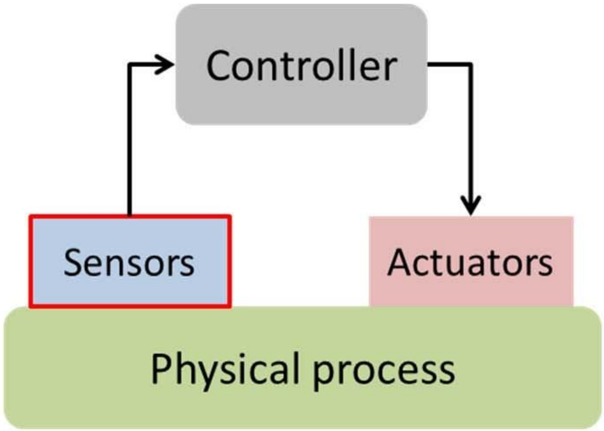


Real-Time Systems

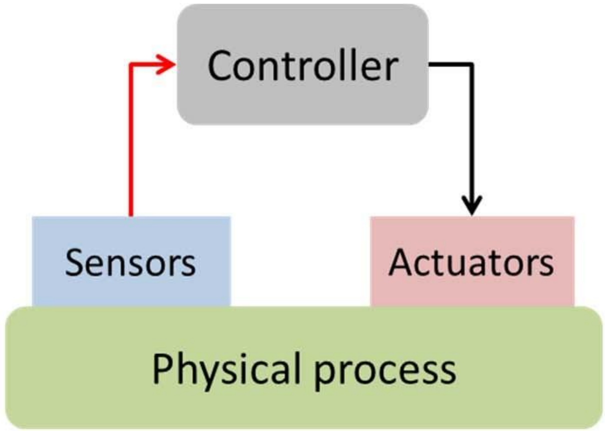
In many *cyber-physical systems (CPSs),* correct timing is a matter of *correctness*, not performance: *an answer arriving too late is consider to be an error.*



