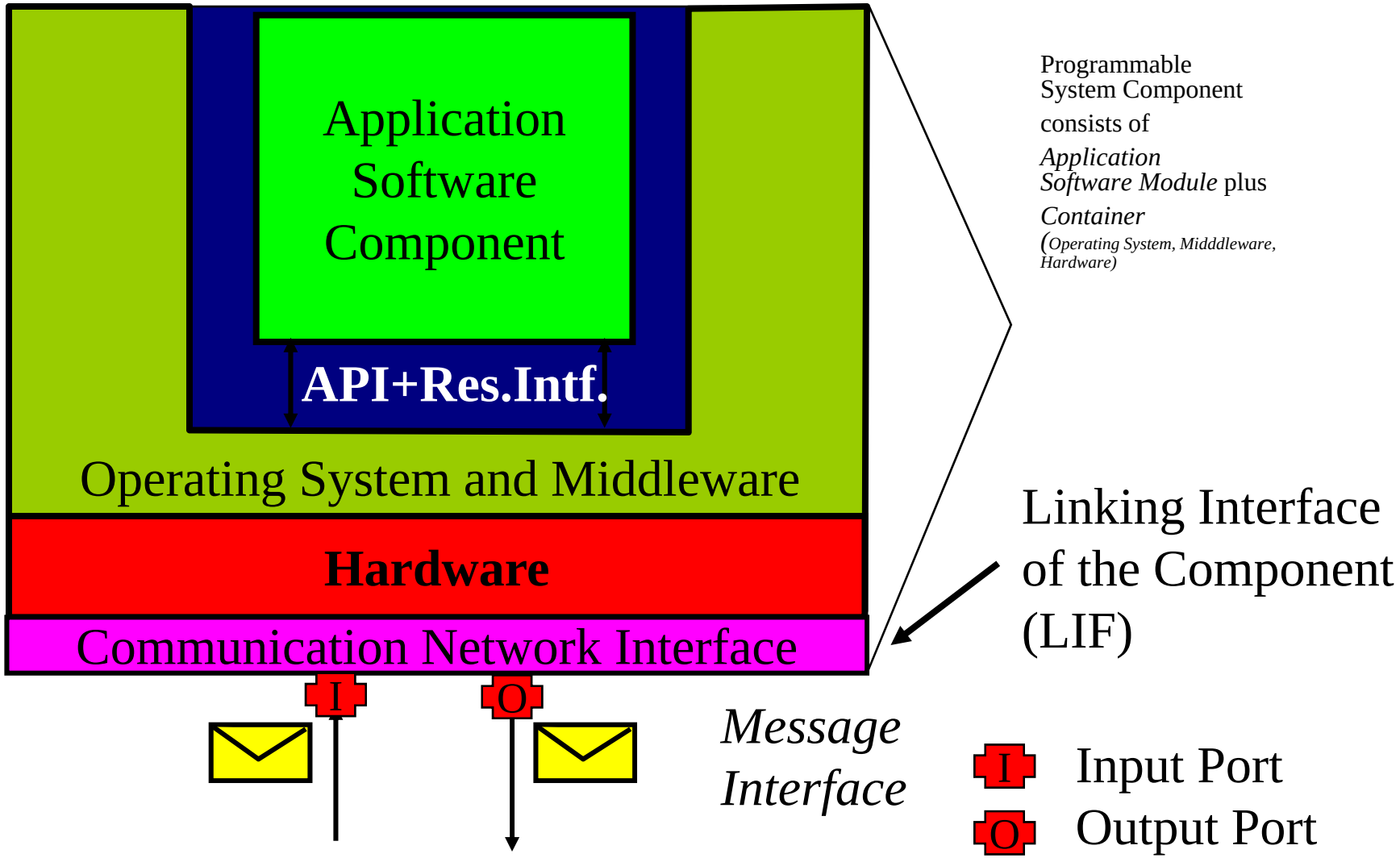
Real-Time Operating Systems

Outline

- Task Structure
- Resource Management
- Input/Output
- Scheduling
- The Priority Ceiling Protocol

Operating System and Middleware



ARINC Standard WG 48-1999

The Avionics Computing Resource (ACR) shall include internal hardware and software management methods as necessary to ensure that time, space and I/O allocations are deterministic and static. "Deterministic and static" means in this context, that time, space and I/O allocations are determined at compilation, assembly or link time, remain identical at each and every initialization of a program or process, and are not dynamically altered during runtime.

