

TinyTimber

Johan Nordlander

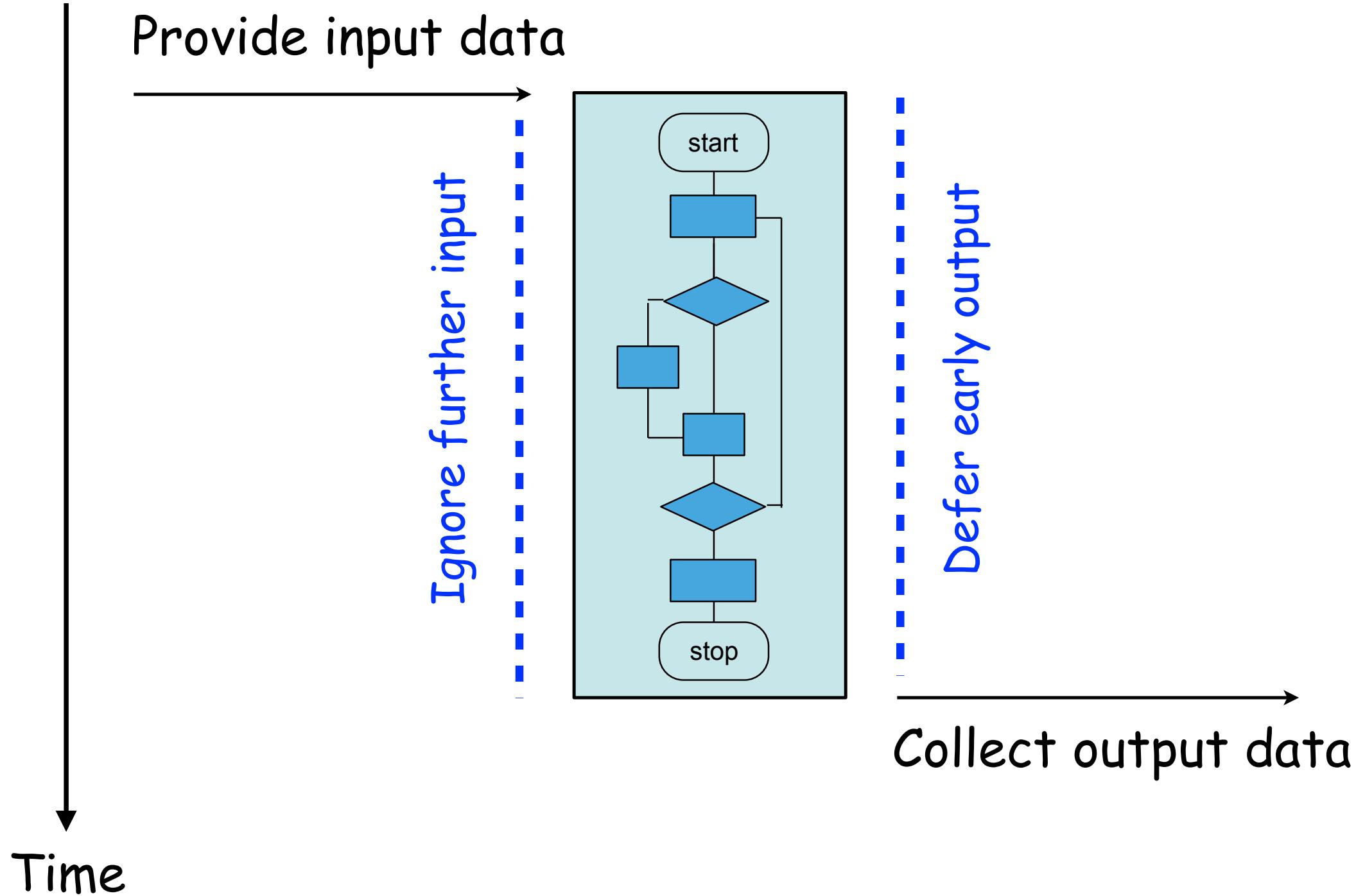
johan.nordlander@dataductus.se

Data Ductus AB / Chalmers

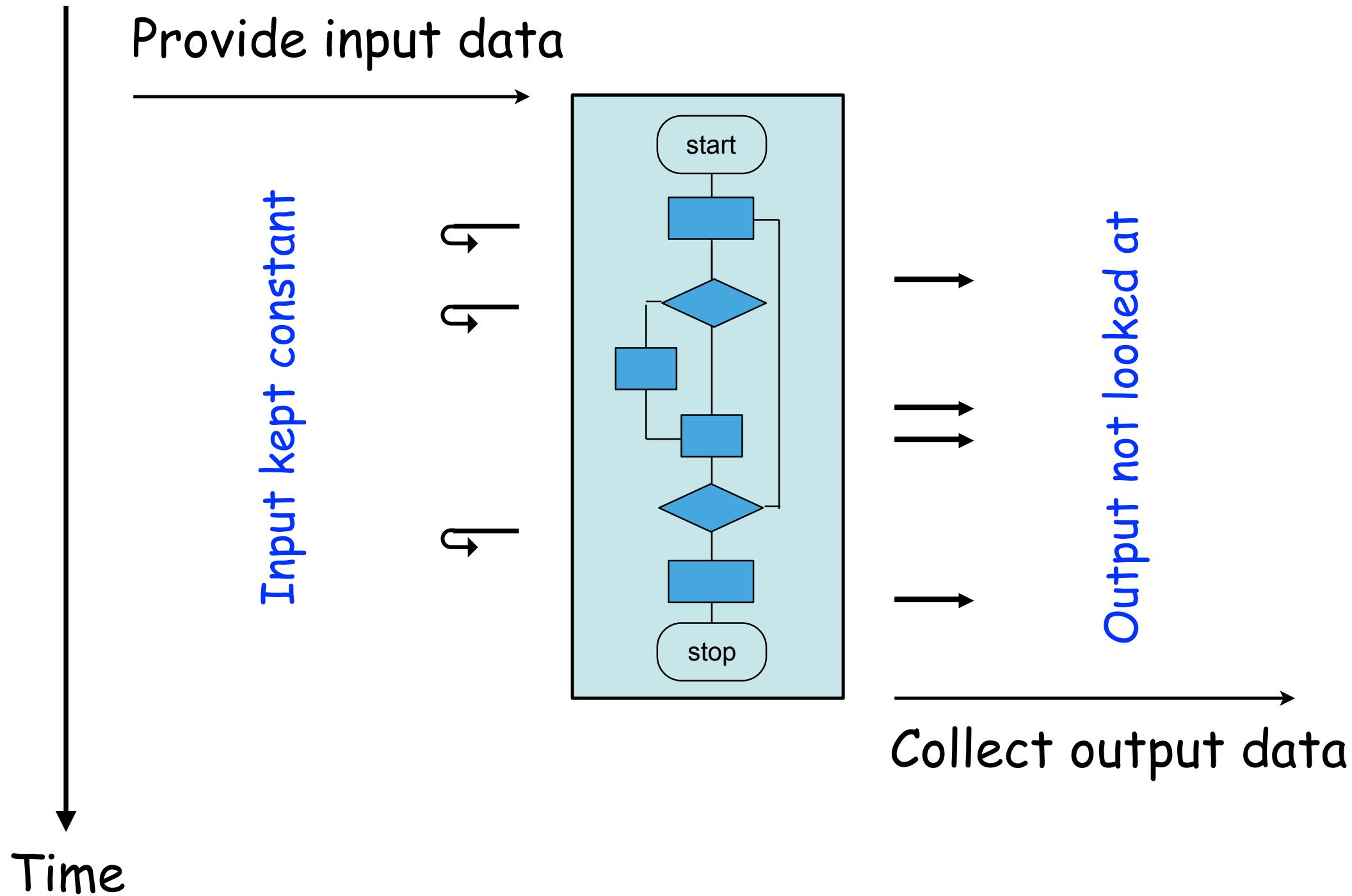
Real-Time Systems EDA223

Jan 24, 2019

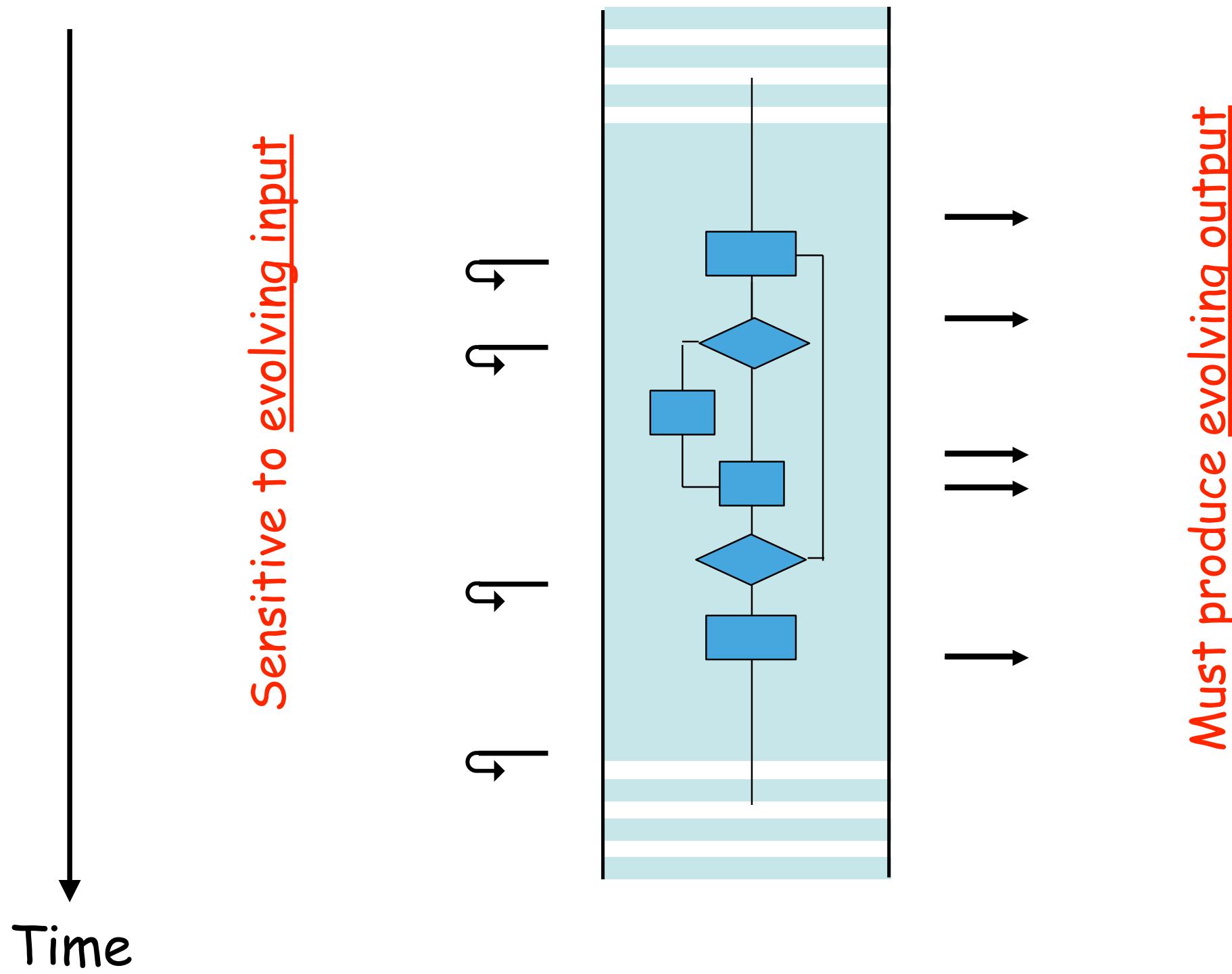
The classical program



In practice



Modern programs



Why?

- Because modern computers are **components among other evolving components** like
 - Keyboards, mice and displays
 - Human users behind these components
 - Network interfaces
 - Other computers behind these components
 - Sensors and actuators
 - Real physical objects behind these components
- Because a modern computer program is very **rarely in superior control** of its environment

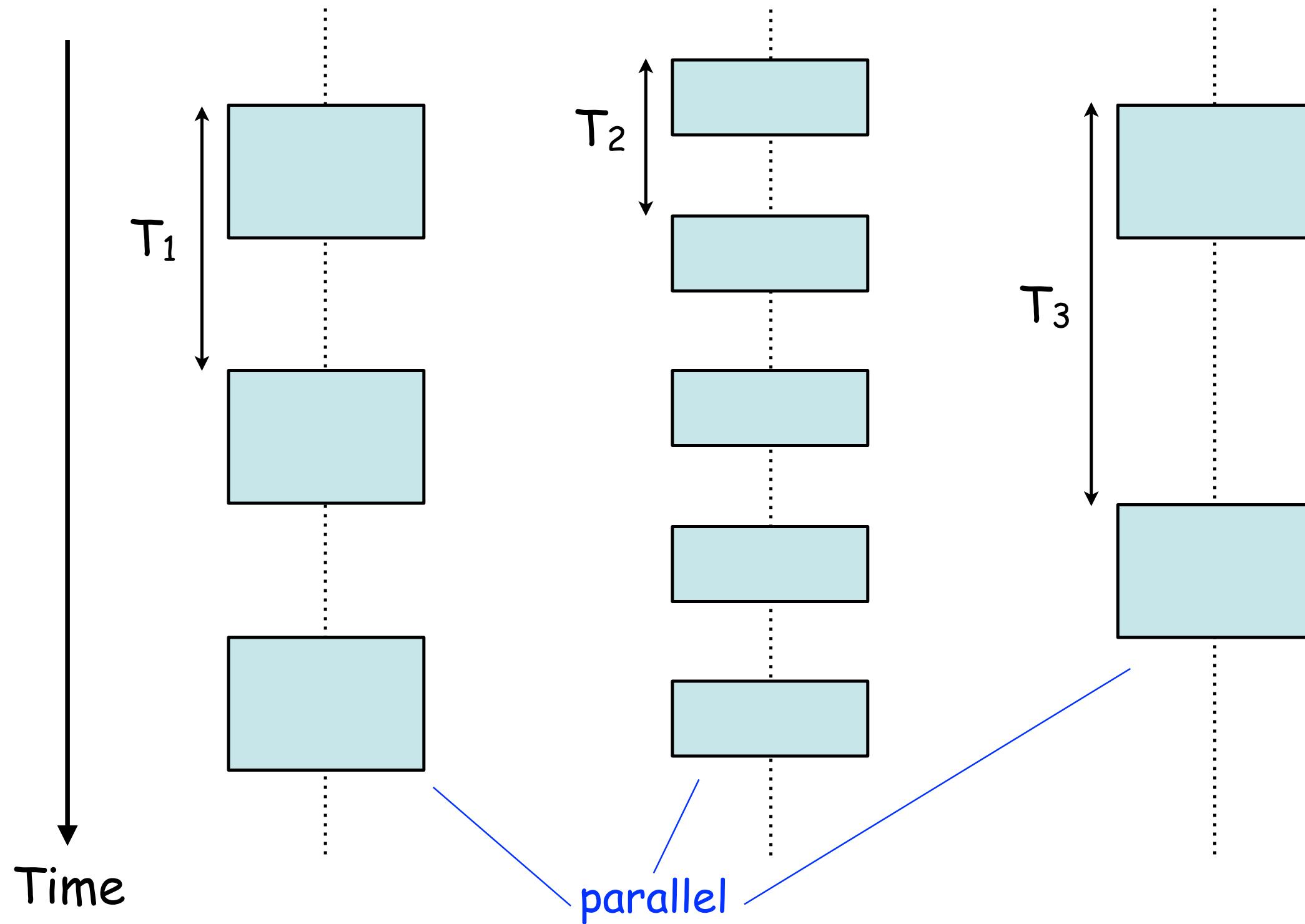
Dealing with evolving input

- Approach 1: New input is **read** from the environment at the initiative of the **program**
 - (As often as "possible"...)
 - (Or in an ad hoc fashion...)
 - Or at **well-defined times!**
- Approach 2: New input is **written** into the program at the initiative of the **environment**
 - (Just to be stored somewhere...)
 - Or guaranteed to trigger an associated **reaction!**

