

# Embedded Software Engineering

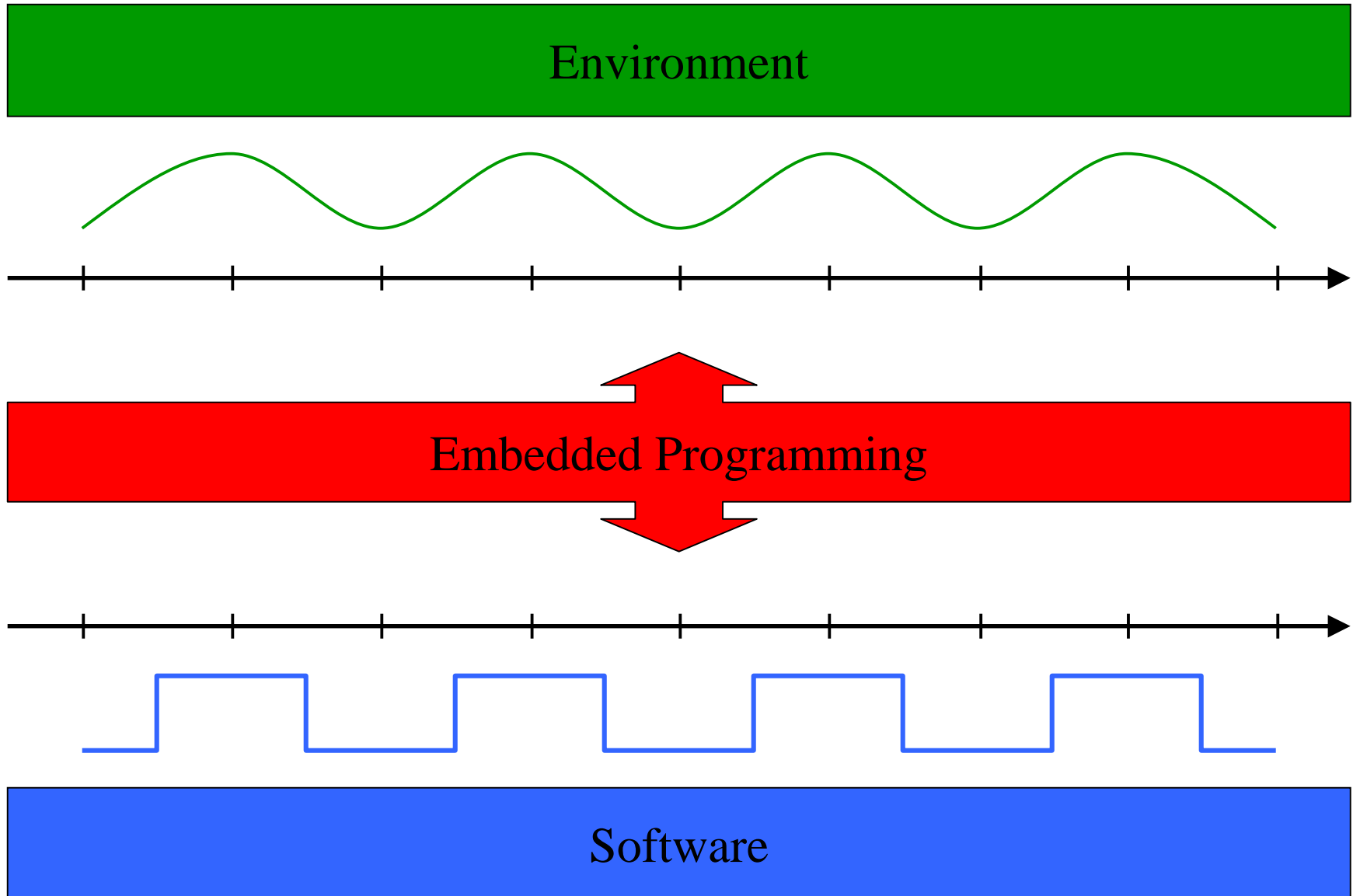
3 Unit Course, Spring 2002  
EECS Department, UC Berkeley

## Chapter 1: RTOS Concepts

Christoph Kirsch

[www.eecs.berkeley.edu/~fresco/giotto/course-2002](http://www.eecs.berkeley.edu/~fresco/giotto/course-2002)