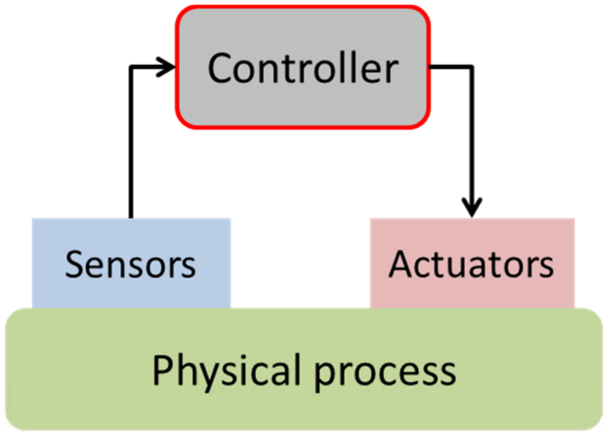
Real-Time Systems

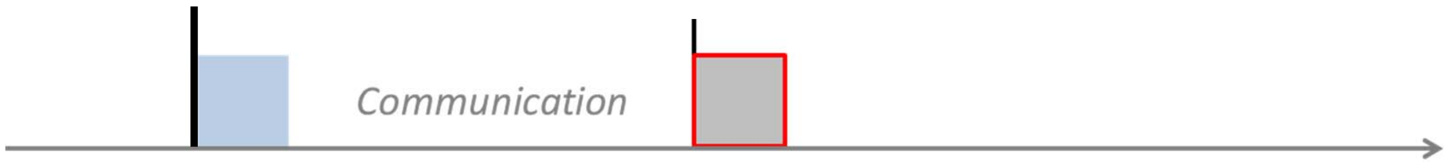


Real-Time Systems

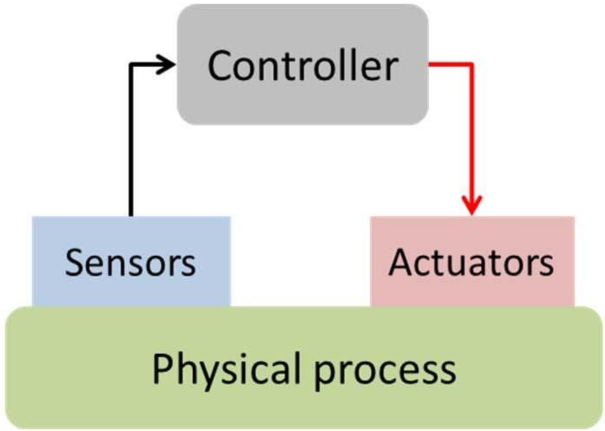


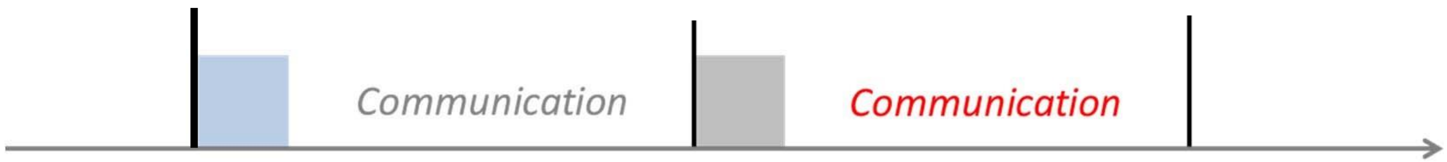
Real-Time Systems



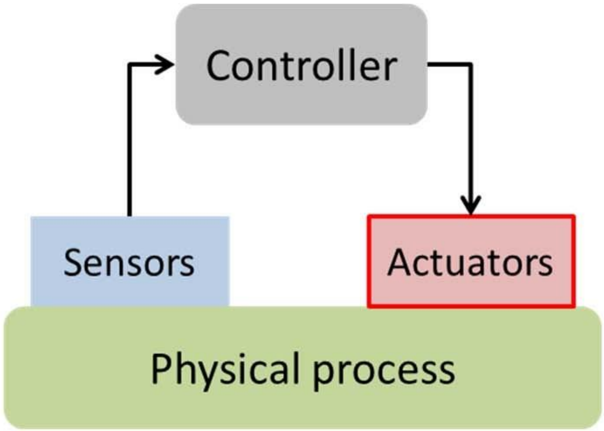


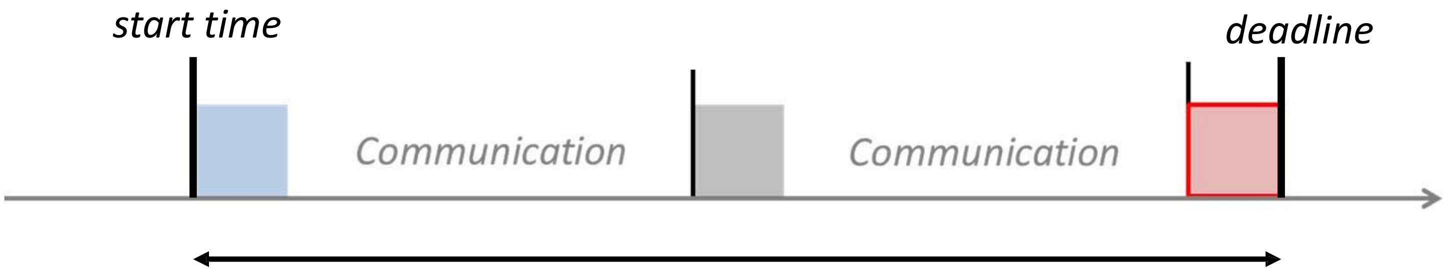
Real-Time Systems





Real-Time Systems





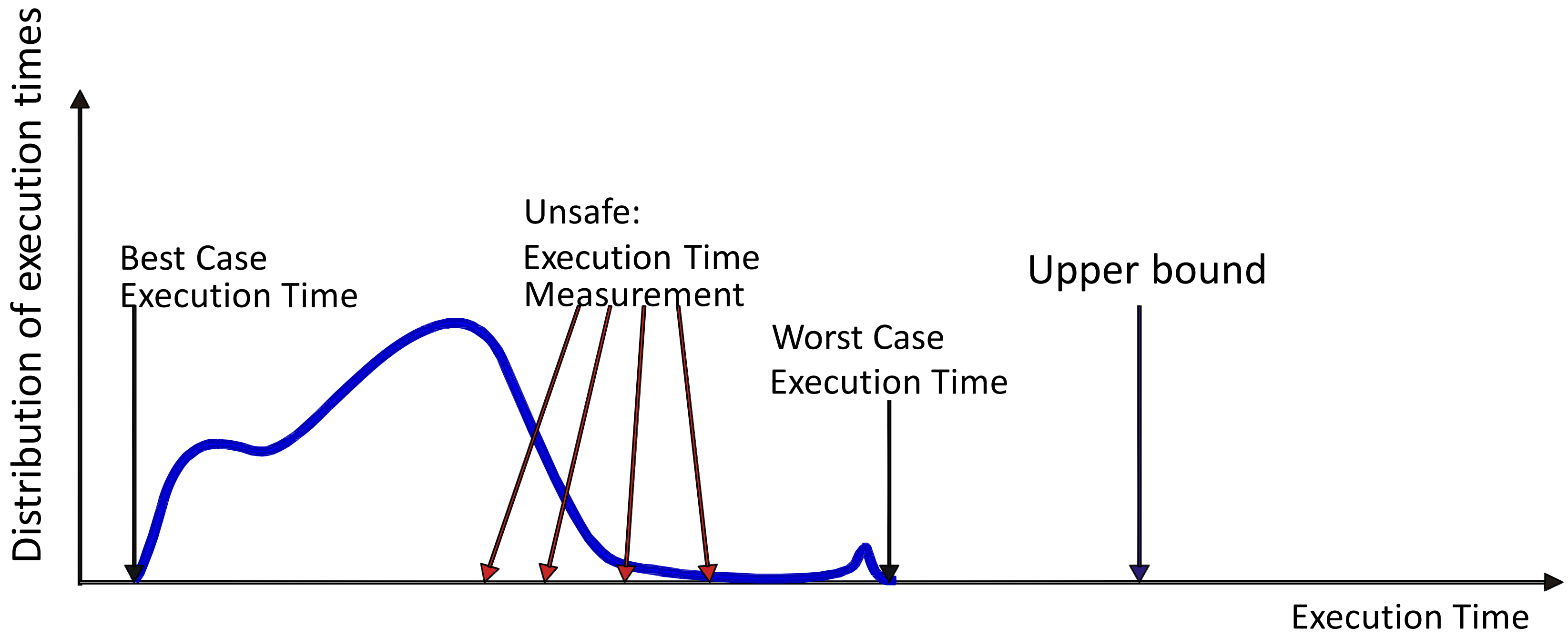
Real-Time Systems

- *Embedded controllers* 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.

- Essential for the analysis and design of a real-time system: *Upper bounds on the execution times* of all tasks are statically known. This also includes the communication of information via a wired or wireless connection.

  - 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).