S-Tasks versus C-Task

The software of a component is structured into a set of tasks (execution of a sequential program) that can be executed in parallel. The operating system must provide the execution environment for each task.

We distinguish between

- ***Simple tasks (S-Task),*** that execute from the beginning to the end without any delay, given the CPU has been allocated.
- ***Complex tasks (C-Task),*** that may contain a *WAIT* statement in the task body.

Task Structure -- TT

Basically the task structure in a TT system is static. There are some techniques to make the task structure data dependent, but they are limited:

- Mode changes--navigate dynamically between statically validated operating modes
- Sporadic server tasks: Provide a laxity in the schedule that can be consumed by a sporadic server task
- Precedence graphs with exclusive or: select dynamically one of a number of excluding alternatives (not very effective!)

This limited data dependency of the task structure is at the same time the big advantage and the big disadvantage of TT systems.

Resource Management--TT

In a time-triggered operating system there is hardly any dynamic resource management.

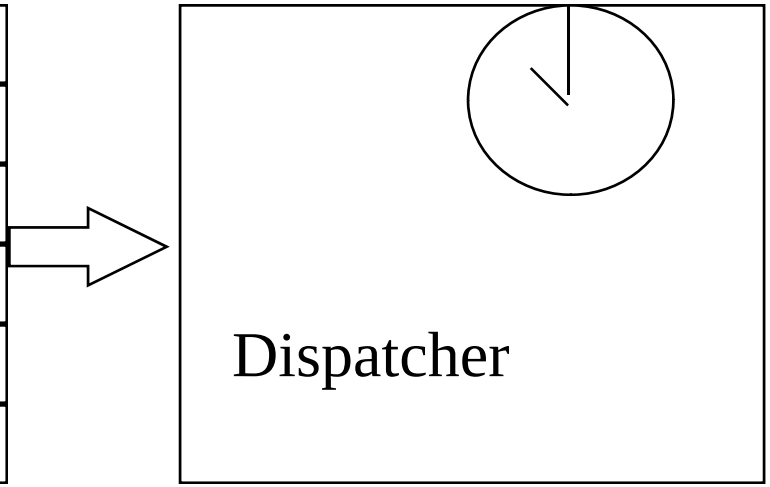
- CPU allocation is static
- Memory management autonomous and hidden behind the memory abstraction. It demands little attention from the operating system.
- Buffer management is non existent
- Explicit synchronization, including the queue management of semaphore queues, does not exist.

Operating systems become simple and can be formally analyzed: examples: TTOS, OSEK time

