Embedded Systems

2. Software Introduction

Lothar Thiele

Contents of Course

1. Embedded Systems Introduction

2. Software Introduction
3. Real-Time Models
4. Periodic/Aperiodic Tasks
5. Resource Sharing
6. Real-Time OS

7. System Components
8. Communication
9. Low Power Design

10. Models
11. Architecture Synthesis

12. Model Based Design

Software and Programming

Processing and Communication

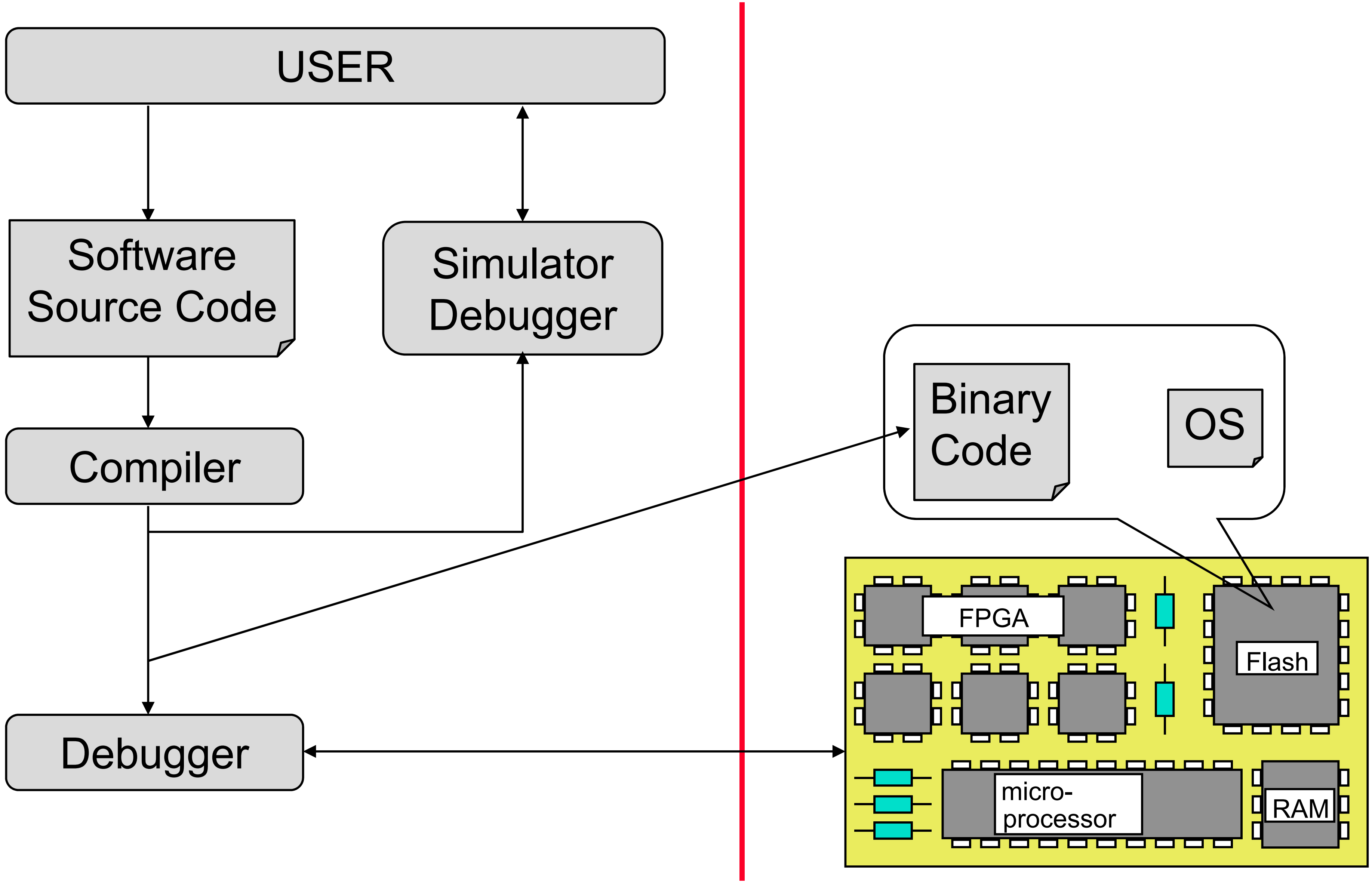
Hardware

Subtopics

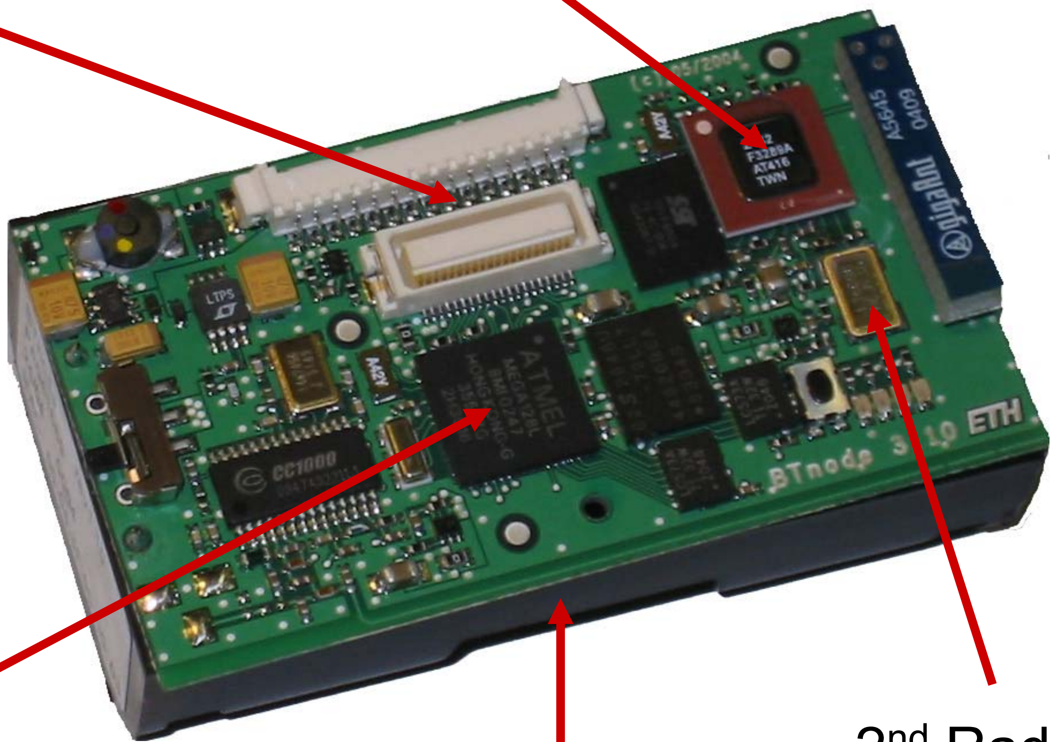
4

- A few introductory remarks.
- Different programming paradigms.

Embedded Software Development



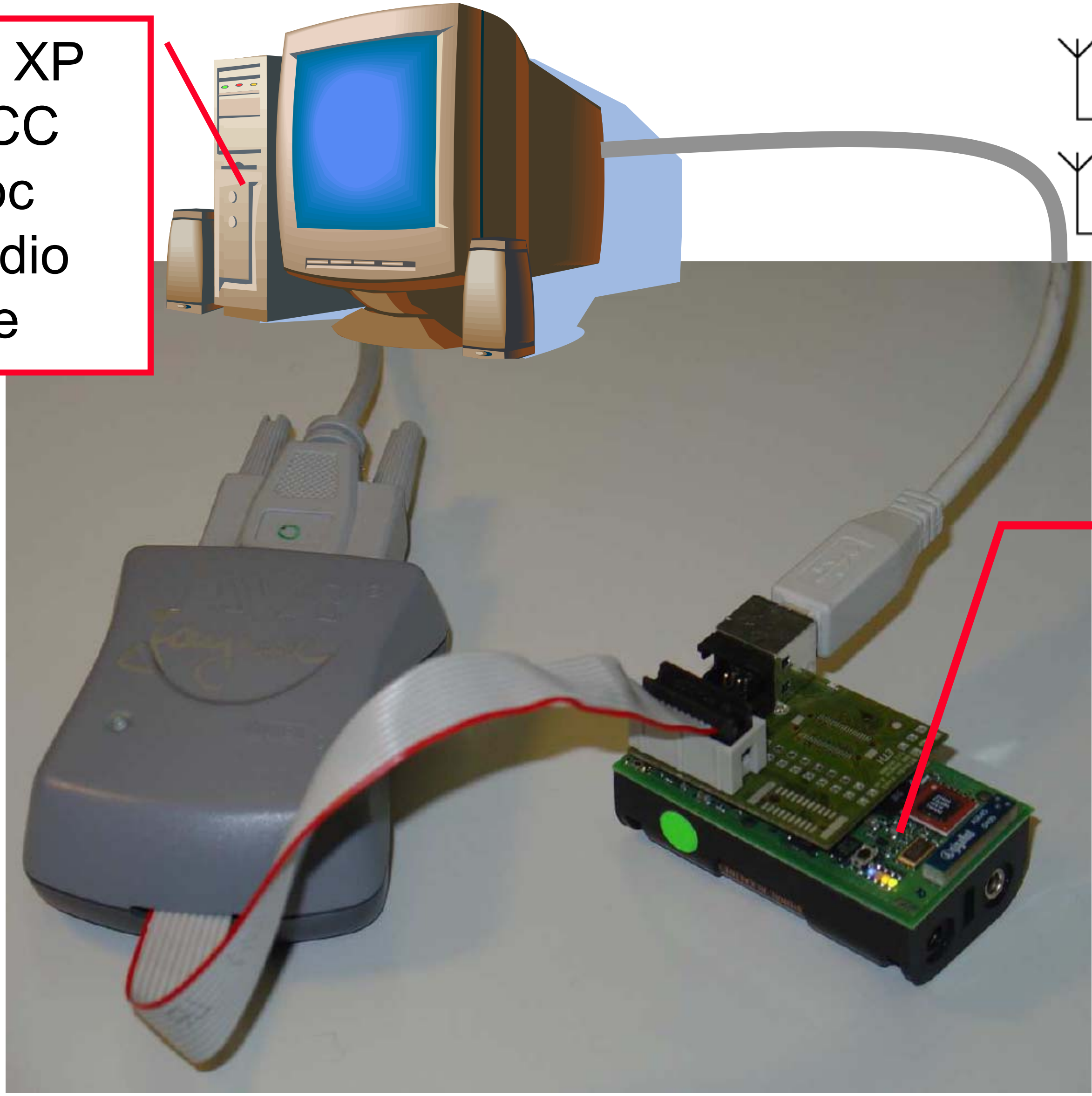
BTnode Platform

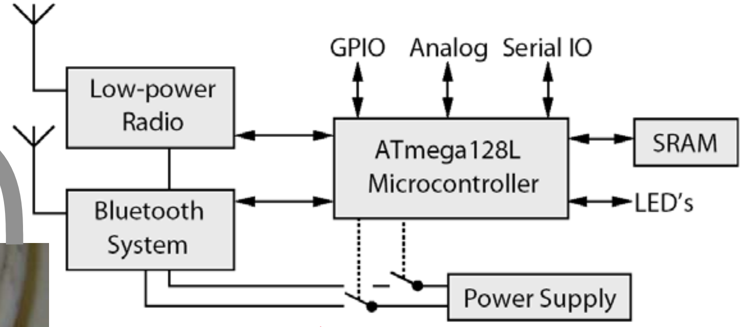


- generic platform for wireless distributed embedded computing
- complete platform including OS
- especially suited for pervasive computing applications (IoT)