# The Art of Embedded Programming



# What Do We Really Need From an RTOS?



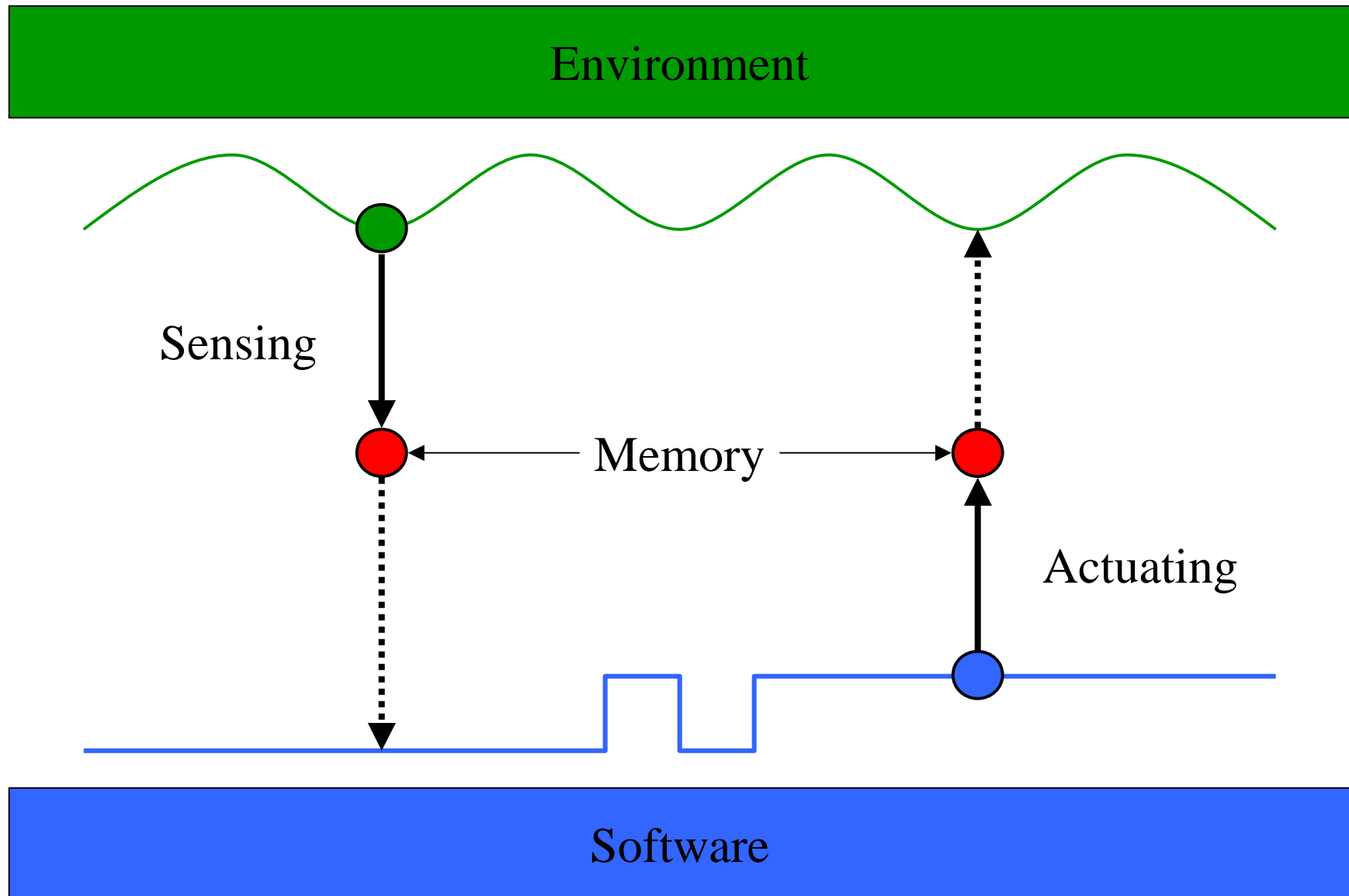
Environment

Environment Processes

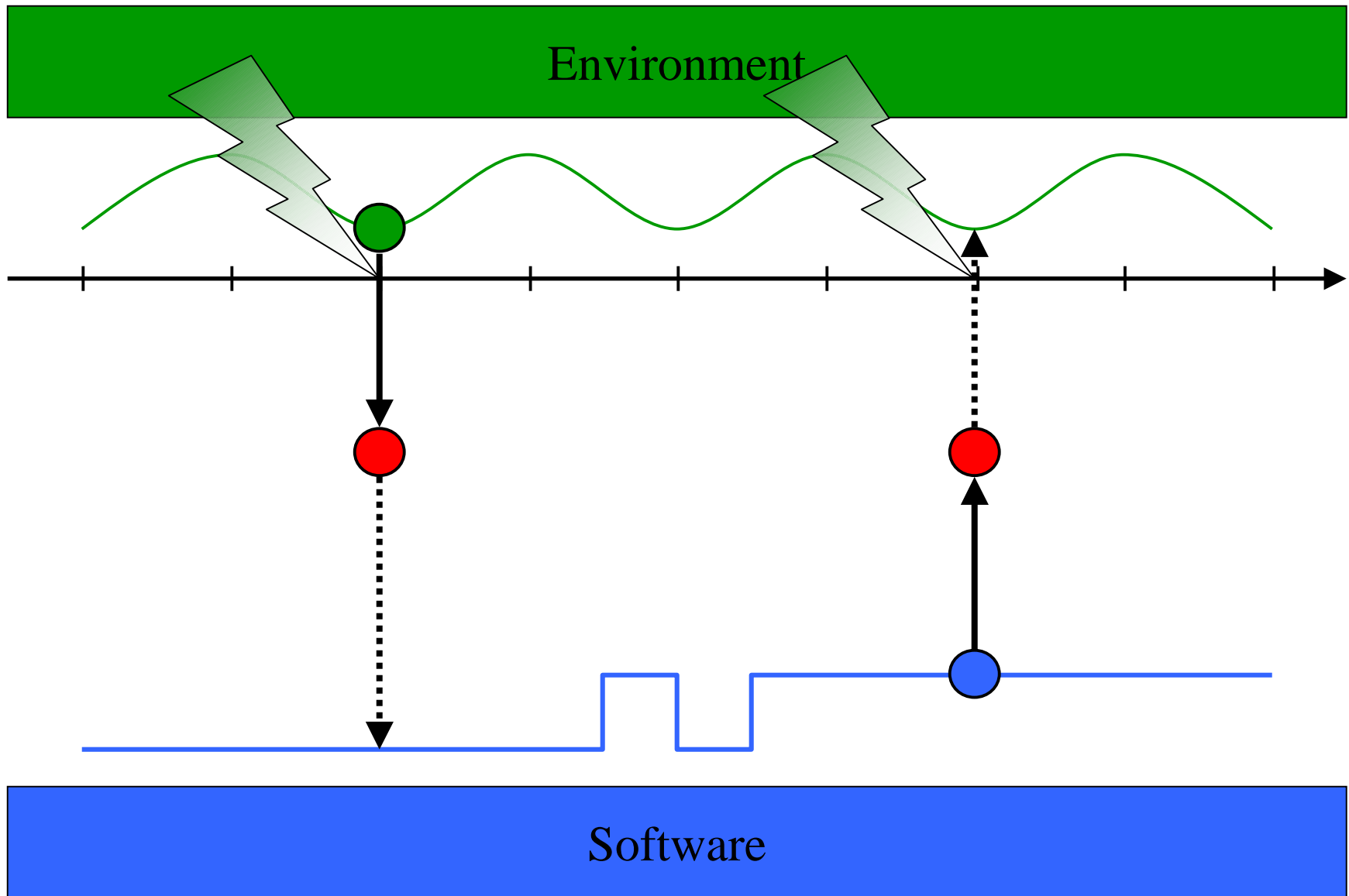
Software Processes

Software

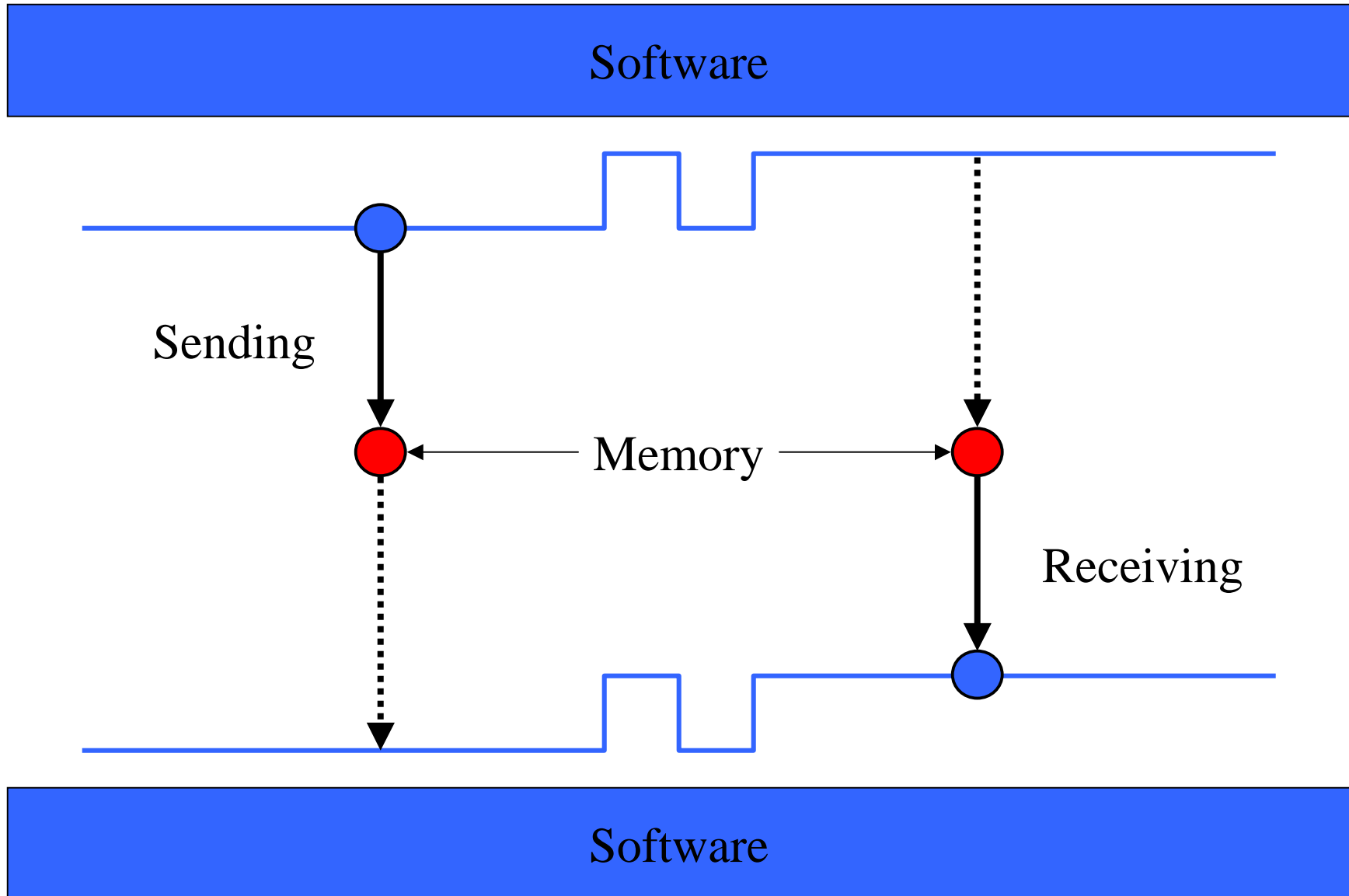
# Environment Communication Services



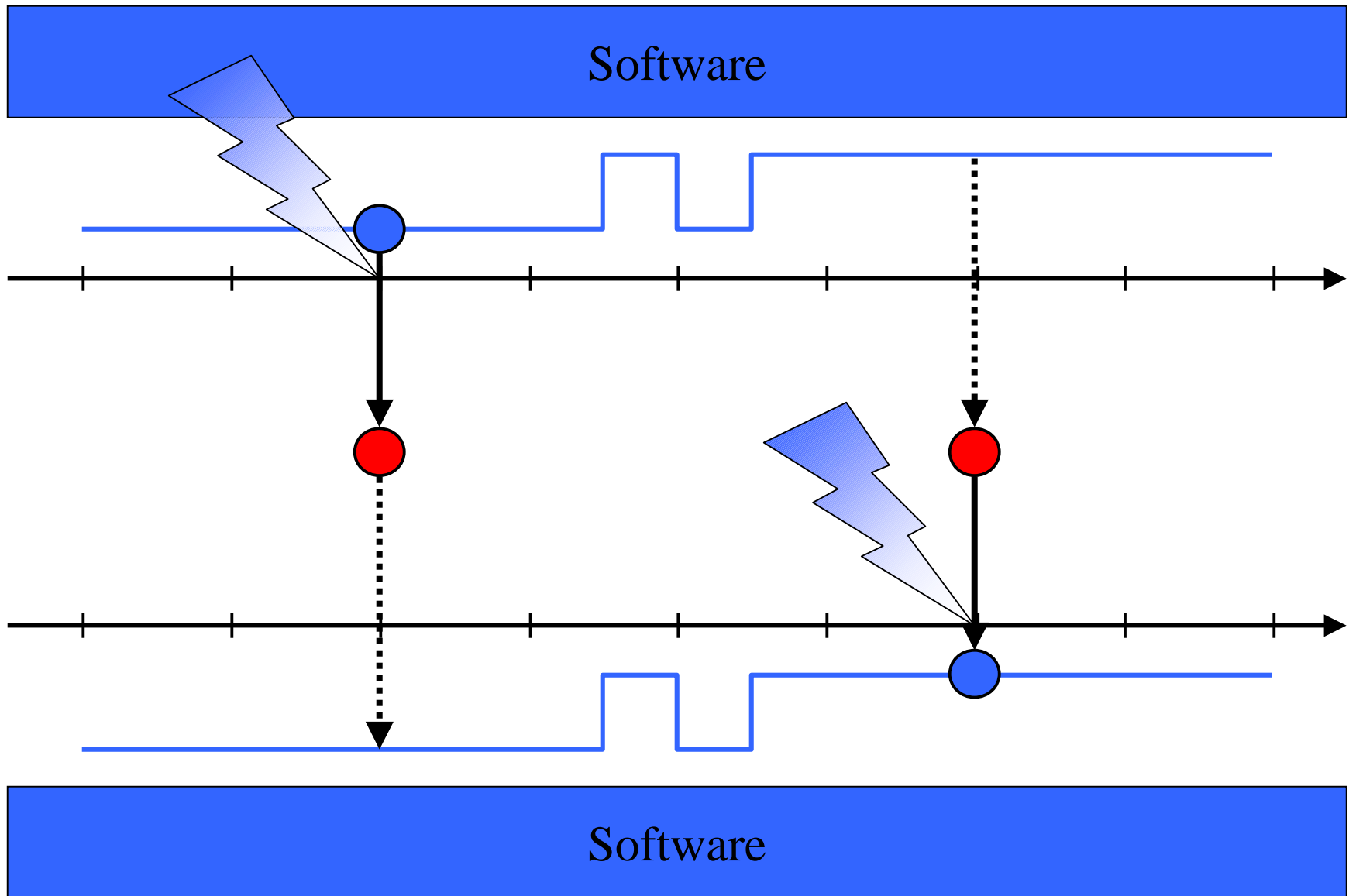
# Environment Trigger Services



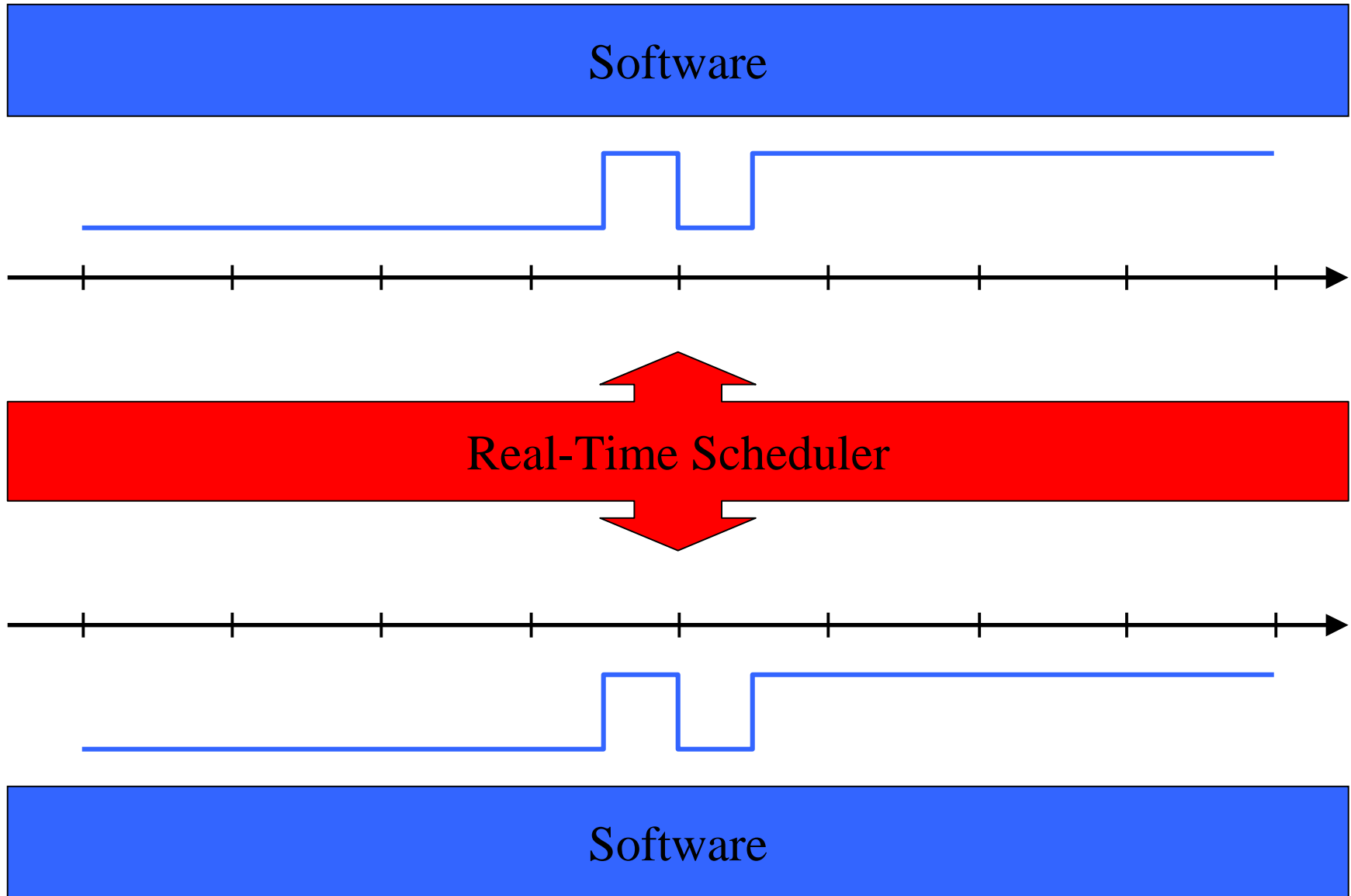
# Software Communication Services



# Software Trigger Services



# Software Scheduling Services

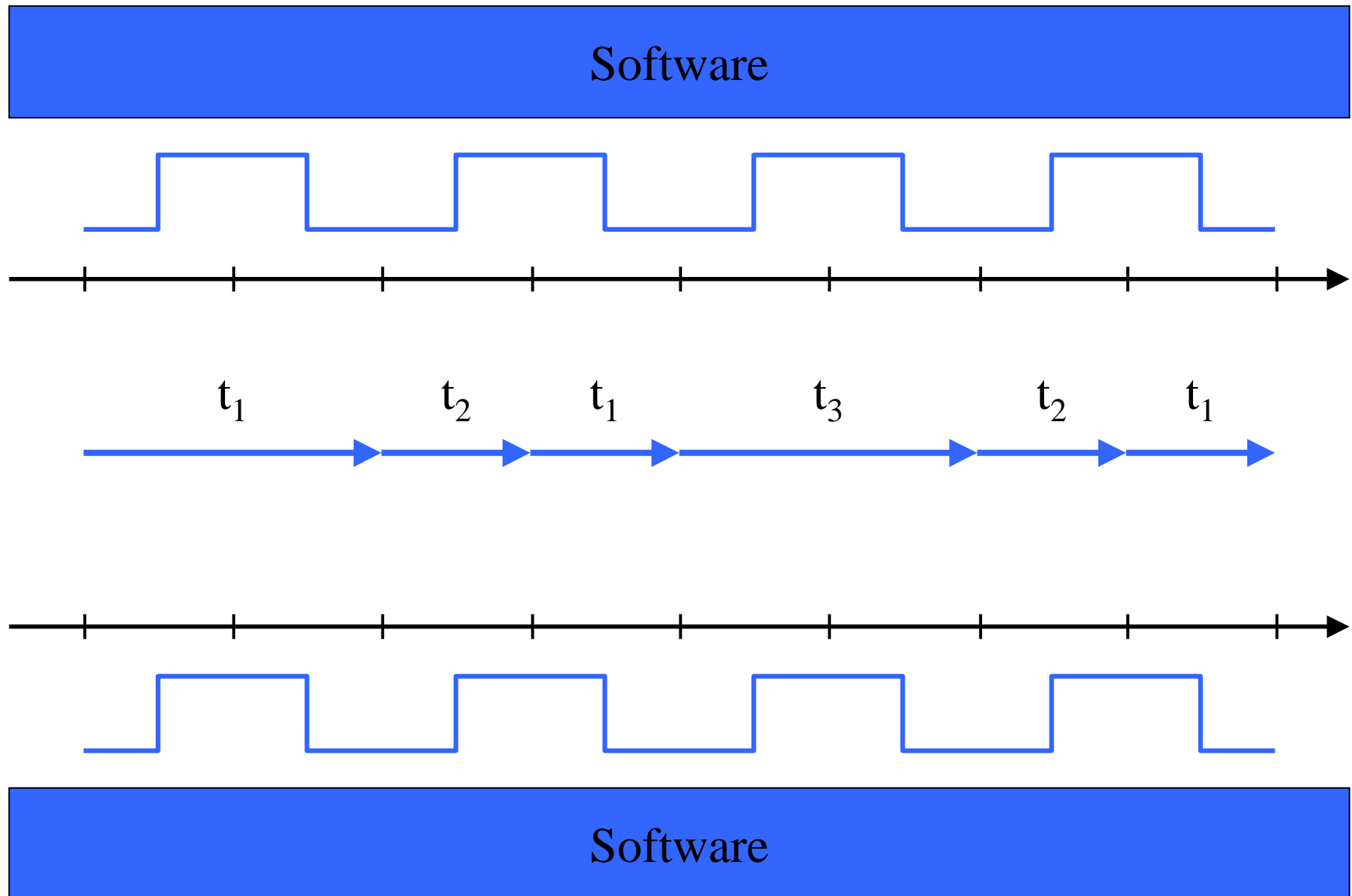




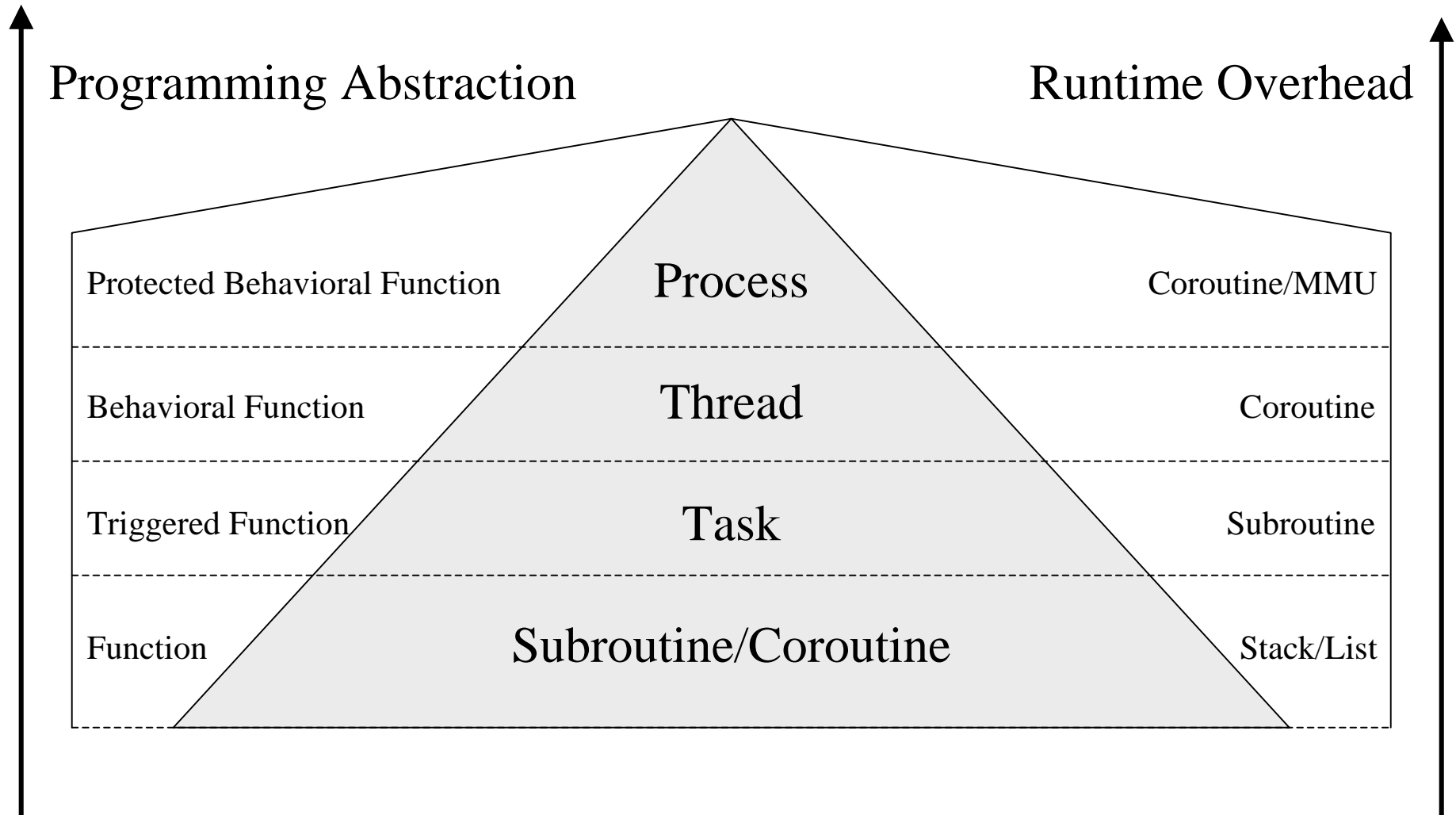
# Summary: RTOS Services

Service	Implementation
Sensing/Actuating	Device Drivers
Environment Triggering	Interrupt Handlers
Software Communication	Shared Variables
Software Triggering	Signals
Software Scheduling	Scheduler

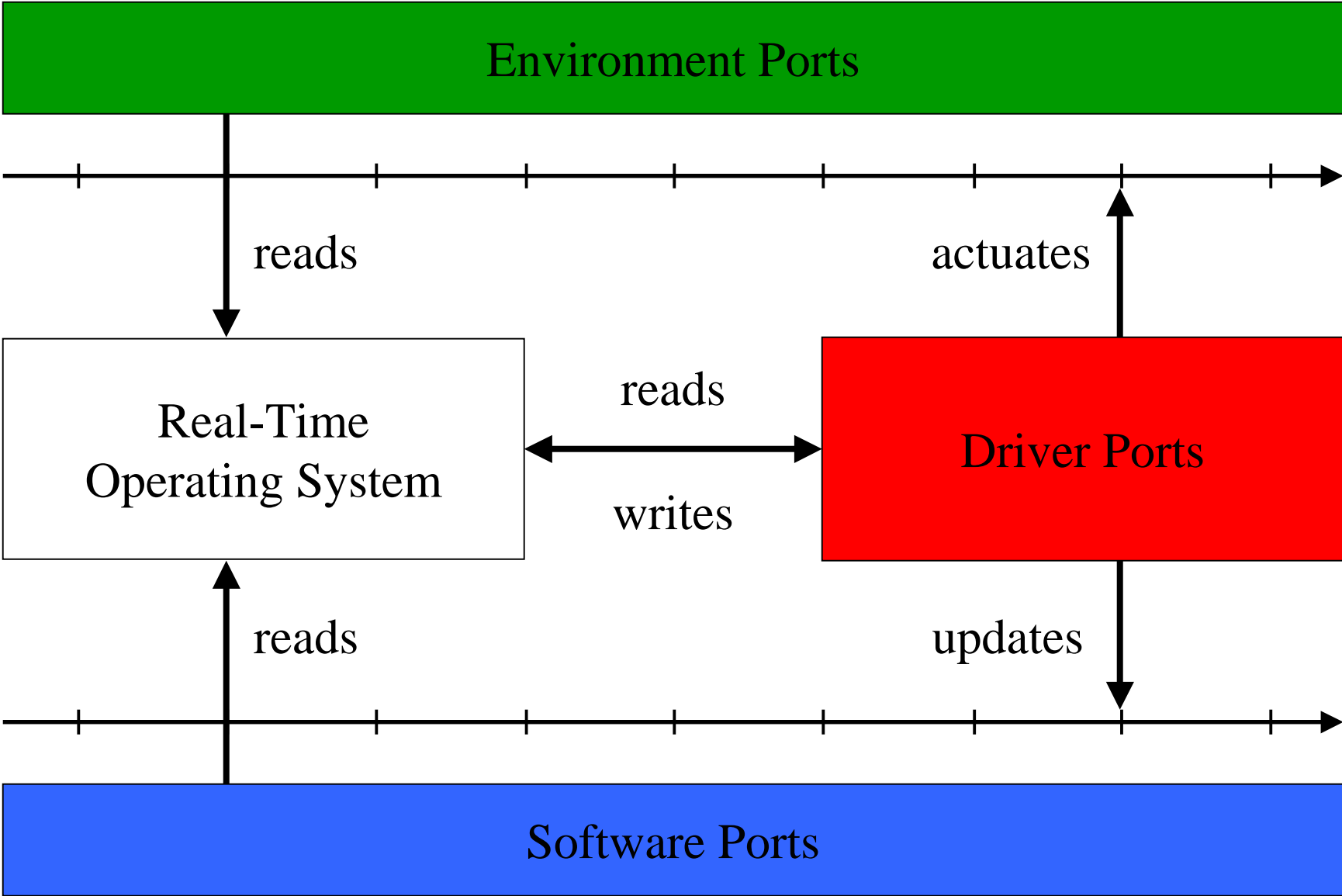
# The Illusion of Concurrent Software



# Abstractions for Multiprogramming



# Memory Model



# Definition: Task

---

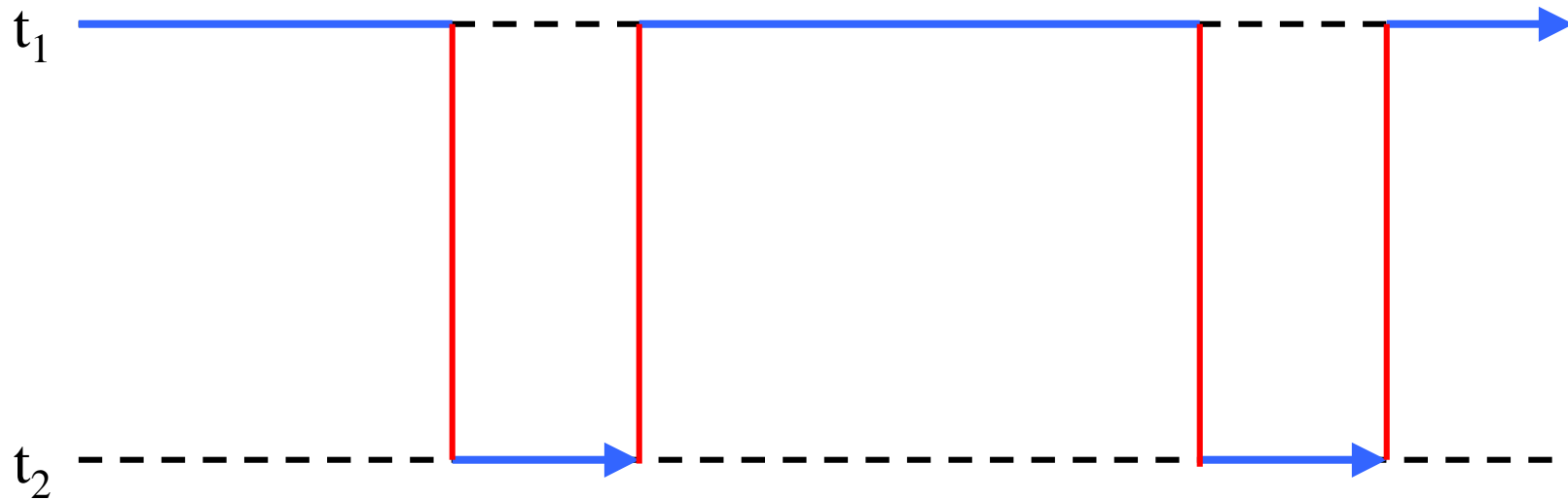
- A task is a *function* from its input and state ports to its output and state ports
- A task *runs to completion* (cannot be killed)
- A task is *preemptable*
- A task does not use *signals* (except at completion)
- A task does not use *semaphores* (as a consequence)
- API (used by the RTOS):
  - `initialize {task: state ports}`
  - `schedule {task}`
  - `dispatch {task: function}`

## So, what's the difference between a task and a function?

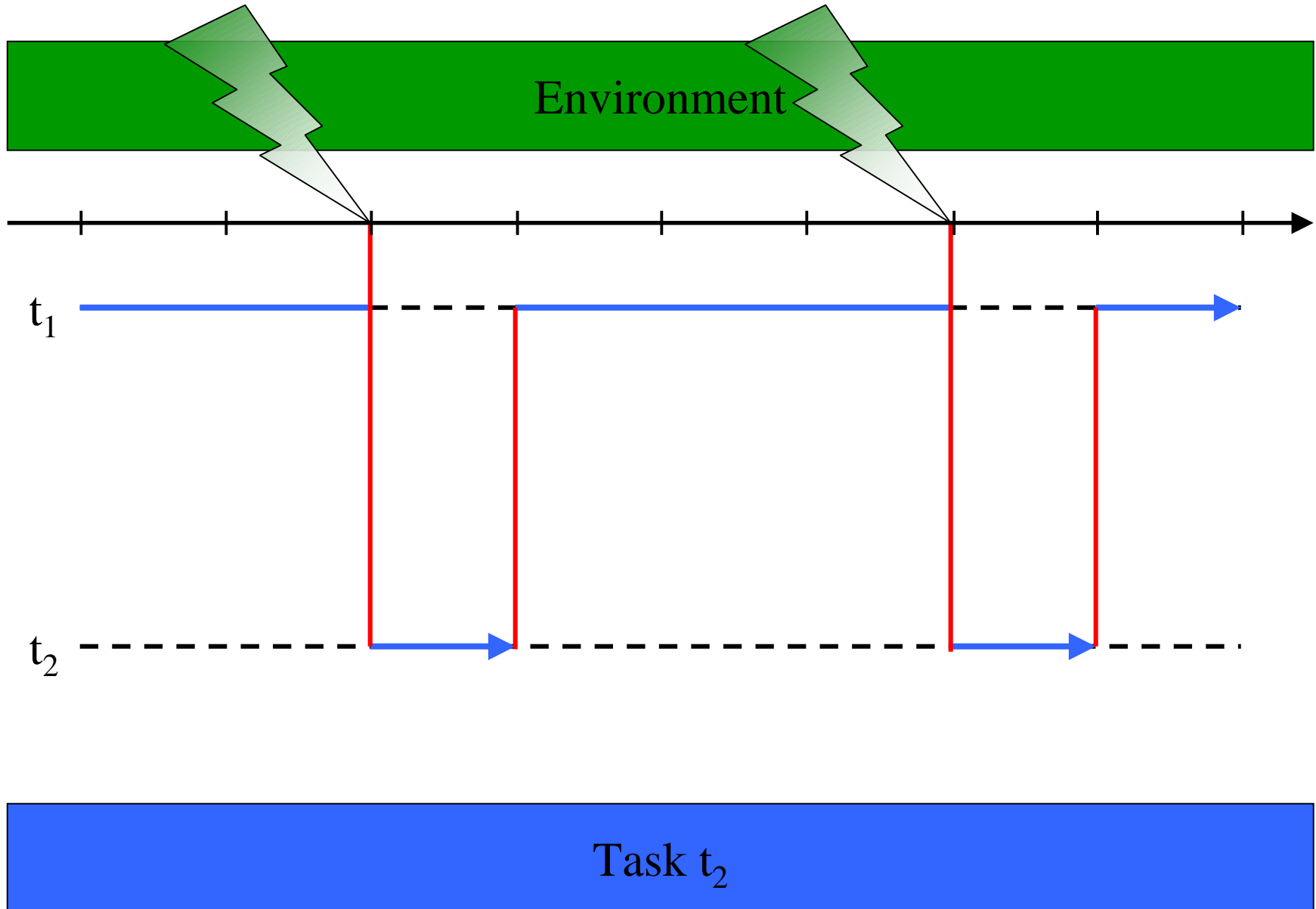
---

- A task has an operational semantics:
  - A task is implemented by a *subroutine* and a *trigger*
  - A task is either *environment-* or *software-triggered*
  - The completion of a task may trigger another task

# Task $t_2$ Preempts Task $t_1$



# Who Triggers Task $t_2$ ?





# Definition: Event and Signal

---

- An event is a *change of state* in some **environment** ports
- A signal is a *change of state* in some **software** ports
- A synchronous signal is a *change of state* in some **driver** ports

# Definition: Trigger

---

- A trigger is a *predicate* on **environment**, **software**, **driver** ports
- A trigger *awaits* events and/or signals
- A trigger is *enabled* if its predicate evaluates to true
- Trigger evaluation is *atomic* (non-preemptable)
  
- A trigger can be *activated* by the RTOS
- A trigger can be *cancelled* by the RTOS
- A trigger can be *enabled* by an event or a signal
  
- API (used by the RTOS):
  - `activate {trigger}`
  - `cancel {trigger}`
  - `evaluate {trigger: predicate}`

# My First RTOS

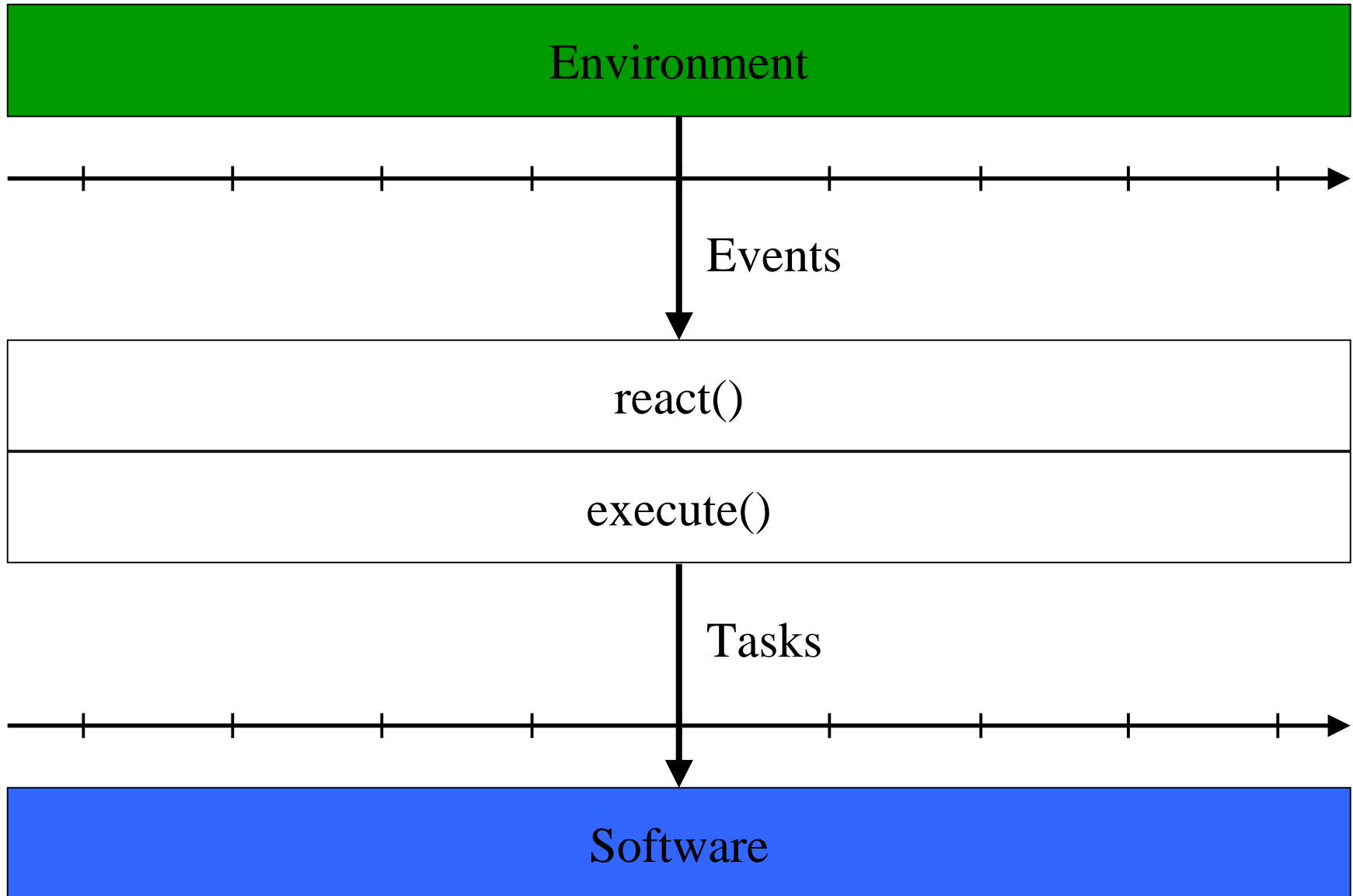
---

```
∀ tasks t: initialize(t);
∀ triggers g: activate(g);
while (true) {
    if ∃ active-trigger g: evaluate(g) == true then
        execute();
}
```