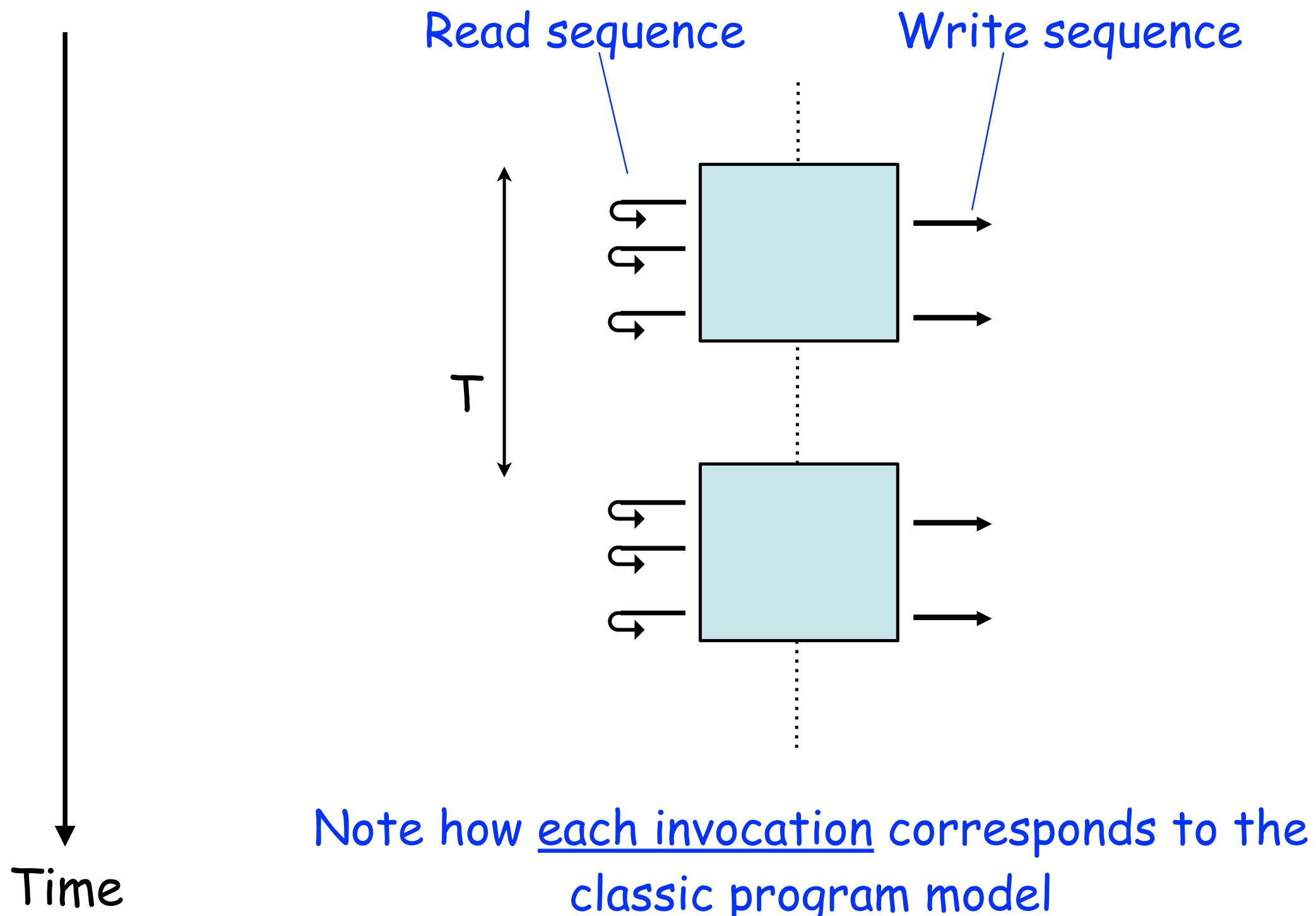
Approach 1: Time-triggered systems

- Idea: **read** input at **pre-defined times**, chosen to match the expected variations in input
- Obvious special case: read input every T time units (the periodic process)
- What happens between the computations? Nothing - the CPU can just shut down!
- How choose T ? Use Nyqvist's sampling theorem!
- What if there are multiple inputs?
 - Let the highest frequency input determine T ...
 - Or run **multiple periodic processes in parallel!**

Periodic time-triggered systems



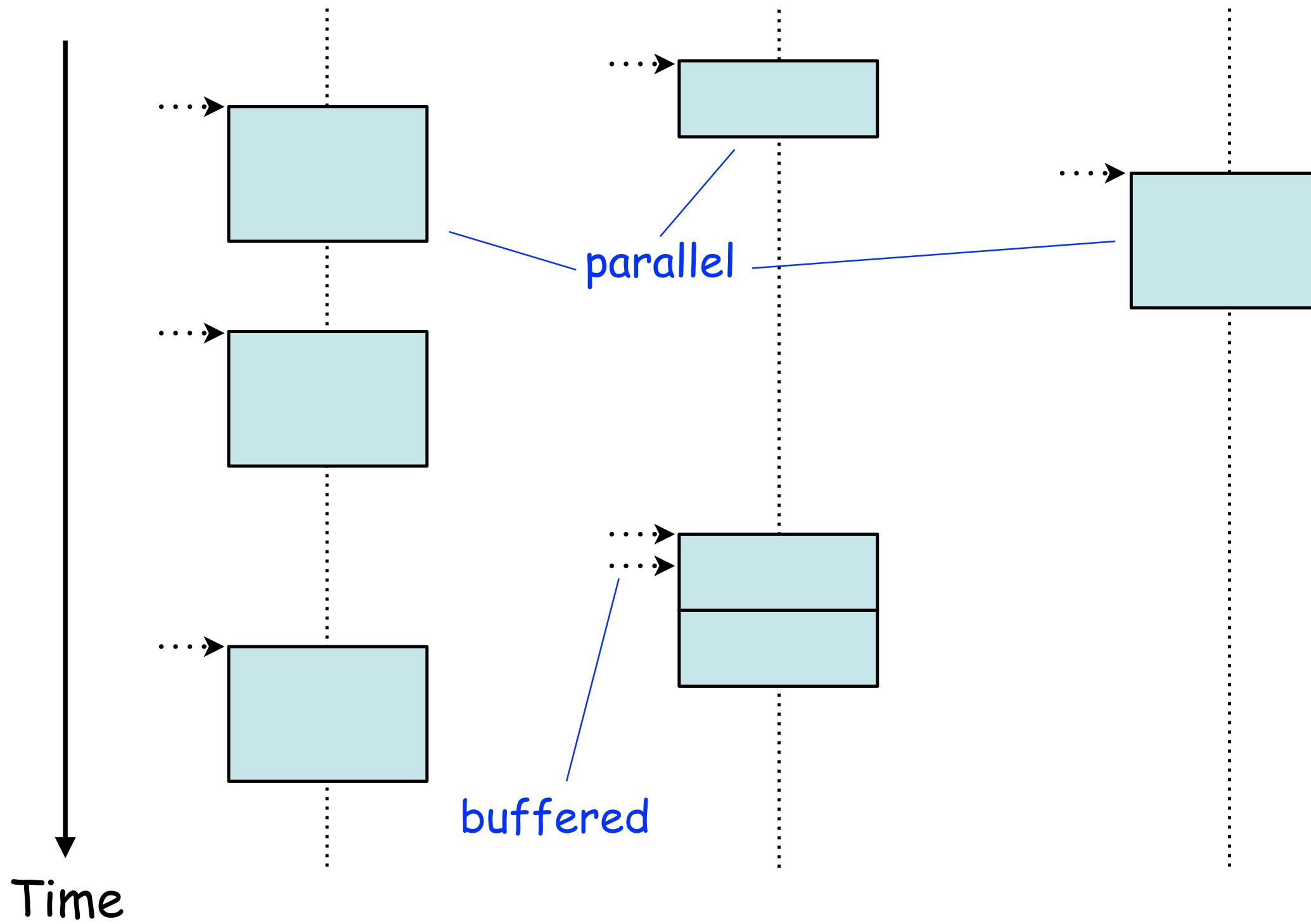
Adding input/output



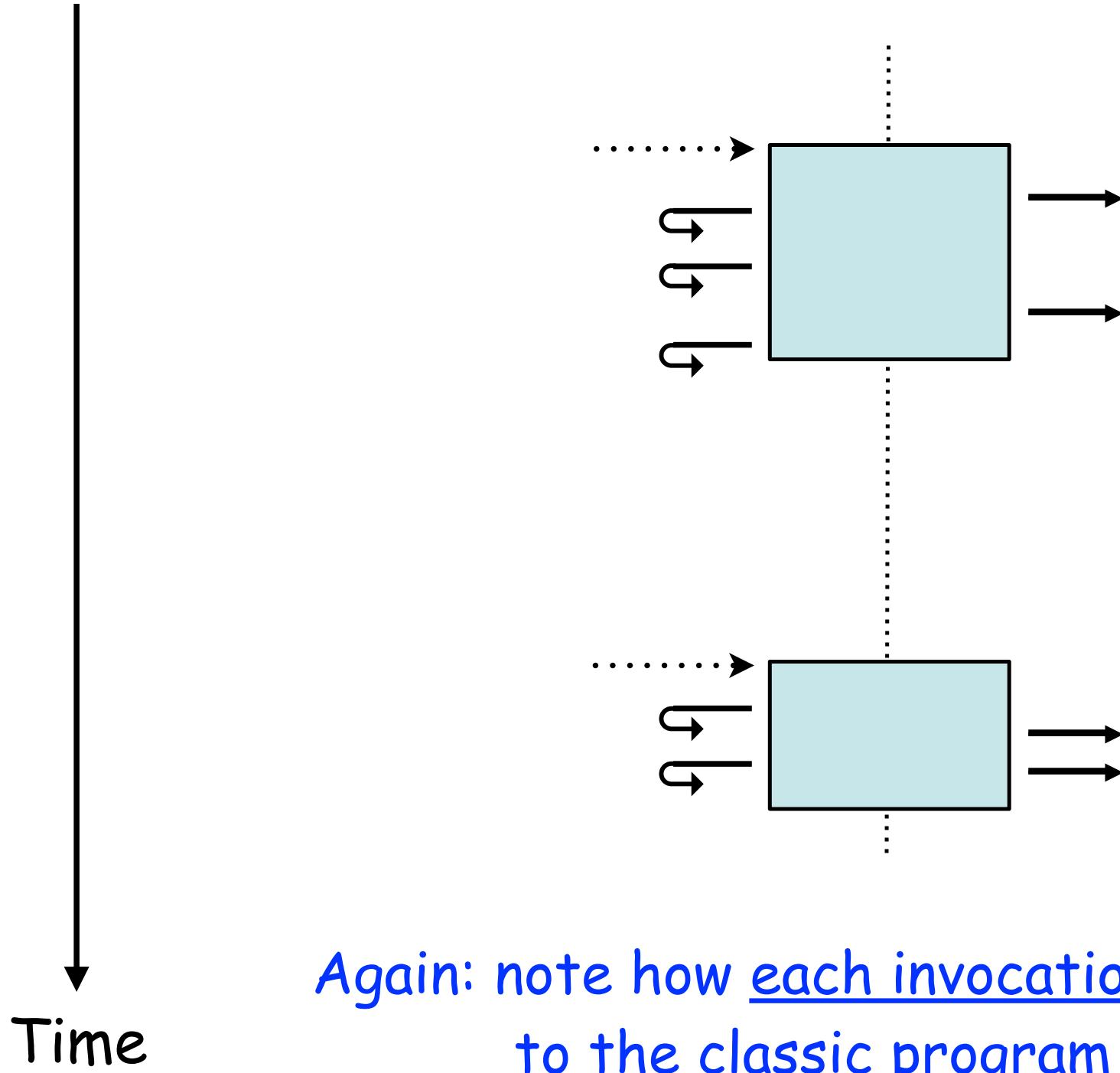
Approach 2: Event-triggered systems

- Idea: let the environment decide when input has changed enough to require some program action;
i.e., when an **event has occurred**
- Well-known concept on the computer hardware level: the **external interrupt!**
- What happens between the event processing phases?
Nothing - the CPU can just shut down!
- What if there are events with overlapping reactions?
 - Buffer up the events...
 - Or run **multiple event-handlers in parallel!**