Development in ES Exercise

Windows XP
GNU GCC
AVR libc
AVR Studio
Eclipse





BTNut OS

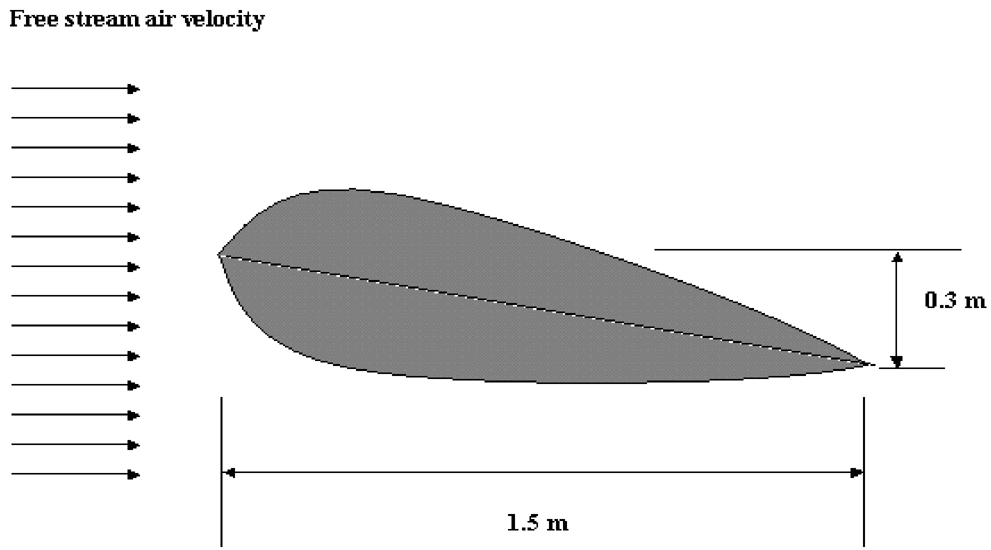
Timing Guarantees

4.1

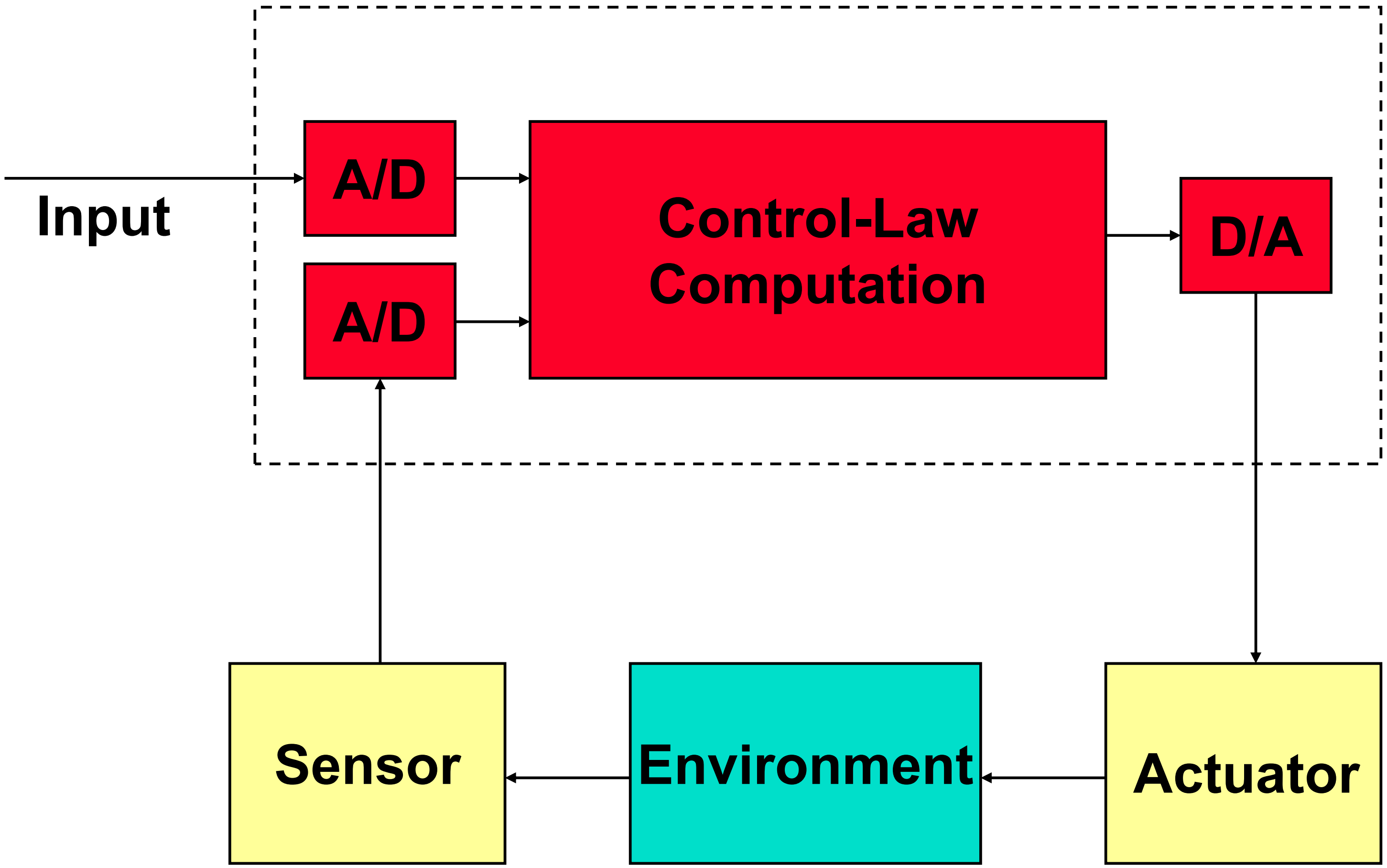
- ***Hard real-time systems***, often in safety-critical applications abound
  - Aeronautics, automotive, train industries, manufacturing control

Sideairbag in car,
Reaction in <10 mSec

Wing vibration of airplane,
sensing every 5 mSec

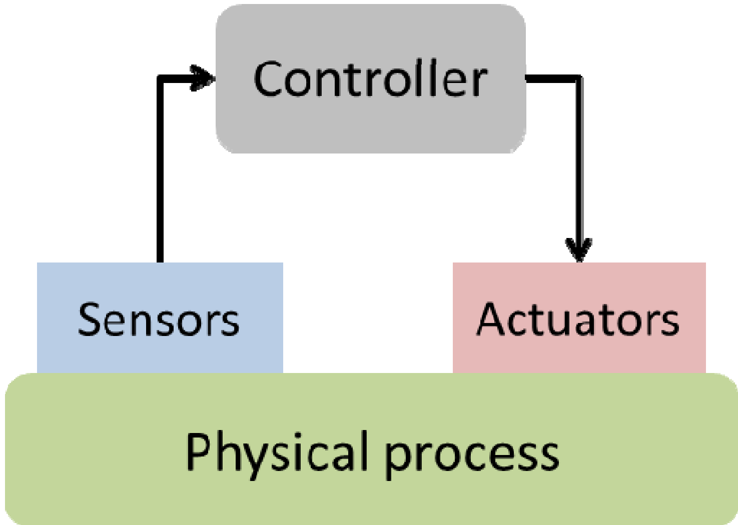


Simple Real-Time Control System

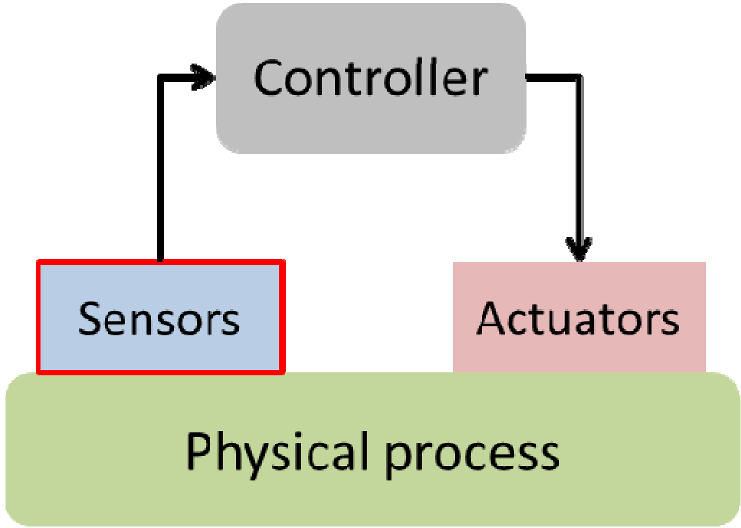


Real-Time Systems

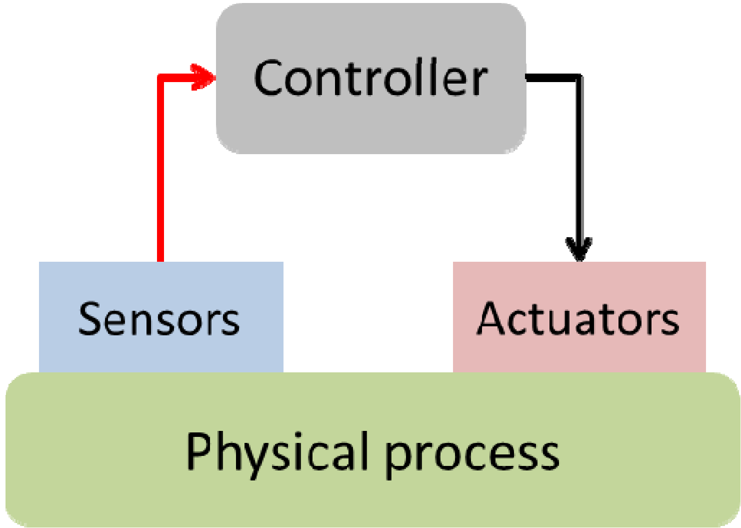
In many cyber-physical systems (CPSs) correct timing is a matter of ***correctness***, not performance.