Distribution of Execution Times



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.*

(Most of) Industry's Best Practice

- ***Measurements:*** determine execution times directly by observing the execution or a simulation on a set of inputs.
  - *Does not guarantee an upper bound* to all executions unless the reaction to all initial system states and all possible inputs is measured.
  - *Exhaustive execution* in general not possible: Too large space of (input domain) x (set of initial execution states).
- ***Simulation*** suffers from the same restrictions.

- ***Compute upper bounds*** along the structure of the program:
  - Programs are *hierarchically* structured: Instructions are "nested" inside statements.
  - Therefore, one may compute the upper execution time bound for a statement from the upper bounds of its constituents, for example of single instructions.
  - *But:* The execution times of individual instructions varies largely!

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.

- *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.

Different Programming Paradigms

Why Multiple Tasks on one Embedded Device?

- The concept of *concurrent tasks* reflects our intuition about the *functionality of embedded systems*.

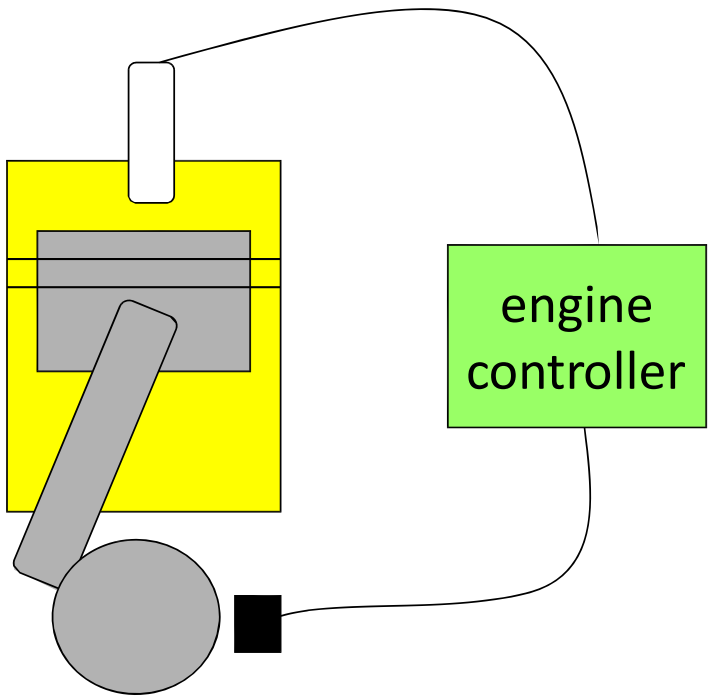
- Tasks help us *manage the complexity of concurrent activities* as happening in the system environment:
  - *Input data* arrive from various *sensors* and input devices.
    - These input streams may have different data rates like in multimedia processing, systems with multiple sensors, automatic control of robots

  - The system may also receive *asynchronous (sporadic) input events*.
    - These input event may arrive from user interfaces, from sensors, or from communication interfaces, for example.

Example: Engine Control

Typical Tasks:

- spark control
- crankshaft sensing
- fuel/air mixture
- oxygen sensor
- Kalman filter – control algorithm



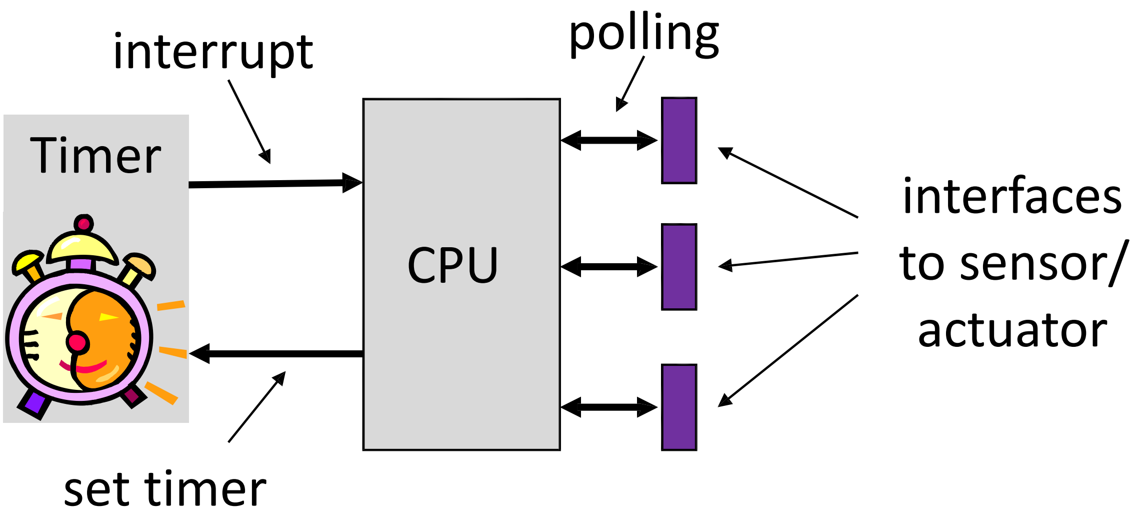
Overview

- 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

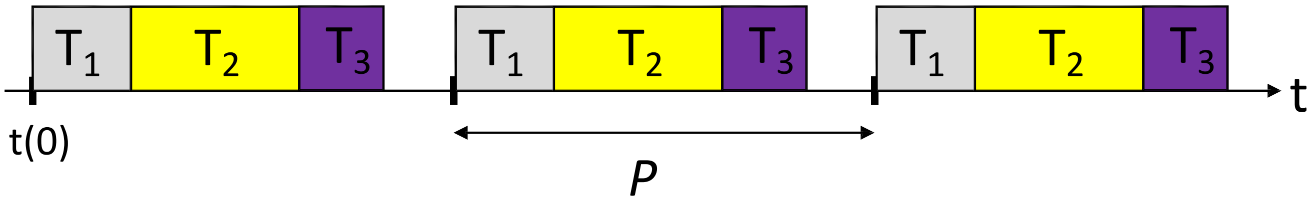
Pure time-triggered model:

- *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*:

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

Simple Periodic Time-Triggered Scheduler

usually done offline

```
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;
```

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

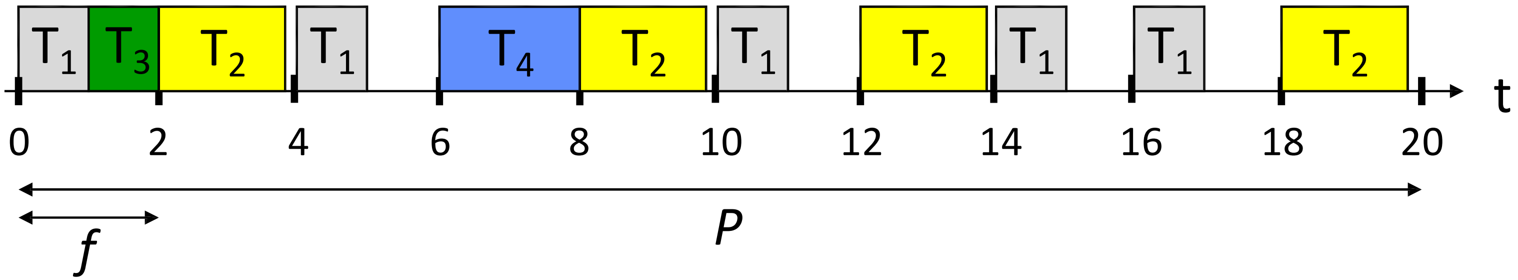
for example using a function pointer in C;
task(= function) returns after finishing.

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*.
- *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*

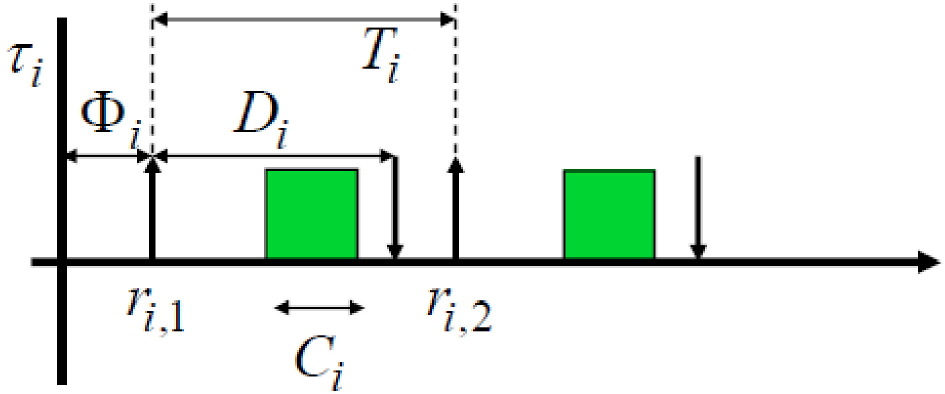
$r_{i,j}$, $d_{i,j}$: denote the release time and absolute deadline of the *j*th instance of task *i*

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

D_i : relative deadline of task *i*

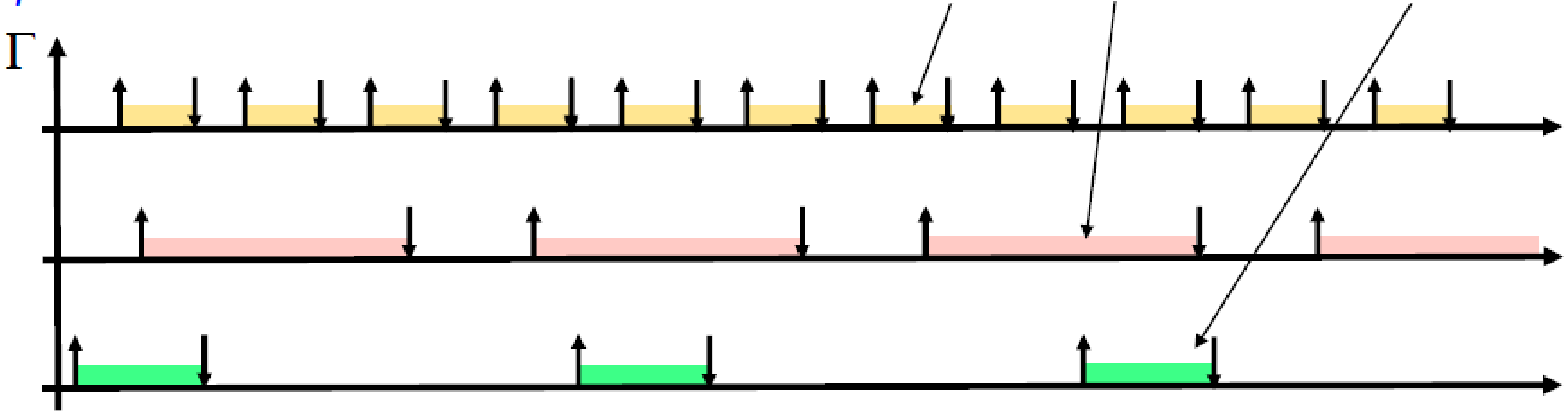
Time-Triggered Cyclic Executive Scheduling