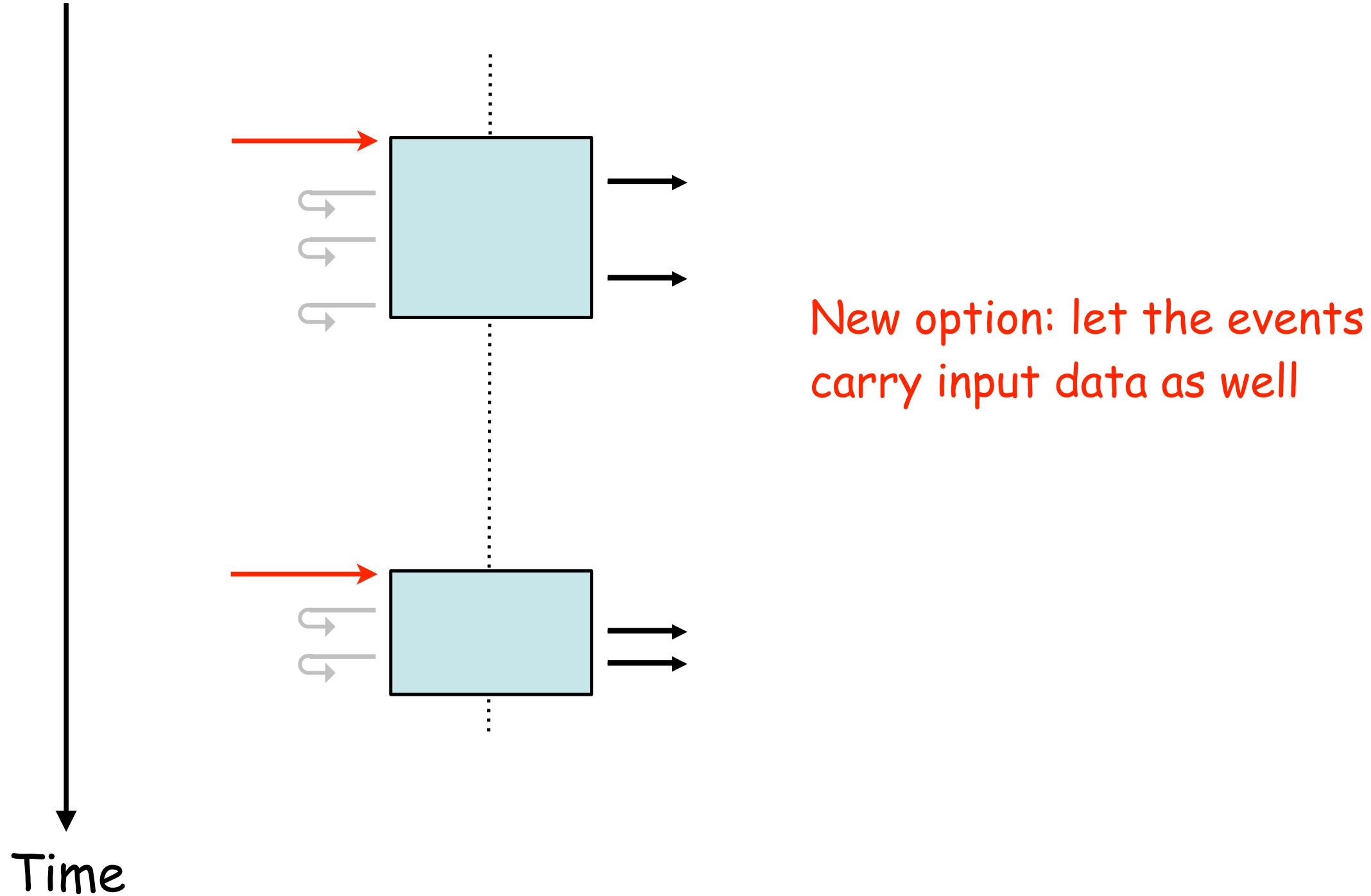
Event-triggered systems



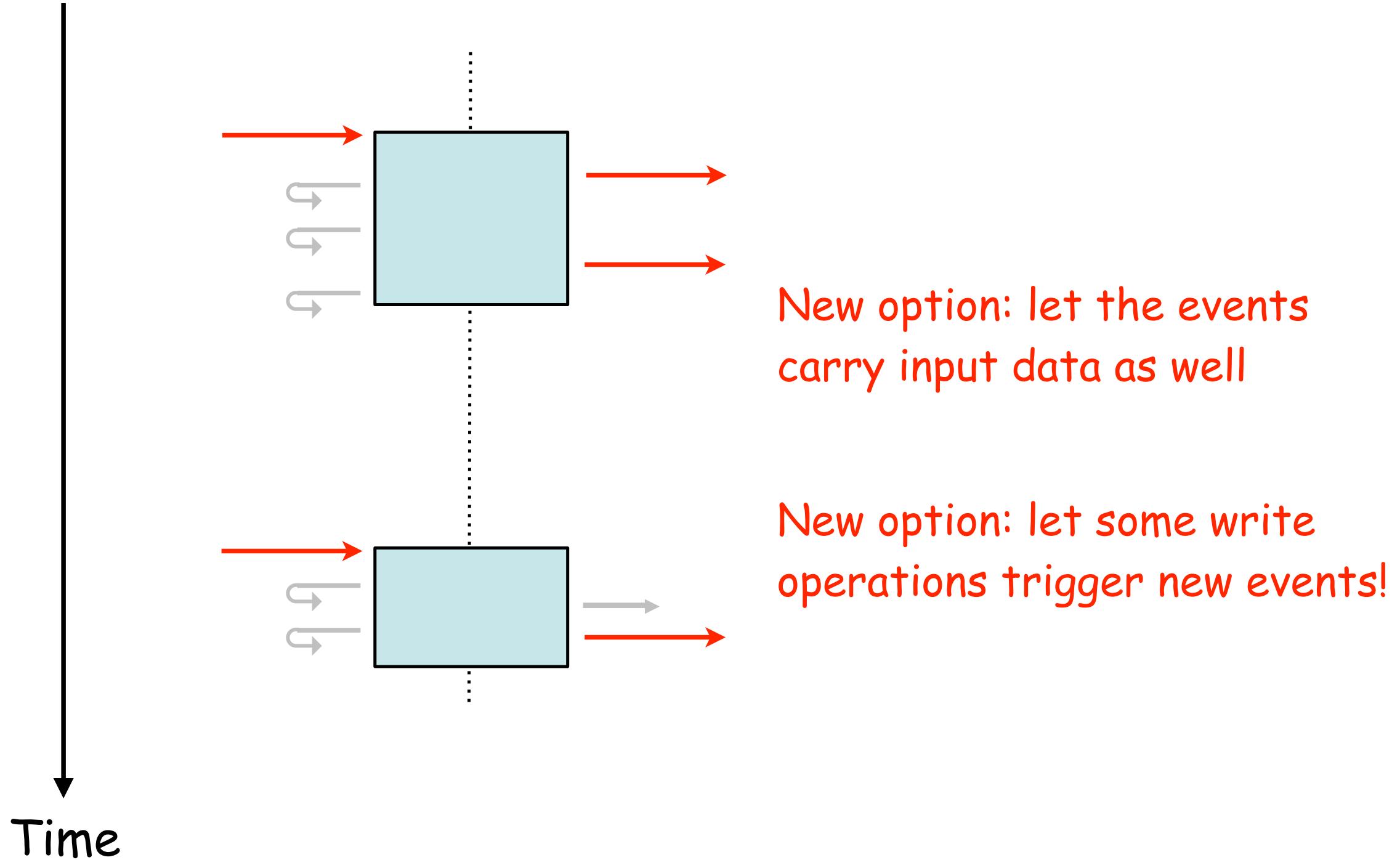
Adding input/output



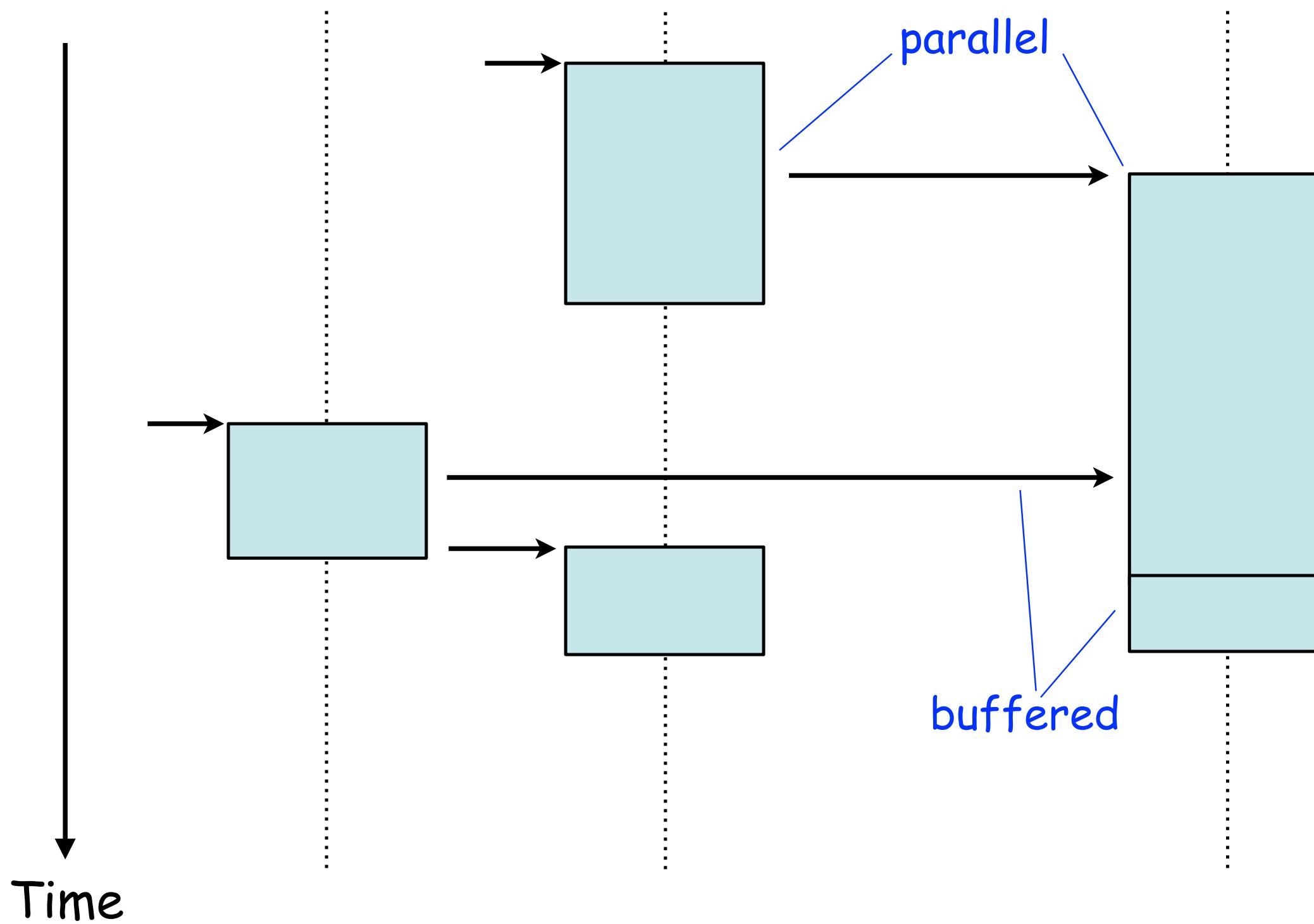
Adding input/output



Adding input/output



Chains of events



Time- vs. event-triggered systems

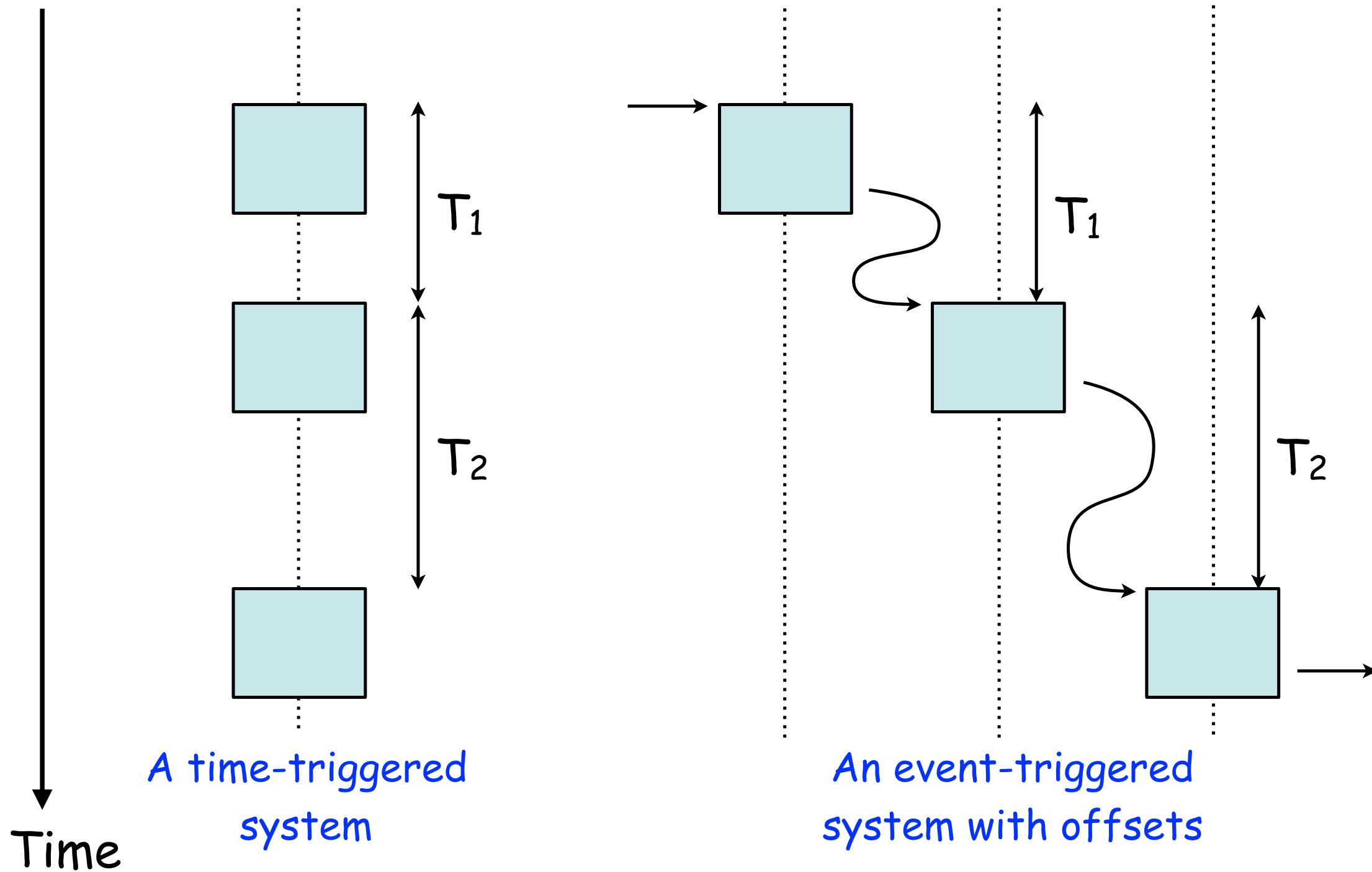
Time-triggered systems **observe** the environment and take action on basis of the changes they see.