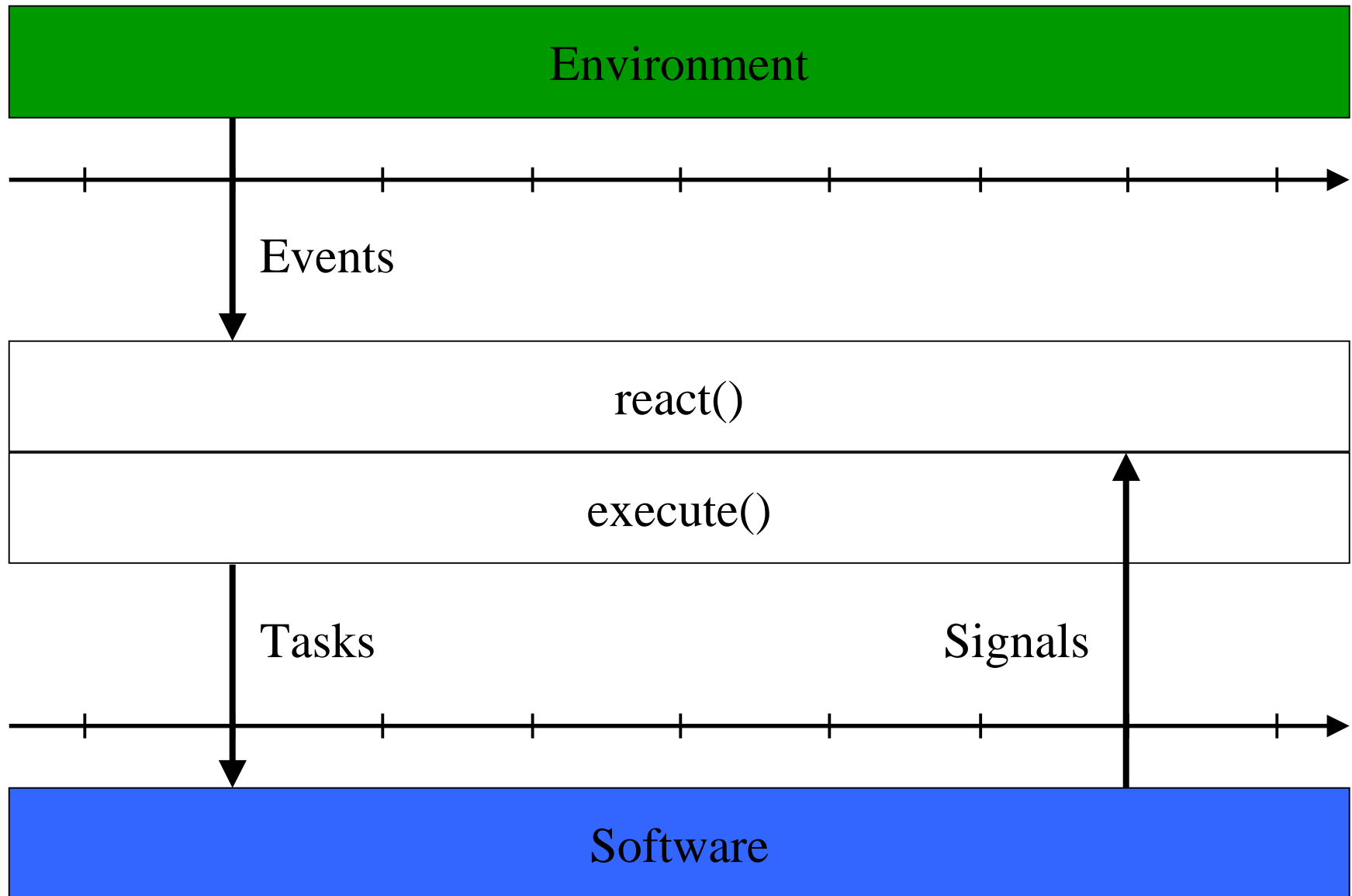
---

```
execute() {
    scheduled-tasks := ∀ triggered-tasks t: schedule(t);
    ∀ scheduled-tasks t: dispatch(t);
}
```

# RTOS Model: Reaction vs. Execution



# RTOS Model with Signals



# RTOS with Preemption

---

```
∀ tasks t: initialize(t);
∀ triggers g: activate(g);
while (true) {
    if ∃ active-trigger g: evaluate(g) == true then
        execute_concurrently();
}
```

---

```
execute_concurrently() {
    scheduled-tasks := ∀ triggered-tasks t: schedule(t);
    ∀ scheduled-tasks t: dispatch(t);
}
```

# Corrected RTOS with Preemption

---

```
∀ tasks t: initialize(t);
∀ triggers g: activate(g);
while (true) {
    if ∃ active-trigger g: evaluate(g) == true then
        execute_concurrently();
}
```

---

```
execute_concurrently() {
    ∀ triggers g: cancel(g);
    scheduled-tasks := ∀ triggered-tasks t: schedule(t);
    ∀ triggers g: activate(g);

    ∀ scheduled-tasks t: dispatch(t);
}
```

# Definition: Thread

---

- A thread is a *behavioral function* (with a trace semantics)
- A thread *may be killed*
- A thread is *preemptable*
- A thread may use *signals*
- A thread may use *semaphores*
- API (used by the RTOS or threads):
  - `initialize {thread: ports}`
  - `schedule {thread}`
  - `dispatch {thread: function}`
  - `kill {thread}`

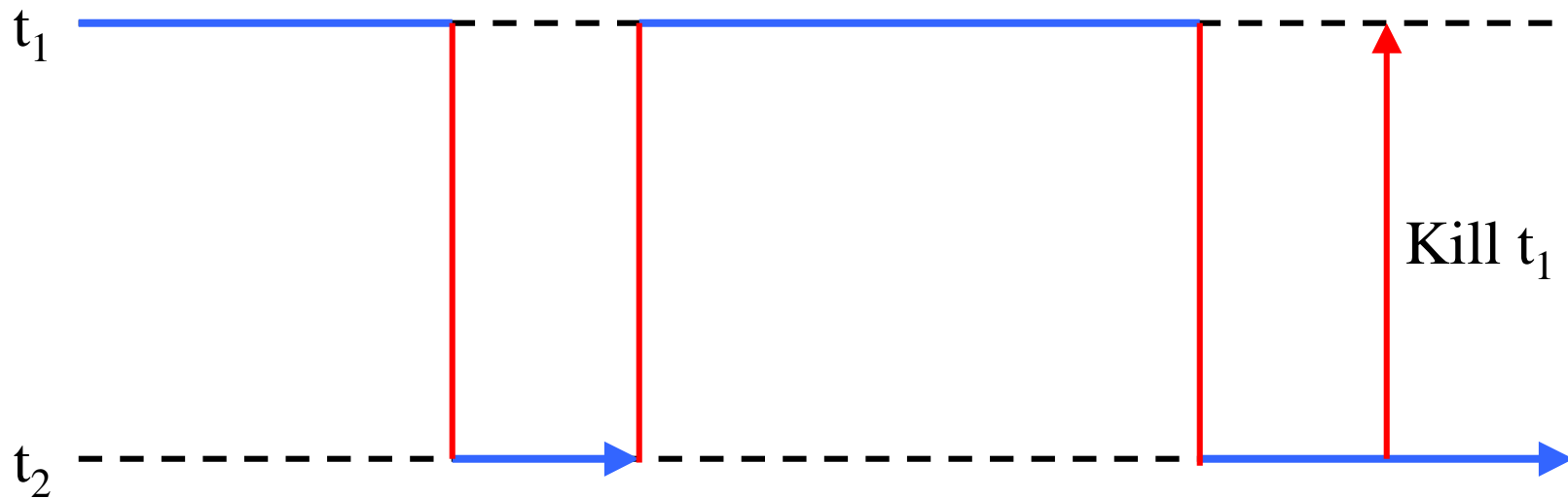


# So, what's the difference between a thread and a task?

---

- A thread is a *collection* of tasks:
  - A thread is implemented by a *coroutine*
  - A thread requires signals

# Task $t_2$ Kills Task $t_1$ : Coroutine



# Signal API

---

- A signal can be *awaited* by a thread
- A signal can be *emitted* by a thread
- Signal emission is *atomic* (non-preemptable)
  
- API (used by threads):
  - `wait {signal}`
  - `emit {signal}`
  
- Literature:
  - `emit: send(signal)`

# Definition: Semaphore

---

- A semaphore consists of a *signal* and a *port*
- A semaphore can be *locked* by a thread
- A semaphore can be *released* by a thread
- Semaphore access is *atomic* (non-preemptable)
- API (used by threads):
  - `lock { semaphore }`
  - `release { semaphore }`
- Literature:
  - `lock: P(semaphore)`
  - `release: V(semaphore)`

# Binary Semaphore (Signal)

---

```
lock(semaphore) {  
    if (semaphore.lock == true) then  
        wait(semaphore.signal);  
    semaphore.lock := true;  
}
```

} *must be atomic*

```
release(semaphore) {  
    semaphore.lock := false;  
    emit(semaphore.signal);  
}
```

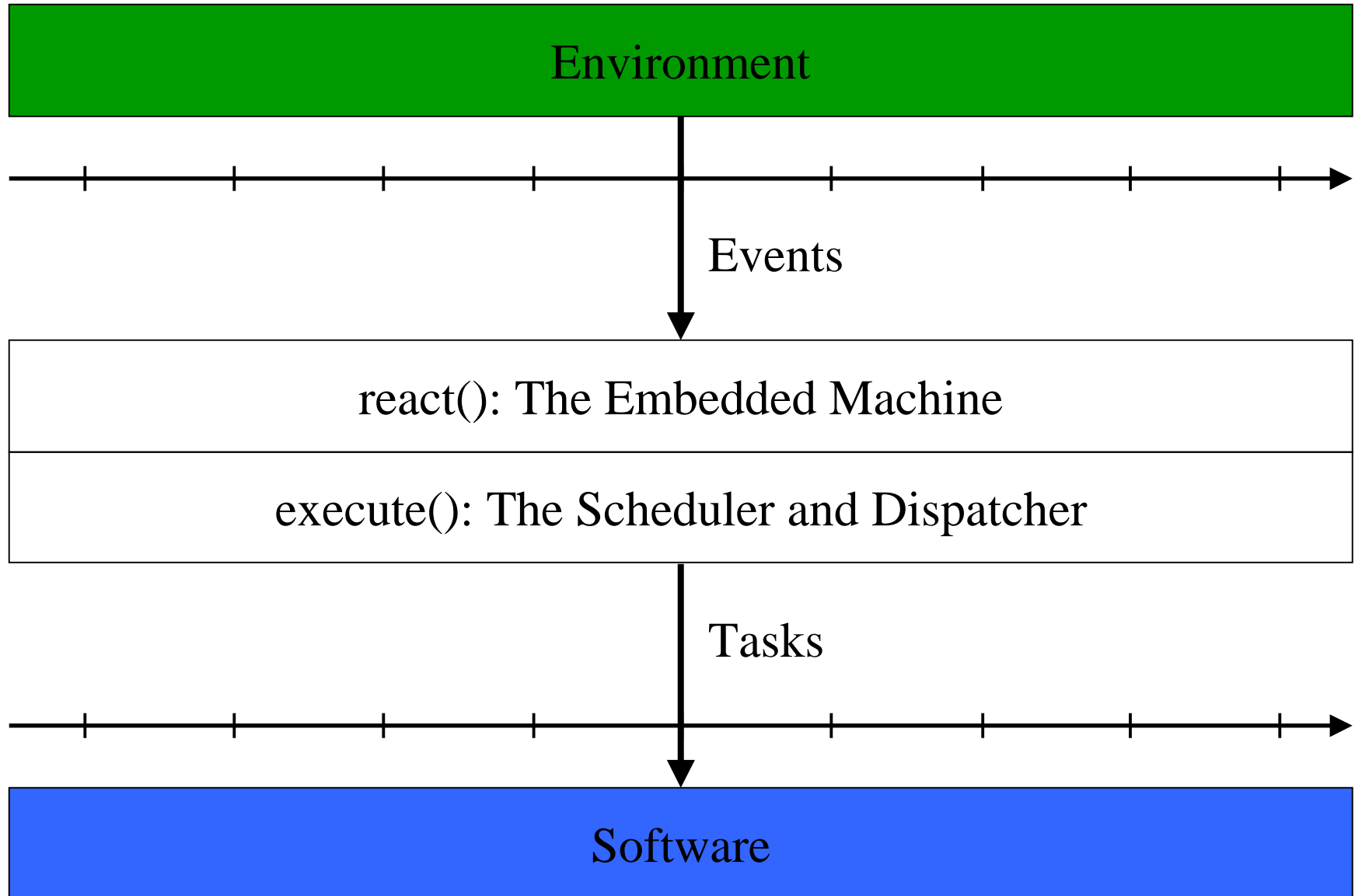
# Binary Semaphore (Busy Wait)

---

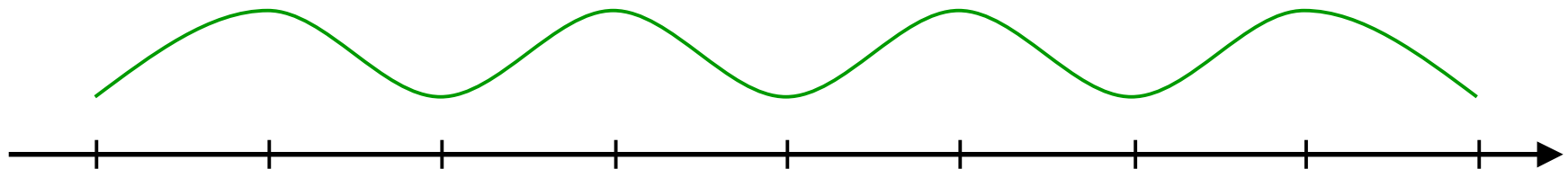
```
lock(semaphore) {  
    while (semaphore.lock == true) do {} } each round  
    semaphore.lock := true; } must be atomic
```

```
release(semaphore) {  
    semaphore.lock := false;  
}
```

# The Embedded Machine



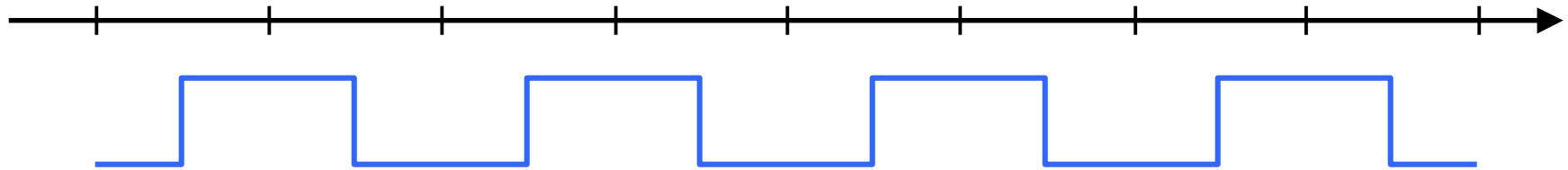
# Proposal



**Human:** Programming in terms of environment time

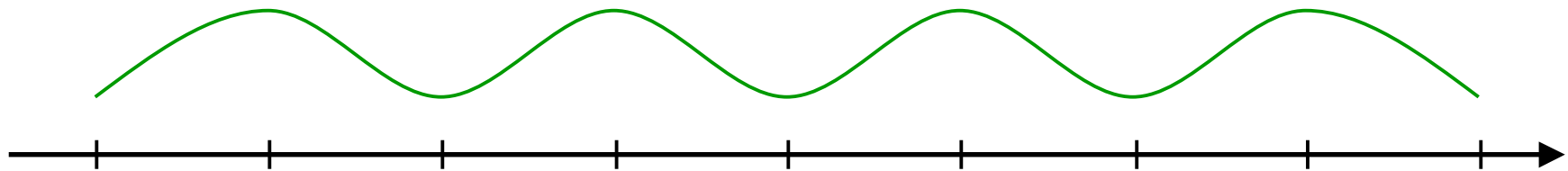


**Compiler:** Implementation in terms of platform time





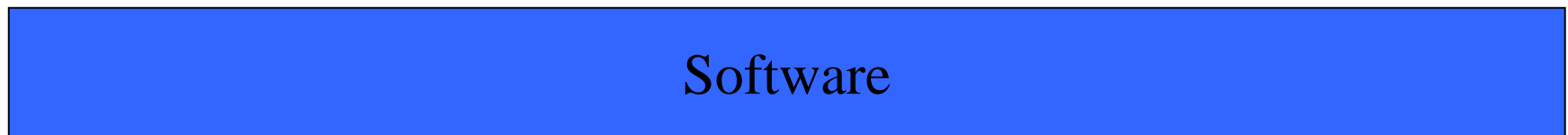
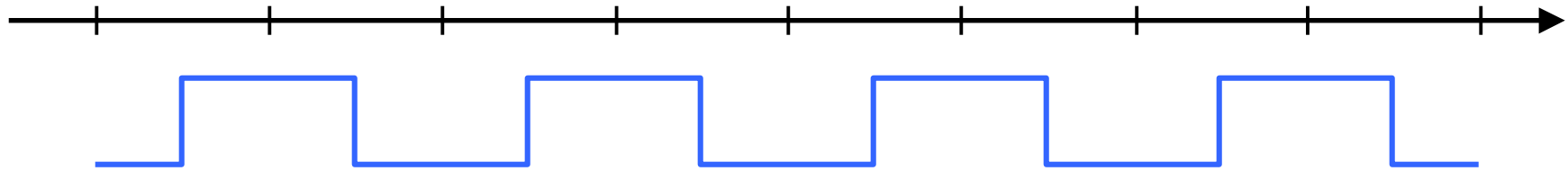
# Platform Time is Platform Memory



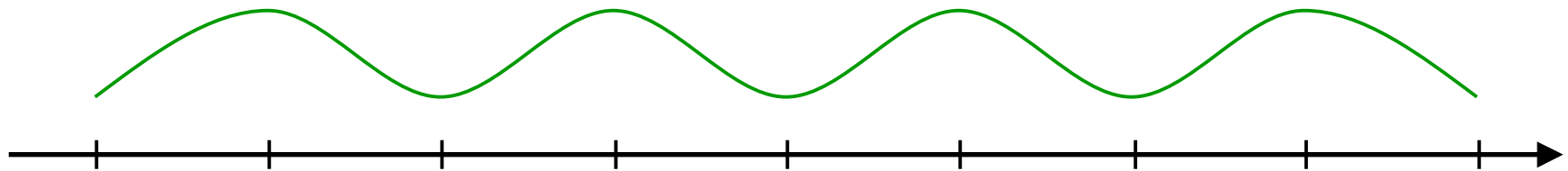
- Programming as if there is enough platform time



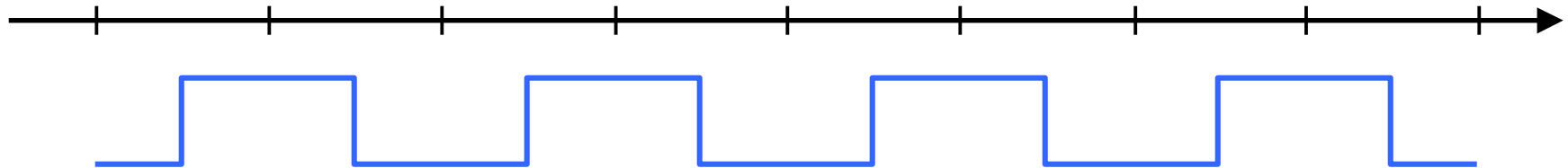
- Implementation checks whether there is enough of it



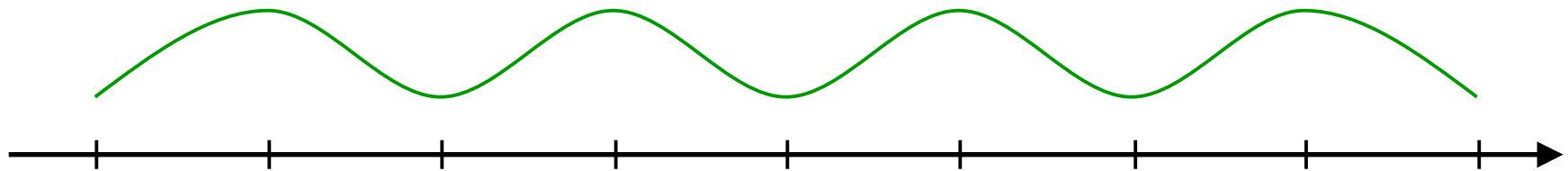
# Portability



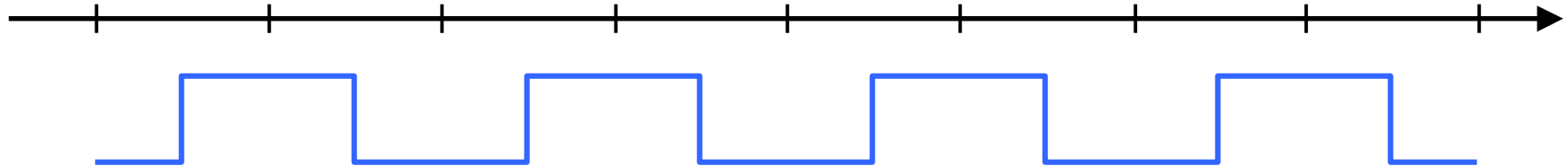
- Programming in terms of environment time yields platform-independent code