- *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

- 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

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

Time-Triggered Cyclic Executive Scheduling

Example

Γ	T_i	Φ_i	D_i	C_i	frame
τ_1	12	2	8	2.8	2
τ_2	12	3	9	3	3
τ_3	4	0	4	1	1, 2, 3

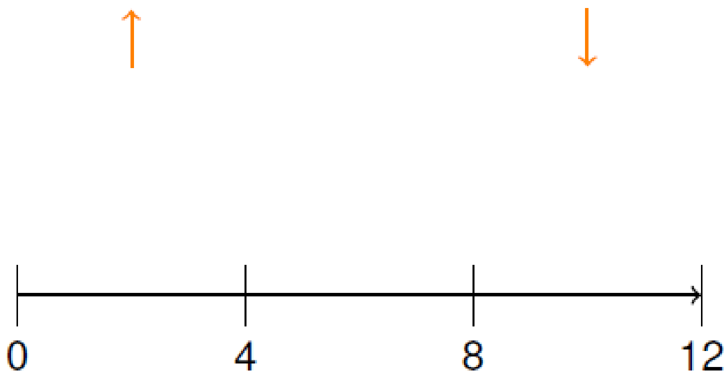
$P = 12, f = 4$

Time-Triggered Cyclic Executive Scheduling

Example

Γ	T_i	Φ_i	D_i	C_i	frame
τ_1	12	2	8	2.8	2
τ_2	12	3	9	3	3
τ_3	4	0	4	1	1, 2, 3

$P = 12, f = 4$

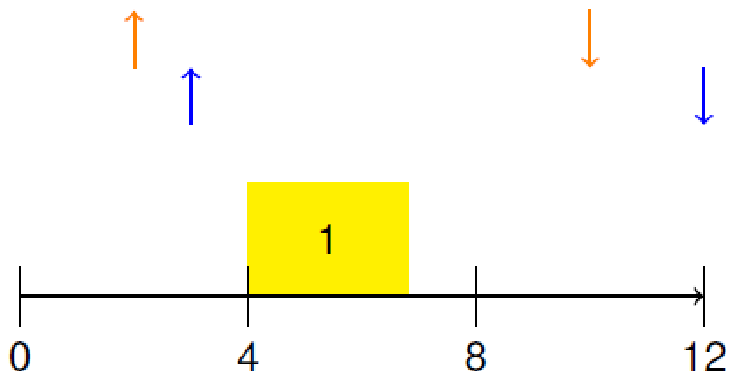


Time-Triggered Cyclic Executive Scheduling

Example

Γ	T_i	Φ_i	D_i	C_i	frame
τ_1	12	2	8	2.8	2
τ_2	12	3	9	3	3
τ_3	4	0	4	1	1, 2, 3

$P = 12, f = 4$

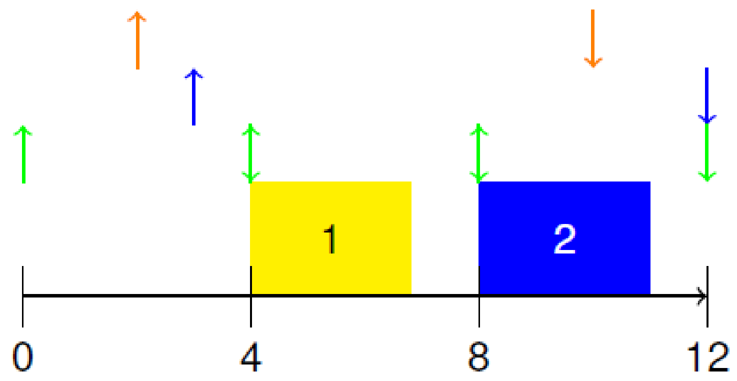


Time-Triggered Cyclic Executive Scheduling

Example

Γ	T_i	Φ_i	D_i	C_i	frame
τ_1	12	2	8	2.8	2
τ_2	12	3	9	3	3
τ_3	4	0	4	1	1, 2, 3

$P = 12, f = 4$

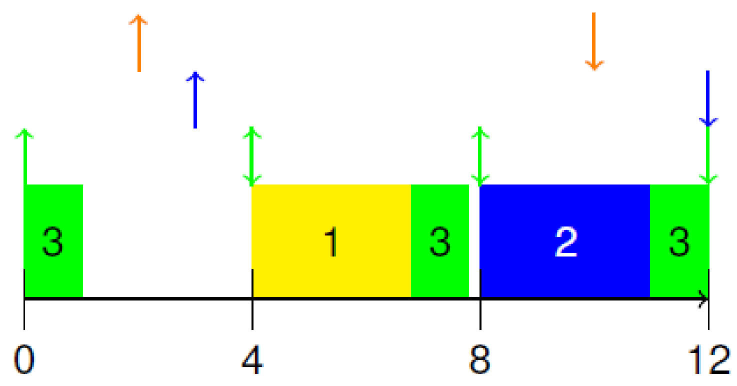


Time-Triggered Cyclic Executive Scheduling

Example

Γ	T_i	Φ_i	D_i	C_i	frame
τ_1	12	2	8	2.8	2
τ_2	12	3	9	3	3
τ_3	4	0	4	1	1, 2, 3

$P = 12, f = 4$



Time-Triggered Cyclic Executive Scheduling

Example with 4 tasks:

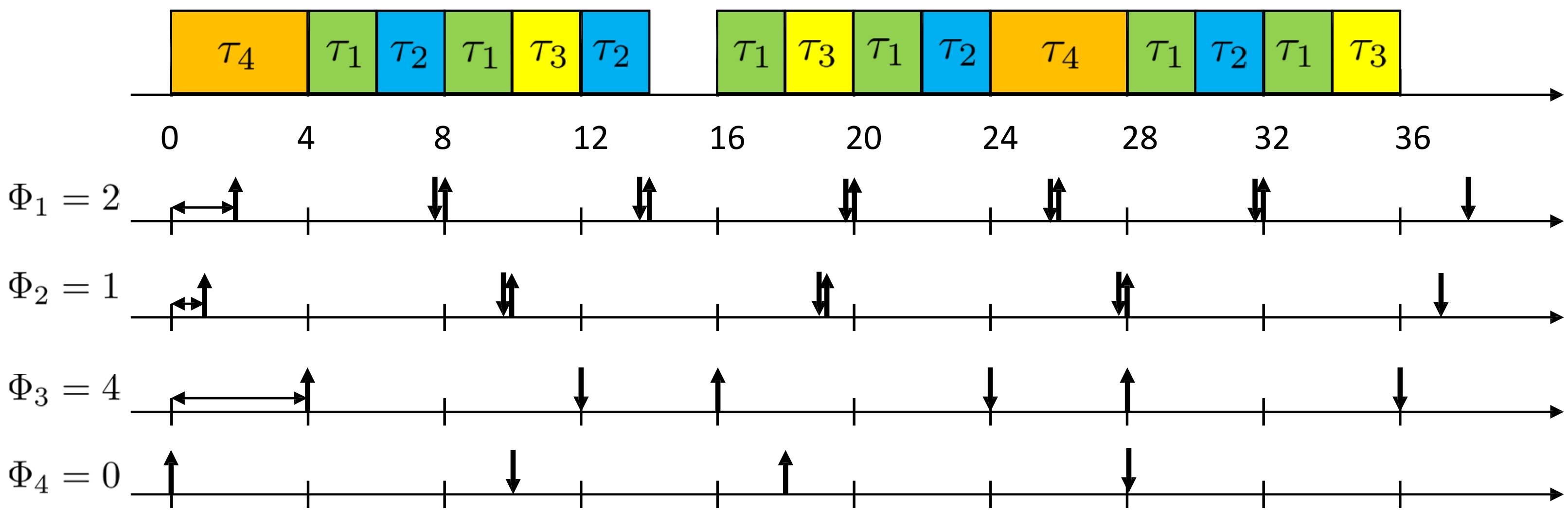
- $\tau_1 : T_1 = 6, D_1 = 6, C_1 = 2$ $\quad$ $\tau_2 : T_2 = 9, D_2 = 9, C_2 = 2$
  $\tau_3 : T_3 = 12, D_3 = 8, C_3 = 2$ $\quad$ $\tau_4 : T_4 = 18, D_4 = 10, C_1 = 4$
- $P = 36, f = 4$

Time-Triggered Cyclic Executive Scheduling

Example with 4 tasks:

- $\tau_1 : T_1 = 6, D_1 = 6, C_1 = 2$ $\quad\tau_2 : T_2 = 9, D_2 = 9, C_2 = 2$

  $\tau_3 : T_3 = 12, D_3 = 8, C_3 = 2$ $\quad\tau_4 : T_4 = 18, D_4 = 10, C_1 = 4$
- $P = 36, f = 4$



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?

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

Example: Cyclic Executive Scheduling

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

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

Example: Cyclic Executive Scheduling

Conditions:

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

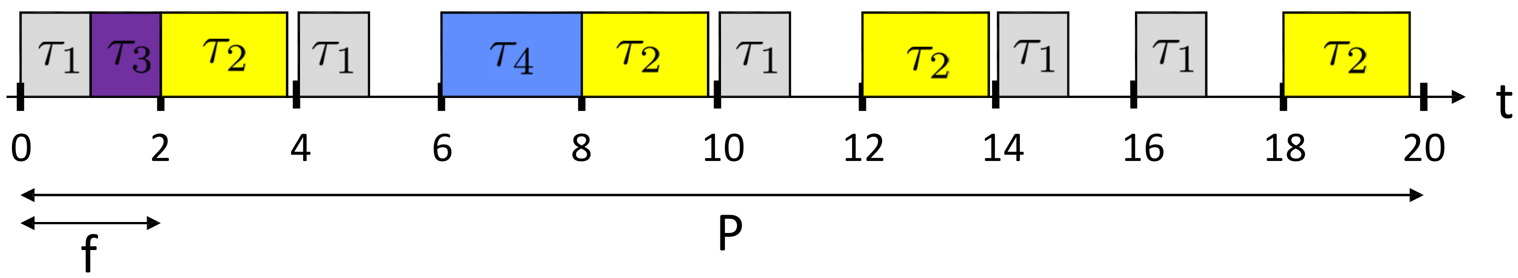
$$f \geq \max\{1.0, 1.0, 1.8, 2.0\} = 2.0$$

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

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

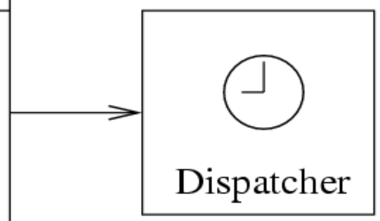
Feasible solution (f=2):