Suitable when input may be **constantly changing** and all value are equally interesting, like in control systems

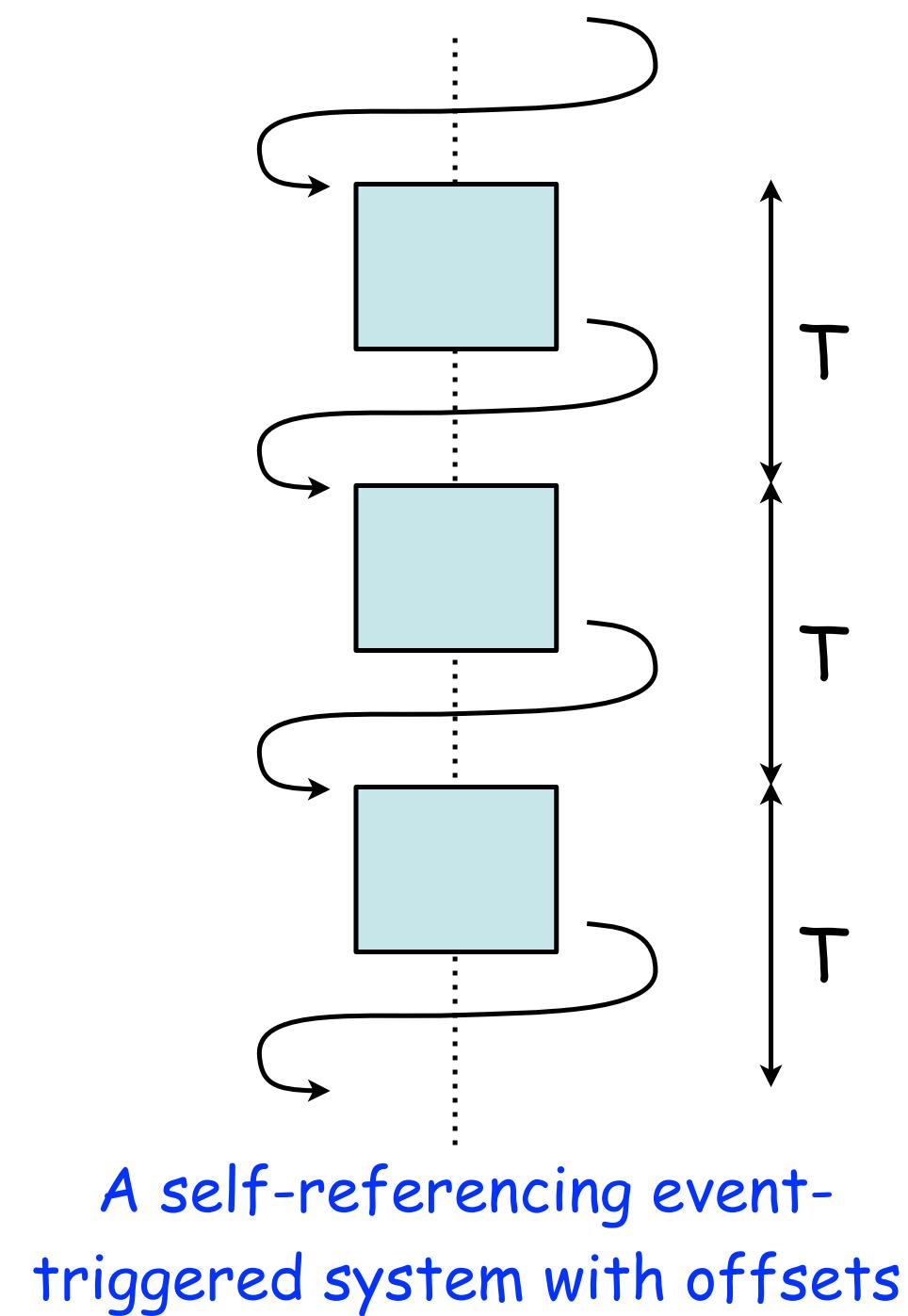
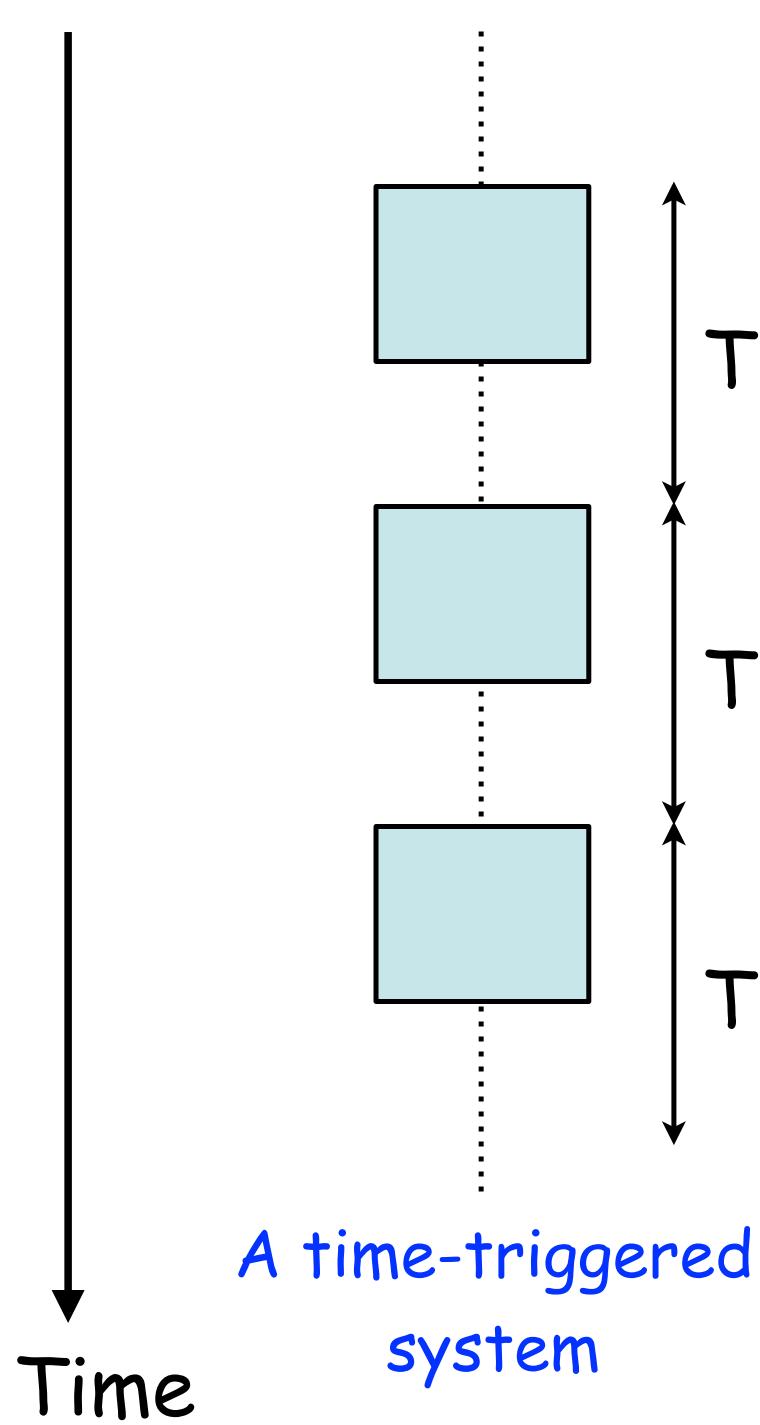
Event-triggered systems are **controlled by** the environment, and take action when the environment so decides.

Suitable when interesting input values are highly **irregular**, or when it is already **discrete**, like in communication systems

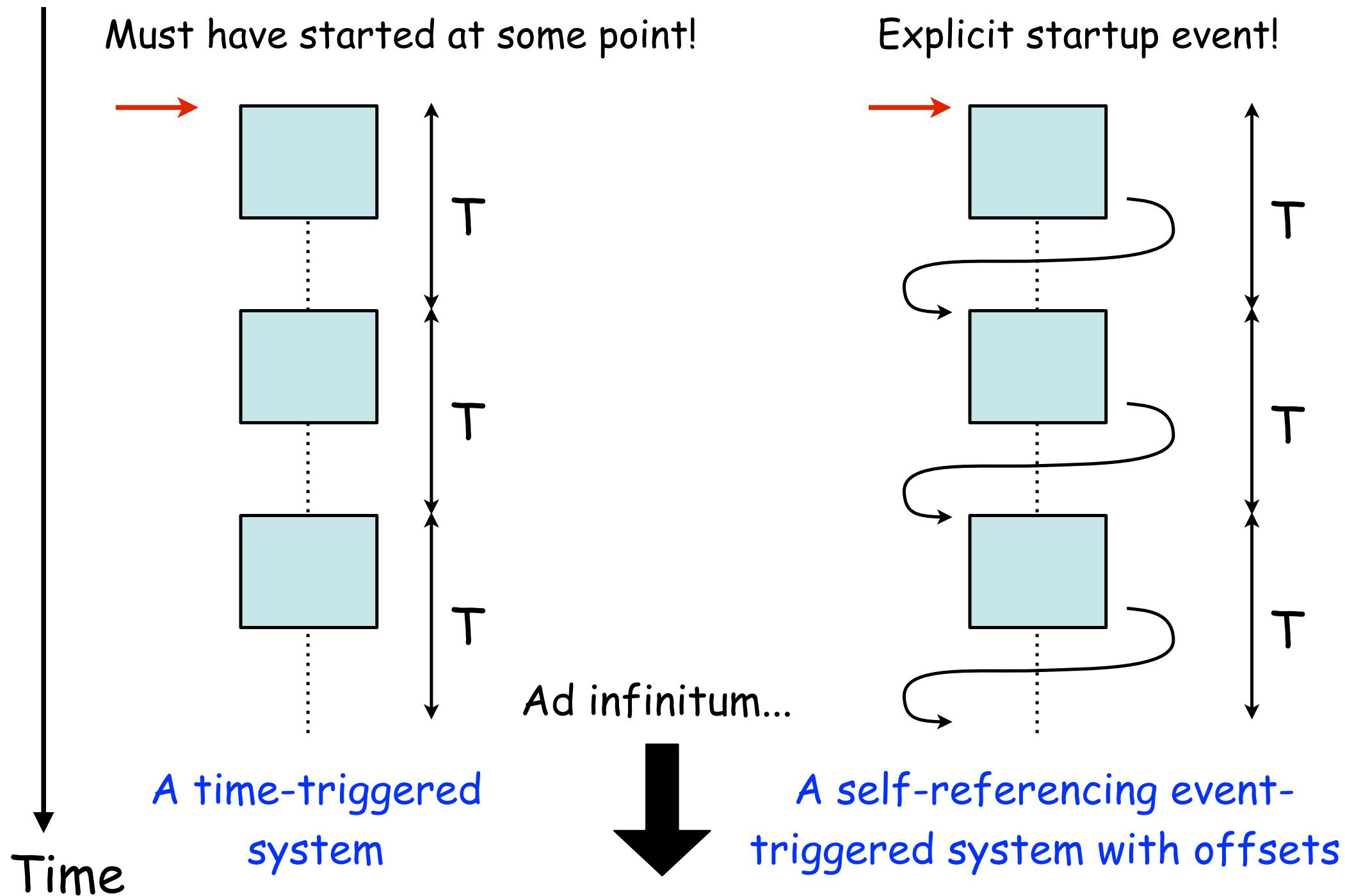
(1) If we allow events with offsets...



(2) If we allow self-referencing...

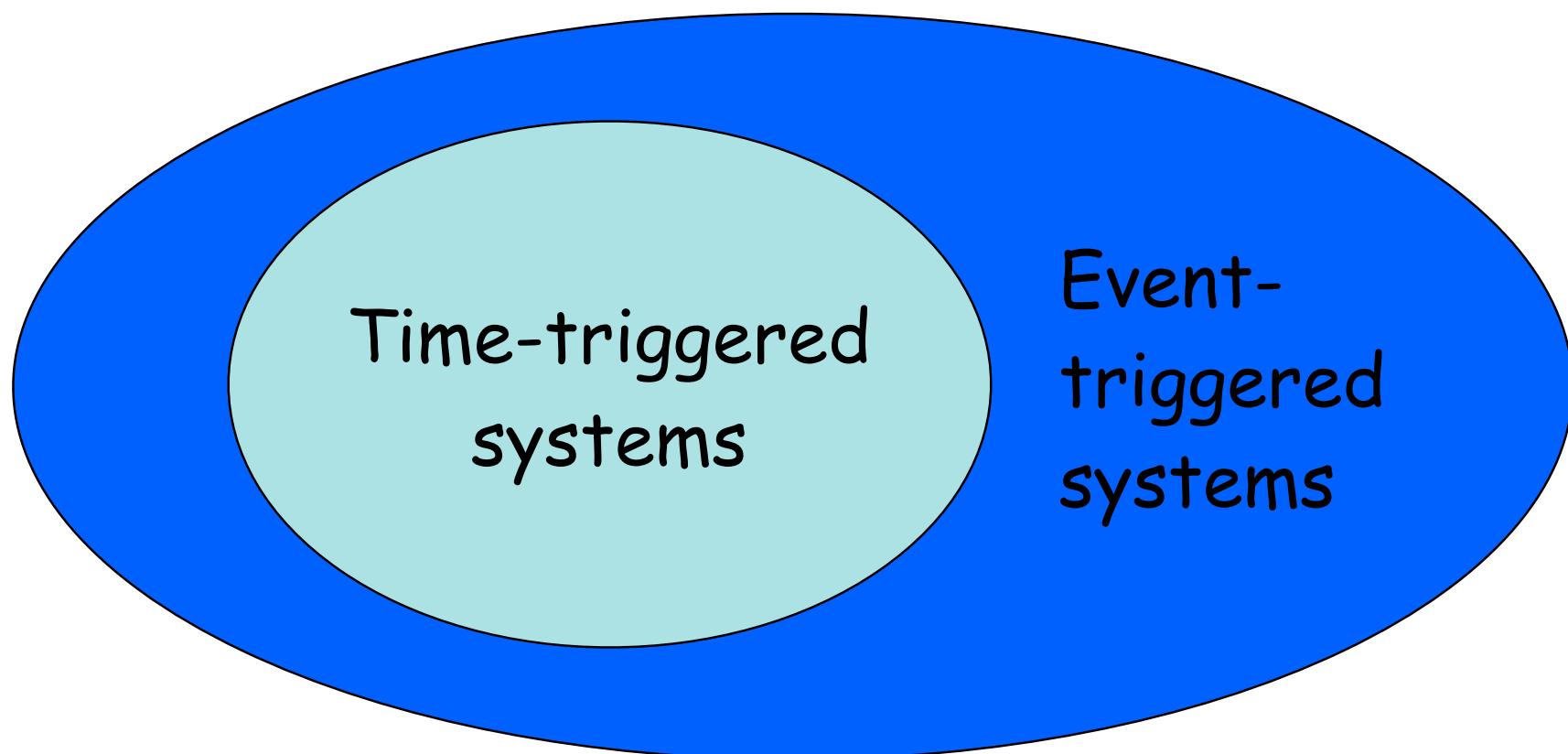


(3) If startup is made explicit...



Then:

Time-triggered behavior emerges as a special case of an event-triggered system!