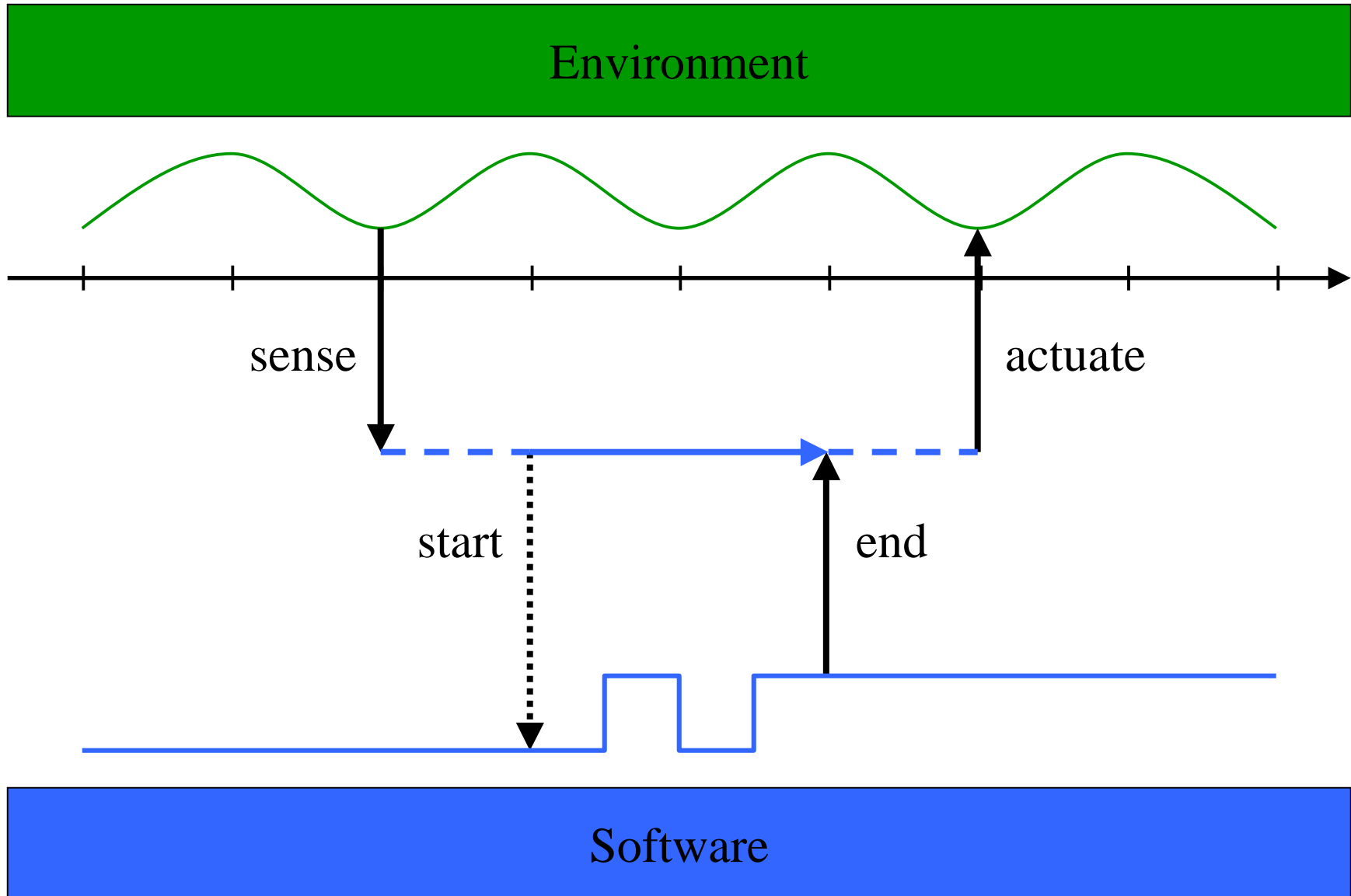
# Predictability



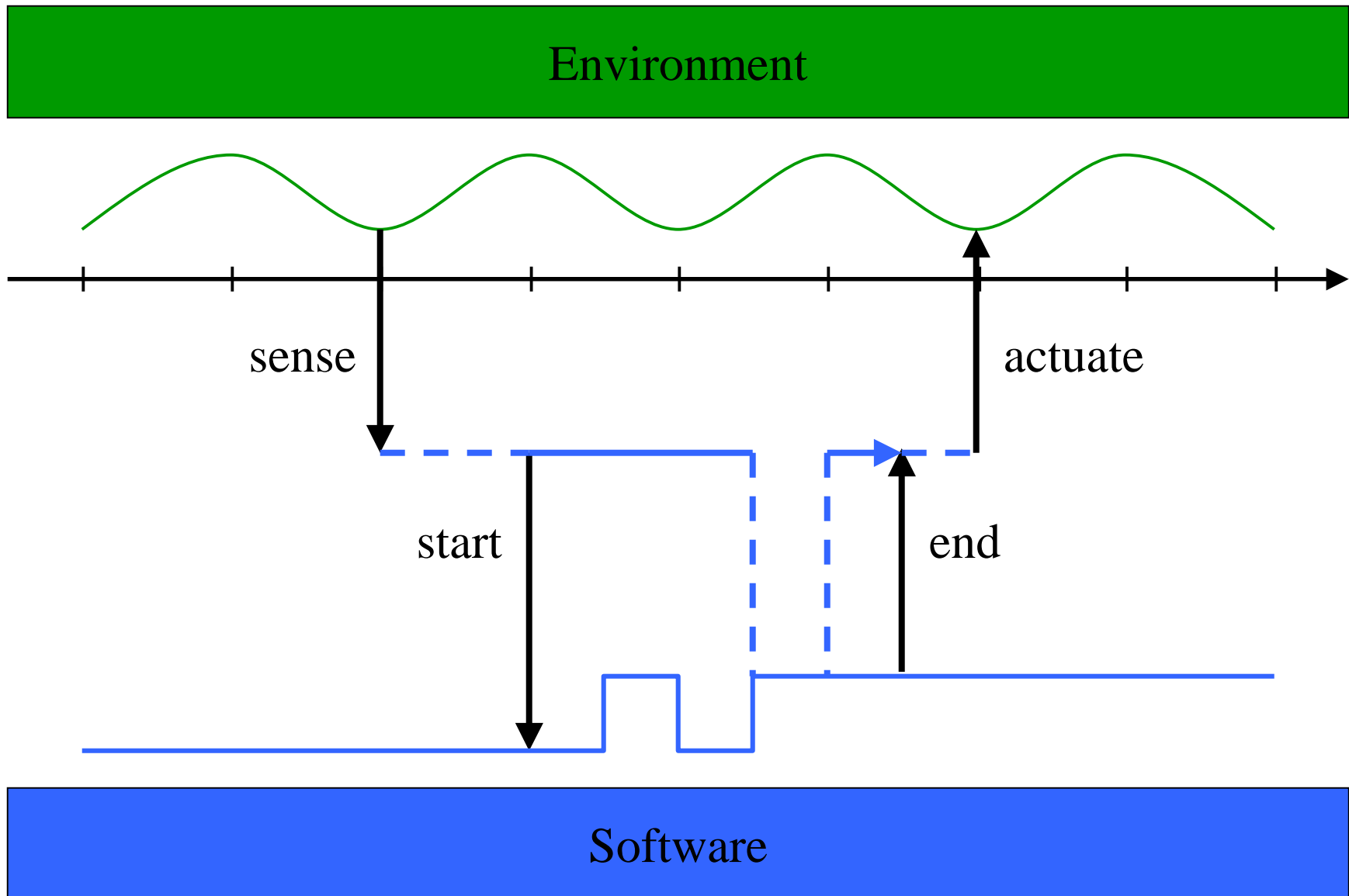
- Programming in terms of environment time yields deterministic code



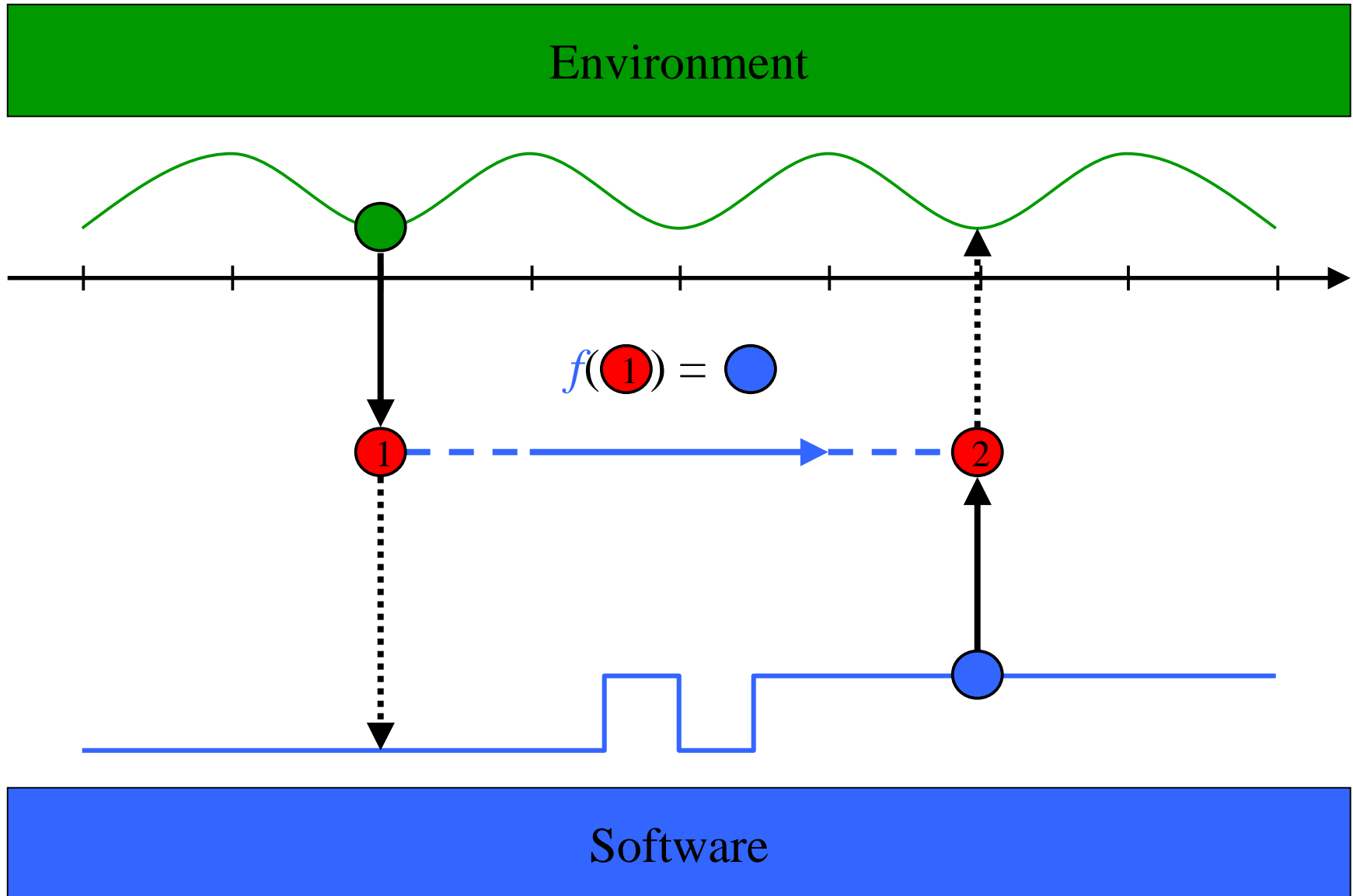
# The Task Model



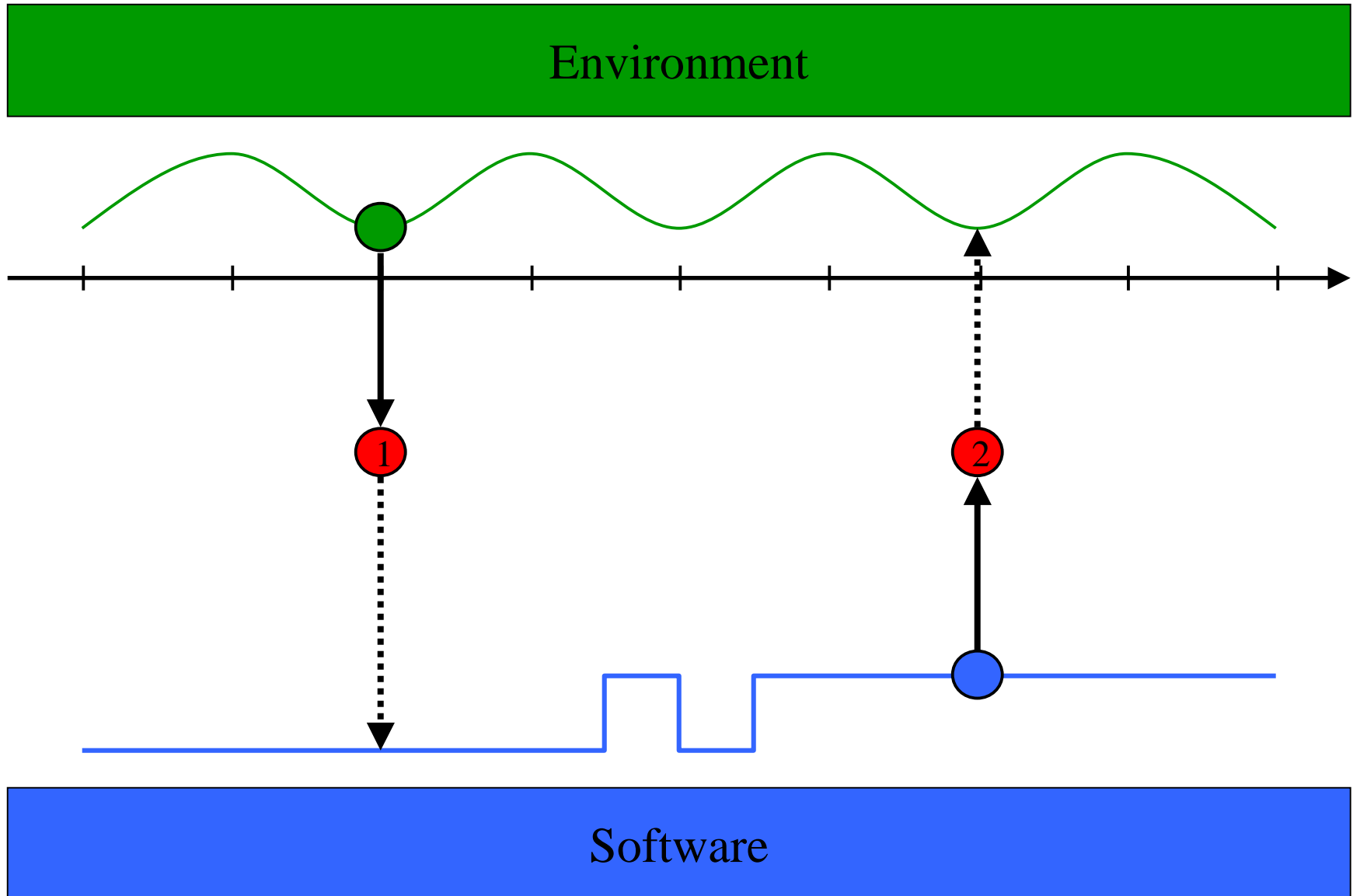
# Preemptable...