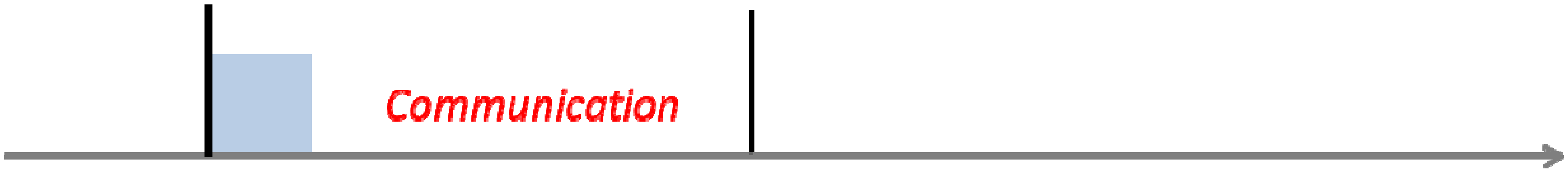


Real-Time Systems

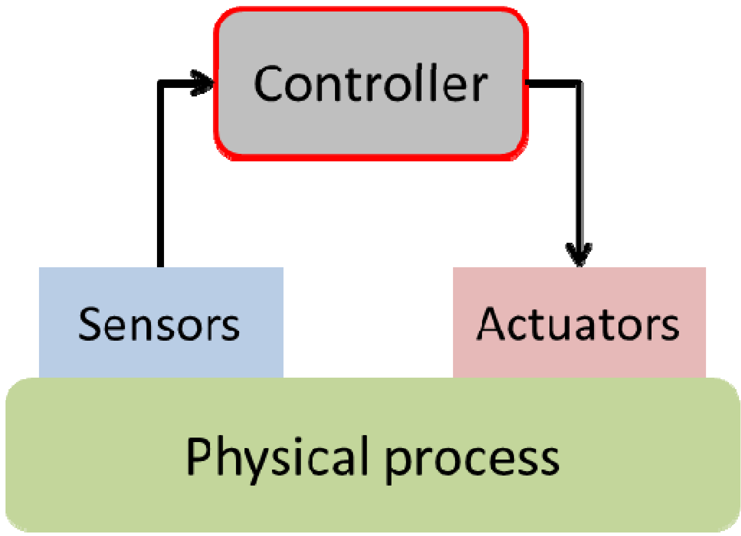


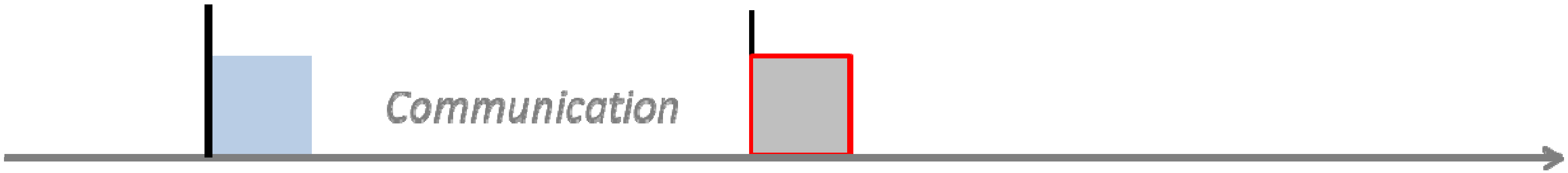
Real-Time Systems



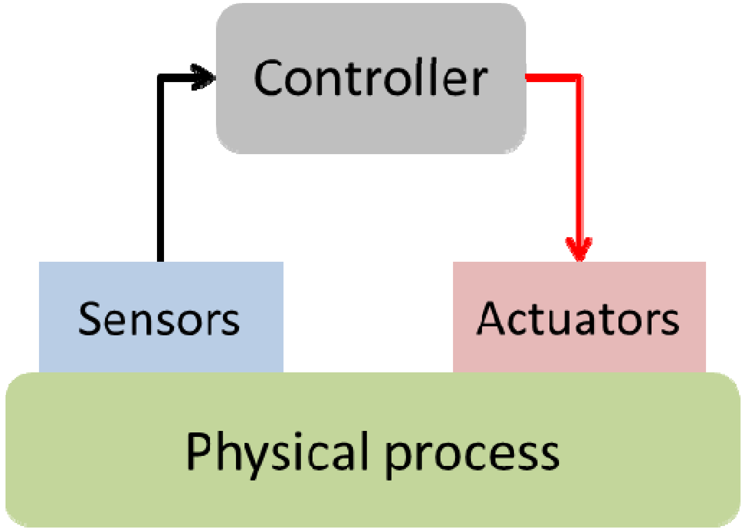


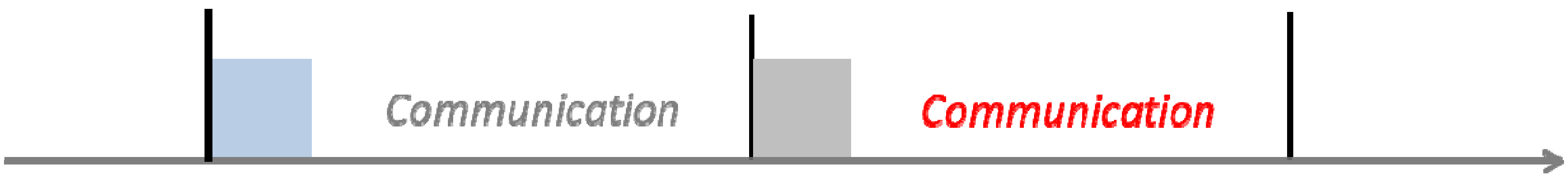
Real-Time Systems



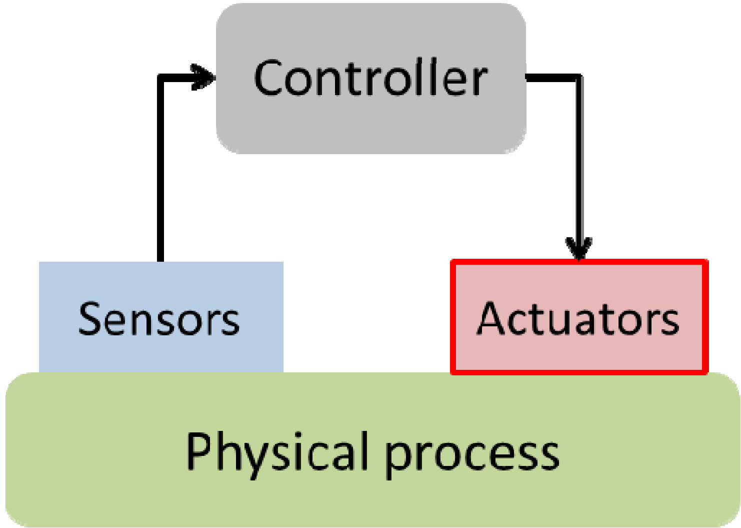


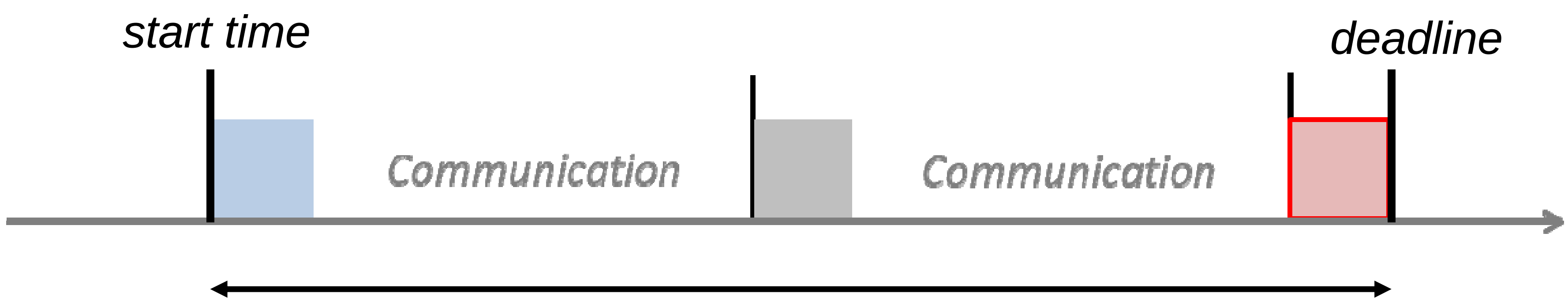
Real-Time Systems





Real-Time Systems

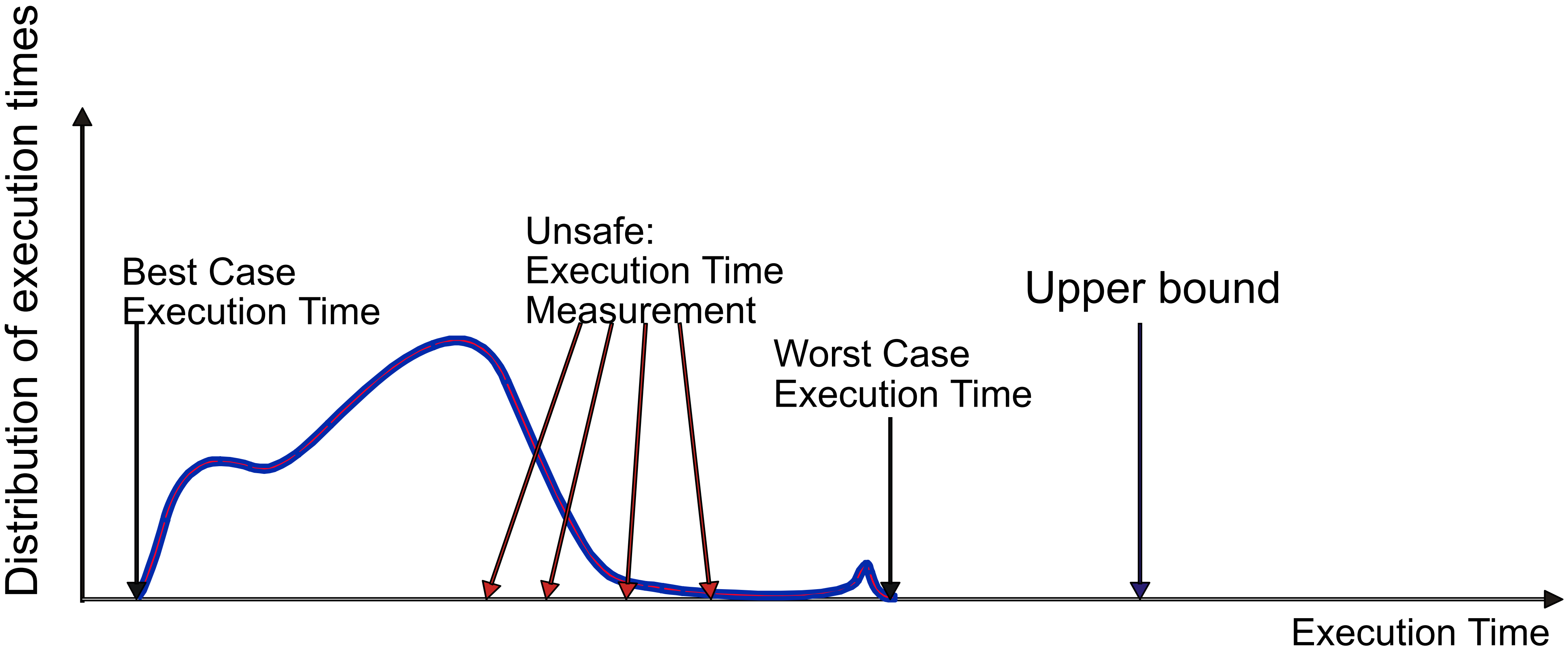




Real-Time Systems

- Embedded controllers are often expected to finish their tasks reliably within time bounds.
- Essential: ***upper bound on the execution times*** of all tasks statically known.
- Commonly called the ***Worst-Case Execution Time*** (WCET)
- Analogously, ***Best-Case Execution Time*** (BCET)

Distribution of Execution Times

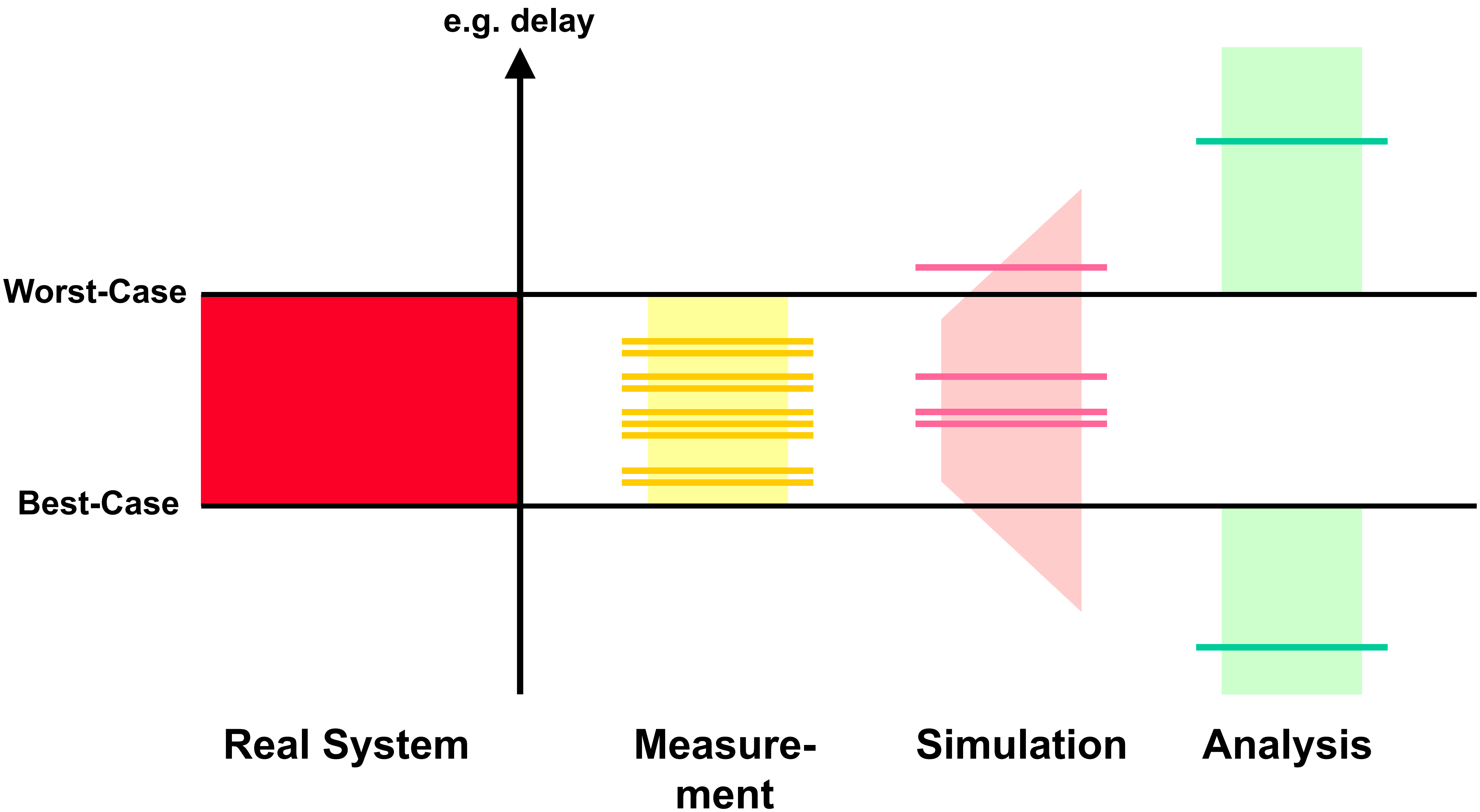


Modern Hardware Features

- Modern processors ***increase performance*** by using: *Caches, Pipelines, Branch Prediction, Speculation*

- These features make ***WCET computation difficult***: Execution times of instructions vary widely.
  - ***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.
  - *Span may be several hundred cycles.*

System-Level Performance Methods



(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.
- ***Exhaustive execution*** in general ***not possible***!
  - Too large space of (input domain) x (set of initial execution states).
- ***Compute upper bounds*** along the ***structure*** of the program:
  - Programs are hierarchically structured.
  - Instructions are "nested" inside statements.
  - So, compute the upper bound for a statement from the upper bounds of its constituents

Determine the WCET

Complexity:

- in the general case: undecidable whether a bound exists.
- for restricted programs: simple for „old“ architectures, very complex for new architectures with pipelines, caches, interrupts, virtual memory, etc.

Analytic (formal) approaches:

- for hardware: typically requires hardware synthesis
- for software: requires availability of machine programs; complex analysis (see, e.g., www.absint.de); requires precise machine (hardware) model [see lecture “hardware software codesign”].

Subtopics

- A few introductory remarks.

- Different programming paradigms.

Why Multiple Processes?

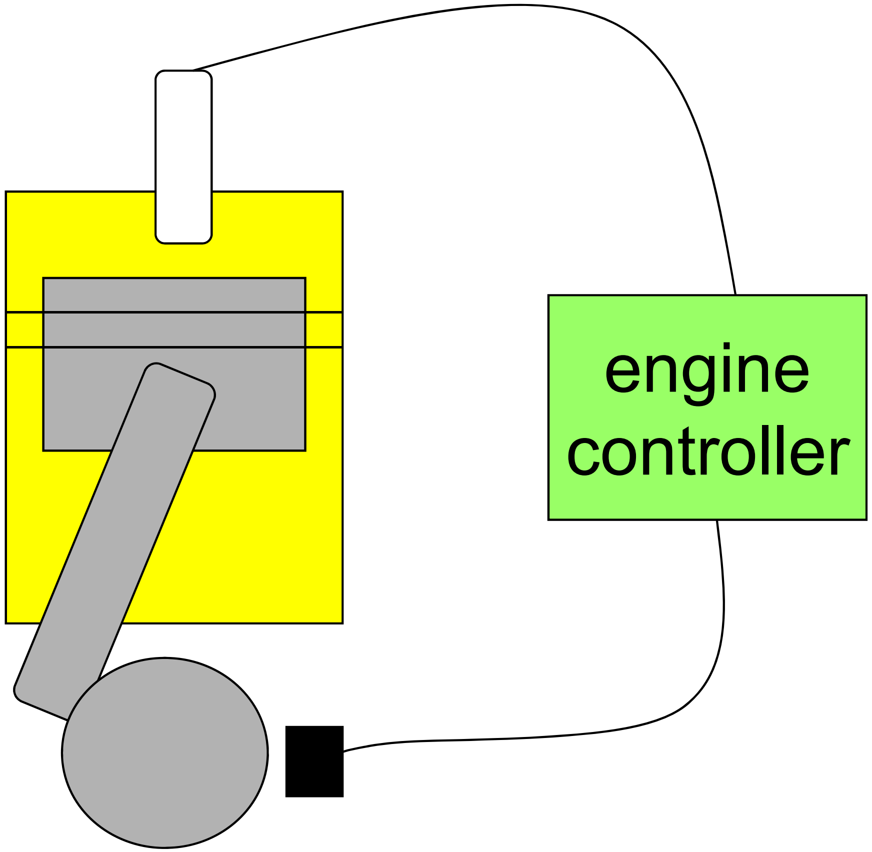
4.2

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

- Processes help us ***manage timing complexity***:
  - multiple rates
    - multimedia
    - automotive
  - asynchronous input
    - user interfaces
    - communication systems