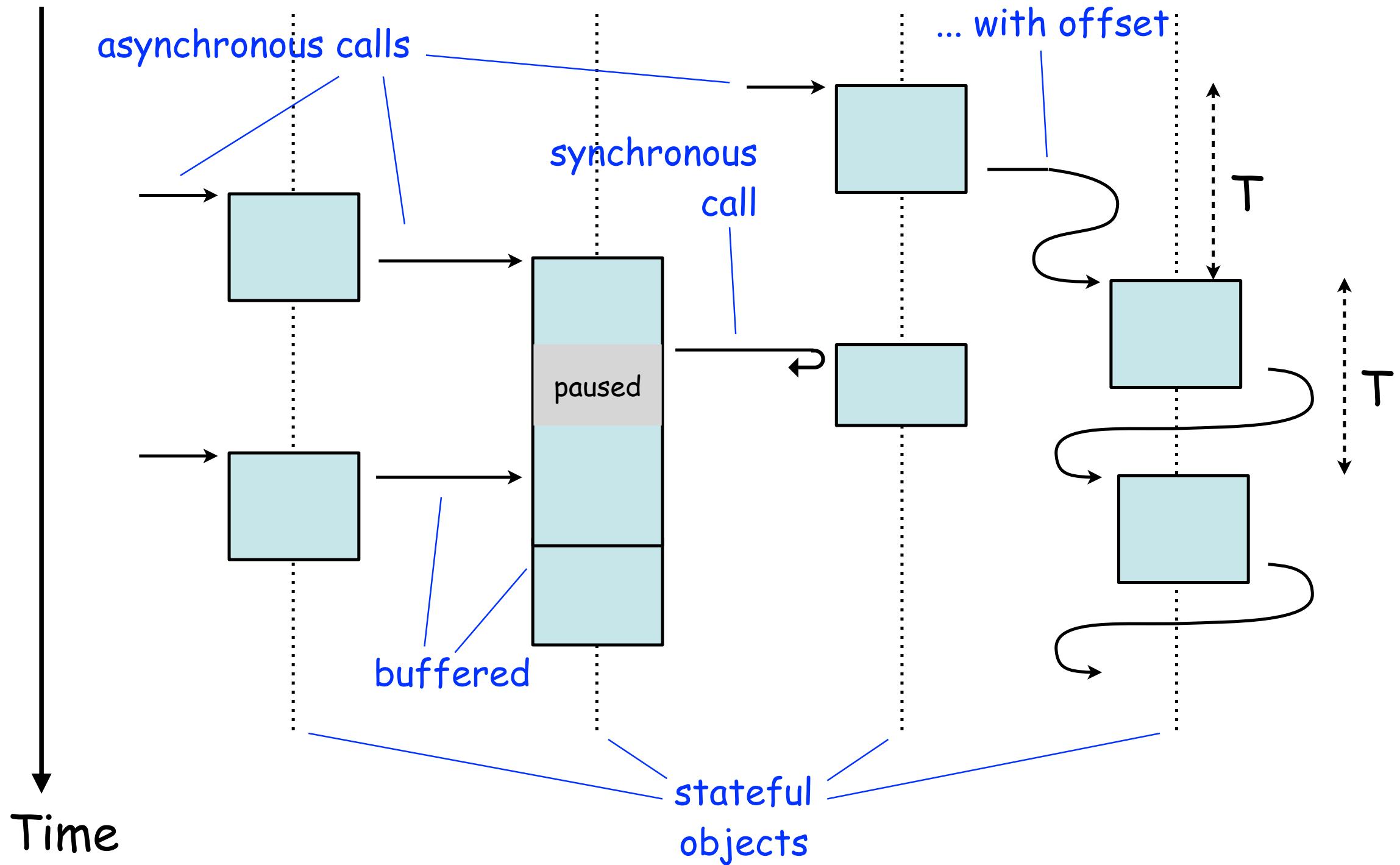
Time-triggerering as a special case

- A time-triggered behavior is just a **chain of event reactions**, separated by well-defined **time offsets**
- A **periodic** process is such a chain-reaction that **oscillates** (produces as many new events as those it reacts to)
- Many hybrid variants exist between the extremes of one single reaction and the oscillating periodic behavior
- Allows us to seamlessly study trade-offs between the basic approaches
- Note: not the commonly taught real-time systems view!
- It is however the view we find in **TinyTimber!**

TinyTimber

- A run-time kernel + a design style for programming embedded real-time systems in C
- Also a cut-down variant of the programming language **Timber** (timber-lang.org)
- Basic ideas:
 - Events can be triggered with time offsets
 - Events = **asynchronous method calls**
 - Methods belong to **objects**
 - Objects = protected sets of state variables
 - Also: **synchronous method calls** (mimic read/write)

A TinyTimber run-time scenario



In concrete C

```
typedef struct {  
    Object super;  
    int value;  
    int enabled;  
} Counter;
```

State layout

Method definitions

```
Counter cA = initCounter(1);  
Counter cB = initCounter(0);
```

Constructor definition

```
#define initCounter(en) { initObject(), 0, en }
```

```
int inc( Counter *self, int arg ) {  
    if (self->enabled)  
        self->value = self->value + arg;  
    return self->value;  
}  
int enable( Counter *self, int arg ) {  
    self->enabled = arg;  
    return 0;  
}
```

Creating global instances

Calling methods

```
... ASYNC( &cA, inc, 1 ); ...
```

— Asynchronous call

```
... int r = SYNC( &cA, inc, 0 ); ...
```

— Synchronous call

```
... AFTER( SEC(2), &cB, enable, 1 ); ...
```

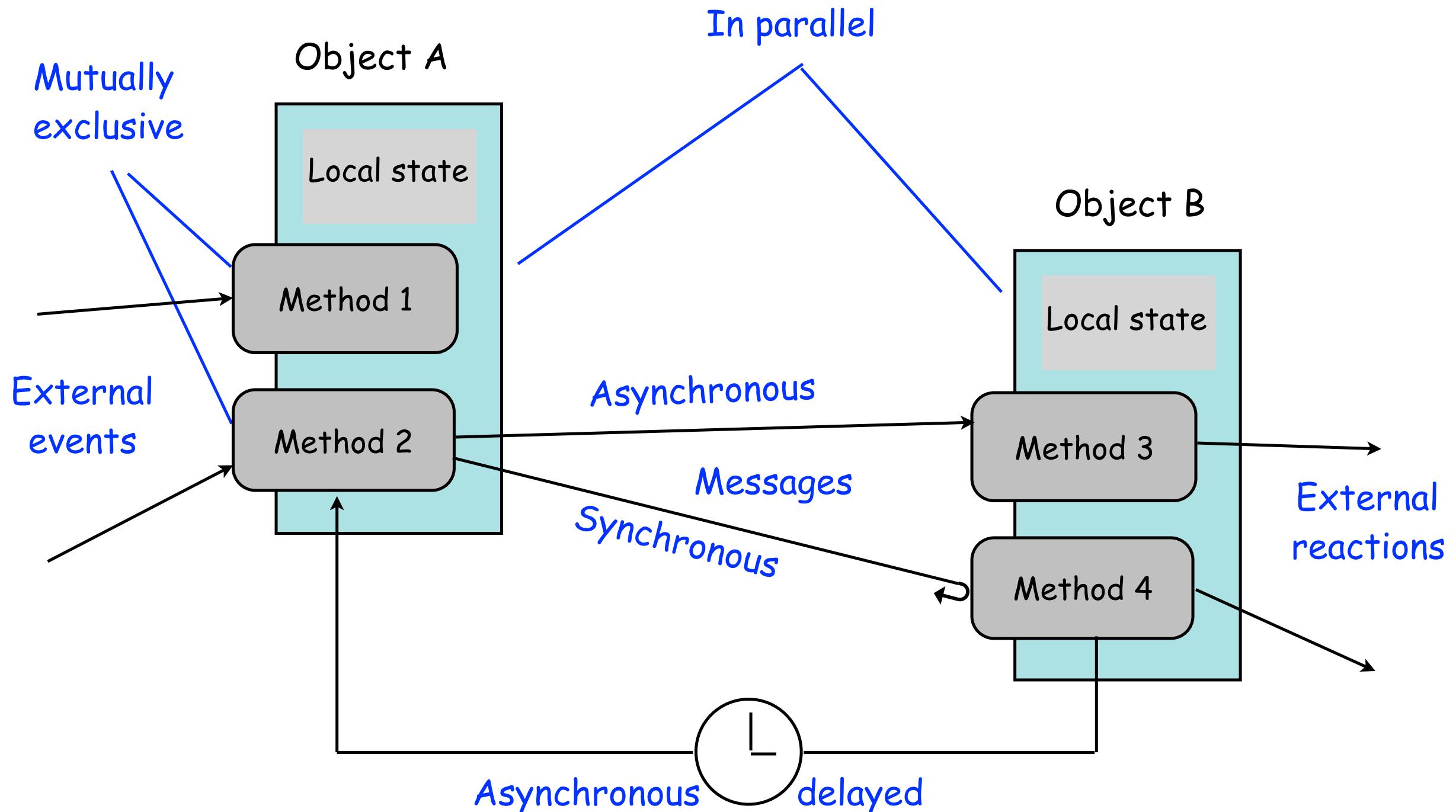
— Asynchronous call with offset

Top-level application setup

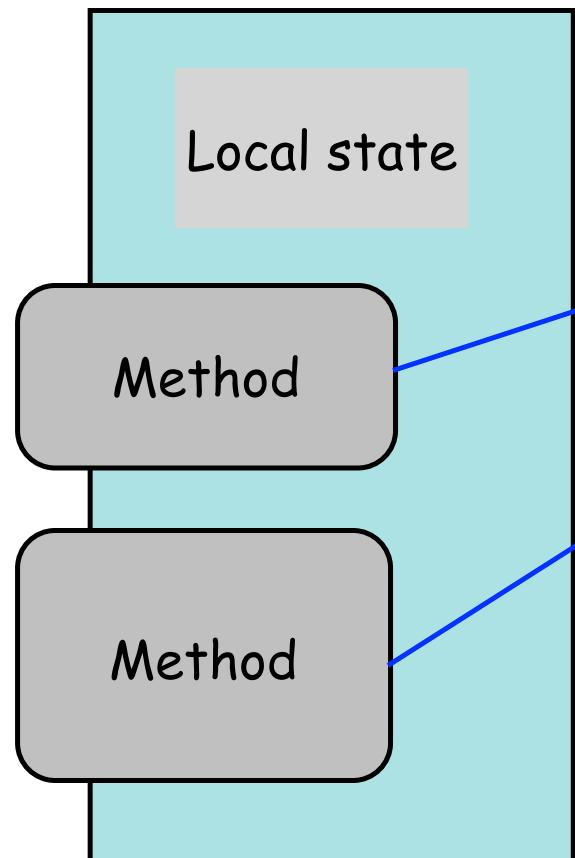
```
MyApplication app = initMyApplication();
```

```
int main() {
    INSTALL( &app, compute, IRQ1 );
    INSTALL( &cB, inc, IRQ2 );
    return TINYTIMBER( &app, reset, 0 );
}
```

Run-time execution model



Methods



Finite sequences that

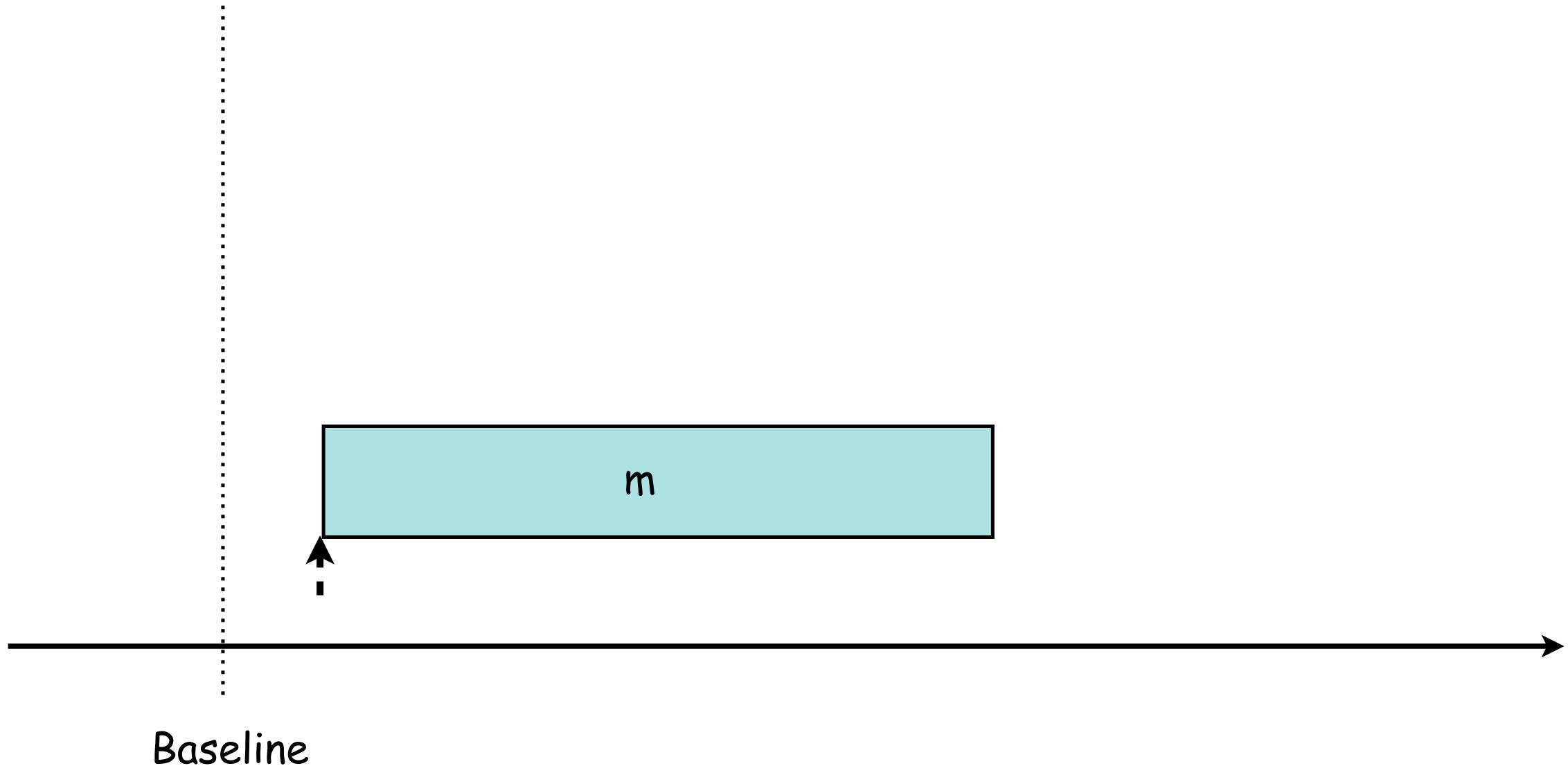
- Read and write local state
- Call other methods
- Perform local computations

No indefinitely blocking operations,
no infinite loops:

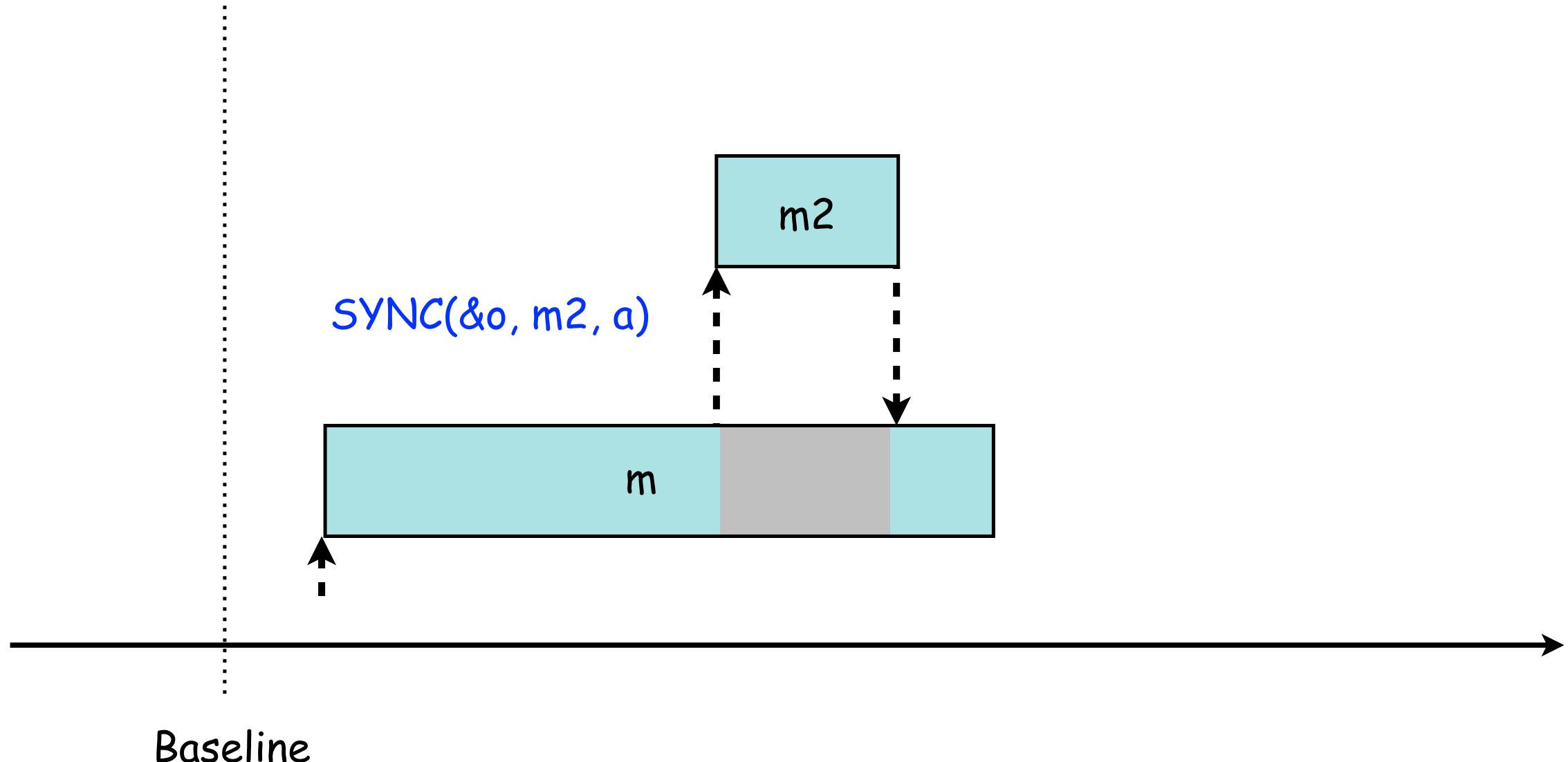
objects sleep between temporary activity

The classical OO intuition recast to a concurrent setting!