Task Control -- TT

In TT systems, the task control consists of a dispatcher that performs

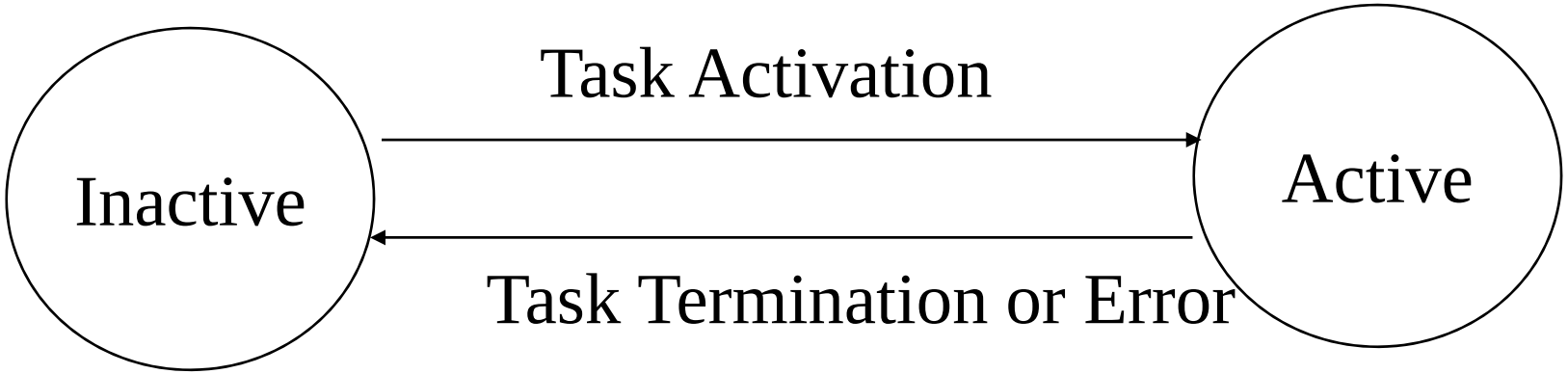
Time	Action
10	Start T1
17	Send M5
22	Stop T1
38	Start T3
47	Send M3



The tables are generated and checked before run time by a static scheduler.

Task States-TT

In a non-preemptive TT system:

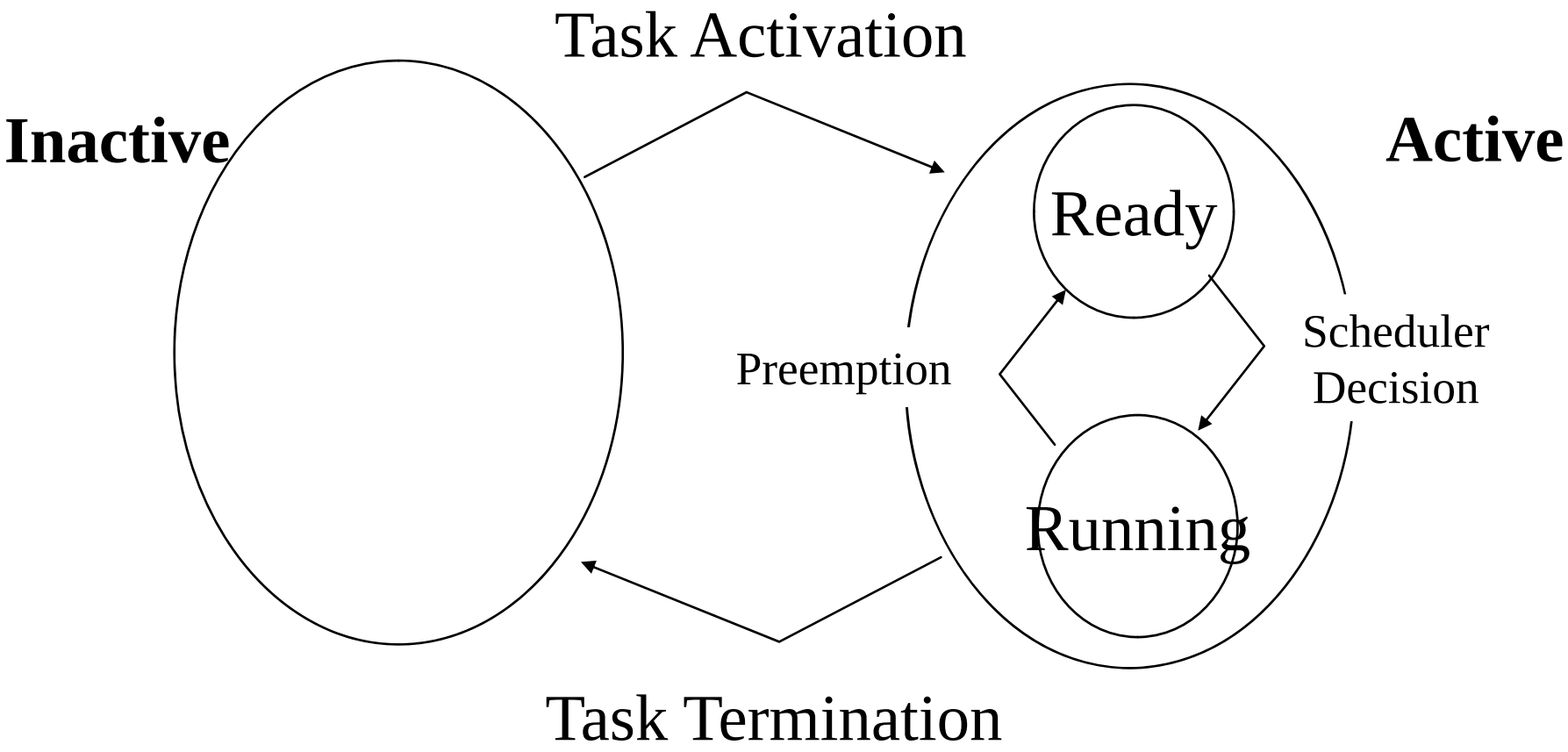


Task Control -- ET with S-Tasks

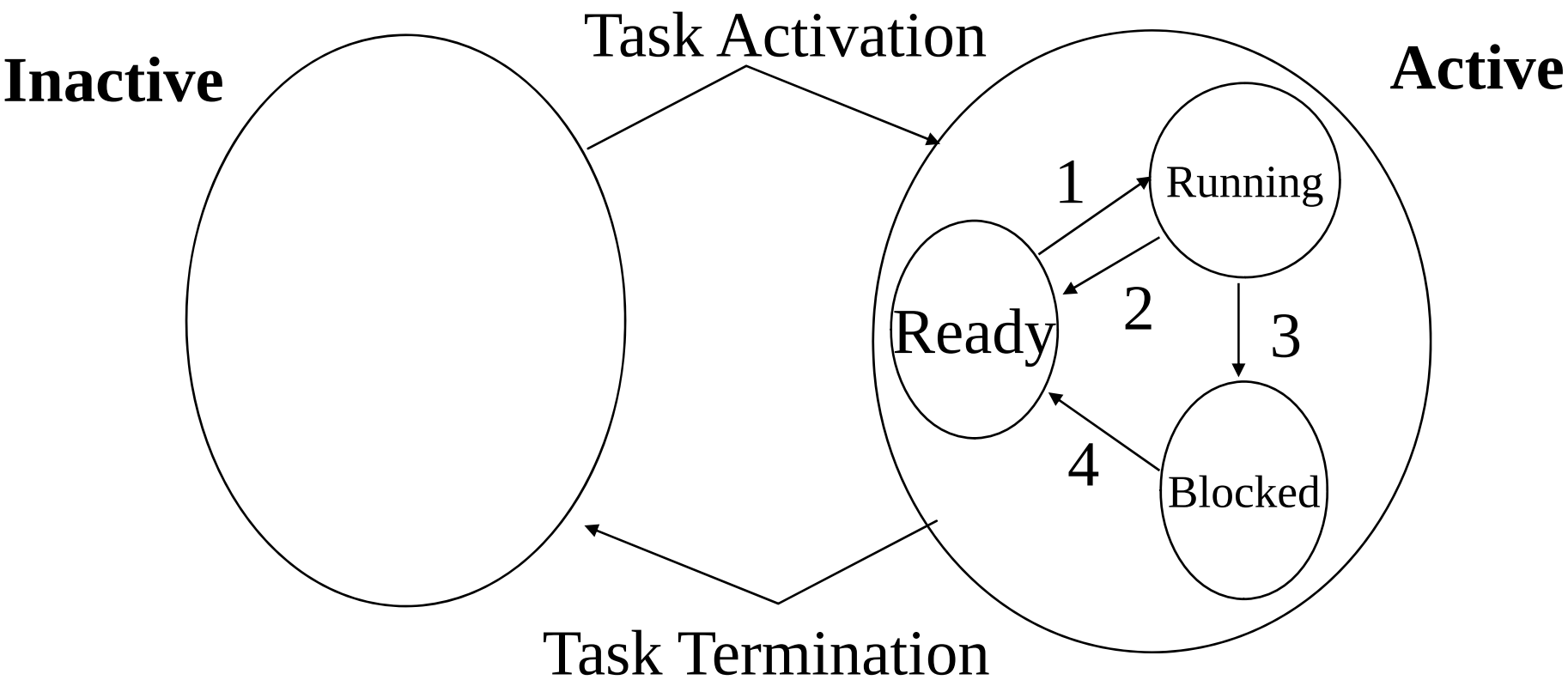
In an ET system, the task control is performed by a dynamic scheduler that decides which task has to be executed next on the basis of the evolving request scenario.