Example: Engine Control

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

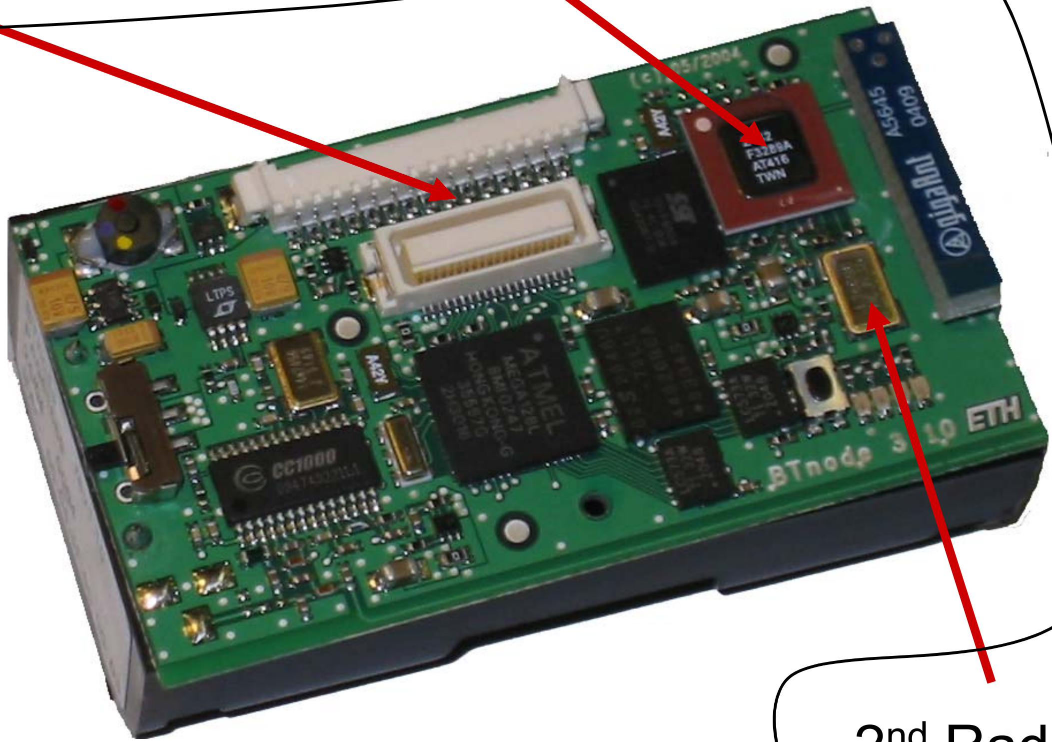


BTnode Platform

Communication via Bluetooth Transceiver

Data Interfaces

at least three interfaces



2nd Radio

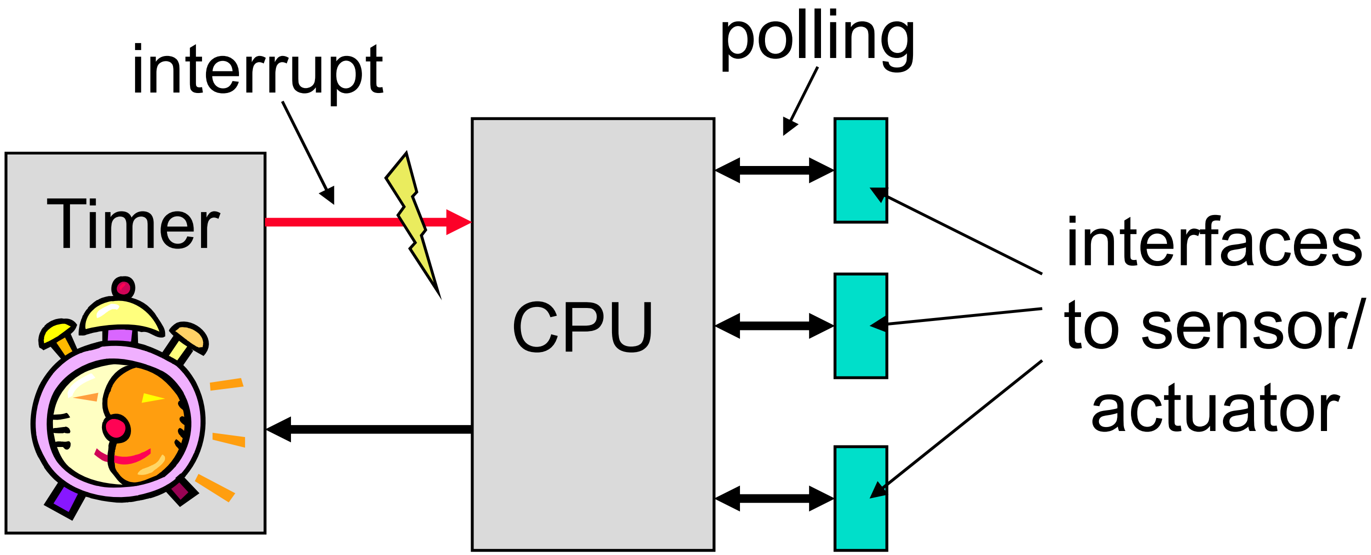
Overview

- There are ***MANY*** structured ways of programming an embedded system.
- Only ***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

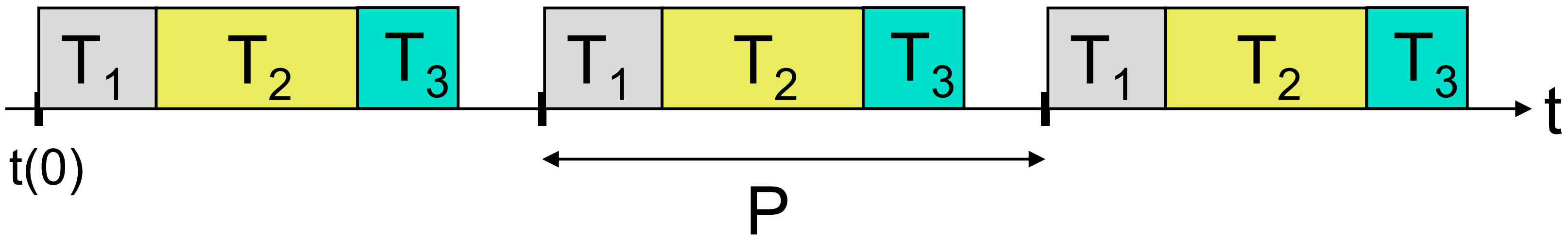
4.2.1

- ***Pure model***:
  - no interrupts except by timer
  - schedule computed off-line → complex sophisticated algorithms can be used
  - deterministic behavior at run-time
  - interaction with environment through polling



Simple Periodic TT Scheduler

- Timer interrupts regularly with period P.
- All processes have same period P.



- ***Properties***:
  - later processes (T_2, T_3) have unpredictable starting times
  - no problem with communication between processes or use of common resources, as there is a static ordering
  - $\sum_{(k)} WCET(T_k) < P$

Simple Periodic TT Scheduler

usually done offline

```
main:
   determine table of processes (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 process T(k); }
   return;
```

set CPU to low power mode;
returns after interrupt

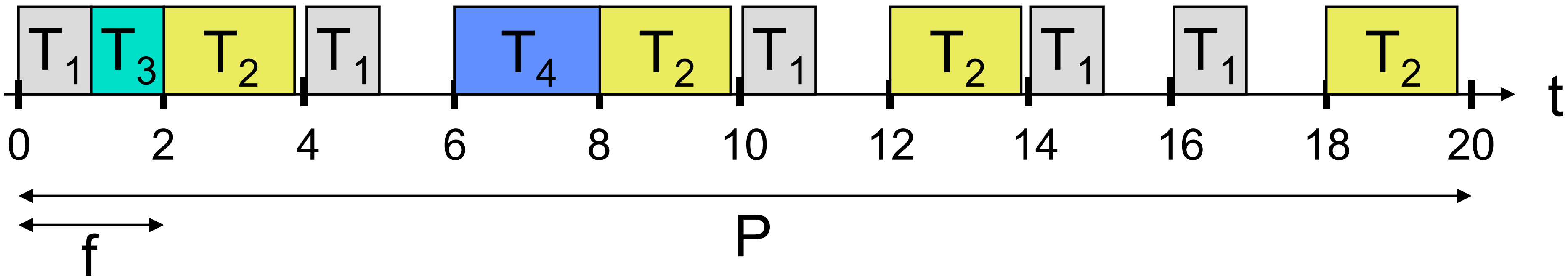
for example using a
function pointer in C;
task returns after finishing.

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

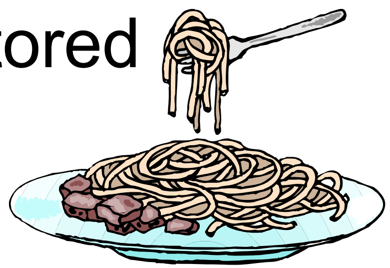
m=5

TT Cyclic Executive Scheduler

- Processes may have different periods.
- The period P is partitioned into frames of length f.



- Problem, if there are long processes; they need to be partitioned into a sequence of small processes; this is TERRIBLE, as local state must be extracted and stored globally:



TT Cyclic Executive Scheduler

- Some conditions:
  - A process executes at most once within a frame:

period of process k

$$f \leq p(k) \quad \forall k$$

  - Period *P* is least common multiple of all periods *p(k).*
  - Processes start and complete within a single frame:

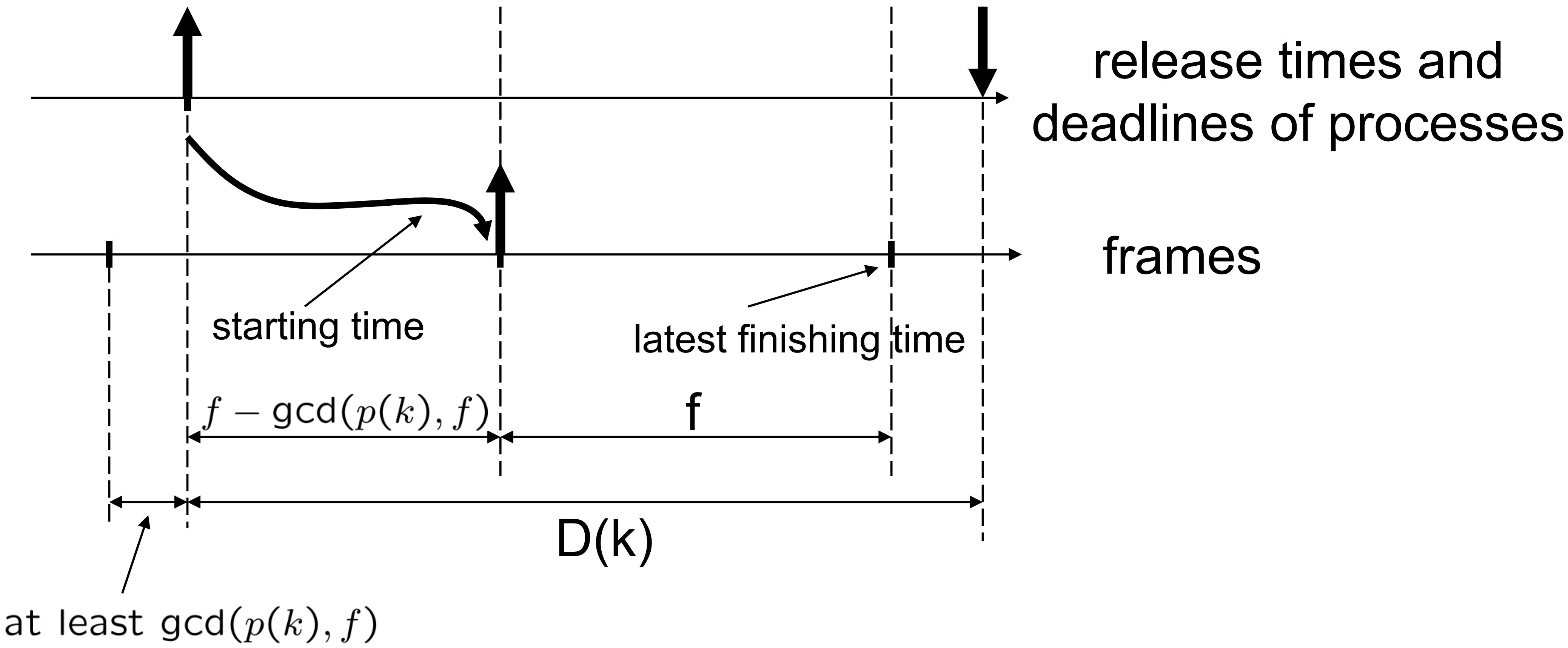
$$f \geq WCET(k) \quad \forall k$$

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

$$2f - \mathrm{gcd}(p(k), f) \leq D(k) \quad \forall k$$

relative deadline of process k

Sketch of Proof for Last Condition



Example: Cyclic Executive Scheduler

- Conditions:

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

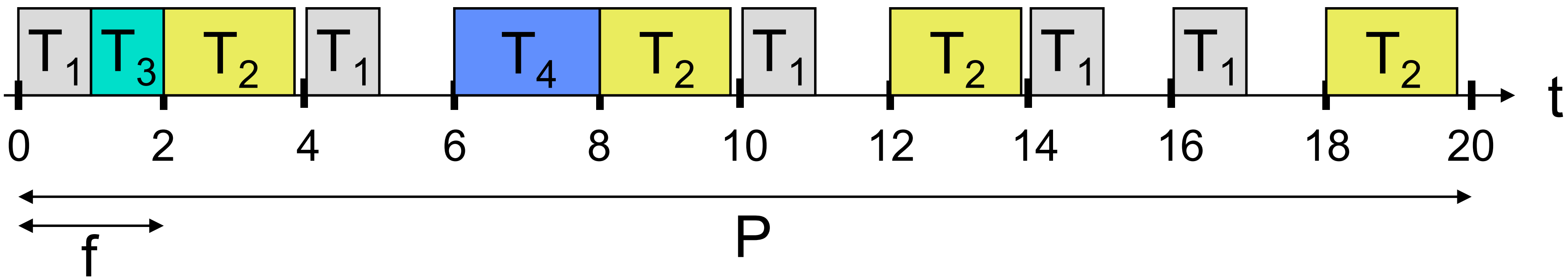
$$f \geq \max\{1.0, 1.0, 1.8, 2.0\} = 2.0$$

$$2f - \gcd(p(k), f) \leq D(k) \quad \forall k$$

possible solution: f = 2

T(k)	D(k)	p(k)	WCET(k)
T_1	4	4	1.0
T_2	5	5	1.8
T_3	20	20	1.0
T_4	20	20	2.0

- Feasible solution (f=2):