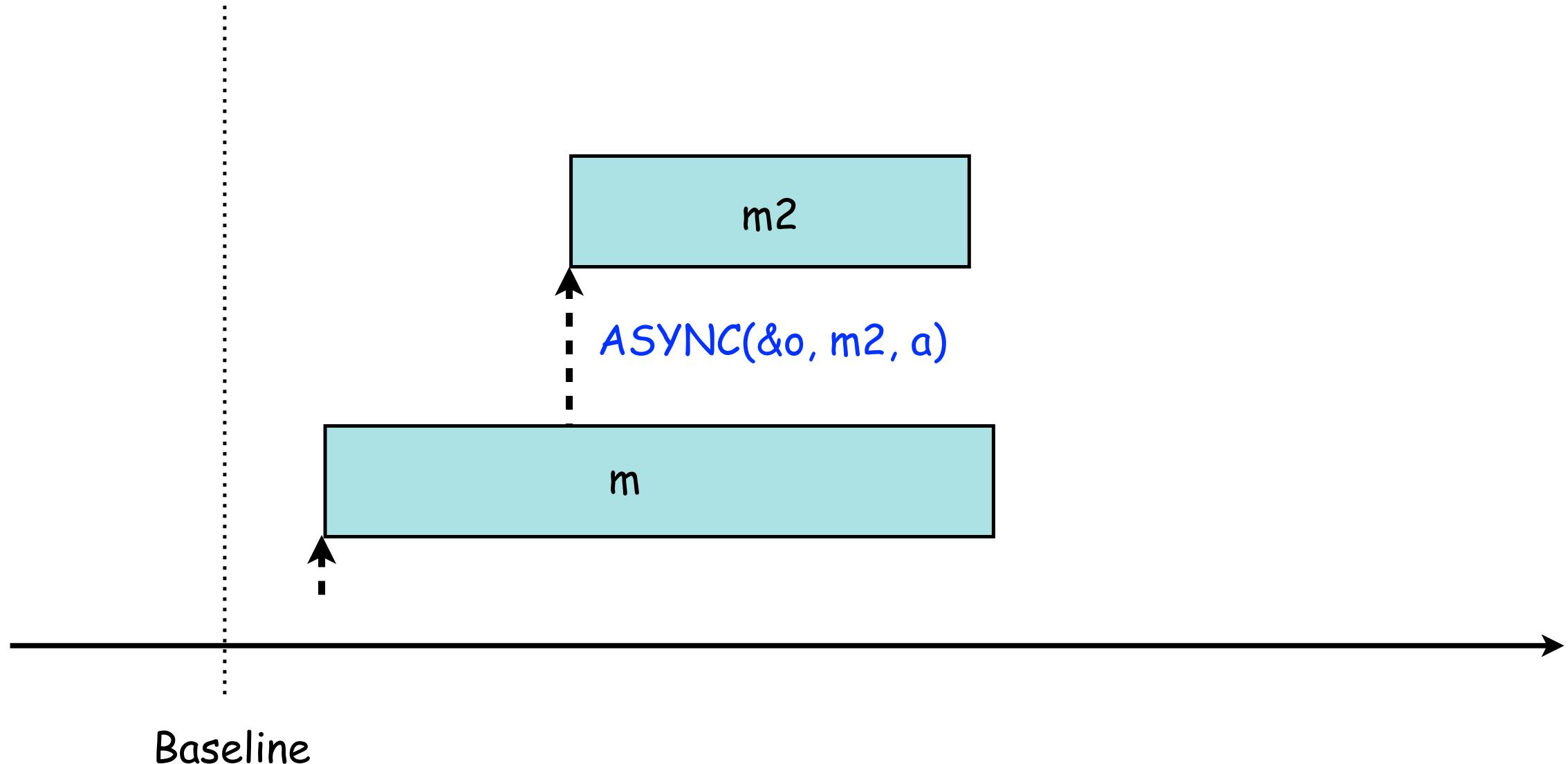
Timing reference



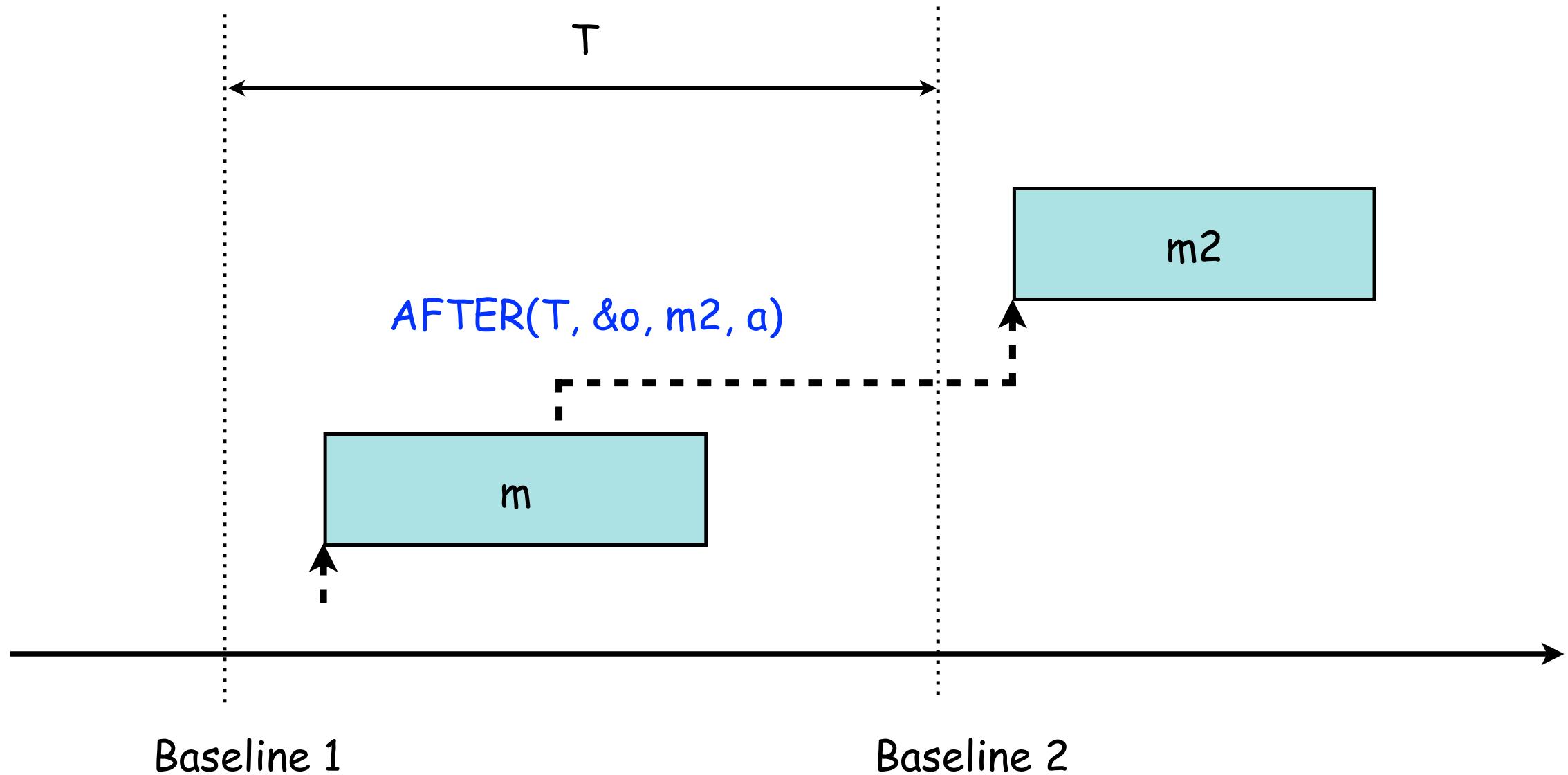
Timing reference



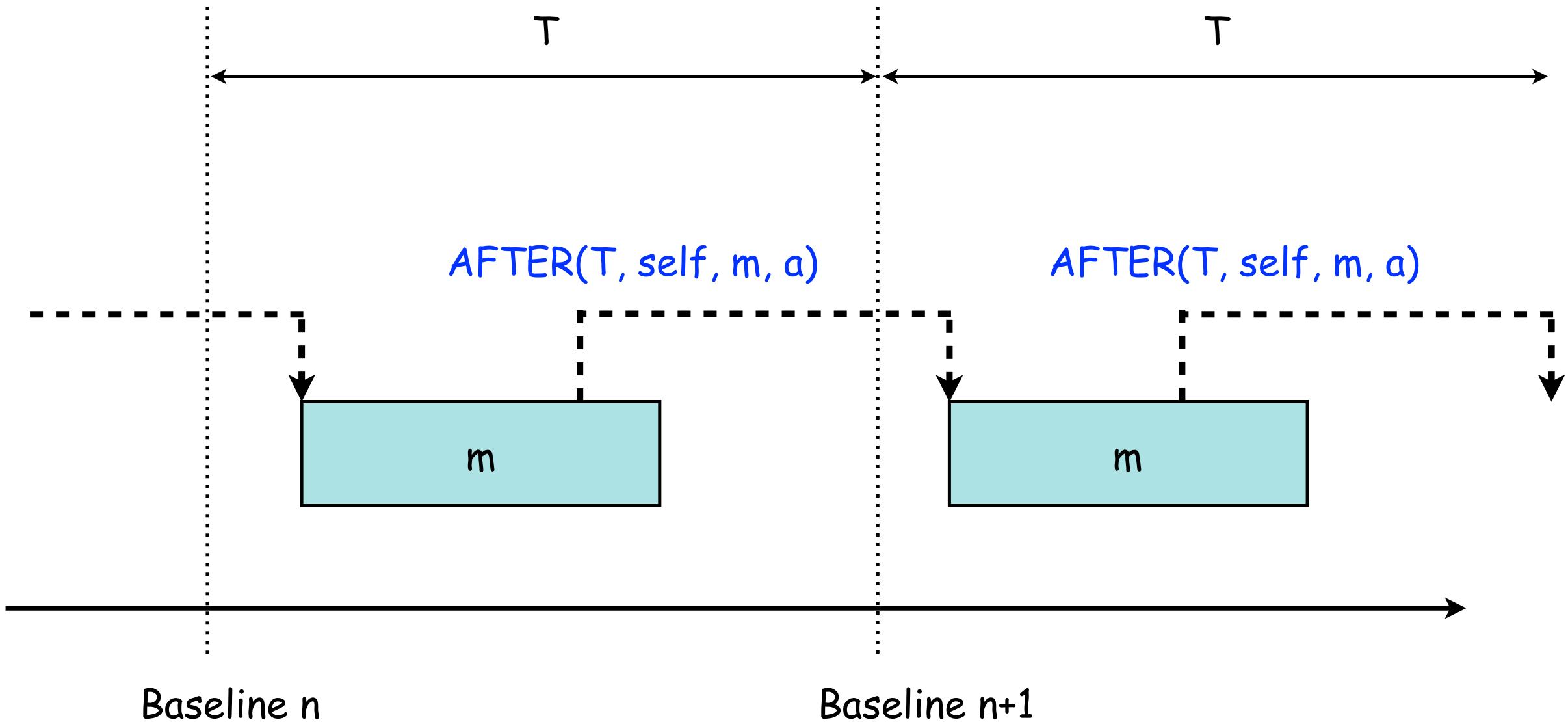
Timing reference



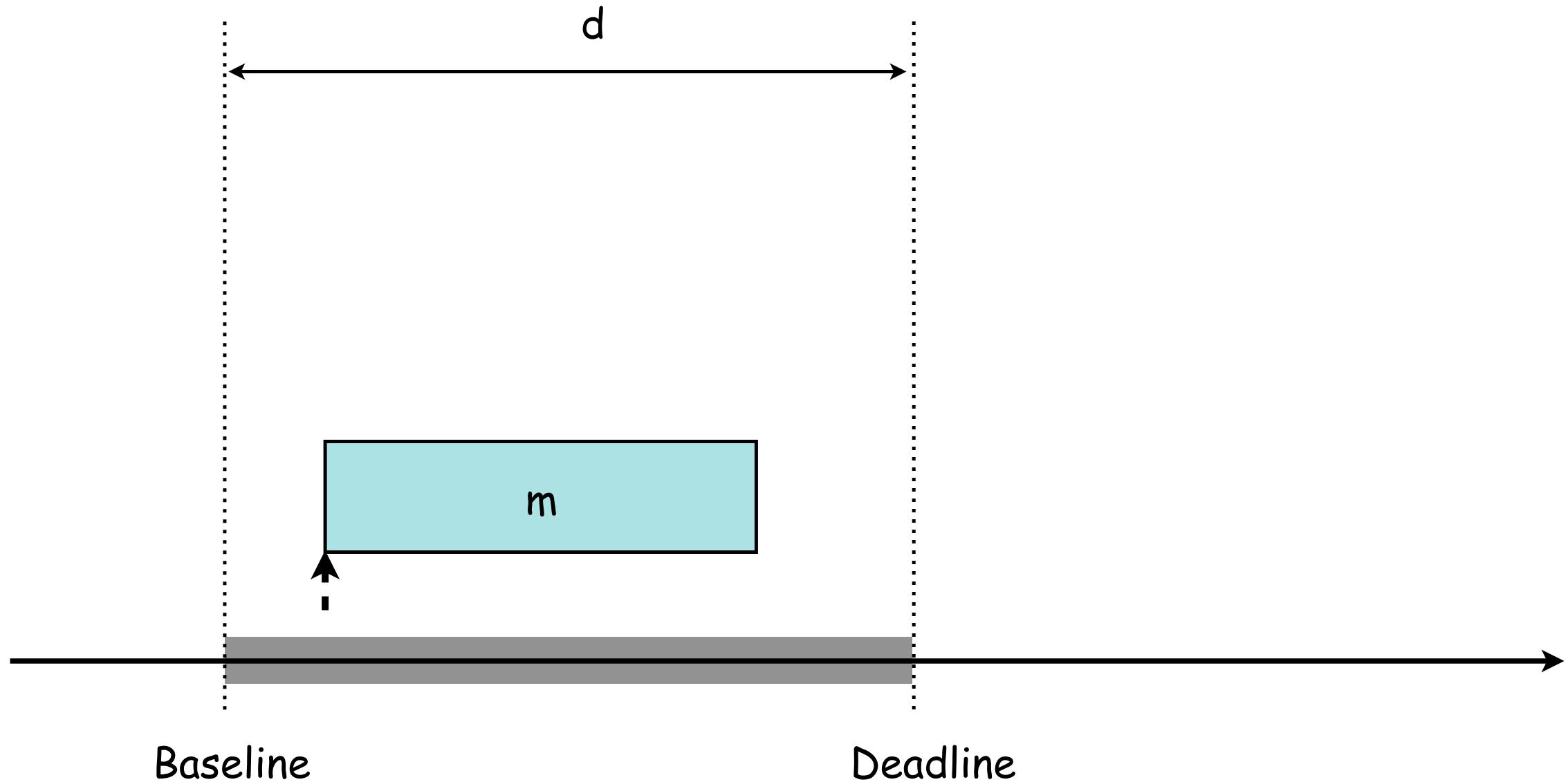
Baseline move



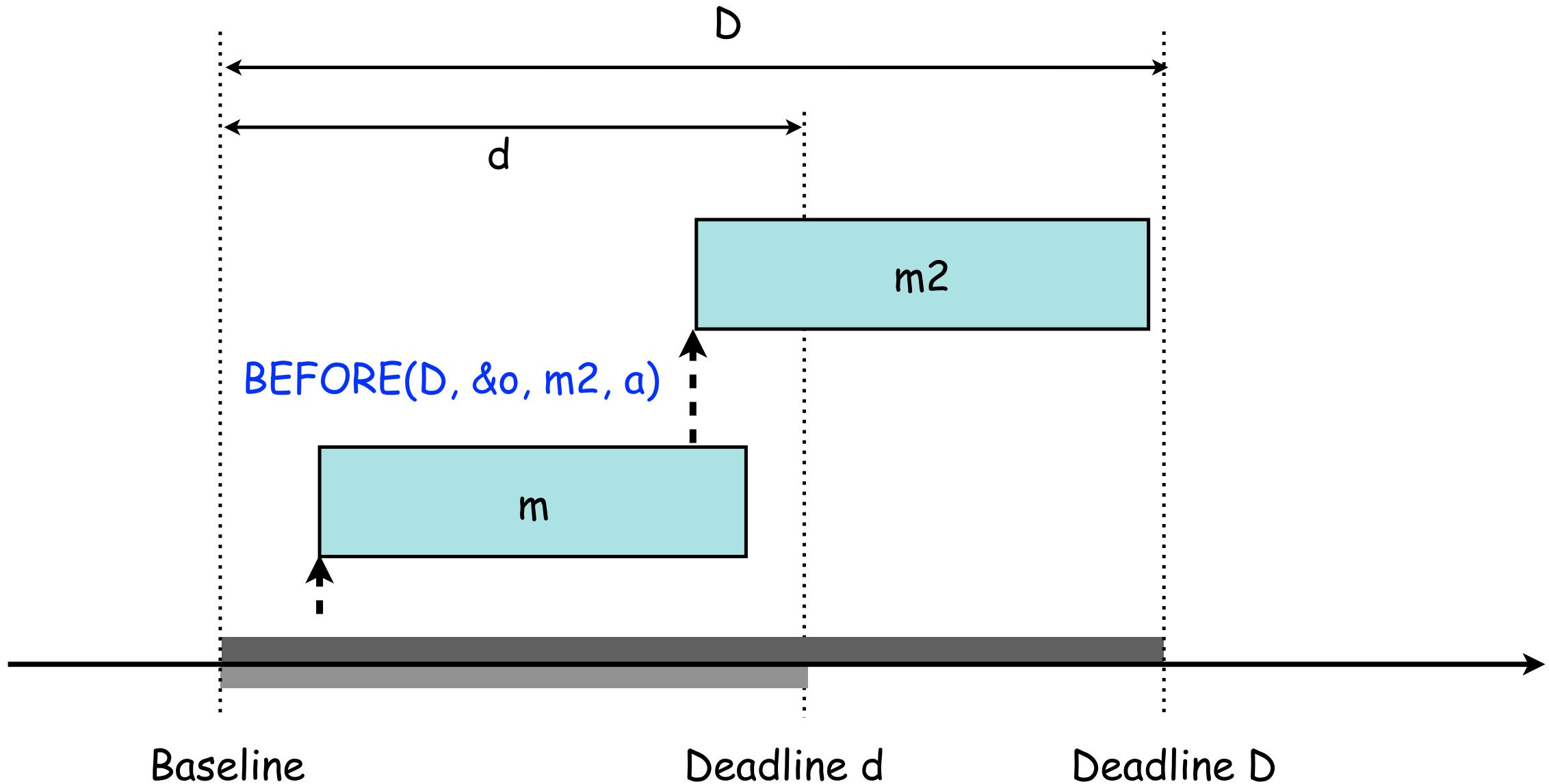
Periodicity



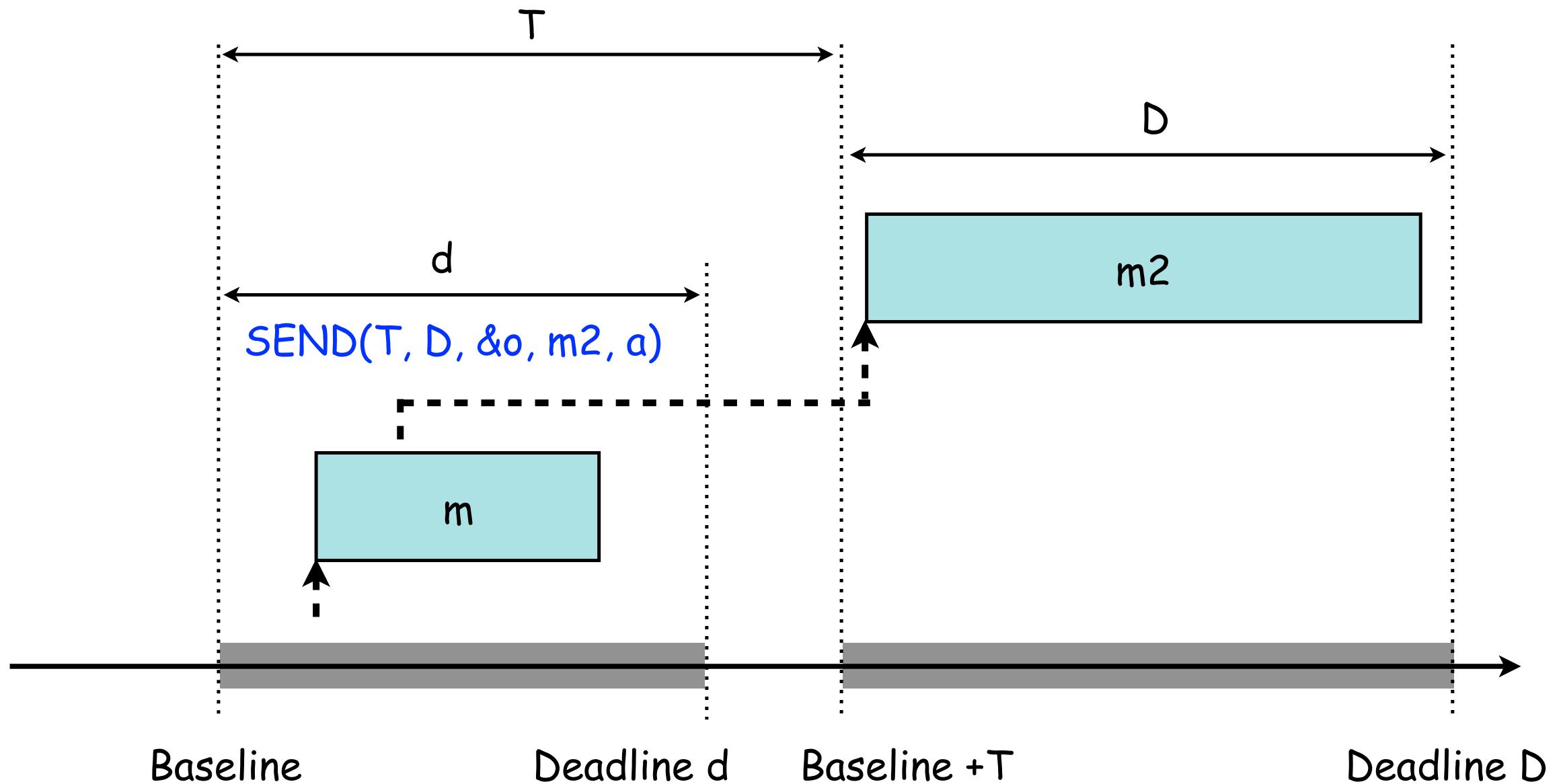
Timing windows



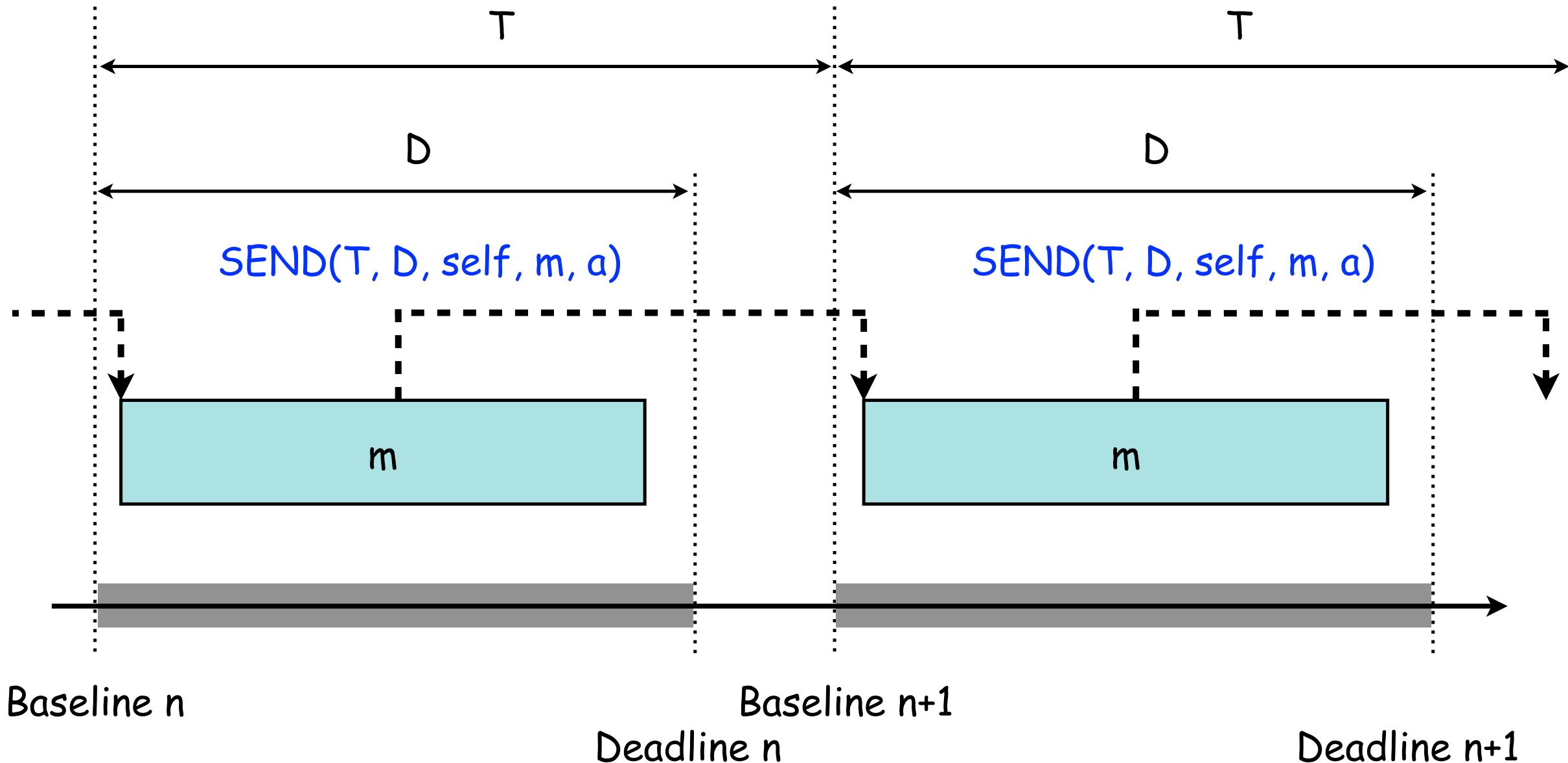
Window resize



Window move



Constrained periodicity



A clock

```
typedef struct {
    Object super;
    int sec, min, hour;
} Clock;

#define initClock() { initObject(), 0, 0, 0 }

int tick( Clock *self, int arg ){
    self->sec++;
    if (self->sec == 60) { self->sec = 0; self->min++; }
    if (self->min == 60) { self->min = 0; self->hour++; }
    AFTER( SEC(1), self, tick, 0 )
}

int sample( Clock *self, CalendarTime *arg ) {
    arg->sec = self->sec; arg->min = self->min; arg->hour = self->hour;
}
```

typedef struct {
 int sec, min, hour;
} CalendarTime;