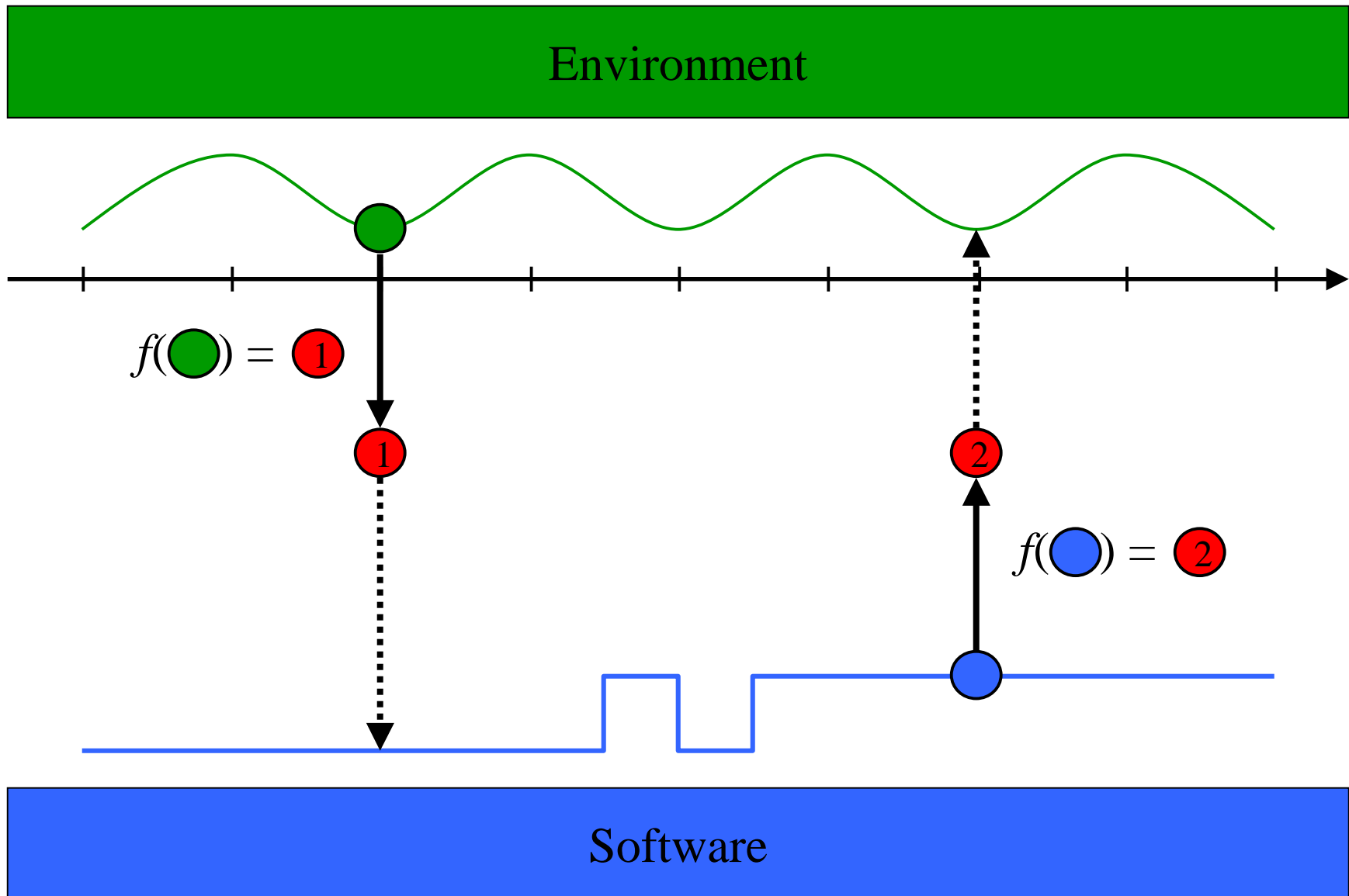
# ...but Atomic



# The Driver Model

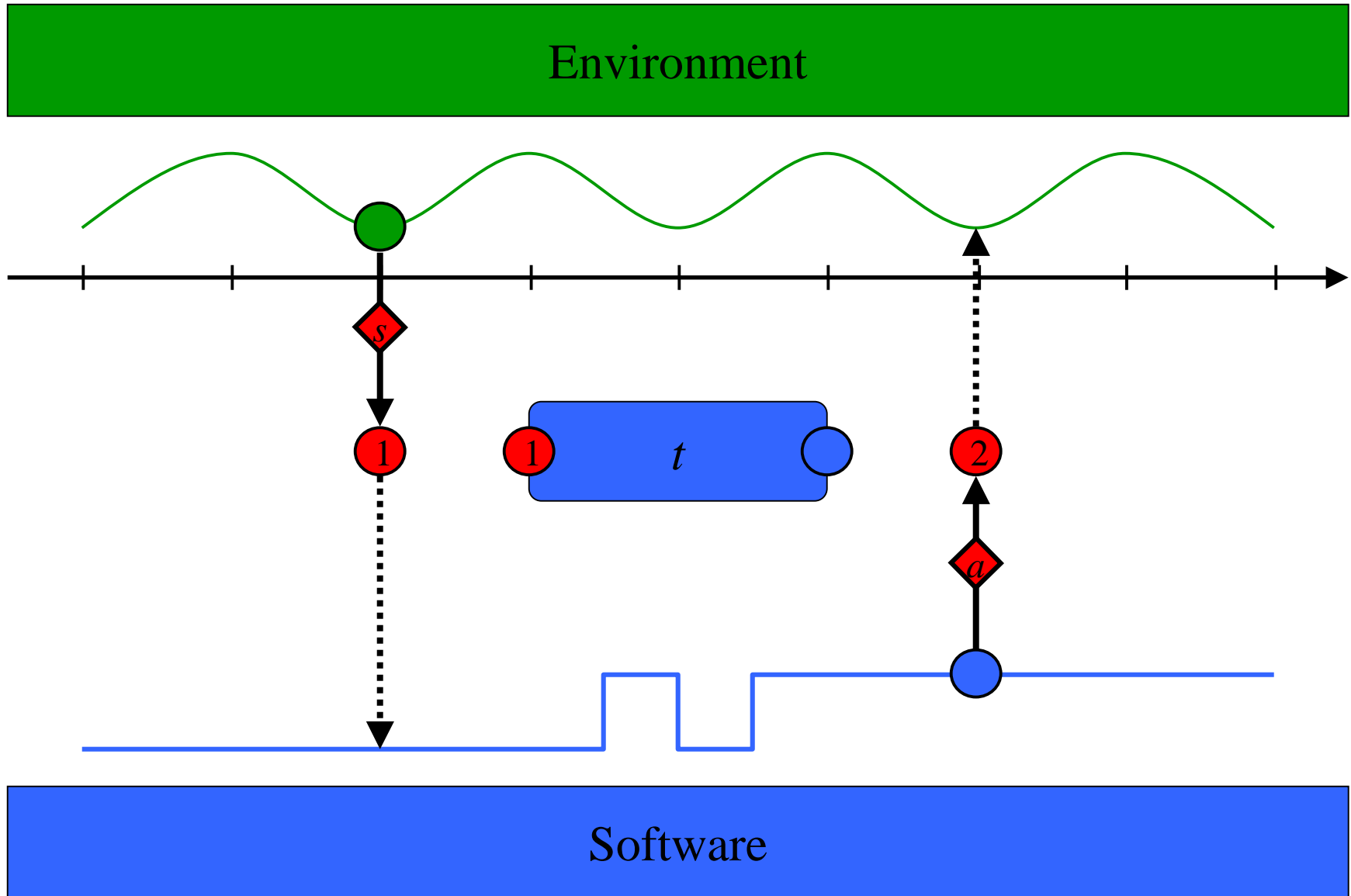


# Non-preemptable, Synchronous

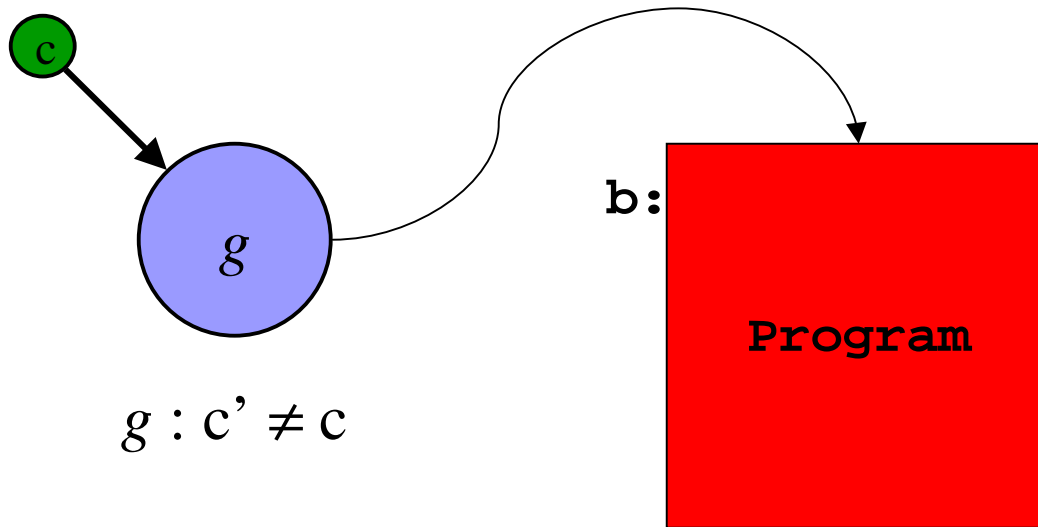




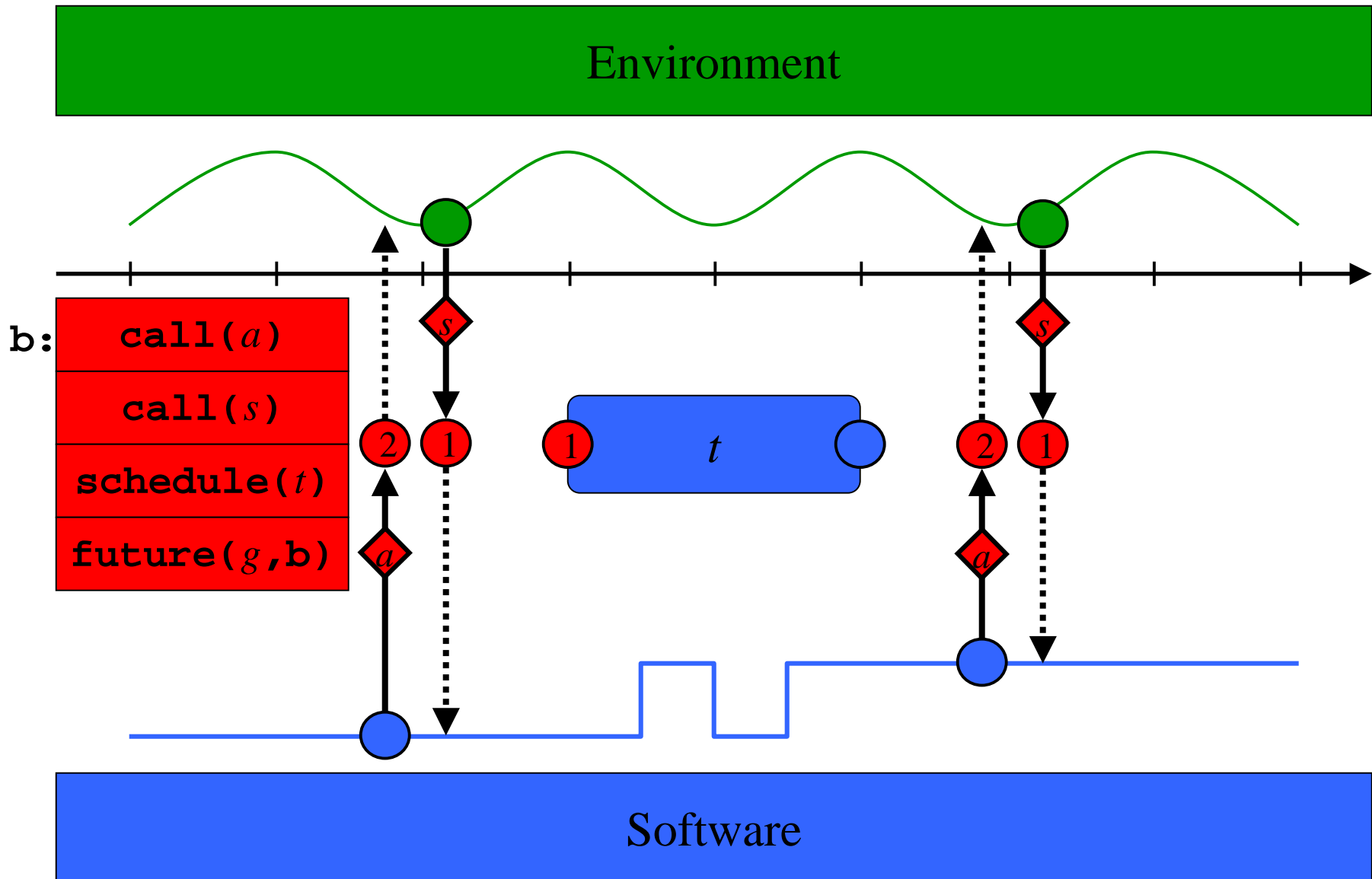
# Syntax



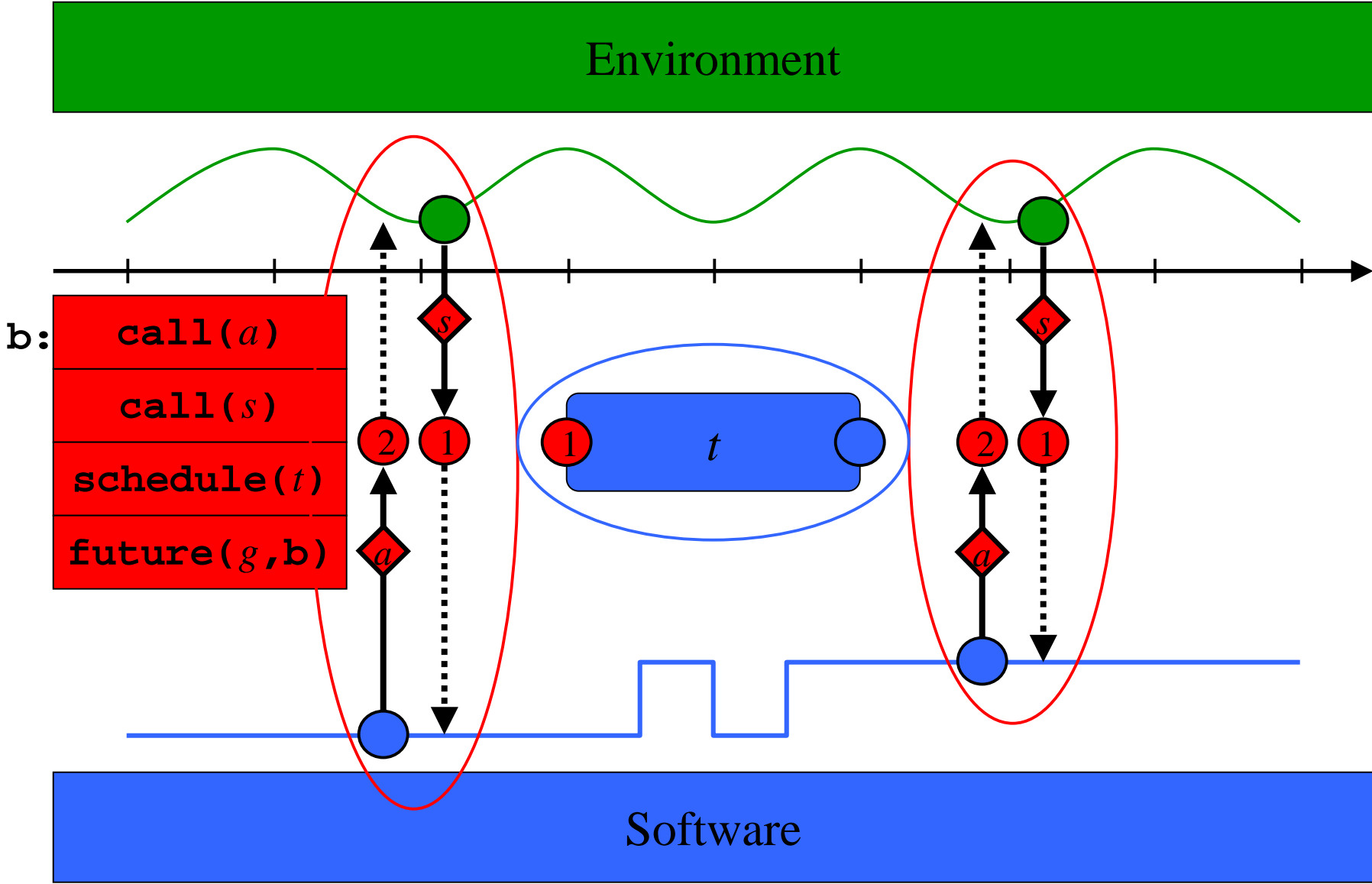
# A Trigger $g$



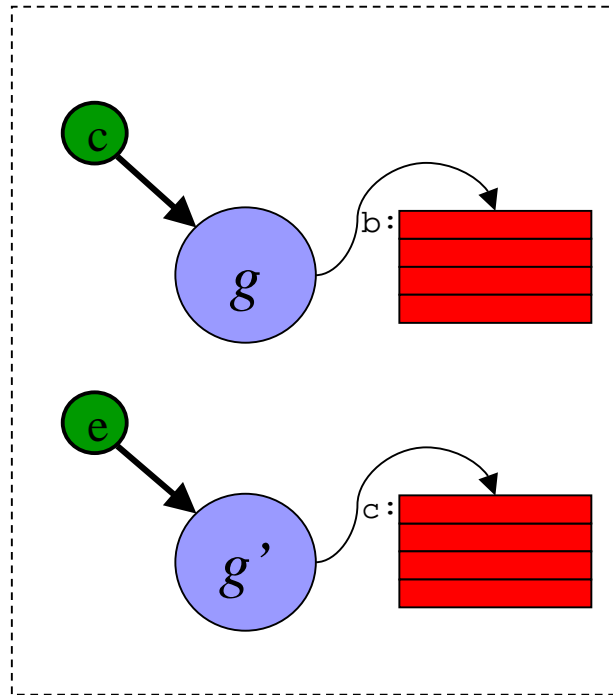
# An Embedded Machine Program



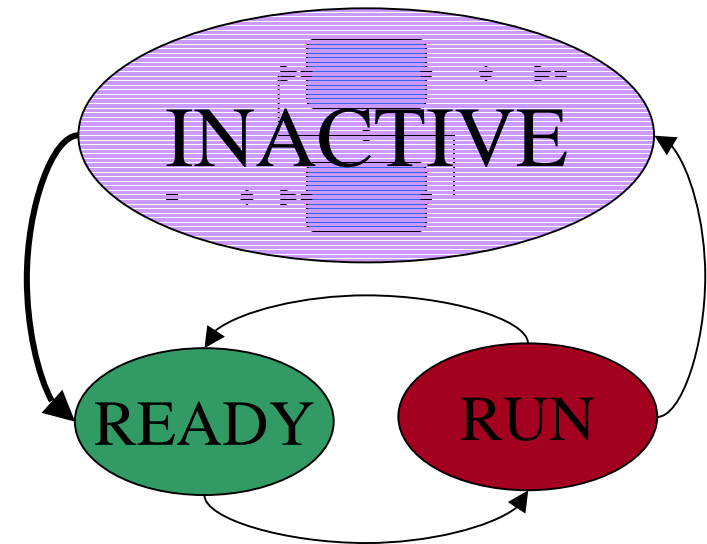
# Synchronous vs. Scheduled Computation



# Synchronous vs. Scheduled Computation



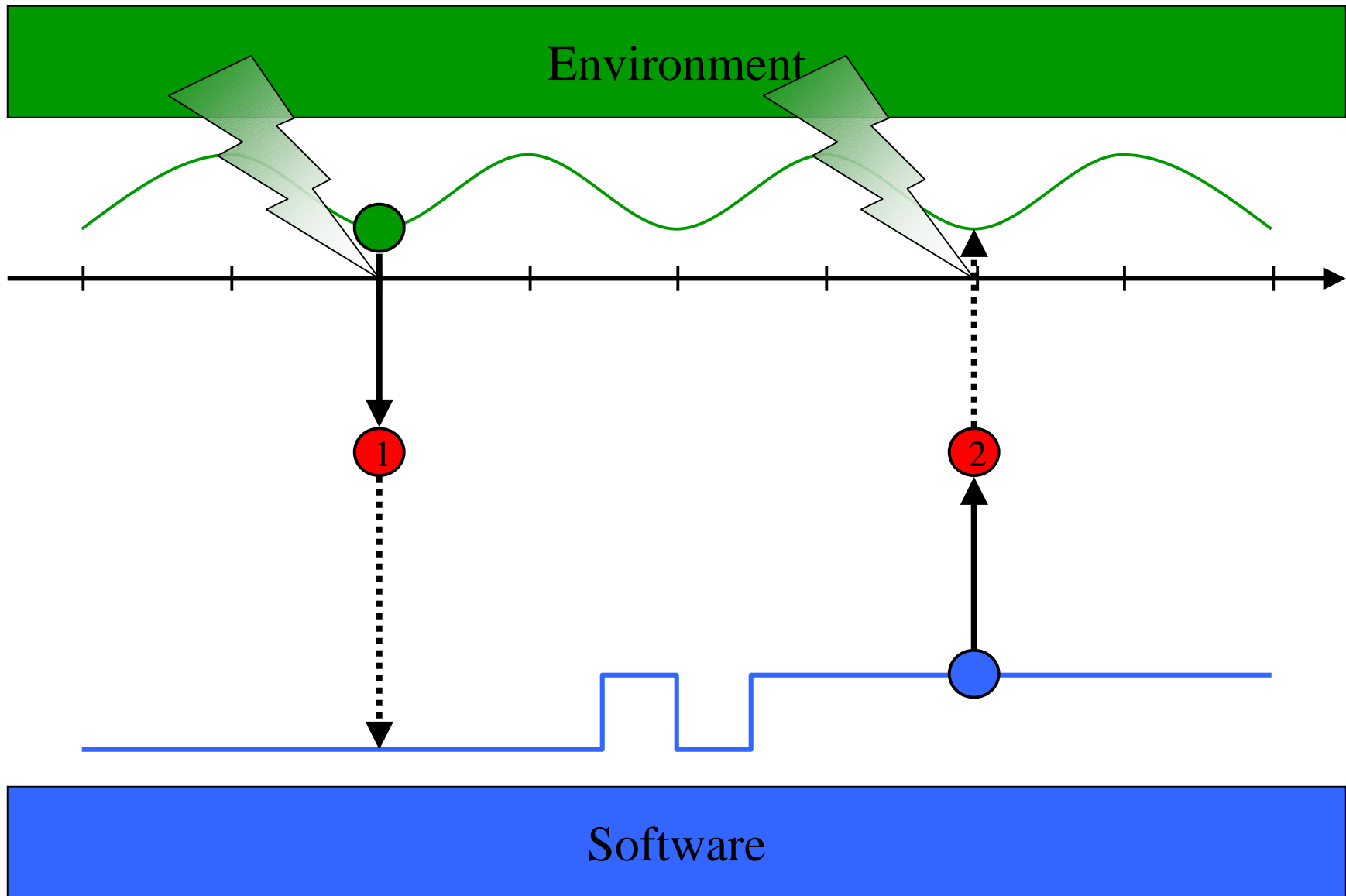
schedules



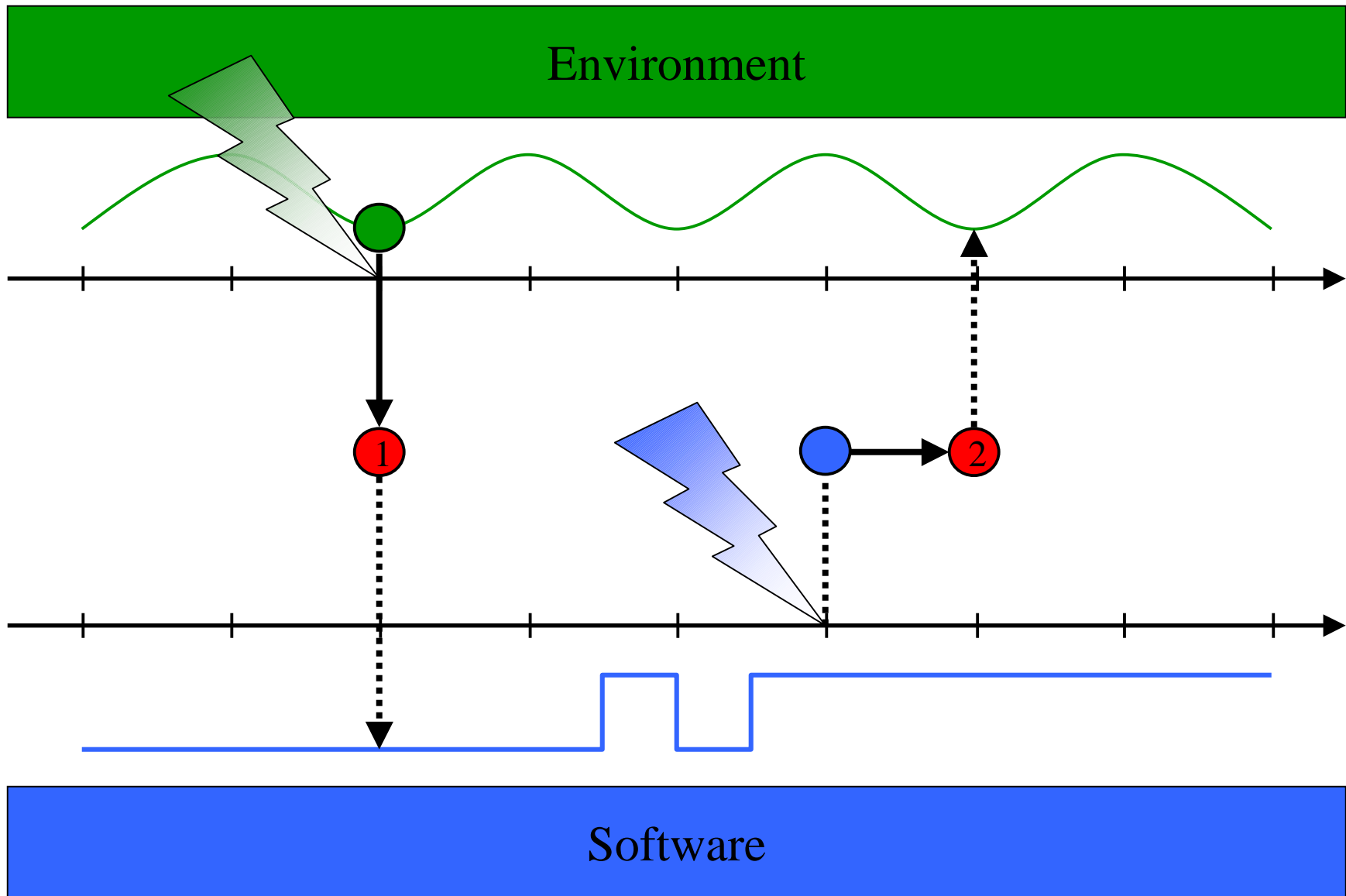
- Synchronous computation
- Kernel context
- Trigger related interrupts disabled