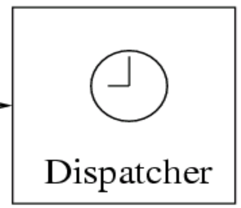


Generic Time-Triggered Scheduler

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 the 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 process T(k_old);
   return;
```

usually done offline

set CPU to low power mode; returns after interrupt

for example using a function pointer in C; process 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

possible extensions: execute aperiodic background tasks if system is idle; check for task overruns (WCET too long)

Summary Time-Triggered Scheduler

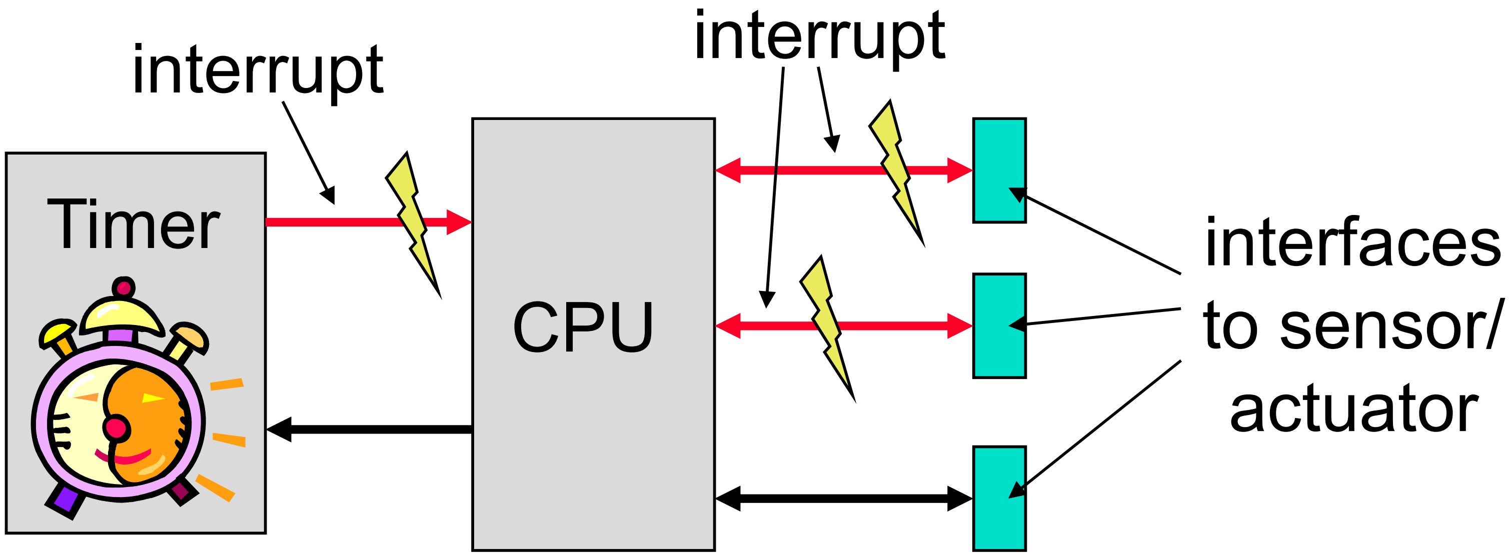
- **deterministic** schedule; conceptually simple (static table); relatively easy to validate, test and certify
- no problems in using ***shared resources***

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

- ***Extensions***:
  - allow interrupts (shared resources ? WCET ?) → ***be careful***!!
  - allow preemptable background processes

Event Triggered Systems

- The schedule of processes is determined by the occurrence of external interrupts:
  - *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 ET Scheduling

- ***Principle***:
  - To each event, there is associated a corresponding process that will be executed.
  - Events are emitted by (a) external interrupts and (b) by processes themselves.
  - Events are collected in a queue; depending on the queuing discipline, an event is chosen for running.
  - Processes can not be preempted.
- ***Extensions***:
  - A background process can run (and preempted!) if the event queue is empty.
  - Timed events enter the queue only after a time interval elapsed. This enables periodic instantiations for example.

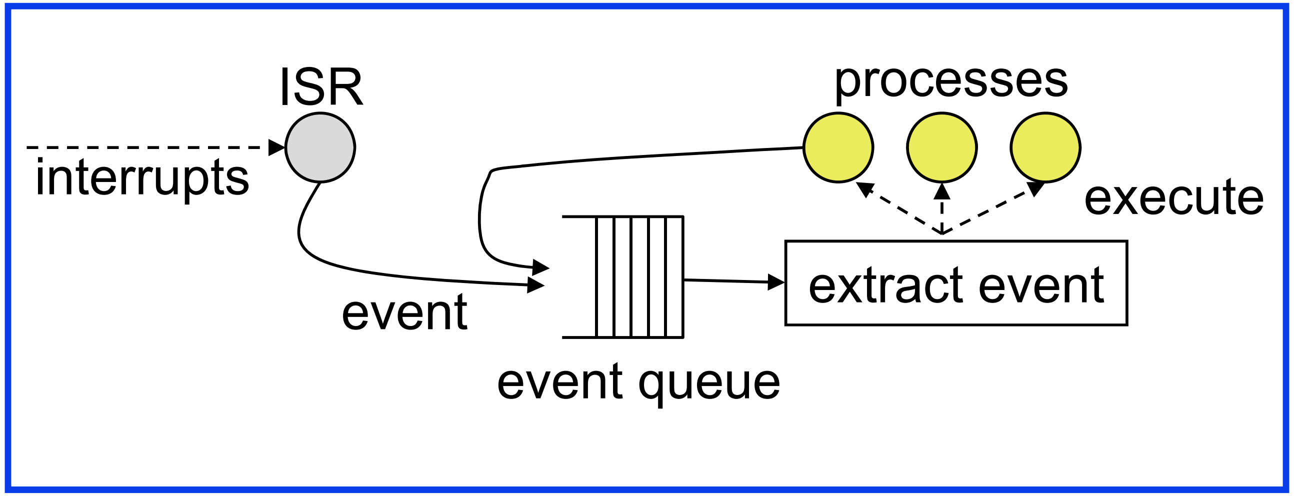
Non-Preemptive ET Scheduling

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

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

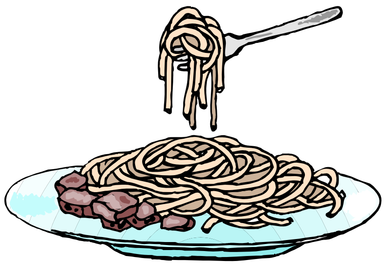
set CPU to low power mode; returns after interrupt

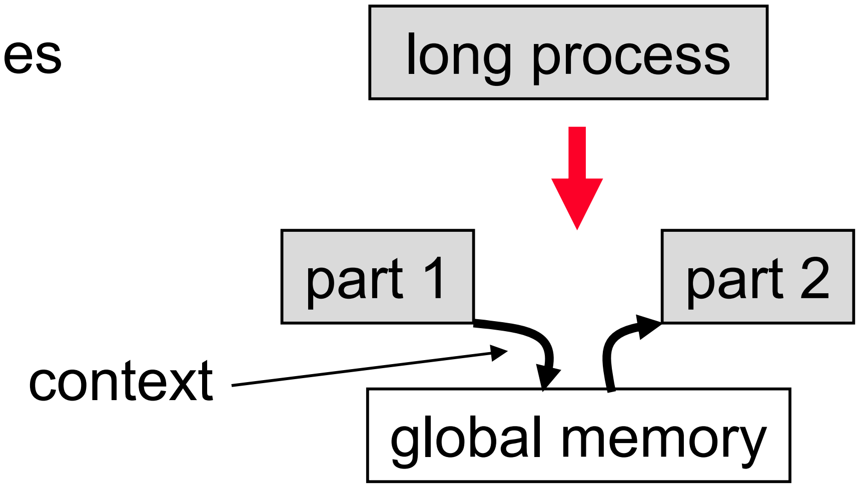
for example using a function pointer in C; process returns after finishing.



Non-Preemptive ET Scheduling

- ***Properties***:
  - communication between processes is simple (no problems with shared resources); interrupts may cause problems with shared resources
  - buffer overflow if too many events are generated by environment or processes
  - long processes prevent others from running and may cause buffer overflow
    - partition processes into smaller ones
    - local context must be stored



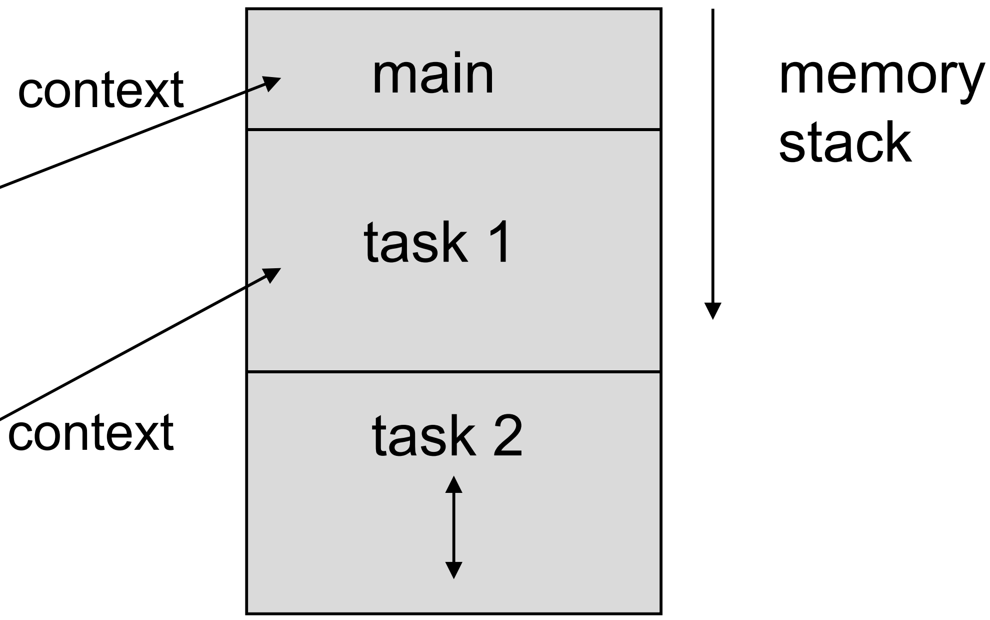


Preemptive ET Scheduling – Stack Policy

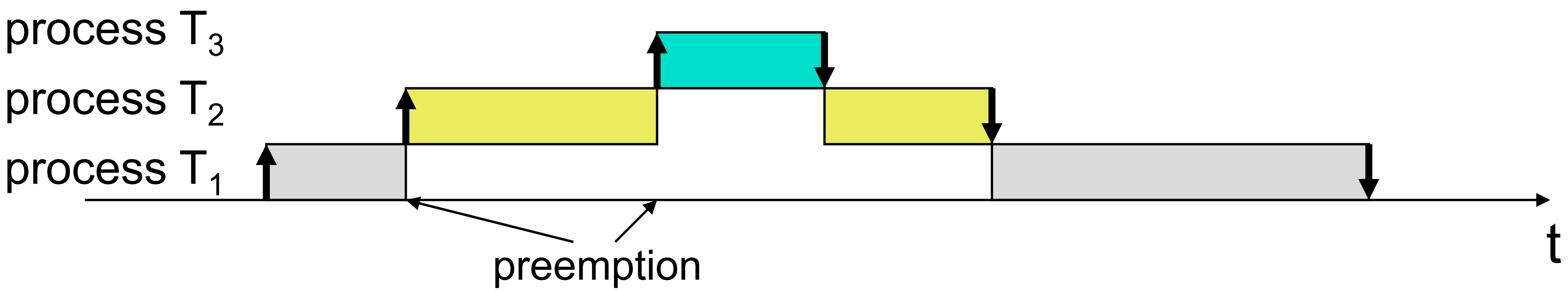
- Similar to non-preemptive case, but ***processes can be preempted*** by others; this resolves partly the problem of long tasks.
- If the order of preemption is restricted, we can use the usual ***stack-based context mechanism of function calls*** (process = function).