Use pointer to
circumvent one-arg-only
restriction.
(Only safe with SYNC)

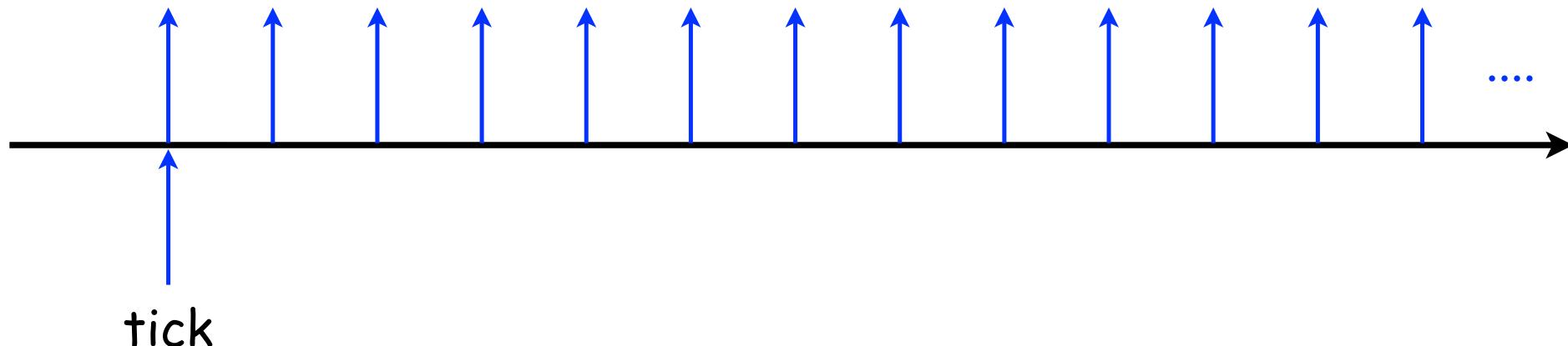
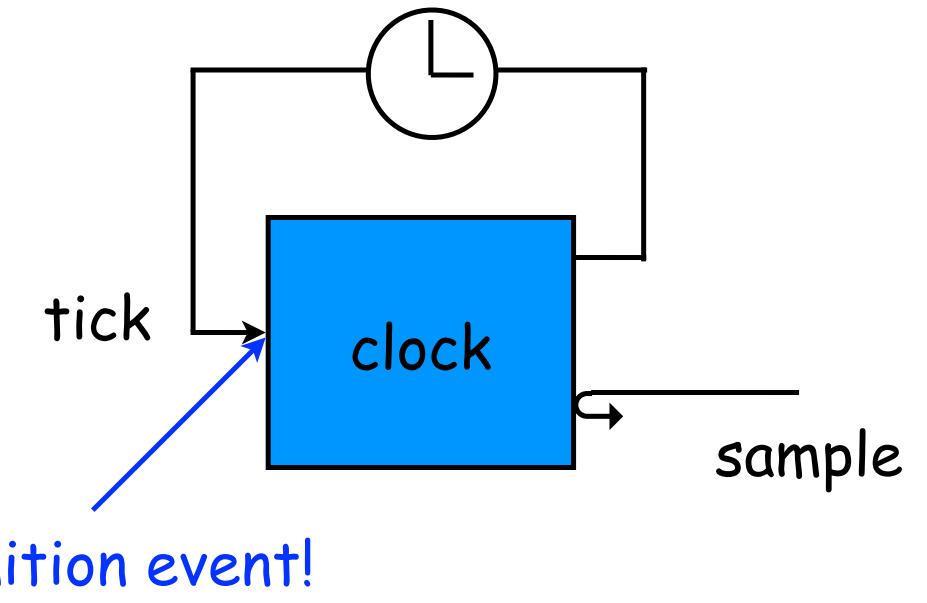
A clock

`Clock clock = initClock();`

Q: Will the clock
start oscillating
by itself?

A: No...

`TINYTIMER(&clock, tick, 0);`



An on-off clock

```
typedef struct {  
    Object super;  
    int sec;  
    int enabled;  
} OnOffClock;
```

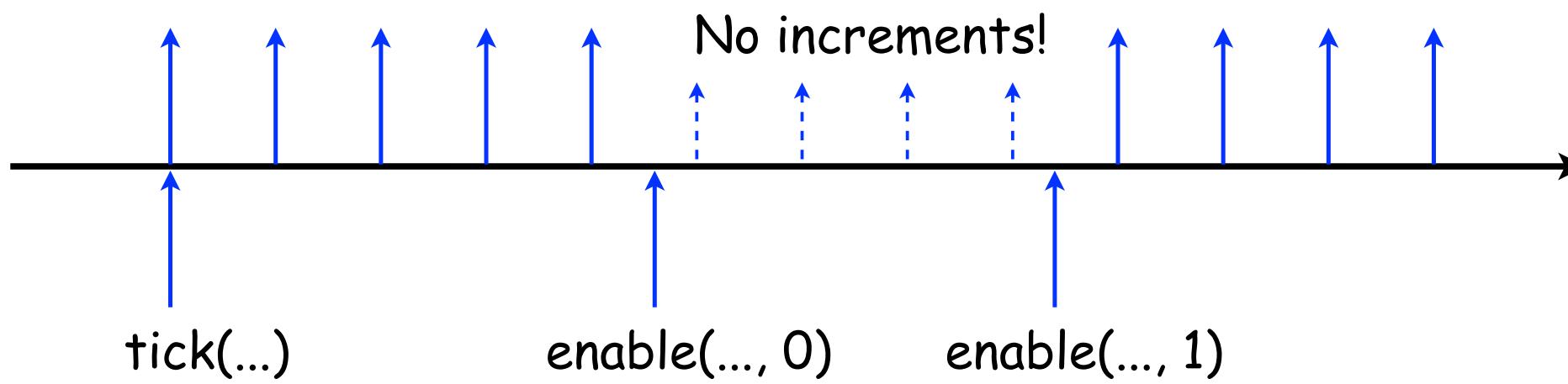
```
#define initOnOffClock() { initObject(), 0, 1 }
```

```
int tick( OnOffClock *self, int arg ) {  
    if (self->enabled)  
        self->sec = self->sec + 1;  
    AFTER( SEC(1), self, tick, 0 )  
}
```

```
int sample( OnOffClock *self, int arg ) { return self->sec; }
```

```
int enable( OnOffClock *self, int en ) { self->enabled = en; }
```

An on-off clock



A different on-off clock

```
typedef struct {
    Object super;
    int sec, enabled;
} OnOffClock2;

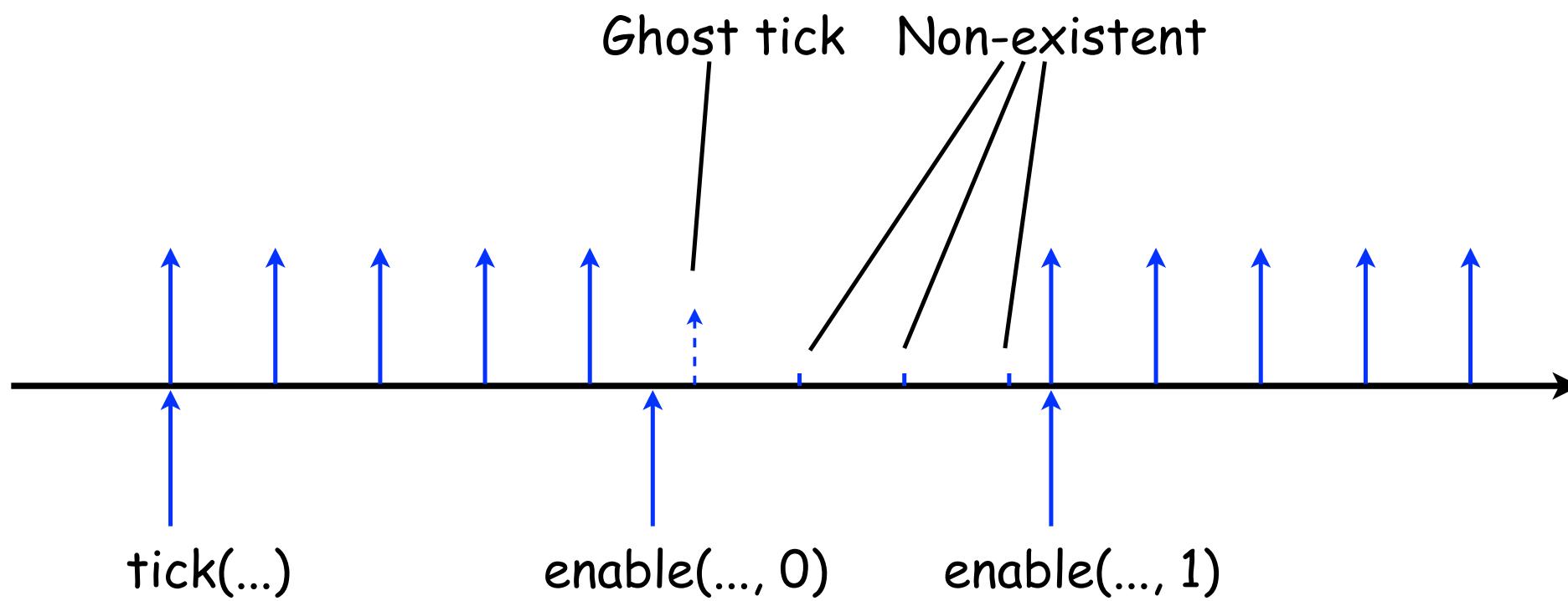
#define initOnOffClock2() { initObject(), 0, 1 }

int tick( OnOffClock2 *self, int arg ) {
    if (self->enabled) {
        self->val = self->val + 1;
        AFTER( SEC(1), self, tick, 0 )
    }
}

int sample( OnOffClock2 *self, int arg ) { return self->sec; }

int enable( OnOffClock2 *self, int en ) {
    if (en && !self->enabled) ASYNC( self, tick, 0 );
    self->enabled = en;
}
```

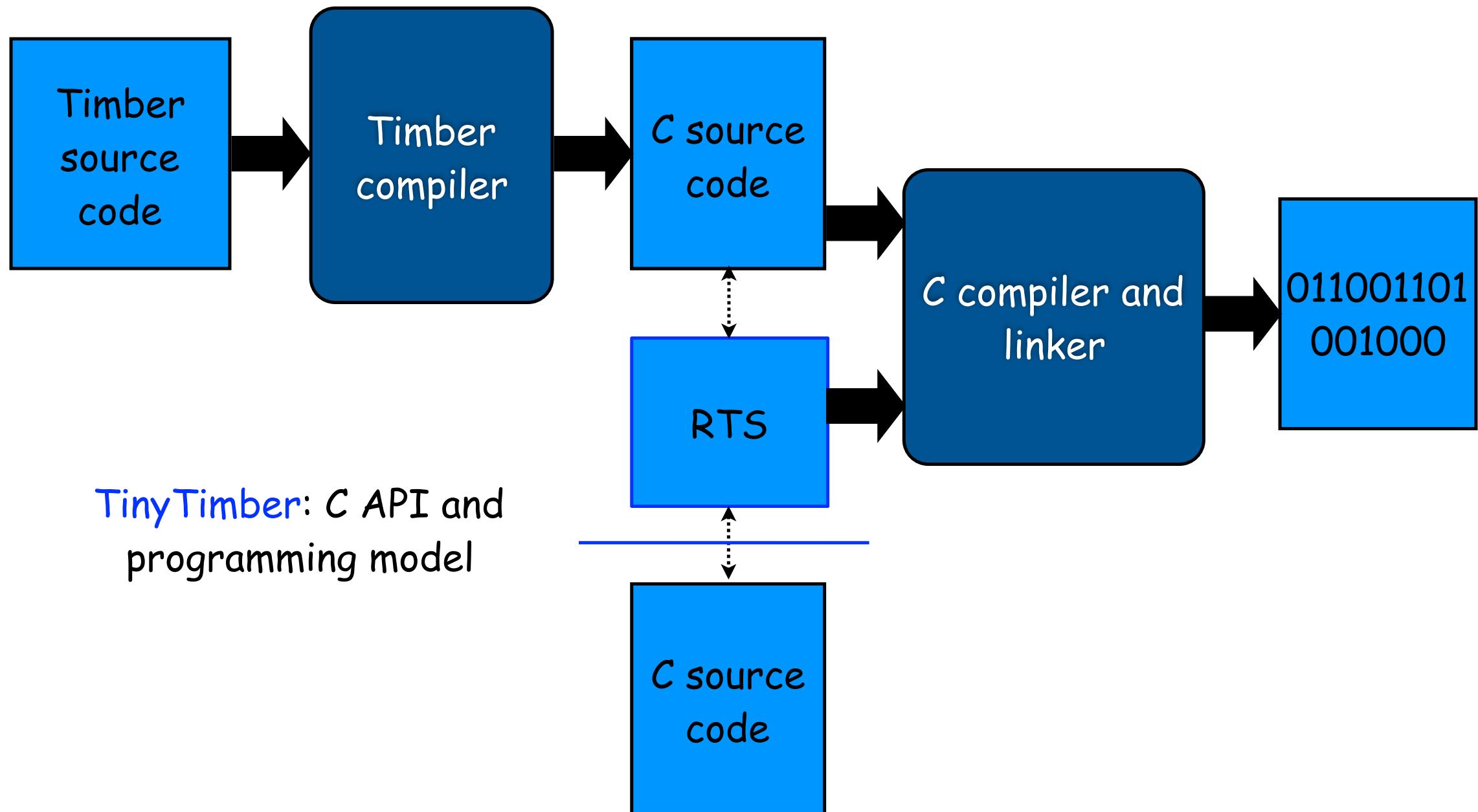
A different on-off clock



Timber

- The big brother of TinyTimber
- Full-featured language:
 - Higher-order & strongly typed
 - Dynamic object creation, garbage-collected heap
 - Haskell-like syntax (but no laziness!)
 - Purely functional computation sub-language
- A real-time successor to O'Haskell (an OO Haskell ext.)
- Developed in part by groups & individuals at Chalmers, Luleå U. of T., Oregon Grad. Inst., Kansas State U.

Compiling Timber



Wrapping up

TinyTimber offers:

- Lightweight real-time facilities for C
- Implicit concurrency
- Implicit state protection
- Object-oriented program structure
- Robust timing semantics

Main conceptual threshold for programmers: **Reactivity!**

The big win of reactivity:

**Modular composition of real-time systems
with composable timing!**