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*

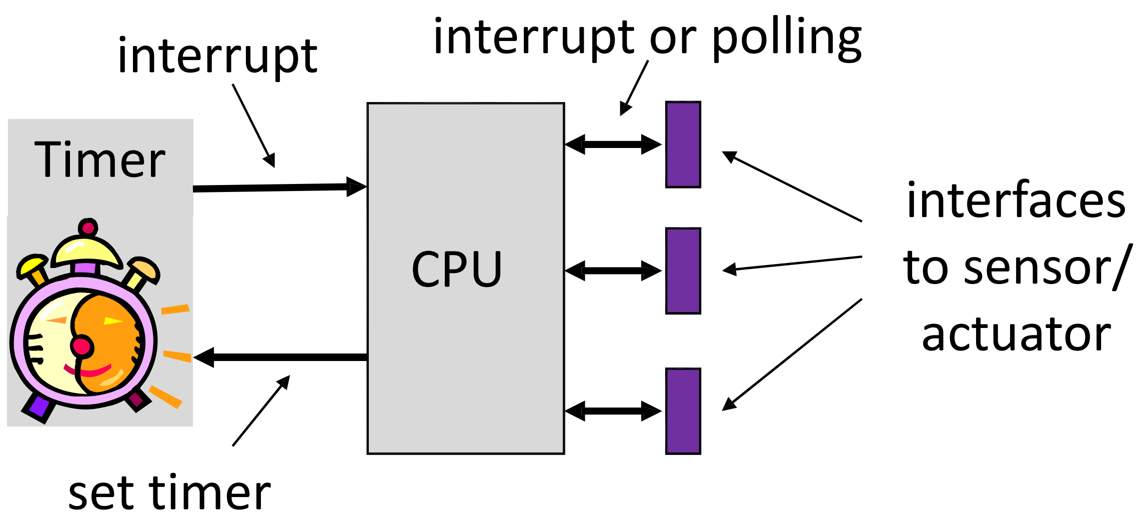
Extensions:

- *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) 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, 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.

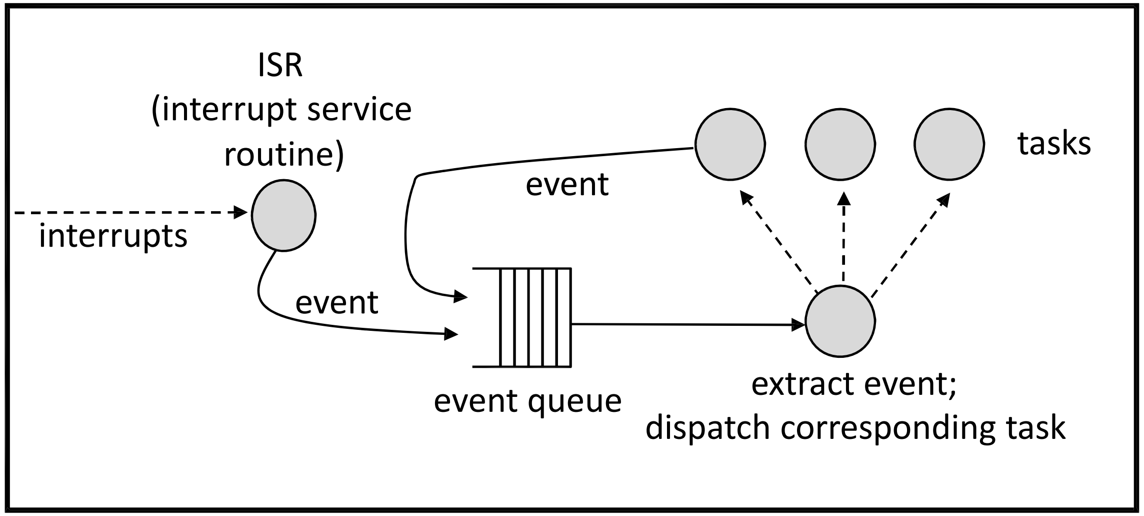
Non-Preemptive Event-Triggered Scheduling

```
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

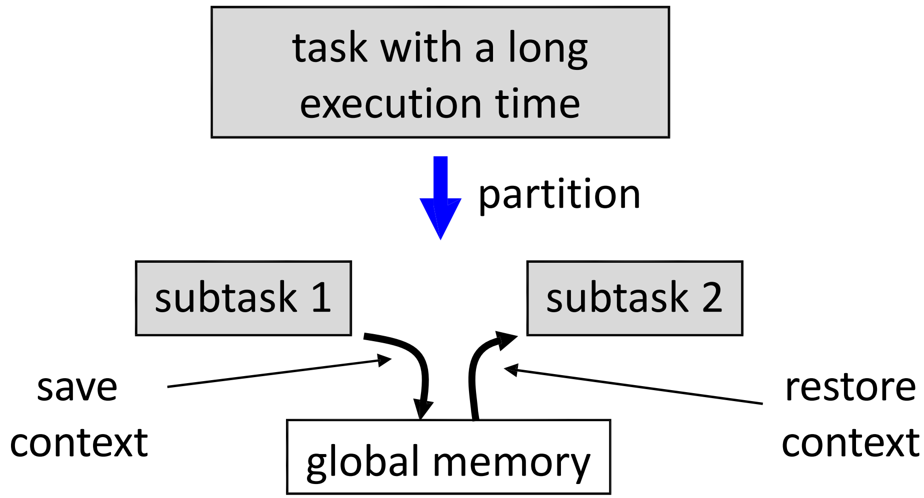
for example using a function pointer in C; task returns after finishing.



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

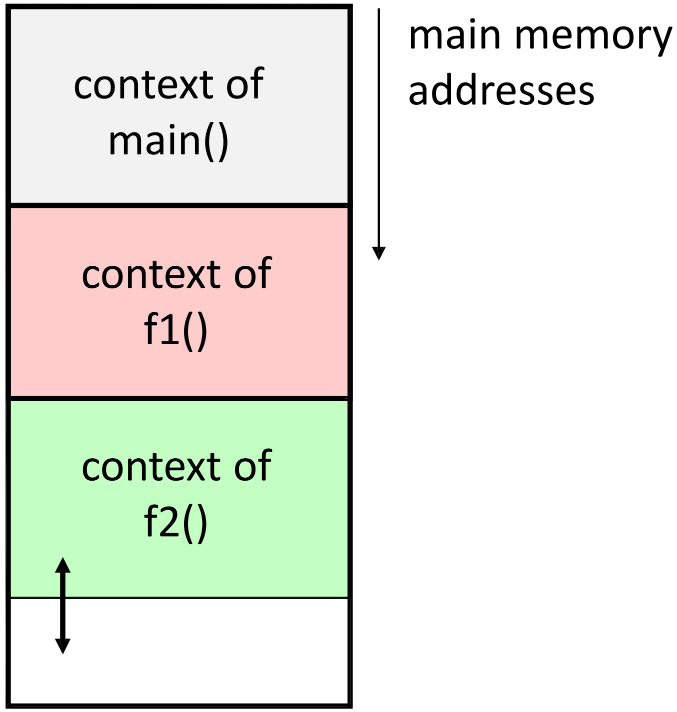


Preemptive Event-Triggered Scheduling – Stack Policy

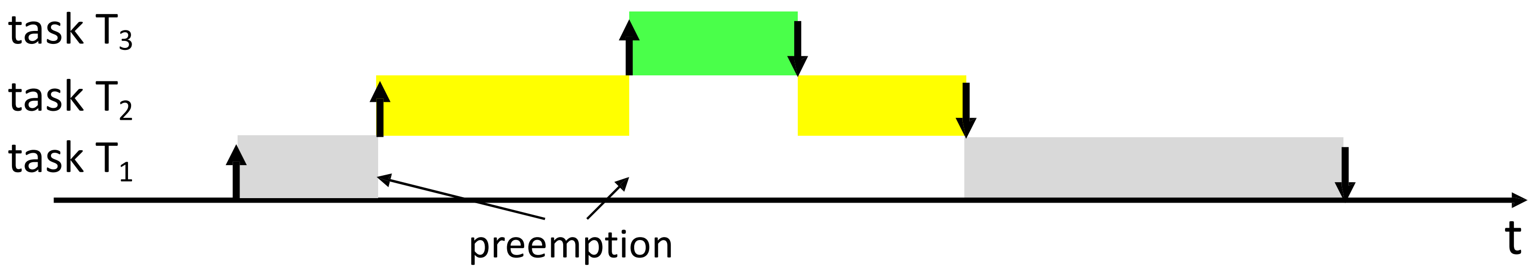
- This case is similar to non-preemptive case, but *tasks can be preempted by others*; this resolves partly the problem of tasks with a long execution time.
- If *the order of preemption is restricted*, we can use the usual stack-based context mechanism of function calls. The context of a function contains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

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

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



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;
        }
   }
```