```
main(){
        …
        task1();
        …

task1(){
        …
        task2();
        …
```



Preemptive ET Scheduling – Stack Policy



- Processes must finish in ***LIFO order*** of their instantiation.
  - restricts flexibility
  - not useful, if several processes wait unknown time for external events
- ***Shared resources*** (communication between processes!) must be ***protected***, for example: disabling interrupts, use of semaphores.

Preemptive ET Scheduling – Stack Policy

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

set CPU to low power mode;
returns after interrupt

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

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

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

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

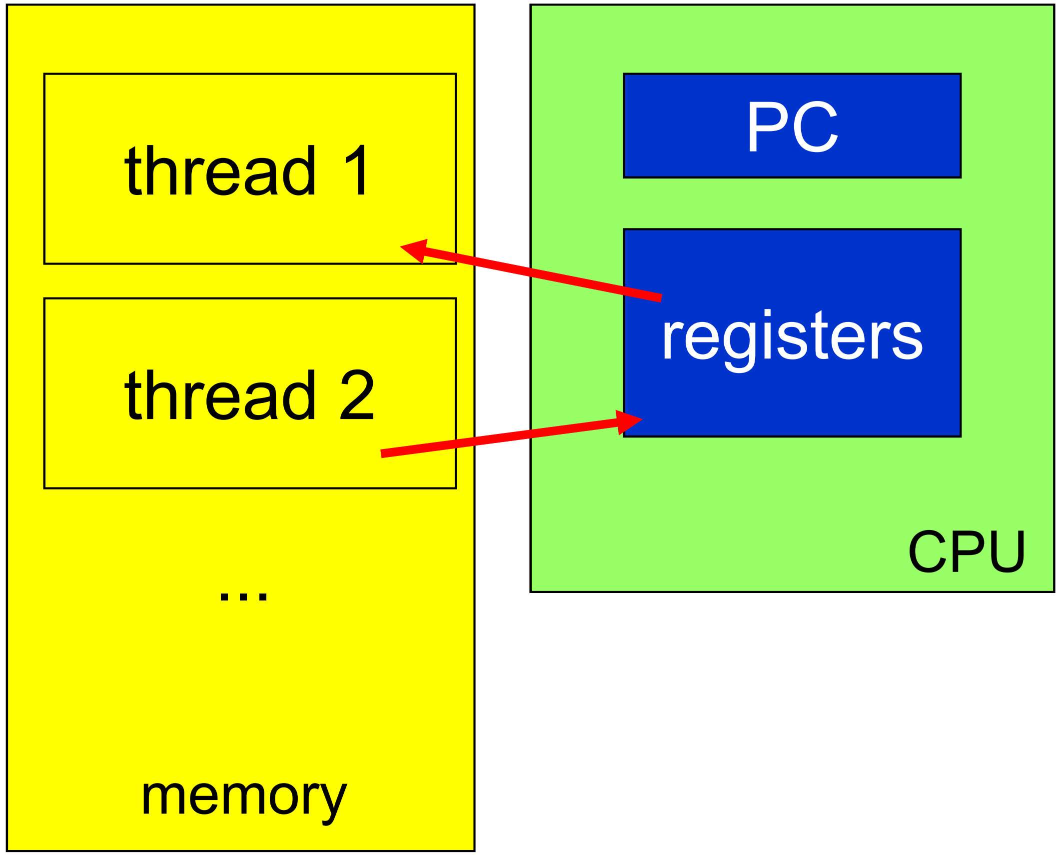
Process

- ***A process is a unique execution of a program*.**
  - Several copies of a “program” may run simultaneously or at different times.

- A ***process has its own state***. In case of a thread, this state consists mainly of:
  - register values;
  - memory stack;

Processes and CPU

- ***Activation record:***
  - copy of process state
  - includes registers and local data structures
- ***Context switch***:
  - current CPU context goes out
  - new CPU context goes in



Co-operative Multitasking

- Each process allows a context switch at `cswitch()` call.
- Separate scheduler chooses which process runs next.

- ***Advantages***:
  - predictable, where context switches can occur
  - less errors with use of shared resources

- ***Problems***:
  - programming errors can keep other threads out, thread never gives up CPU
  - real-time behavior at risk if it takes too long before context switch allowed

Example: co-operative multitasking

Process 1

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

Process 2

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

Scheduler

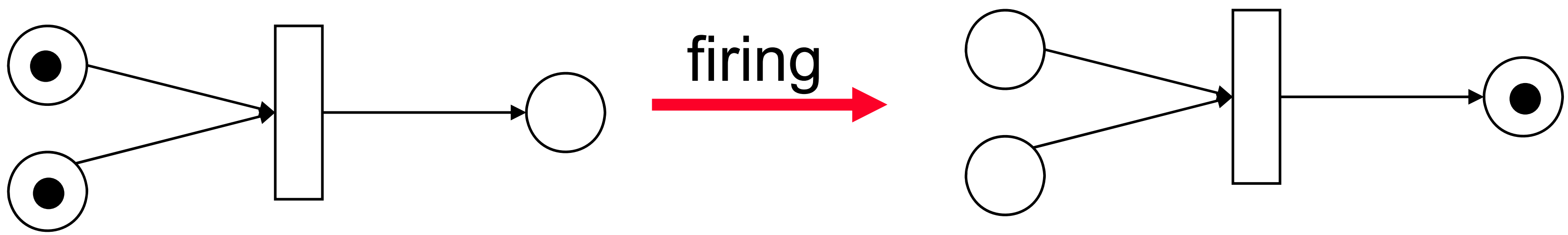
```
save_state(current);
p = choose_process();
load_and_go(p);
```

A Typical Programming Interface

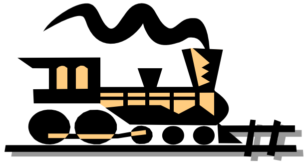
2.6

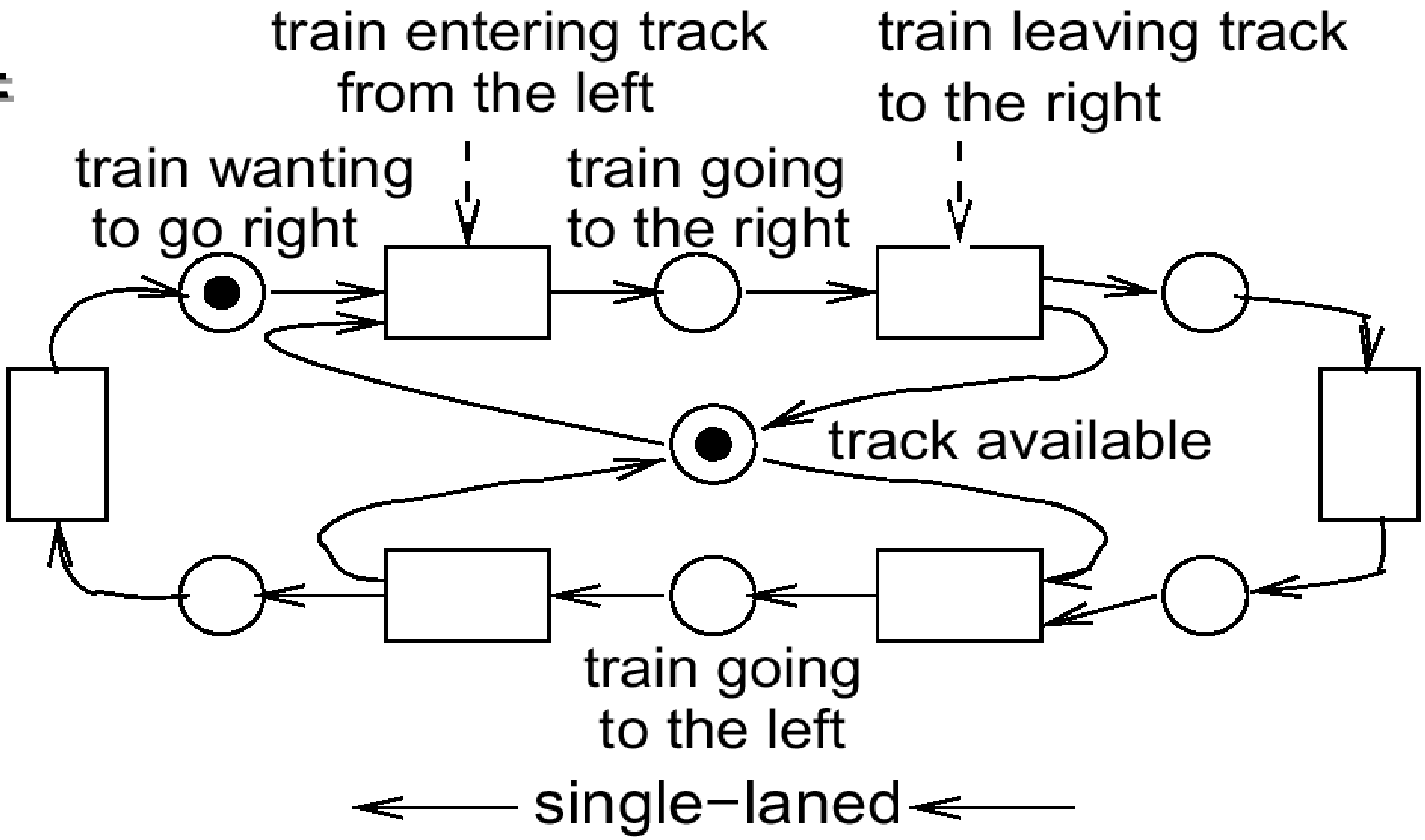
- Example of a co-operative multitasking OS for small devices: ***NutOS*** (as used in the practical exercises).

- Semantics of the calls is expressed using ***Petri Nets***
  - Bipartite graph consisting of ***places*** and ***transitions***.
  - Data and control are represented by moving ***token***.
  - Token are moved by transitions according to ***rules***: A transition can ***fire*** (is enabled) if there is at least one token in every input place. After firing, one token is removed from each input place and one is added to each output place.

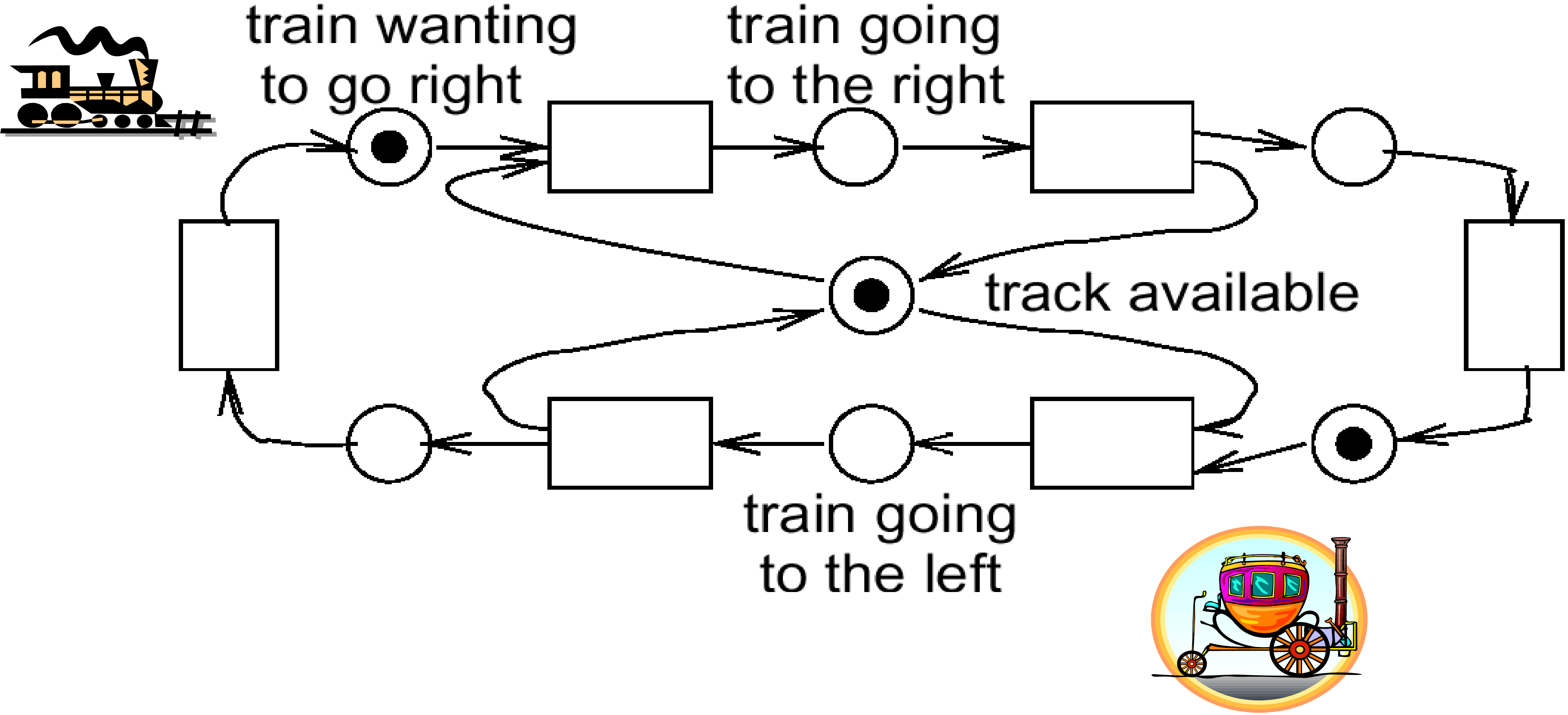


Example: Single Track Rail Segment





Example: Conflict for Resource „Track“



API for Co-Operative Scheduling

- ***Creating a Thread***

```
THREAD(my_thread, arg) {
    for (;;) {
        // do something
    }
}
```

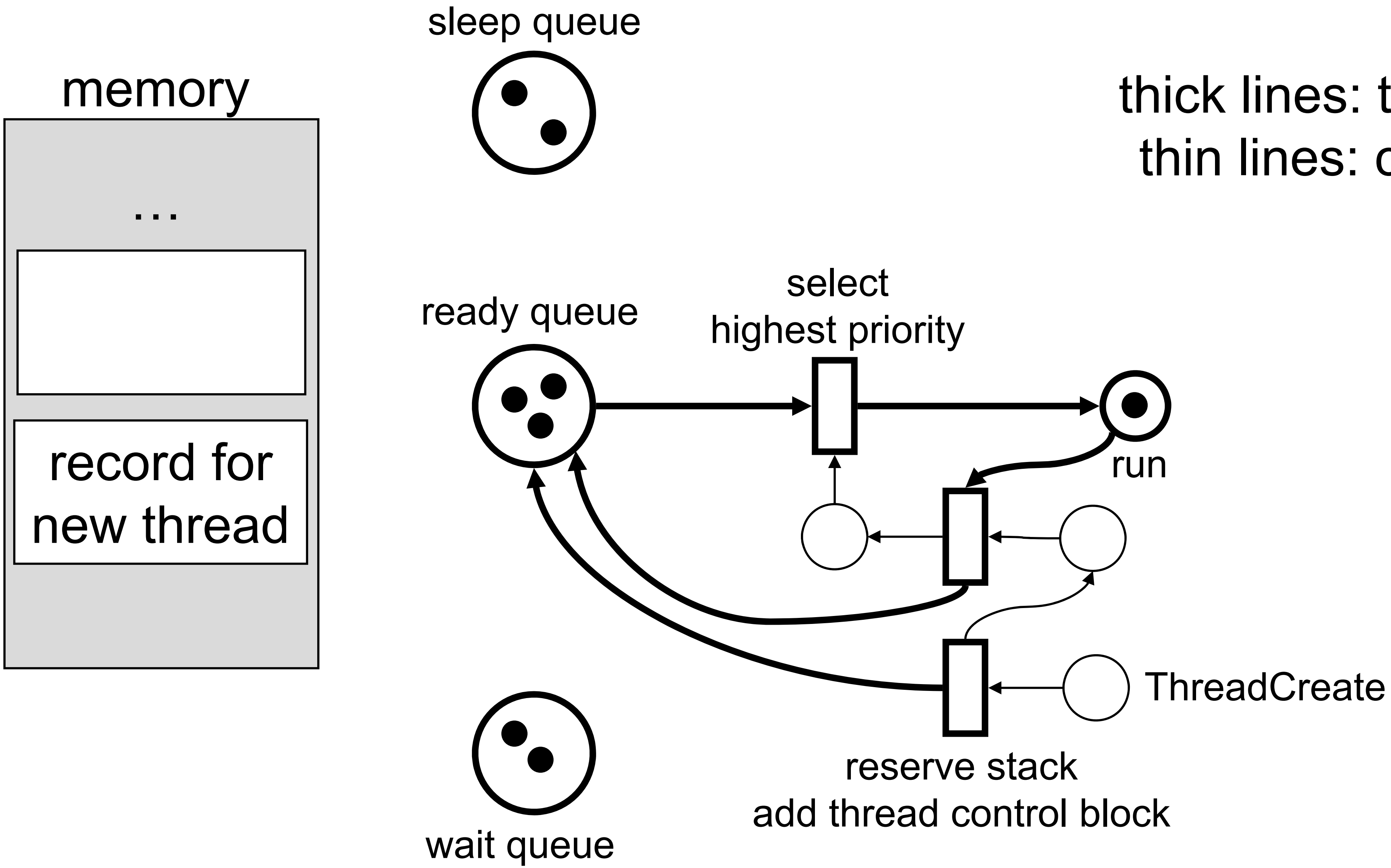
a thread looks like a function that never returns

the thread is put into life