- Scheduled computation
- User context

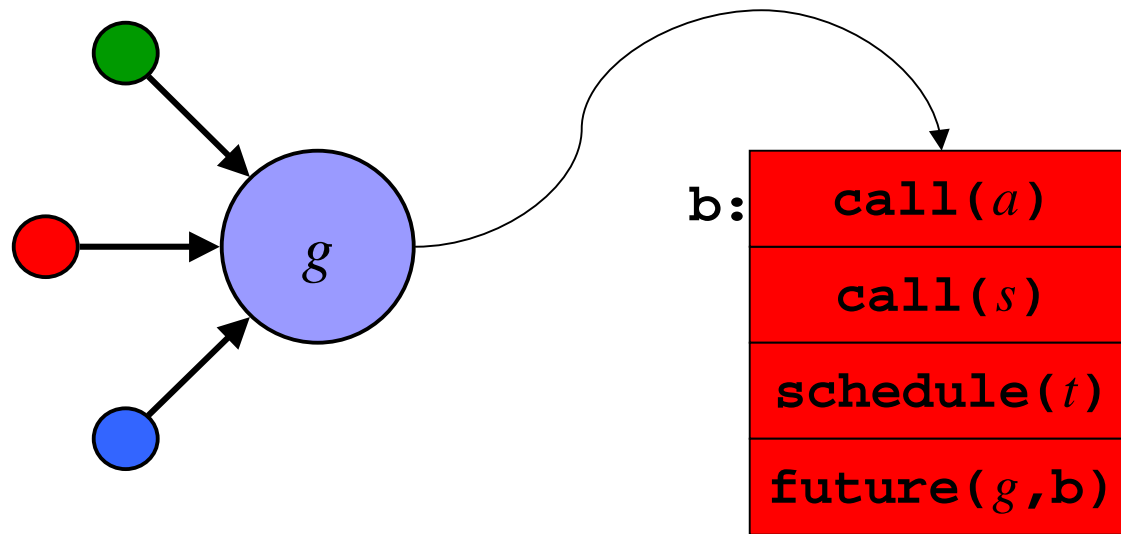
# Environment-triggered Code



# Software-triggered Code

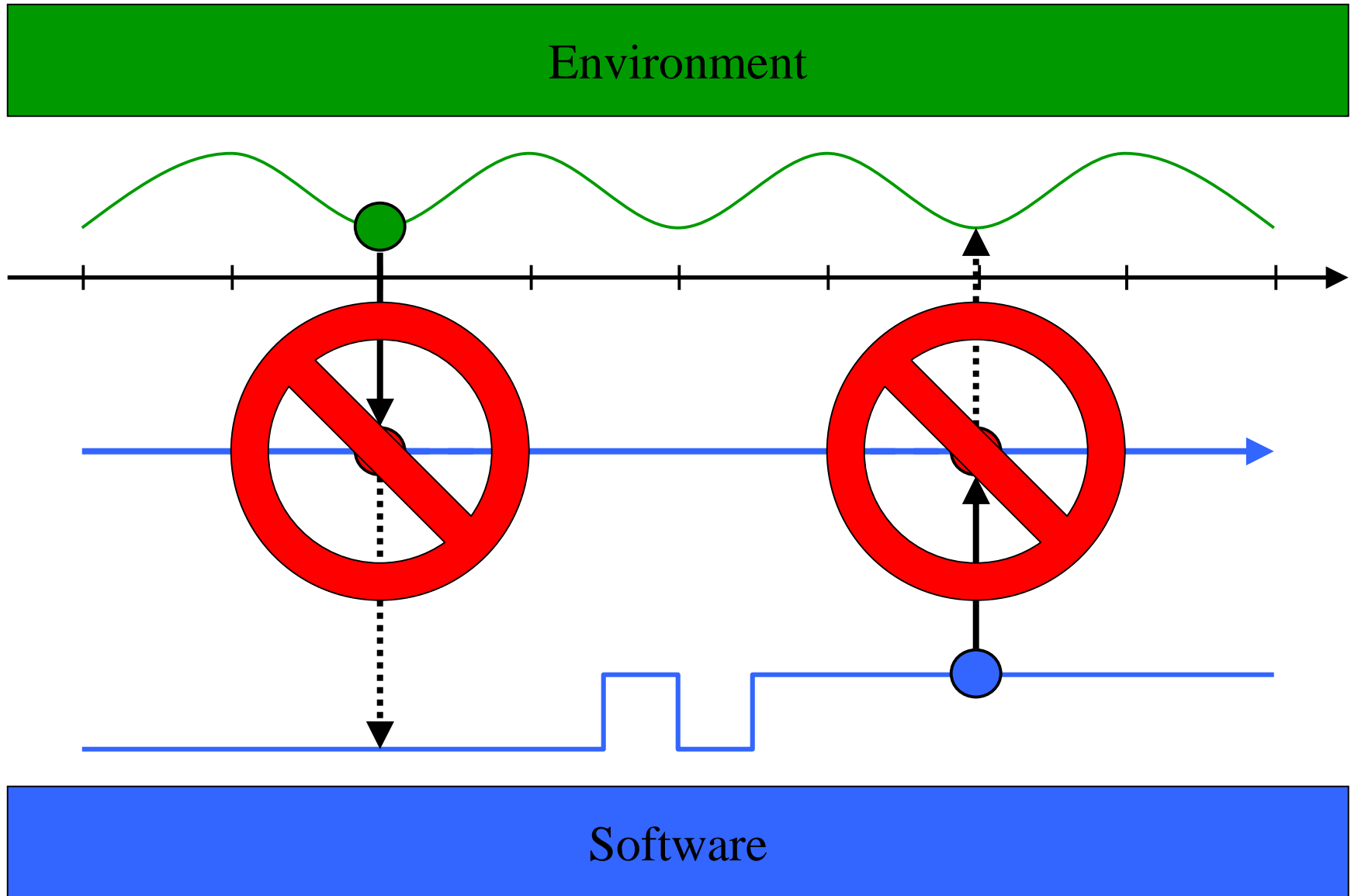


# Trigger $g$ : Input-, Environment-Triggered

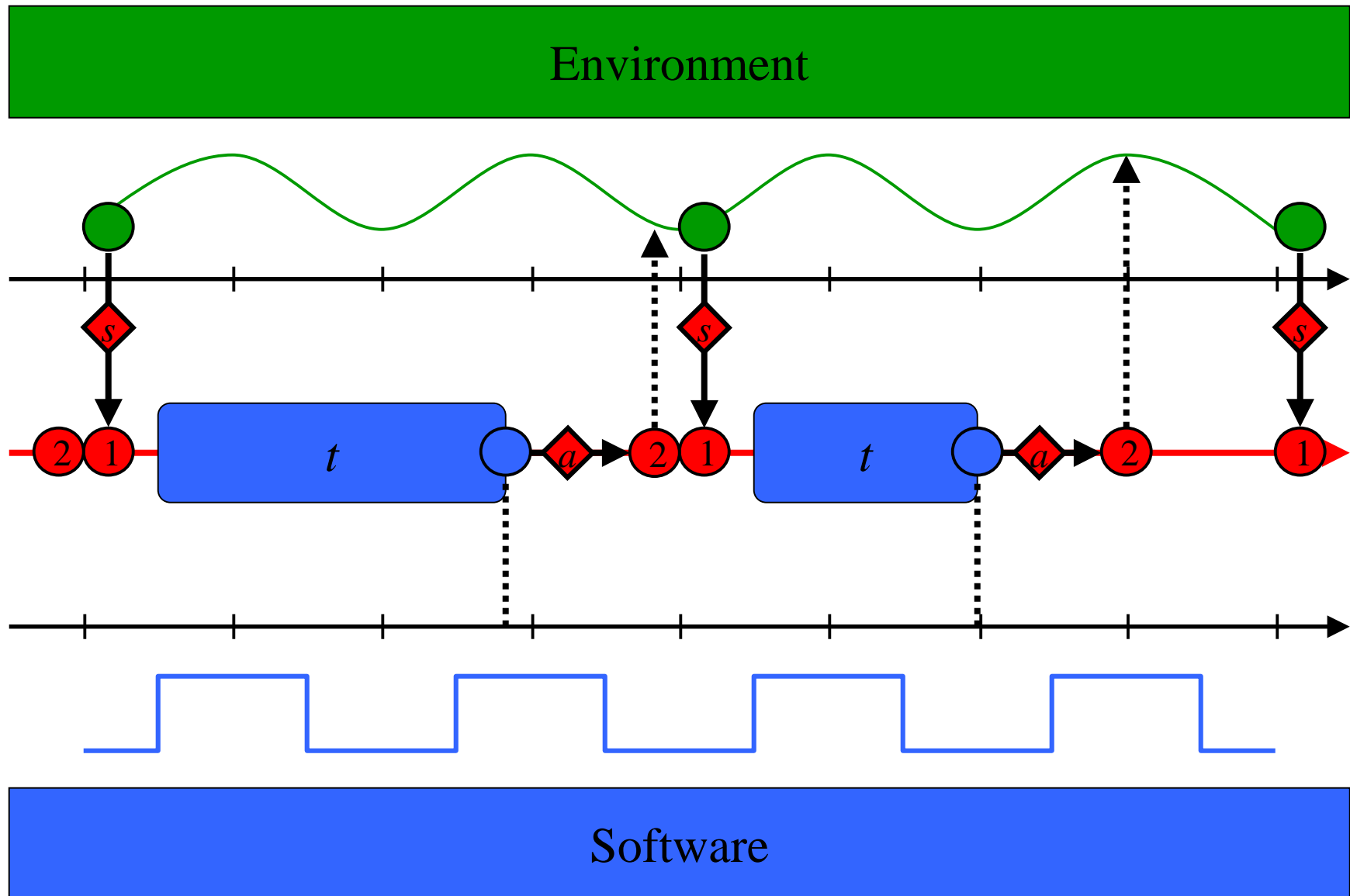




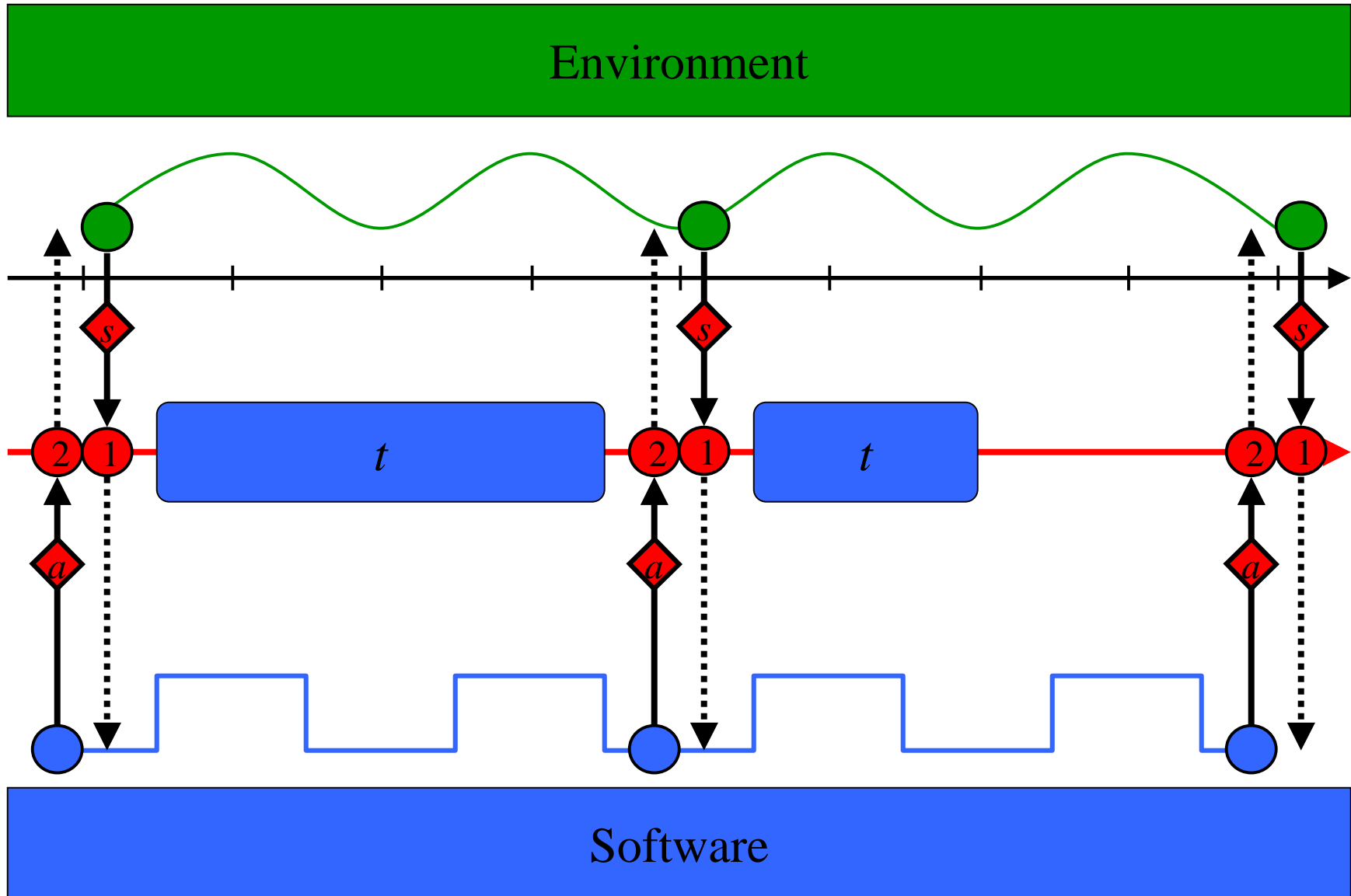
# Time Safety



# Input-deterministic If Time Safe



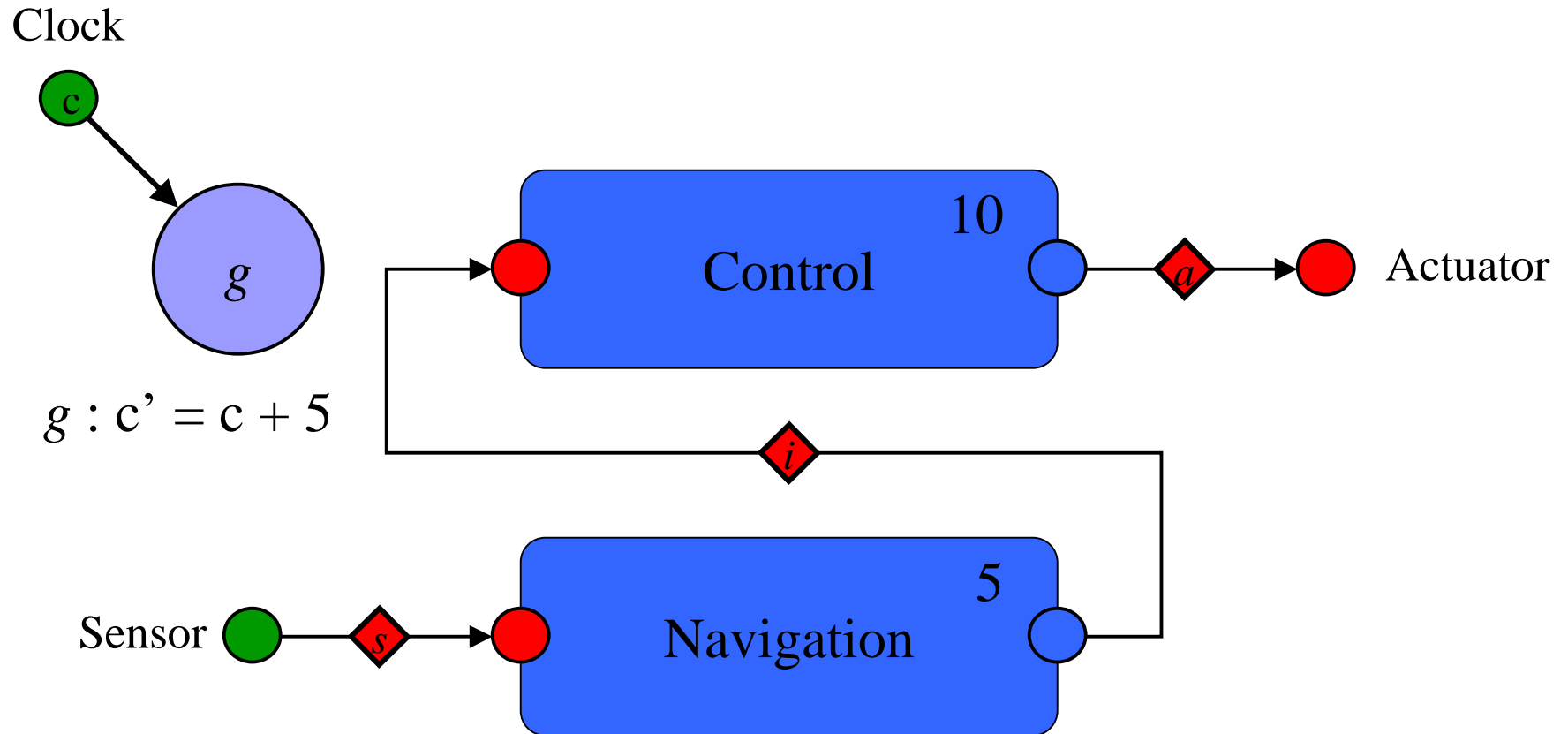
# Environment-deterministic If Environment-triggered



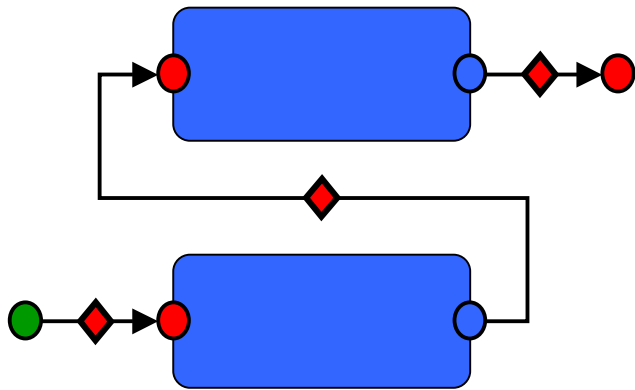
# The Zürich Helicopter



# Helicopter Control Software



# Giotto Syntax (Functionality)



```
sensor gps_type GPS uses c_gps_device ;  
actuator servo_type Servo := c_servo_init  
    uses c_servo_device ;
```

output

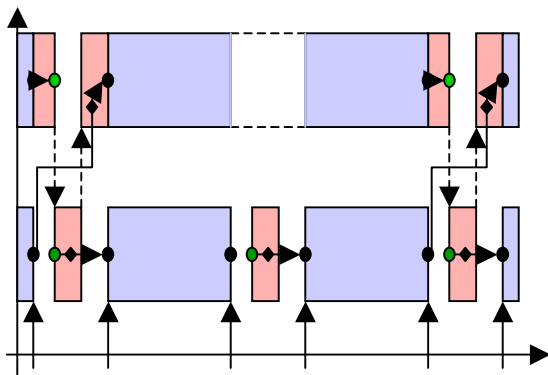
```
ctr_type CtrOutput := c_ctr_init ;  
nav_type NavOutput := c_nav_init ;
```

```
driver sensing (GPS) output (gps_type gps)  
{ c_gps_pre_processing ( GPS, gps ) }
```

```
task Navigation (gps_type gps) output (NavOutput)  
{ c_matlab_navigation_code ( gps, NavOutput ) }
```

...

# Giotto Syntax (Timing)



...

```
mode Flight ( ) period 10ms
```

```
{
```

```
actfreq 1 do Servo ( actuating ) ;
```

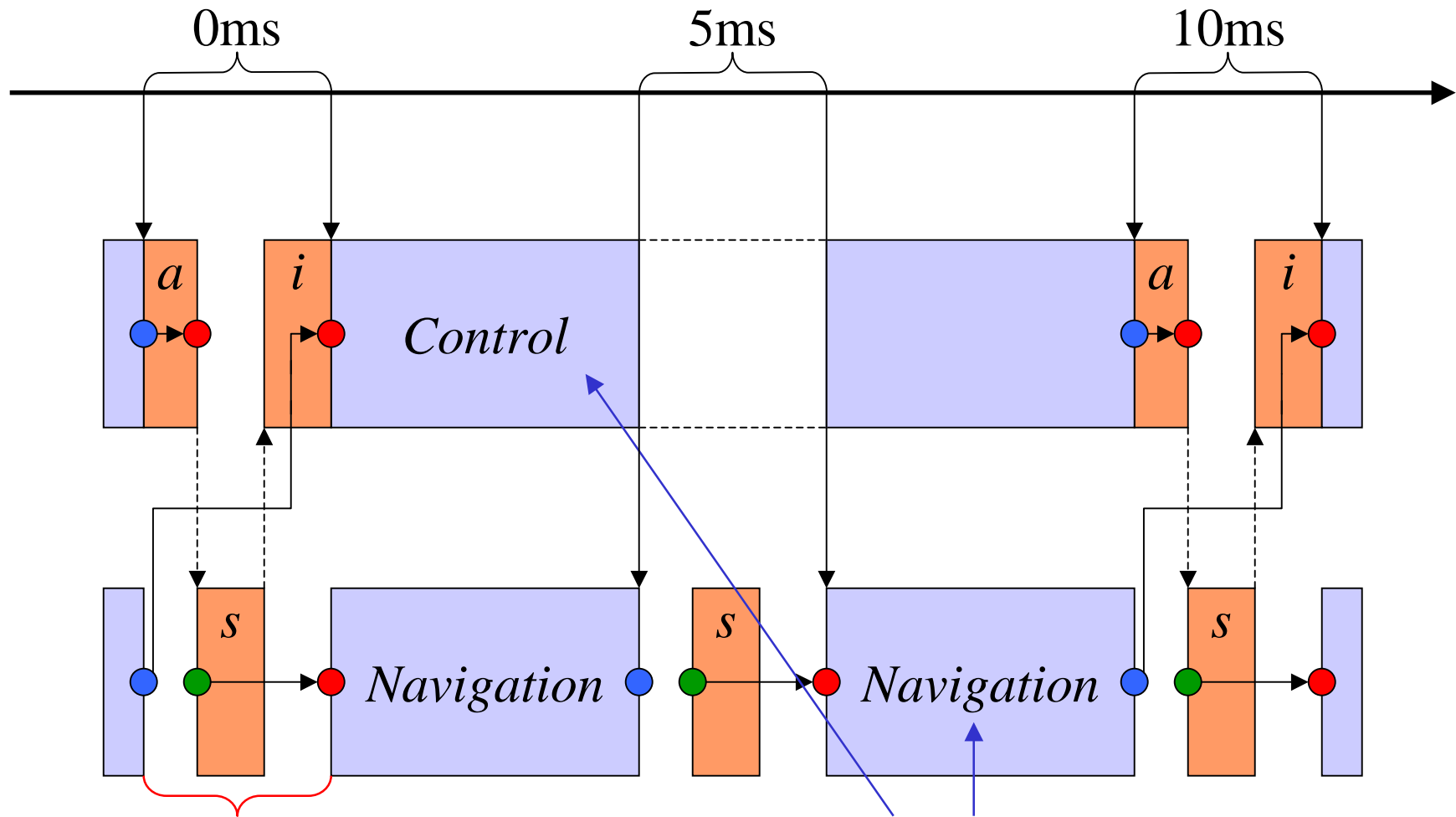
```
taskfreq 1 do Control ( input ) ;
```

```
taskfreq 2 do Navigation ( sensing ) ;
```

```
}
```

...