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

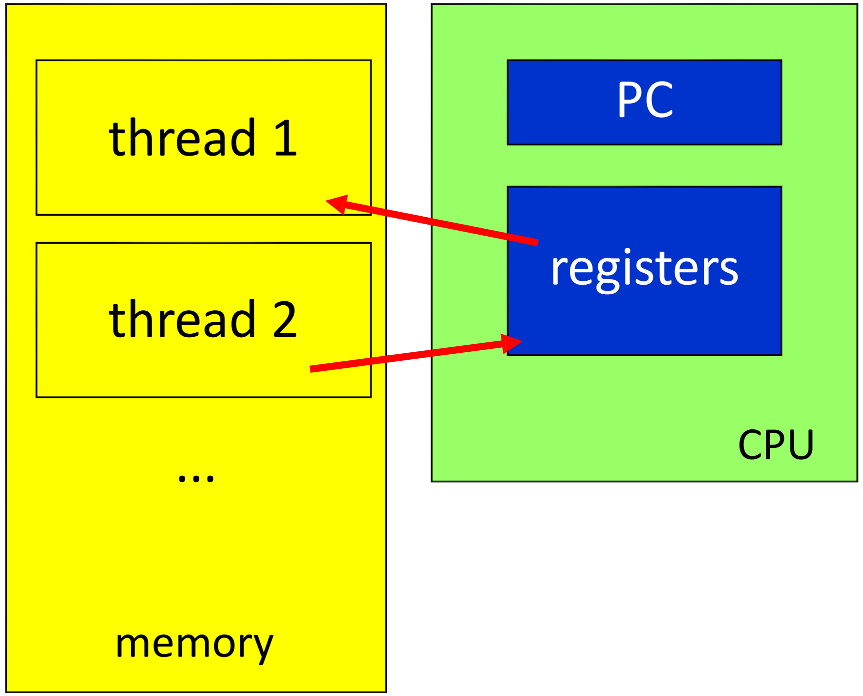
Thread

- *A thread is a unique execution of a program.*
  - 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* (also denoted as the thread context) contains the thread local state which includes registers and local data structures.

- *Context switch:*
  - current CPU context 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.
- ***Advantages***:
  - predictable, where context switches can occur
  - less errors with use of shared resources if the switch locations are chosen carefully
- ***Problems***:
  - programming errors can keep other threads out as a thread may never give up CPU
  - real-time behavior may be at risk if a thread runs 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
  - *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.