```
int main(void) {
    if (0 == NutThreadCreate("My Thread", my_thread, 0, 192)) {
        // Creating the thread failed
    }
    for (;;) {
        // do something
    }
}
```

stack size

API for Co-Operative Scheduling



thick lines: threads
thin lines: control

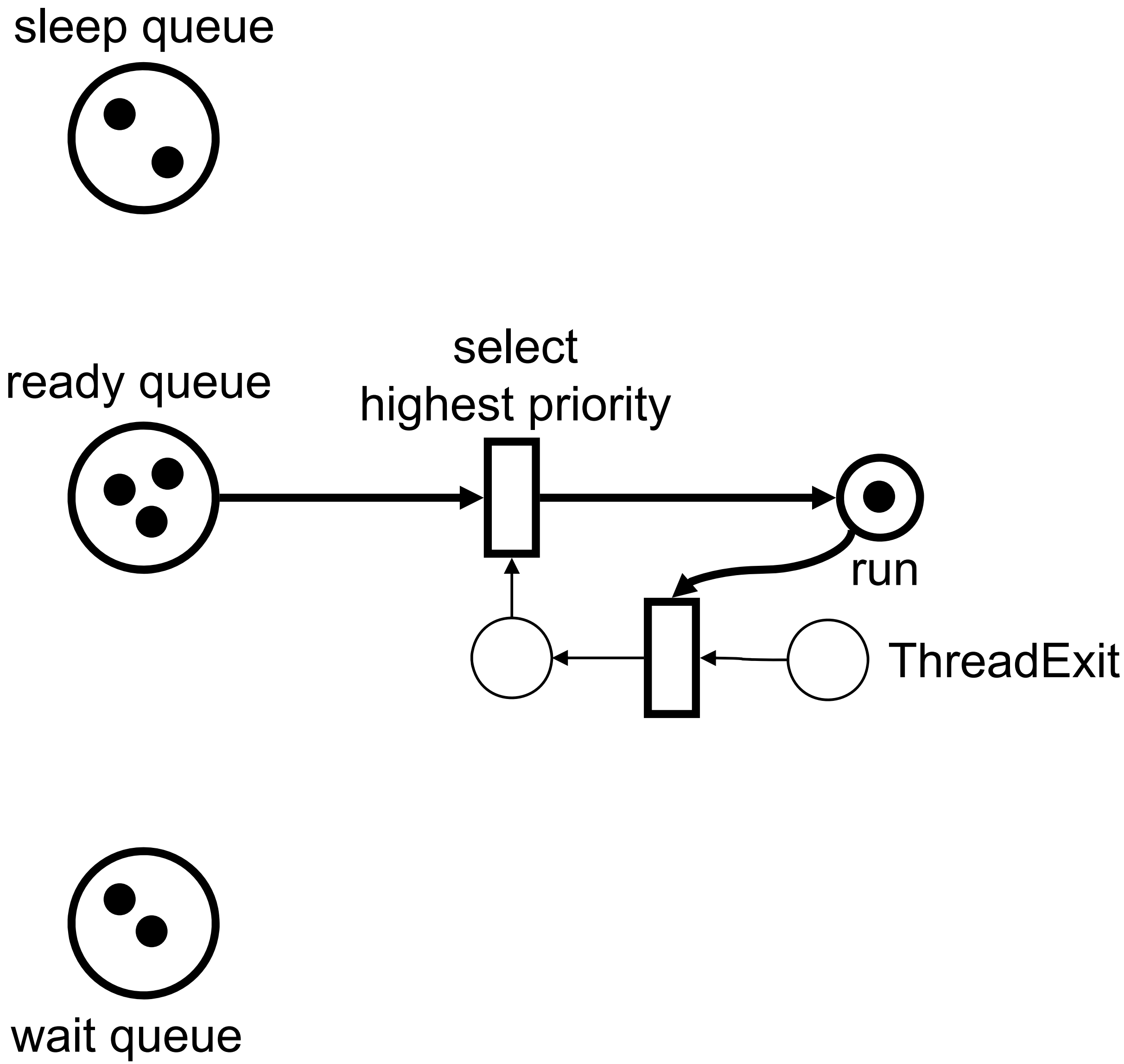
API for Co-Operative Scheduling

- ***Terminating***

```
THREAD(my_thread, arg) {
    for (;;) {
        // do something
        if (some condition)
            NutThreadExit()
    }
}
```

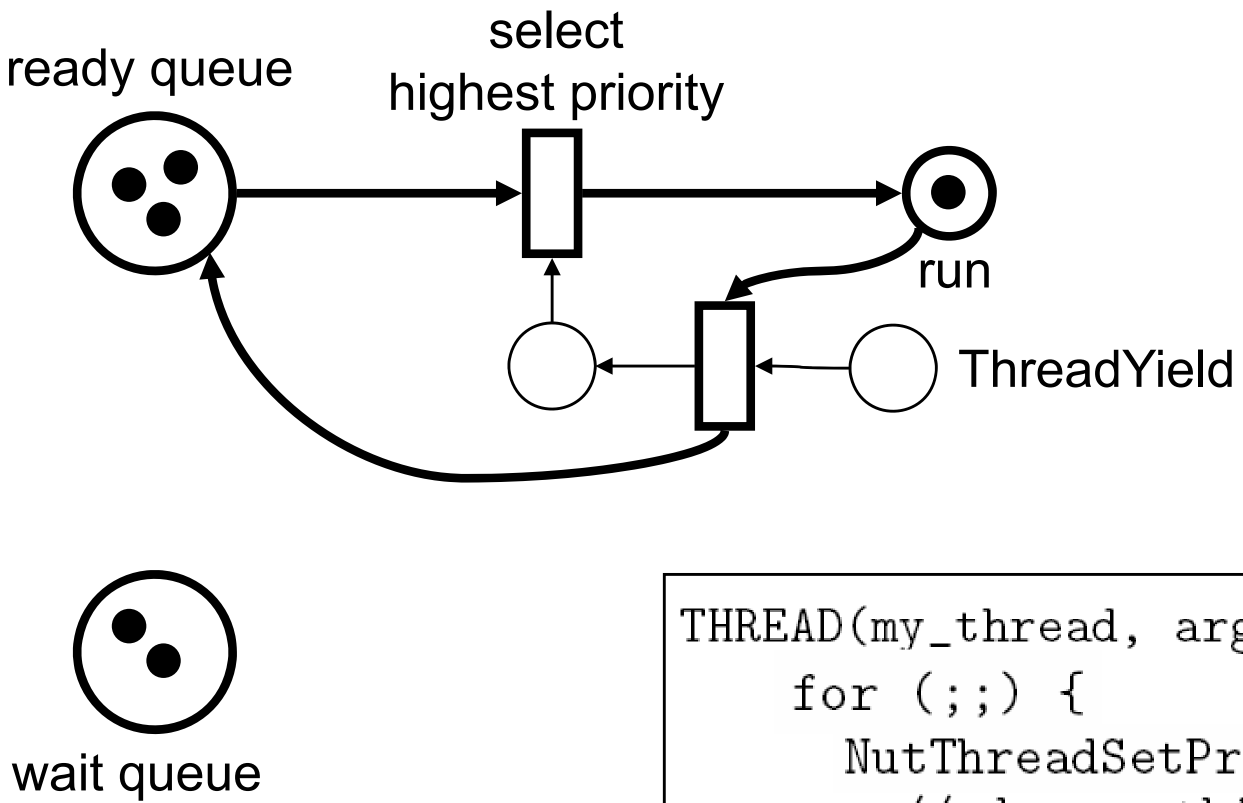
can only kill itself

API for Co-Operative Scheduling



API for Co-Operative Scheduling

- *Yield access to another thread:*



- *Same structure for SetPriority:*

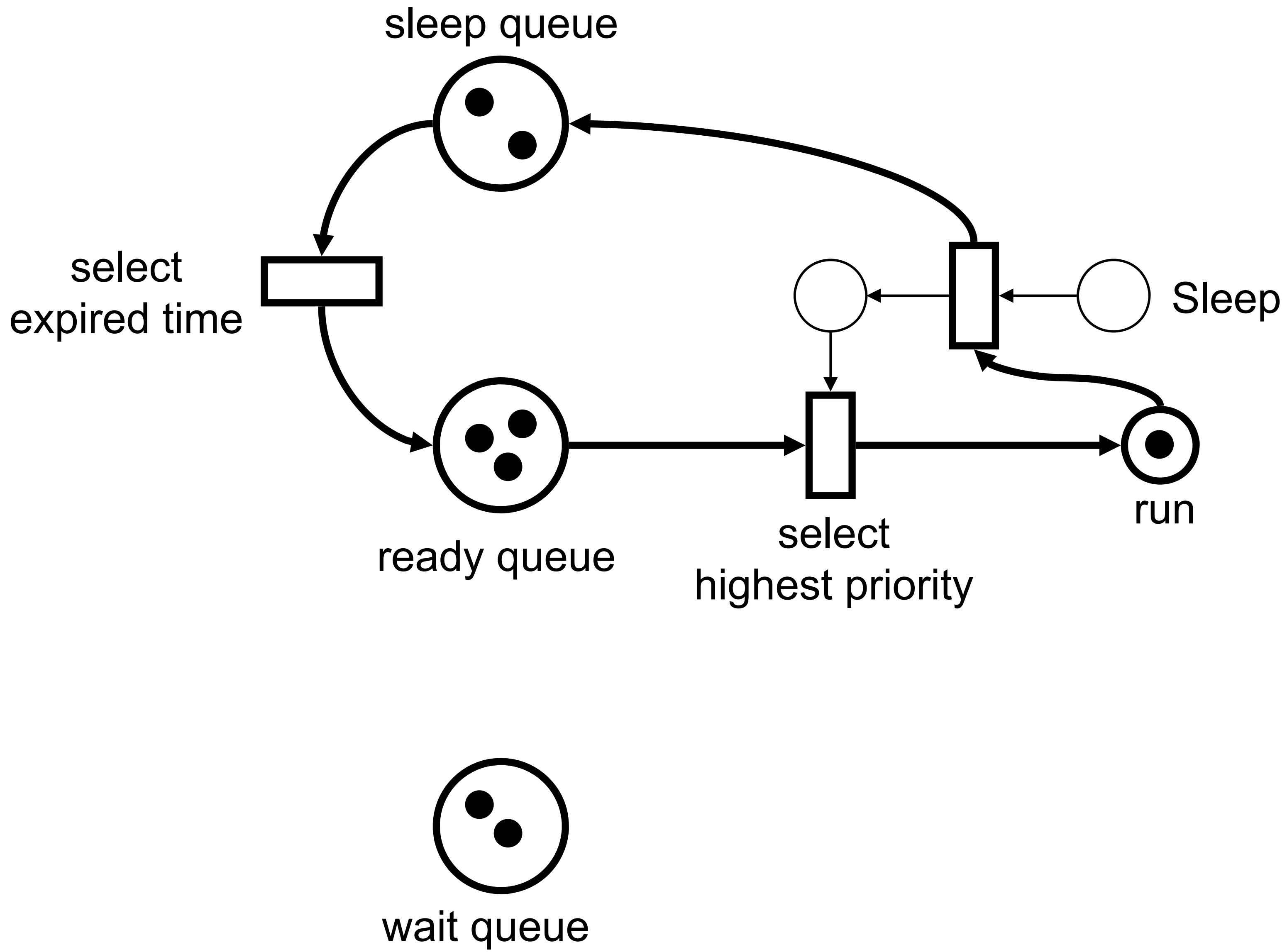
```
THREAD(my_thread, arg) {
    for (;;) {
      NutThreadSetPriority(20);
        // do something
    }
}
```

API for Co-Operative Scheduling

- *Sleep*

```
THREAD(my_thread, arg) {
    for (;;) {
        // do something
        NutSleep(1000);
    }
}
```

API for Co-Operative Scheduling



API for Co-Operative Scheduling

- ***Posting and waiting for events***:

```
#include <sys/event.h>

HANDLE my_event;


THREAD(thread_A, arg) {
    for (;;) {
        // some code
        NutEventWait(&my_event, NUT_WAIT_INFINITE);
        // some code
    }
}