# Environment Timeline

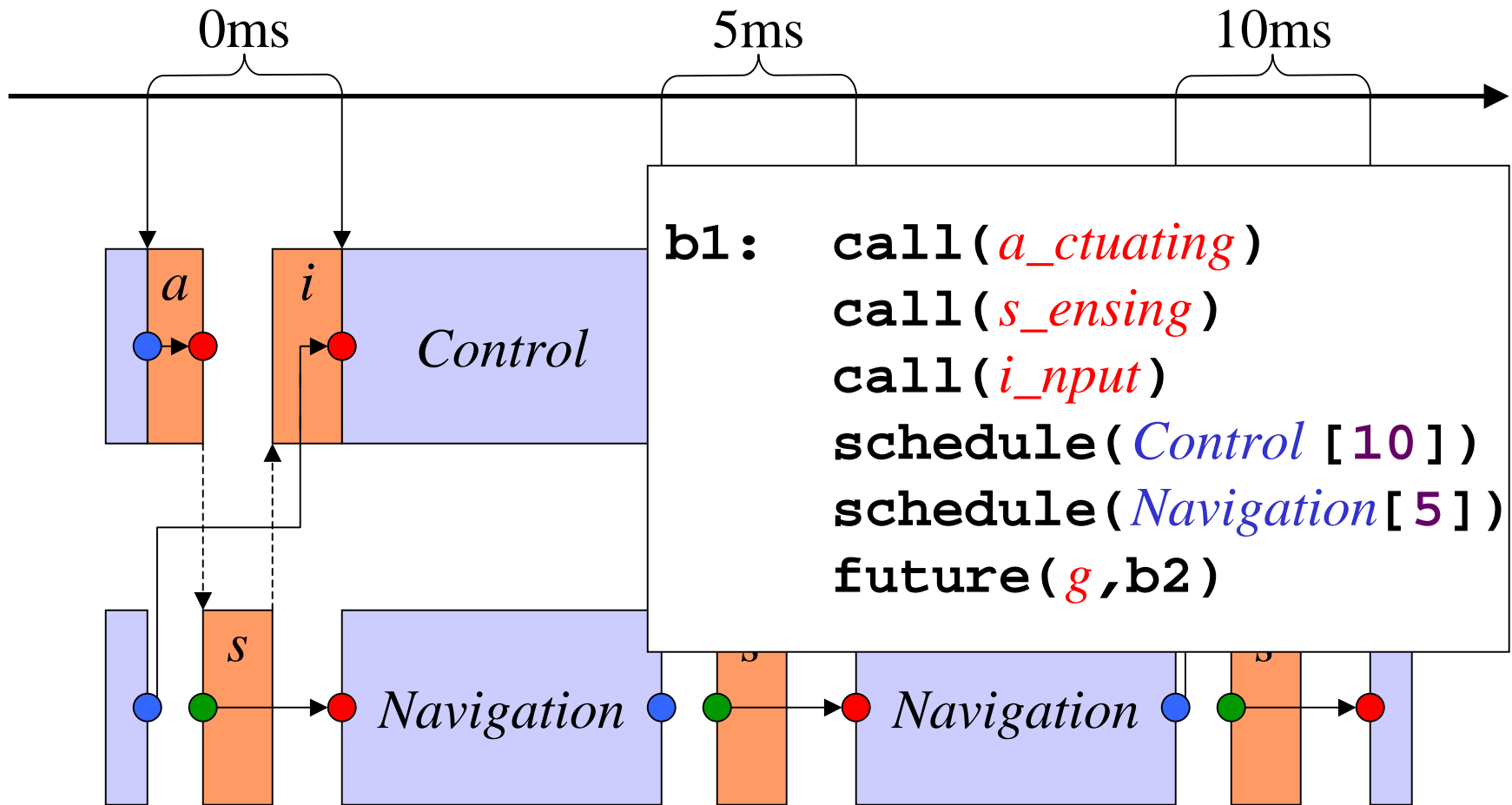


Block of synchronous code  
(nonpreemptable)

Scheduled tasks  
(preemptable)