THREAD(thread_B, arg) {
    for (;;) {
        // some code
        NutEventPost(&my_event);
        // some code
    }
}
```

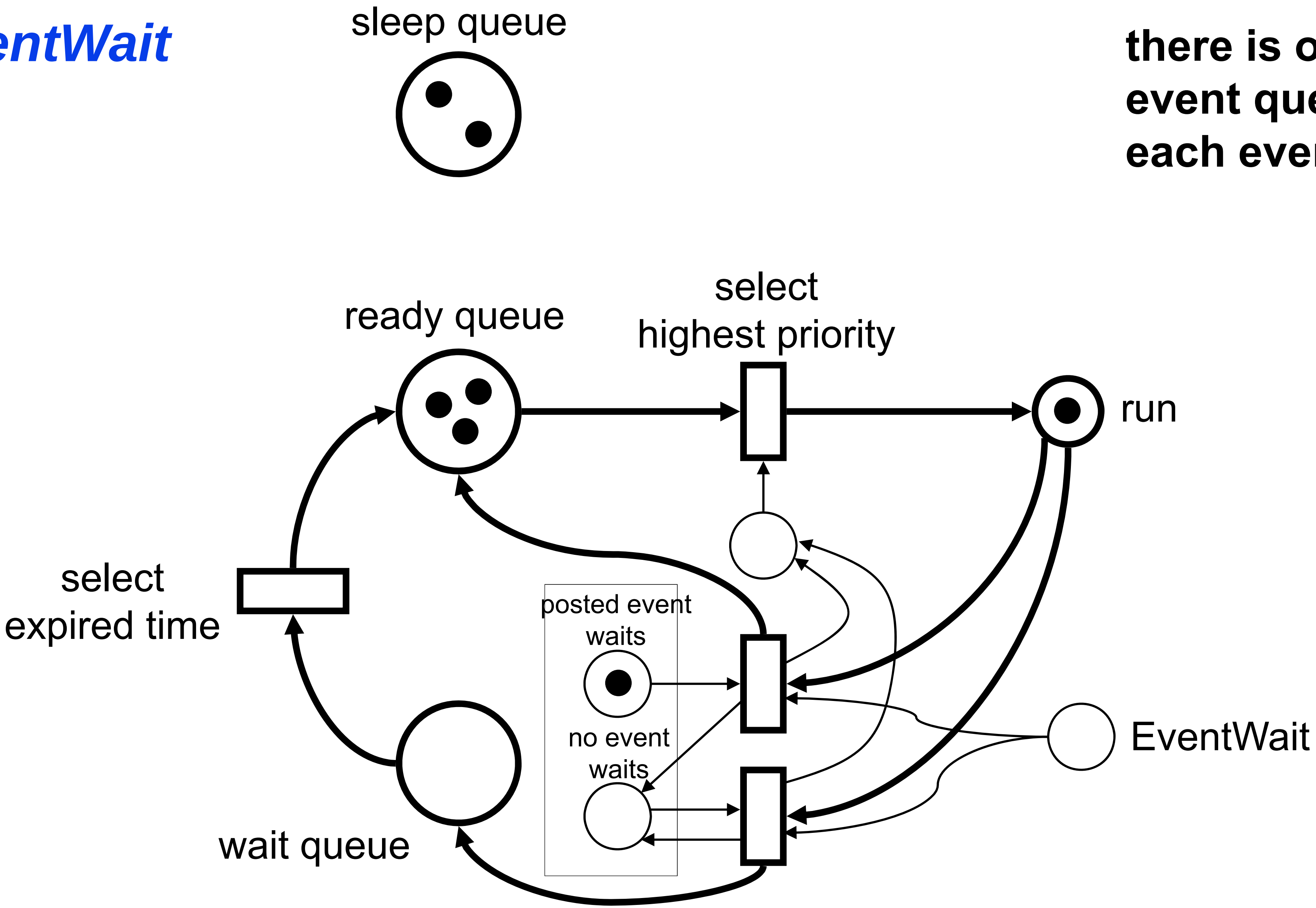
wait for event, but only limited time

post event

API for Co-Operative Scheduling

EventWait

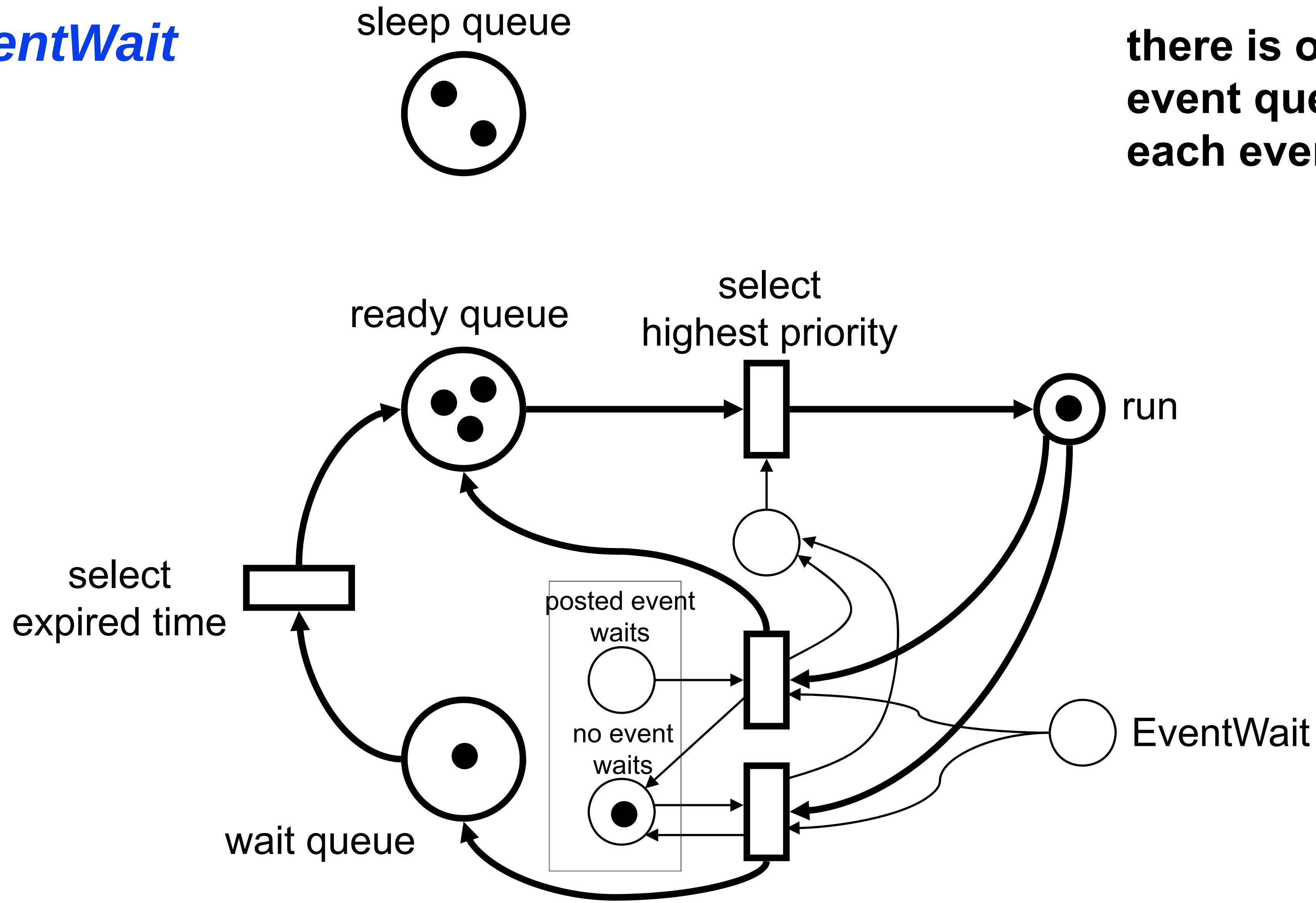
there is one event queue for each event type



API for Co-Operative Scheduling

EventWait

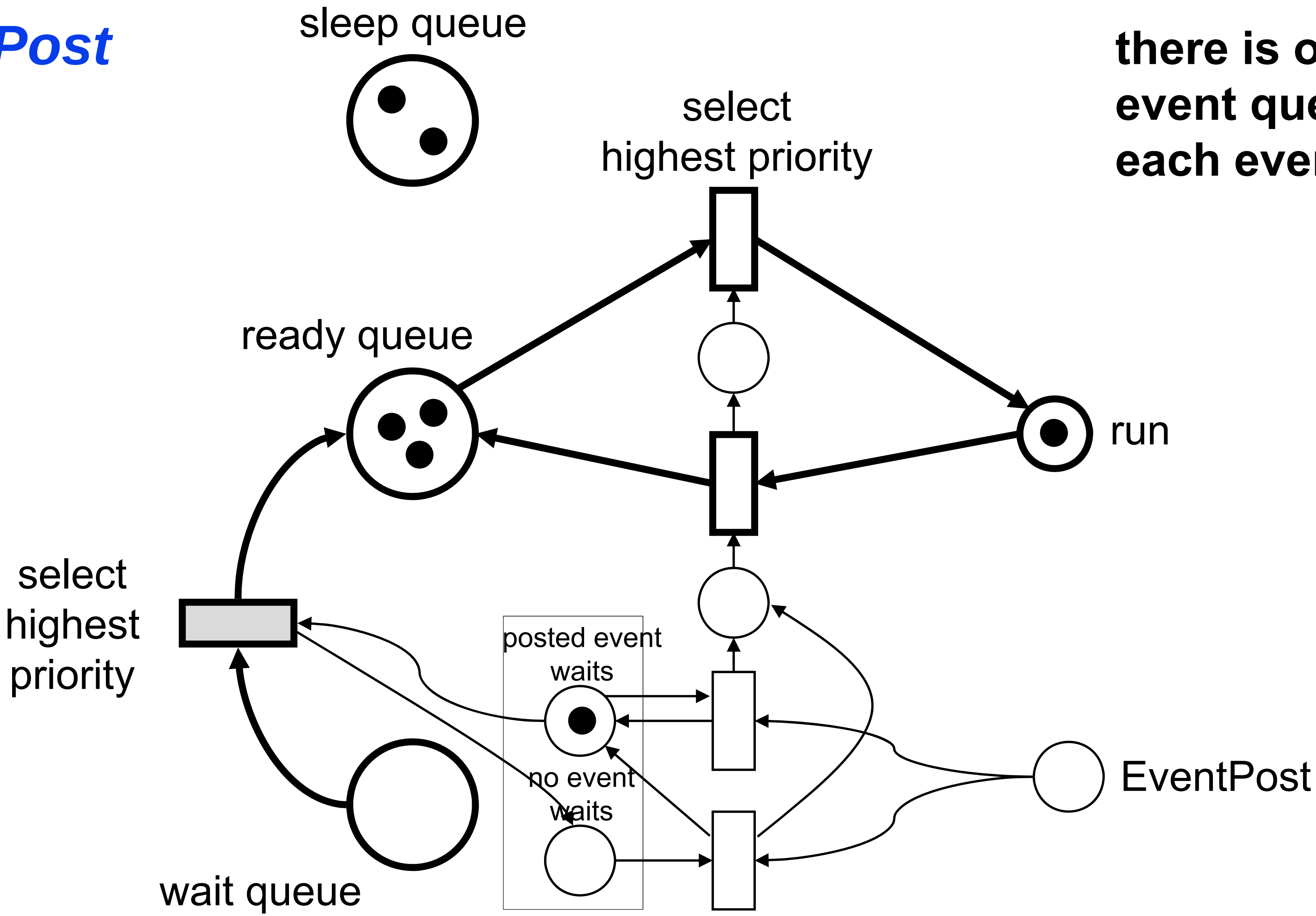
there is one event queue for each event type



API for Co-Operative Scheduling

EventPost

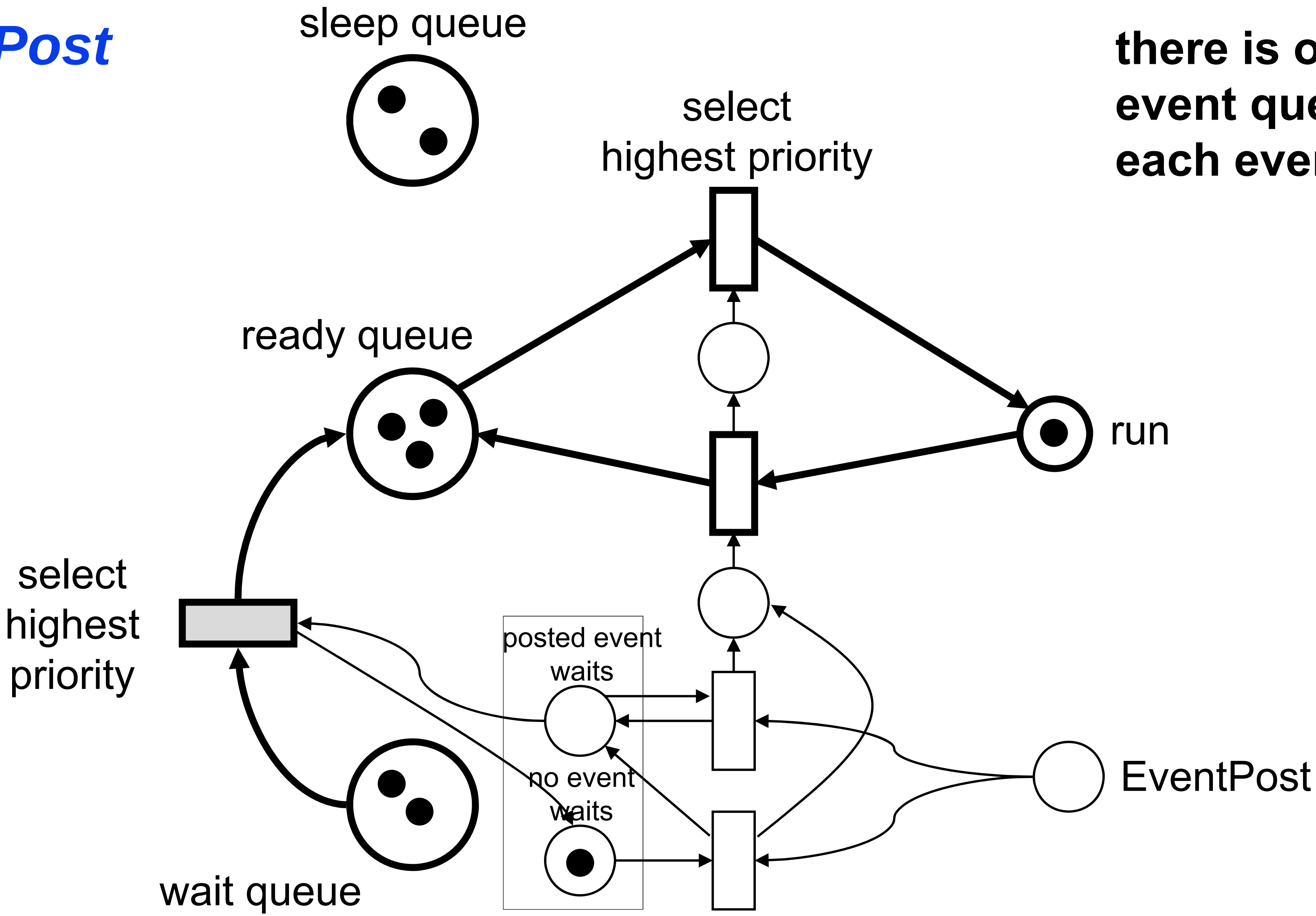
there is one event queue for each event type



API for Co-Operative Scheduling

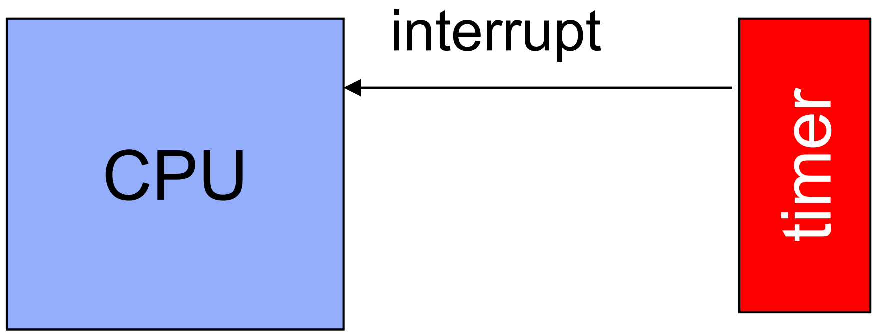
EventPost

there is one event queue for each event type



Preemptive Multitasking

- ***Most powerful form of multitasking***:
  - Scheduler (OS) controls when contexts switches;
  - Scheduler (OS) determines what process runs next.
- Use of ***timers*** to call OS and switch contexts:



- Use ***hardware or software interrupts***, or direct calls to OS routines to switch context.

Flow of Control with Preemption