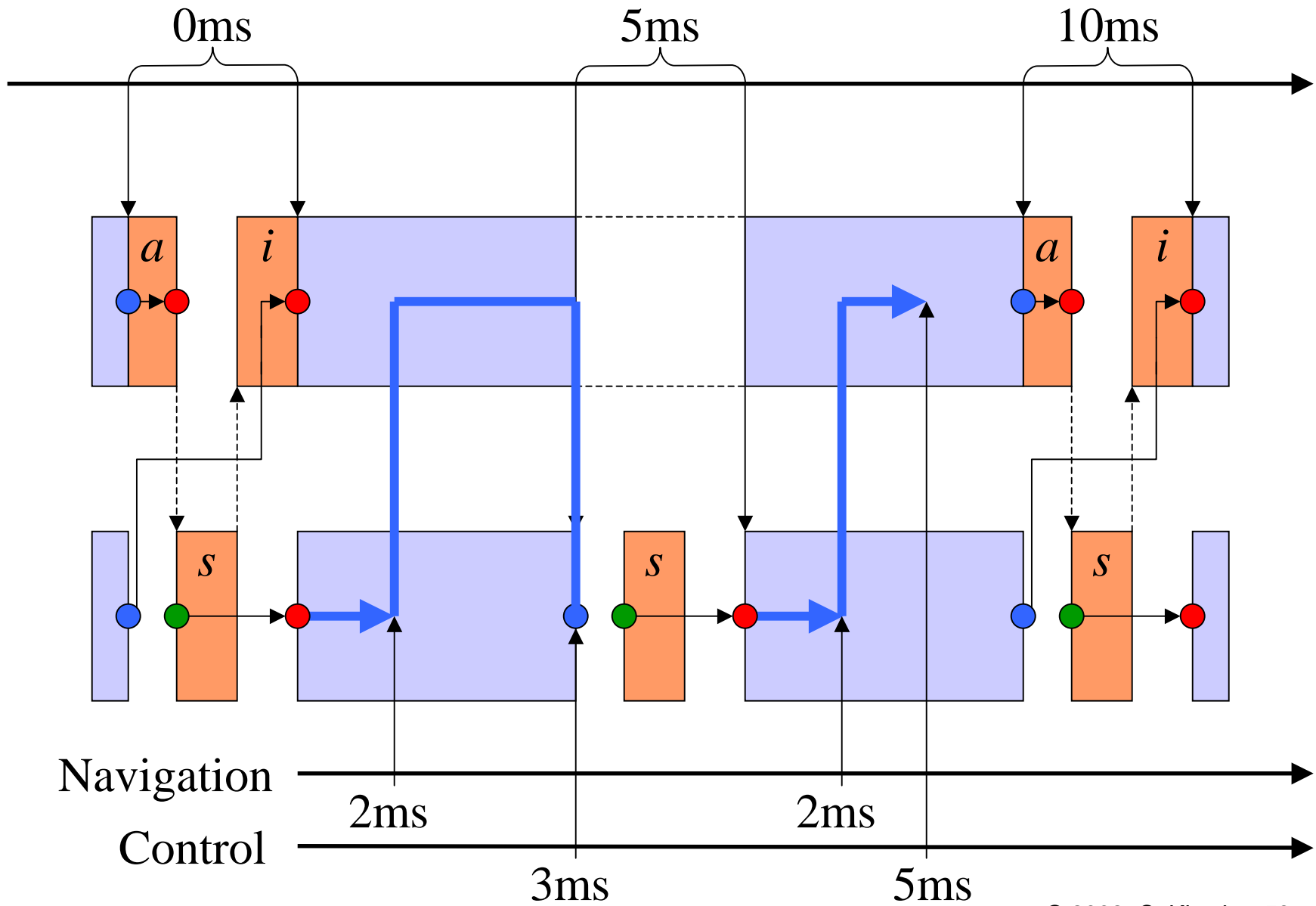


# E Code

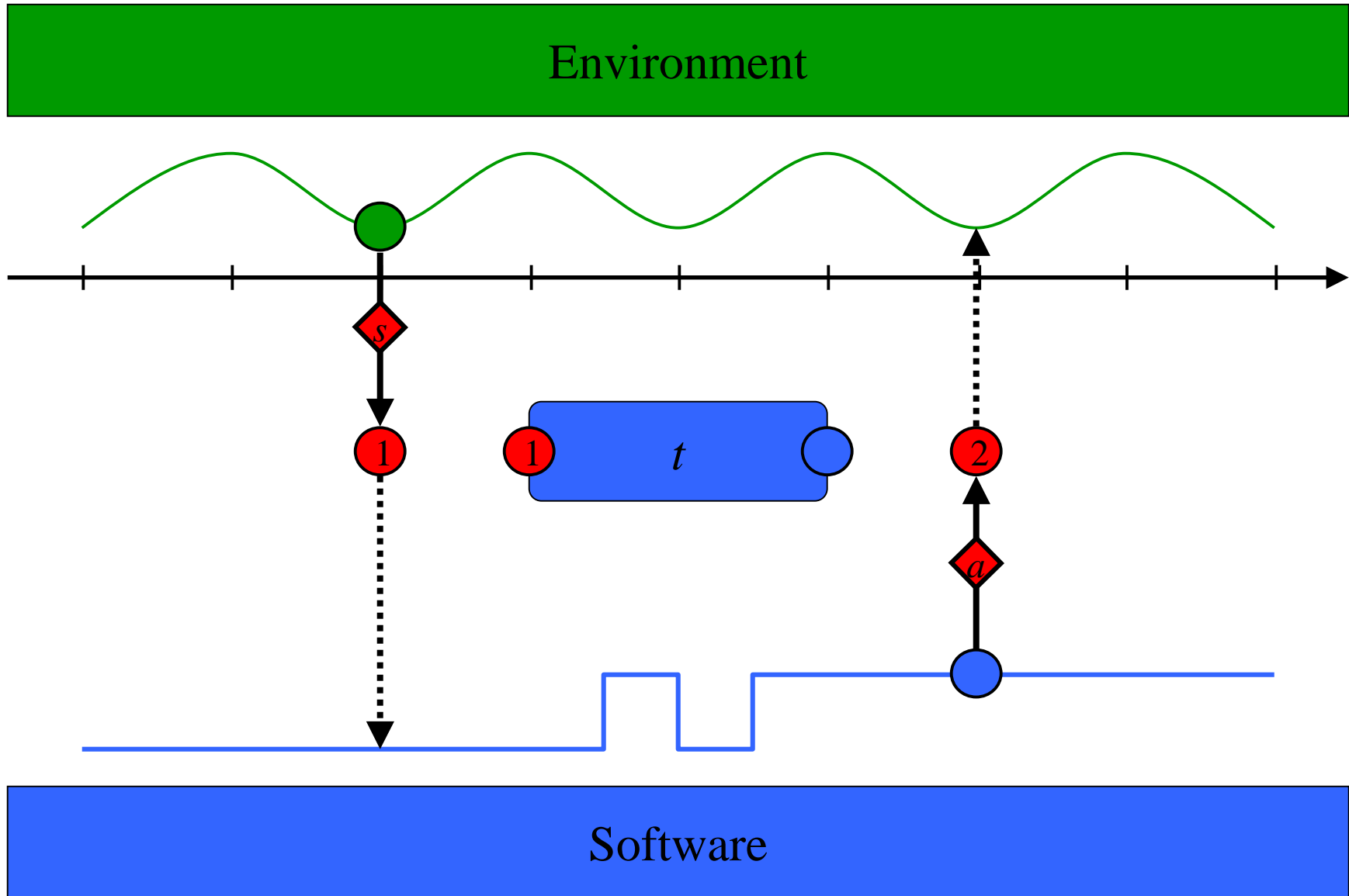




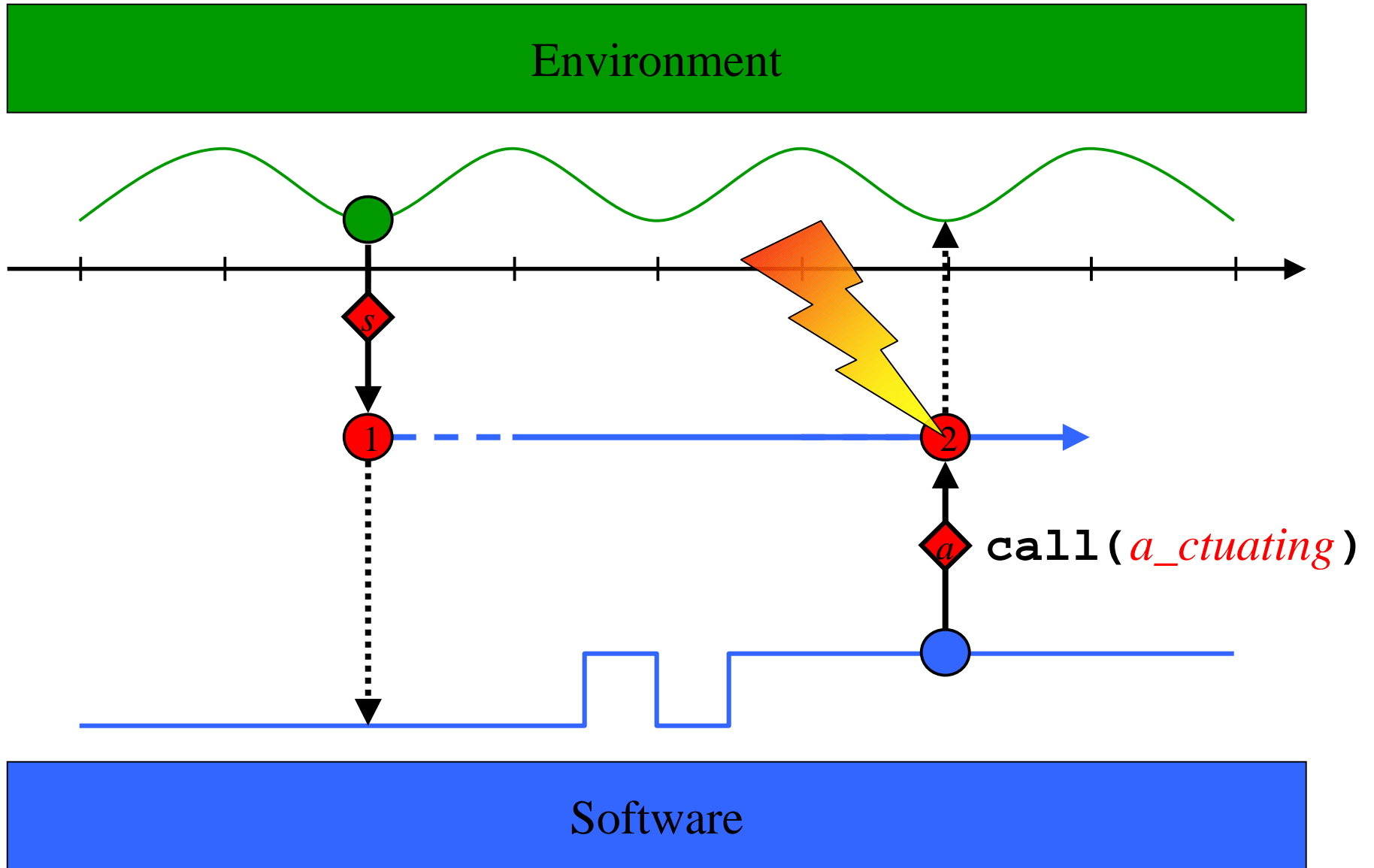
# Platform Timeline: EDF



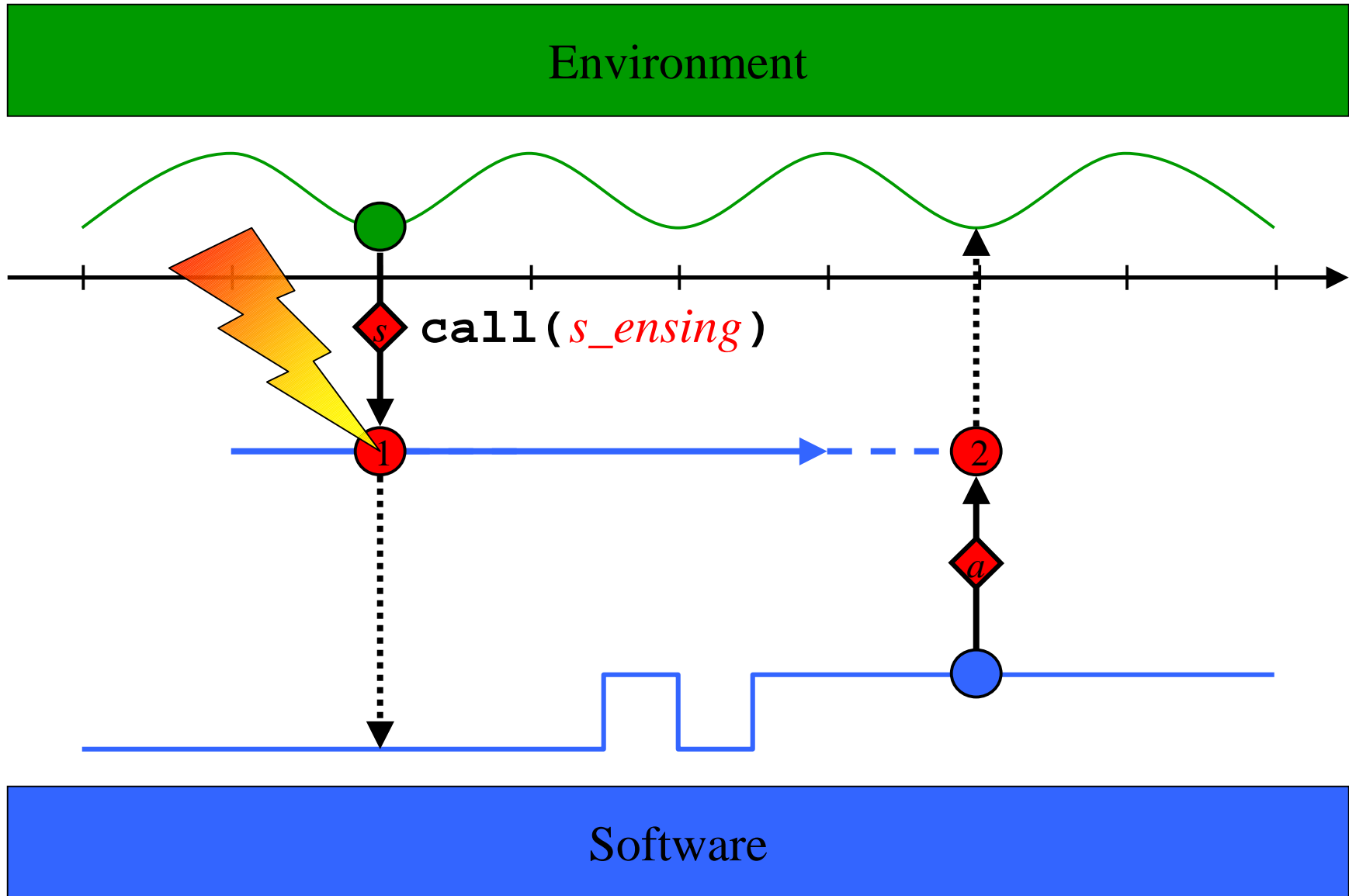
# Time Safety



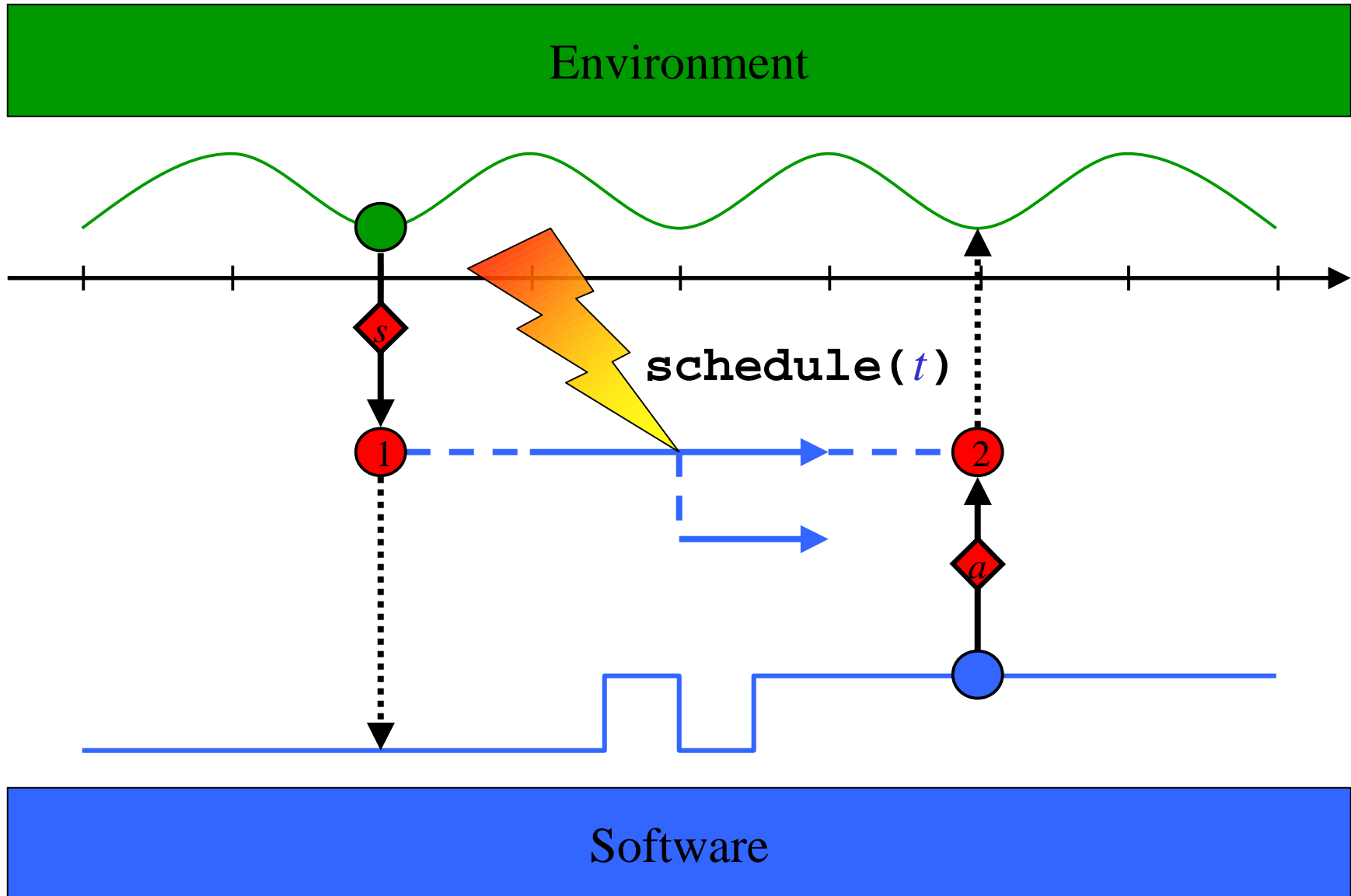
# Runtime Exceptions I



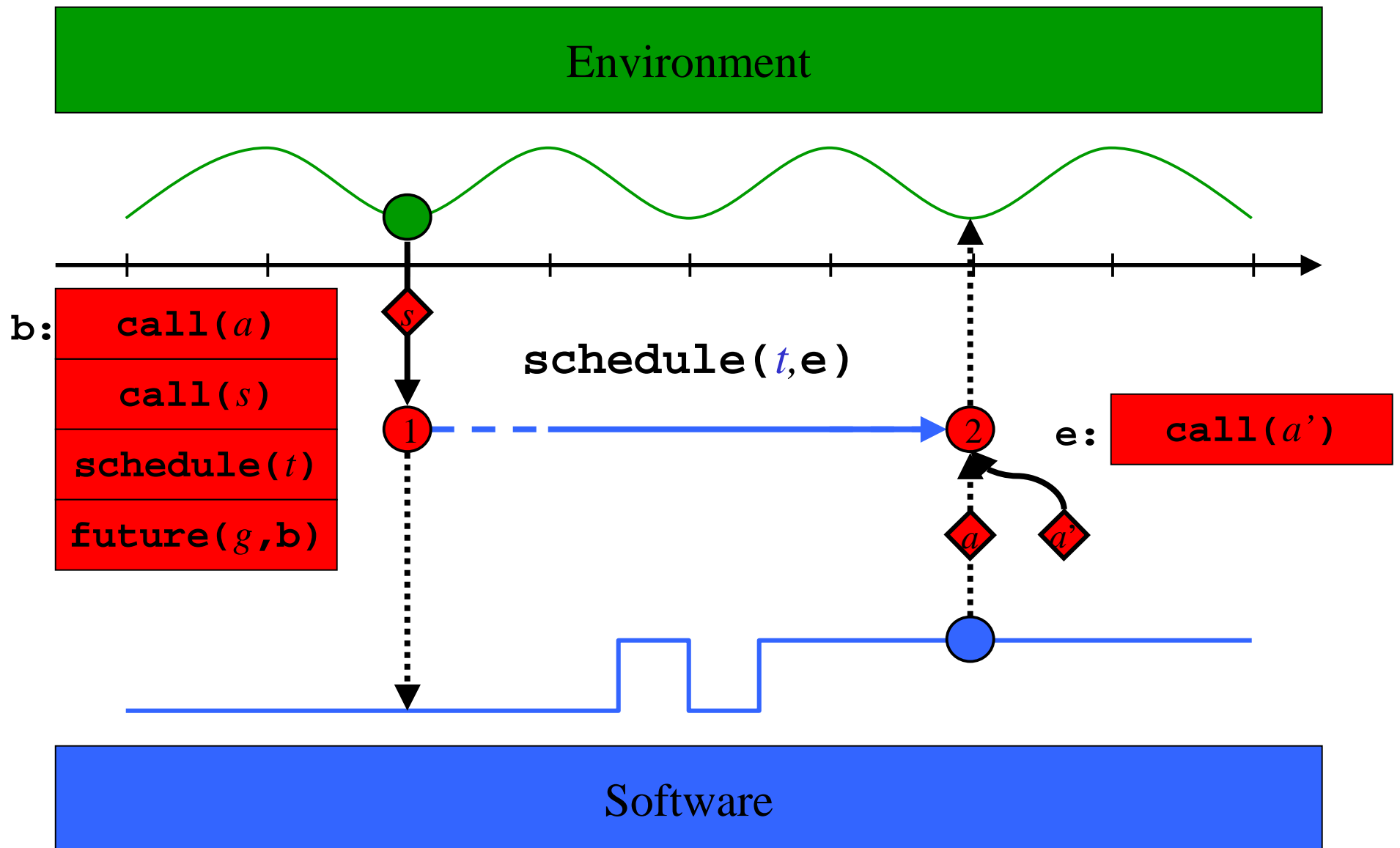
# Runtime Exceptions II



# Runtime Exceptions III

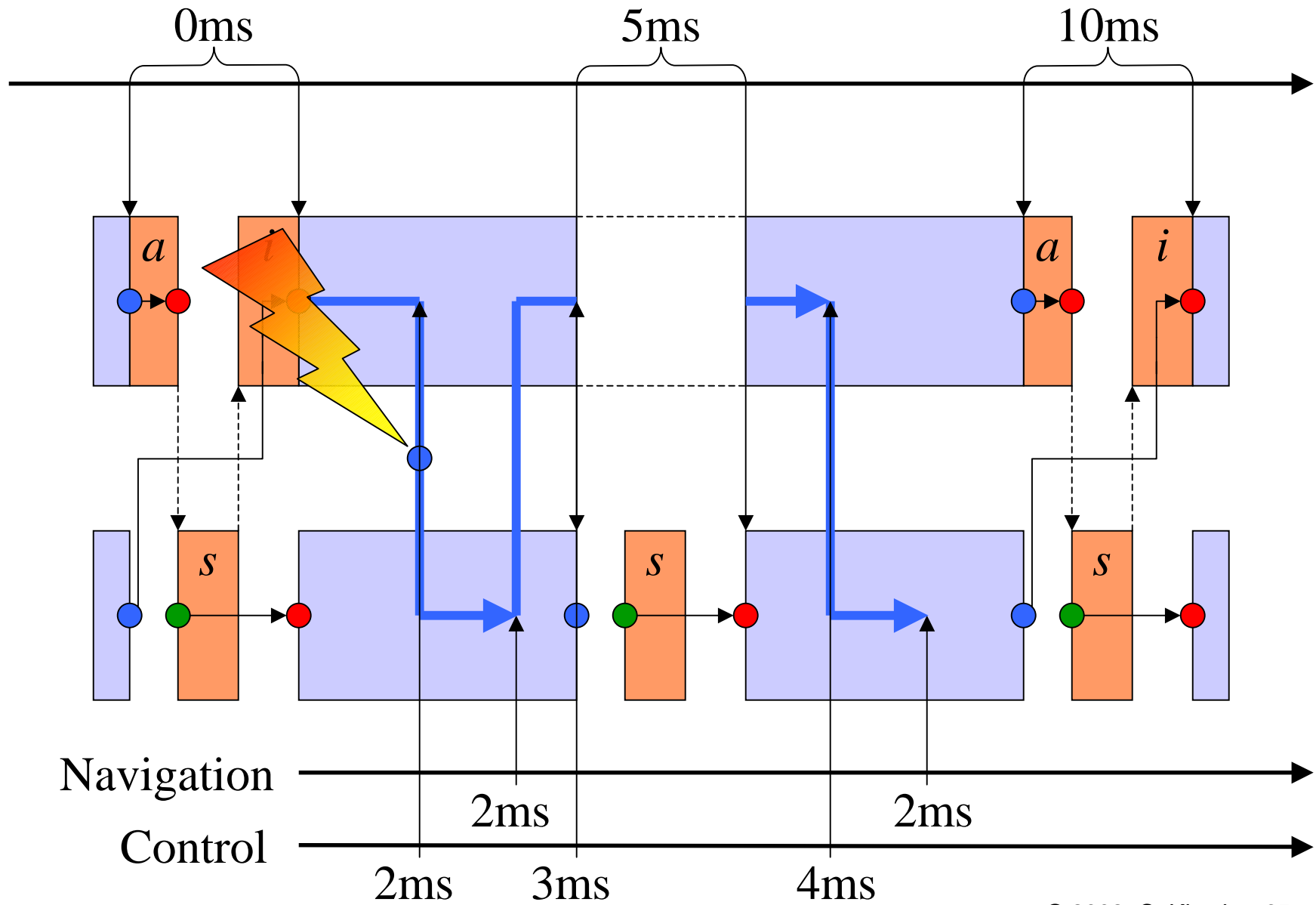


# An Exception Handler $e$

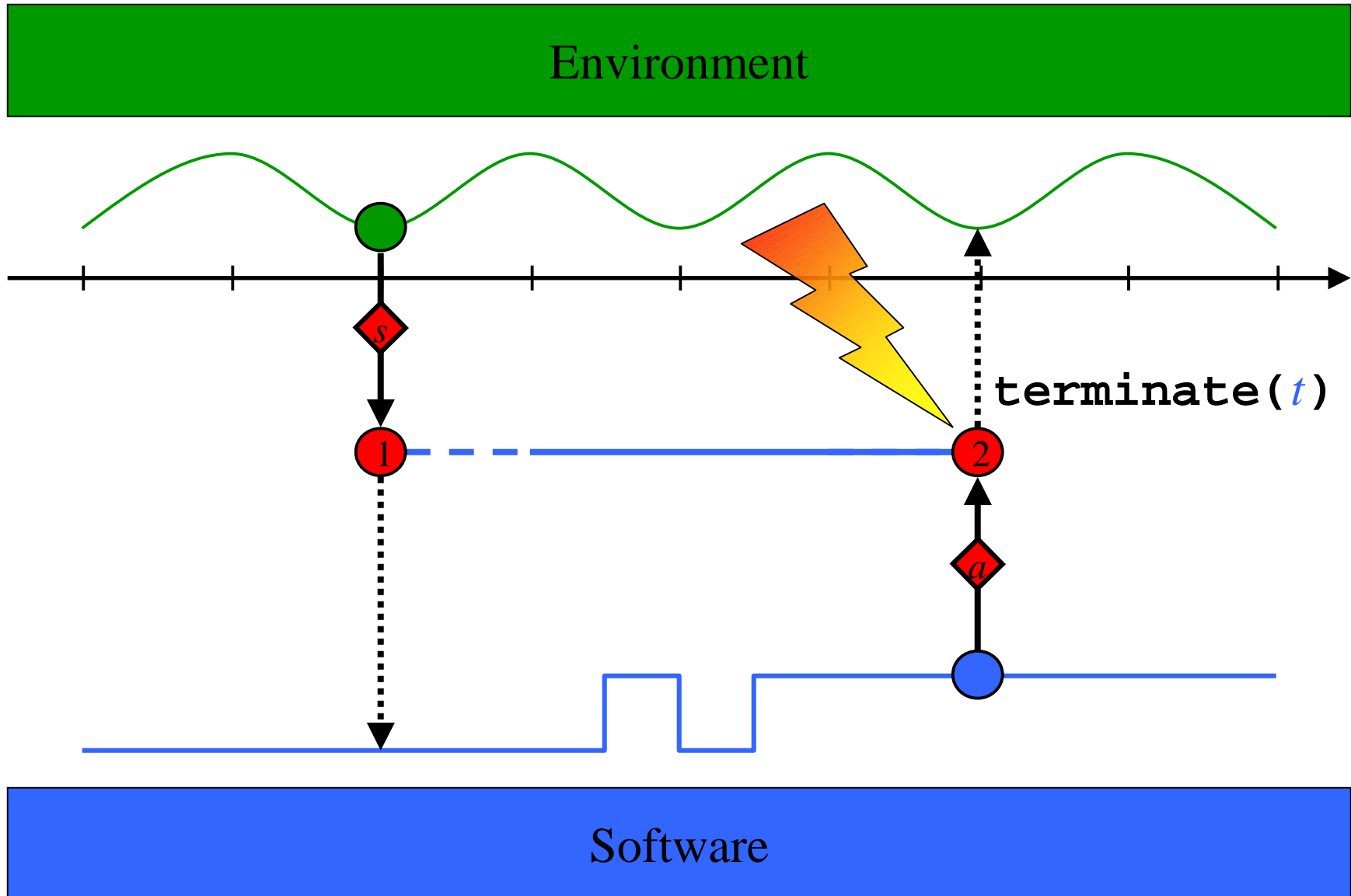




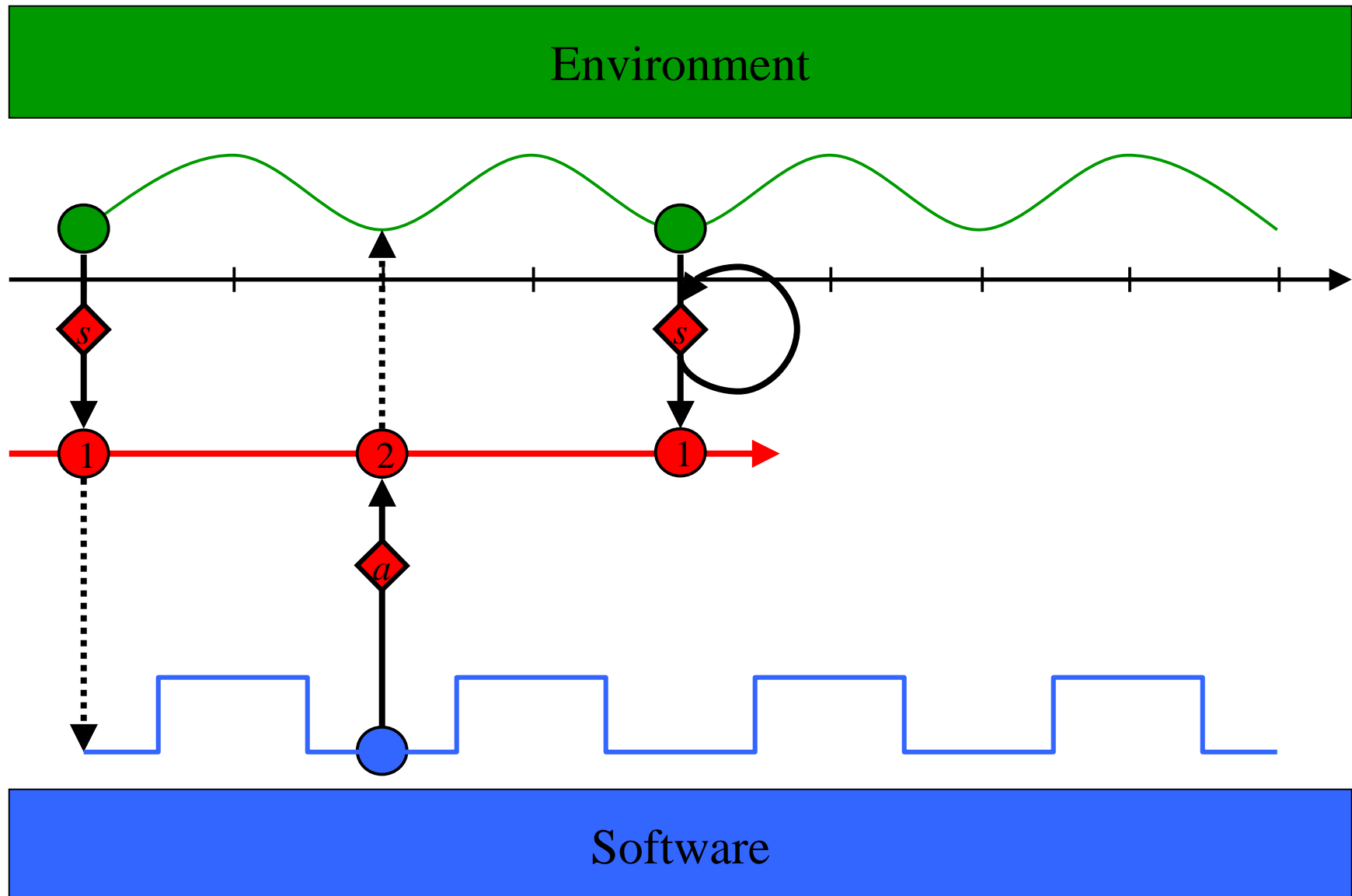
# How to Loose Determinism: Task Synchronization



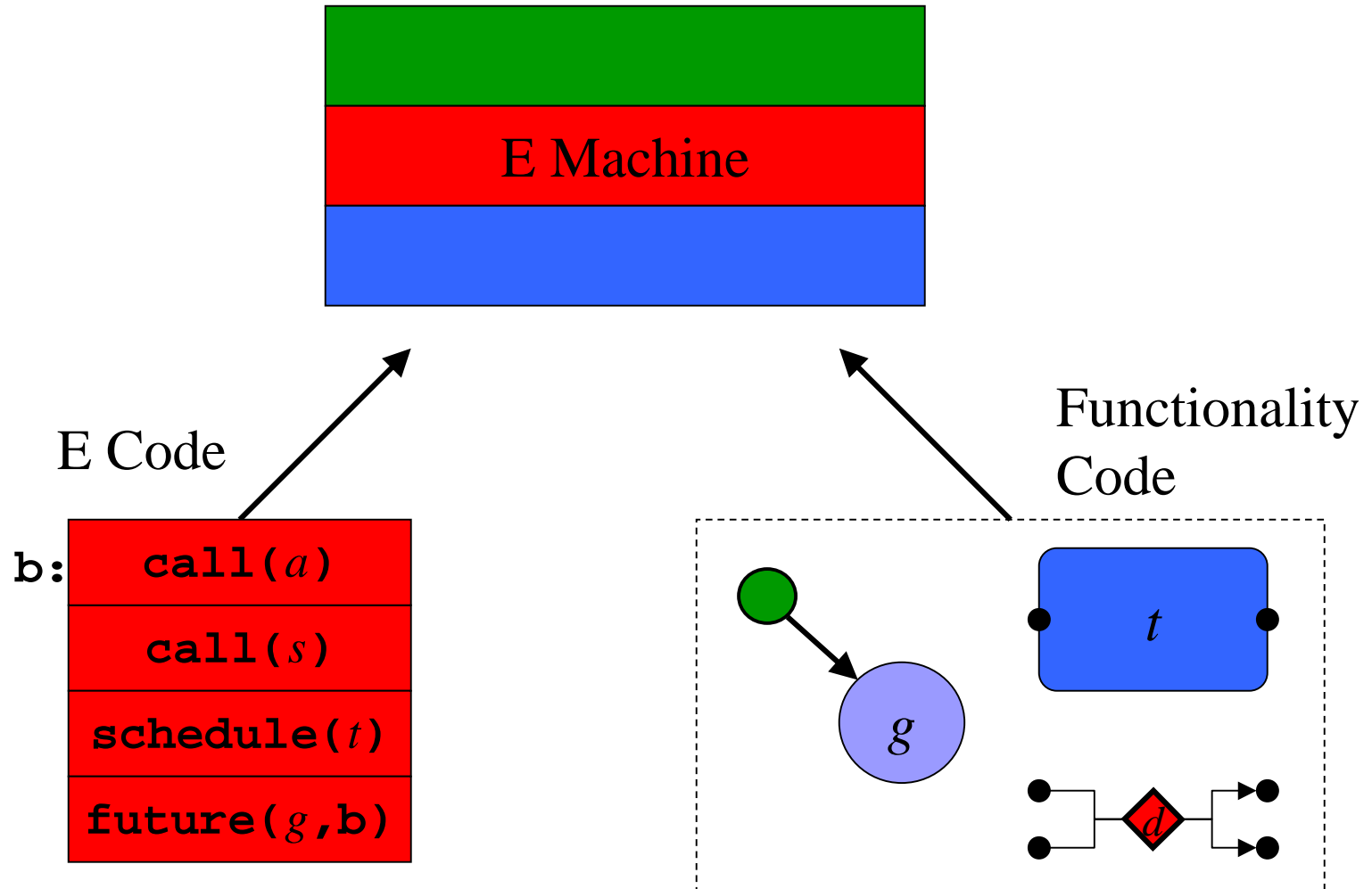
# How to Loose Determinism: Termination



# Time Liveness: Infinite Traces



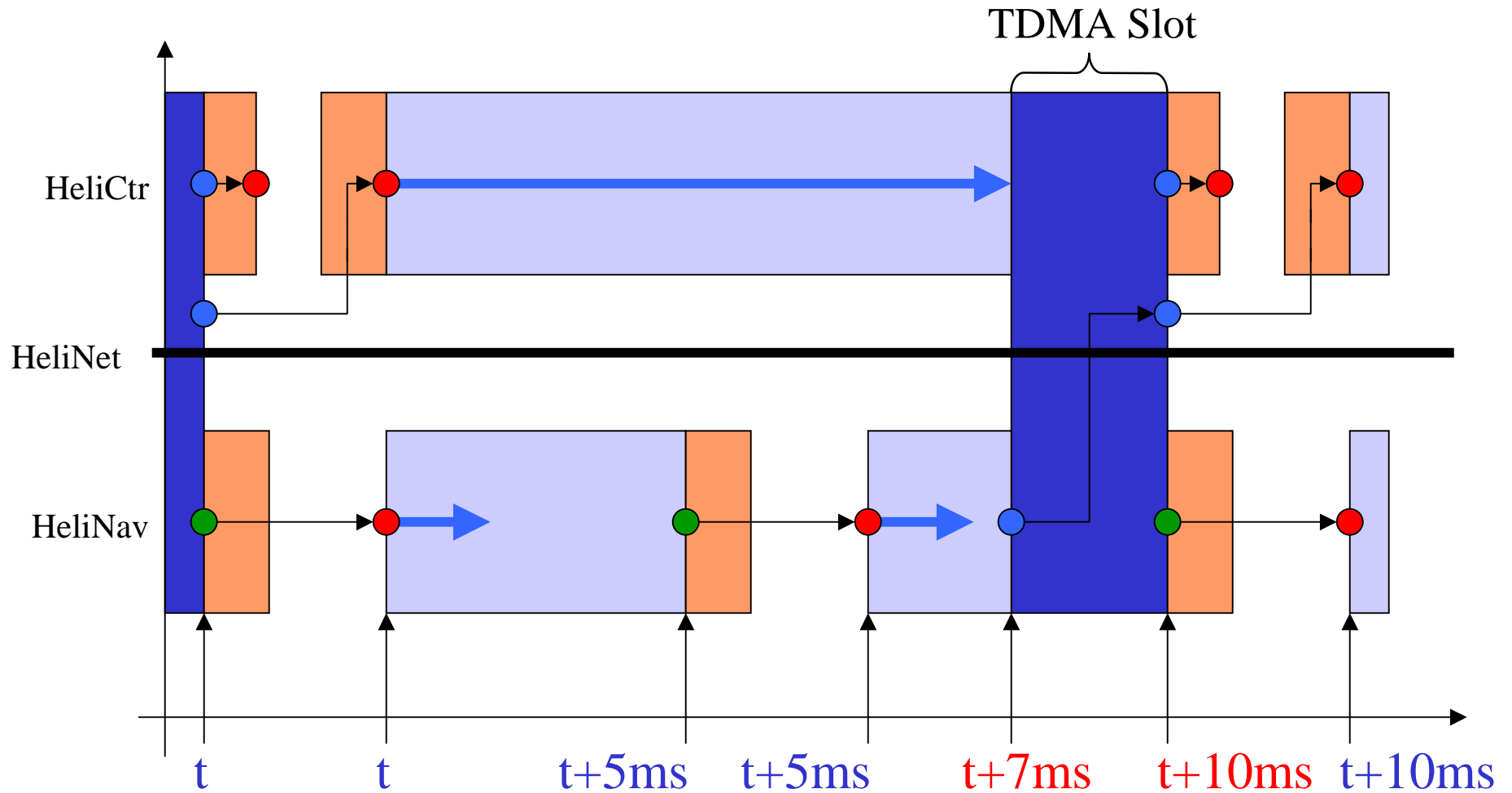
# Dynamic Linking



# The Berkeley Helicopter



# Platform Timeline: Time-triggered Communication





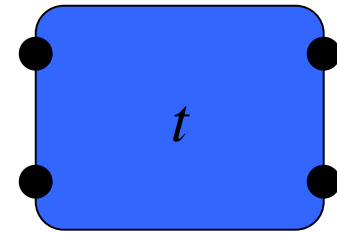
# Instructions

Synchronous  
Driver:



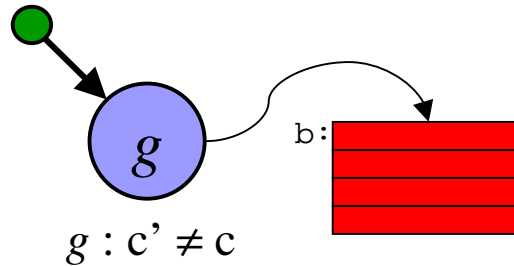
**call(*d*)**

Scheduled  
Task:



**schedule(*t*)**

Triggering:



**future(*g*,*b*)**