- Advantage:
  Actual (and not maximum) load and task execution times form the basis of the scheduling decisions
- Disadvantage:
  In most realistic cases the scheduling problem, that has to be solved on-line, is NP hard.

Task States--ET with S - Tasks



Task States: ET with C-Tasks



1 Scheduler Decision
2 Task Preemption
3 Task executes WAIT for Event
4 Blocking Event occurs

Resource Management

In ET OS the dynamic resource management is extensive:

- Dynamic CPU allocation (next lecture)
- Dynamic memory management
- Dynamic Buffer allocation and event driven management of the communication activities
- Explicit synchronisation between tasks including semaphore queue management and deadlock detection.
- Extensive interrupt management

It is beyond the state of the art to formally analyse the timing of ET operating systems (e.g., OSEK).

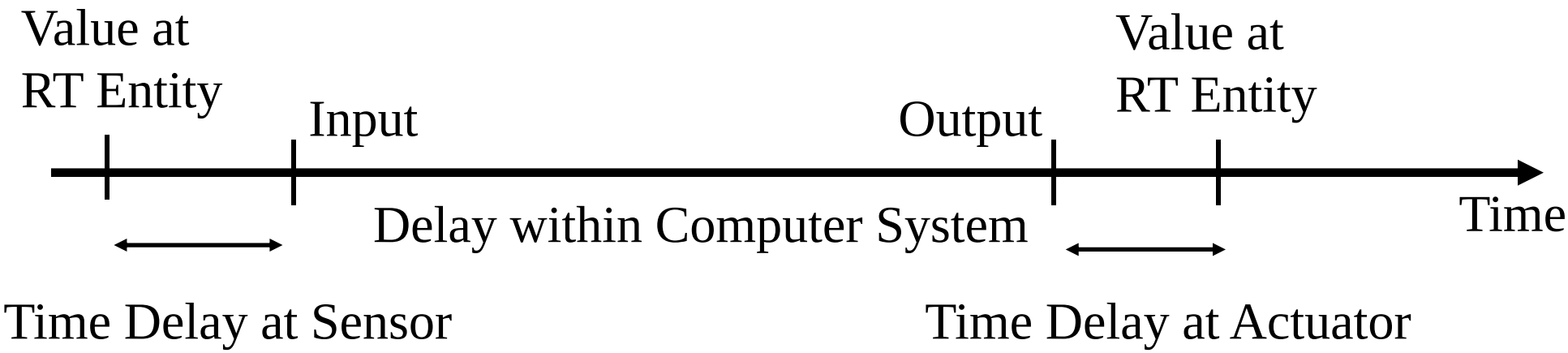
Time Services

Clock Synchronization

Time services:

- Specification of a potentially infinite sequence of events at absolute time-points (off-line and on-line).
- Specification of a future point in time within a specified temporal distance from "now" (timeout service)
- Time stamping of events immediately after their occurrence
- Output of a message (or a control signal) at a precisely defined point in time in the future, either relative to "now" or at an absolute future time point
- Gregorian calendar function to convert TAI (UTC) to calender time and vice versa

Timing at an I/O Interface



The Dual Role of Time

A significant event that happens in the environment of a real-time computer can be seen from two different perspectives:

- It defines the point in time of a value change of a RT entity. The precise knowledge of this point in time is an important input for the later analysis of the consequences of the event (time as data)

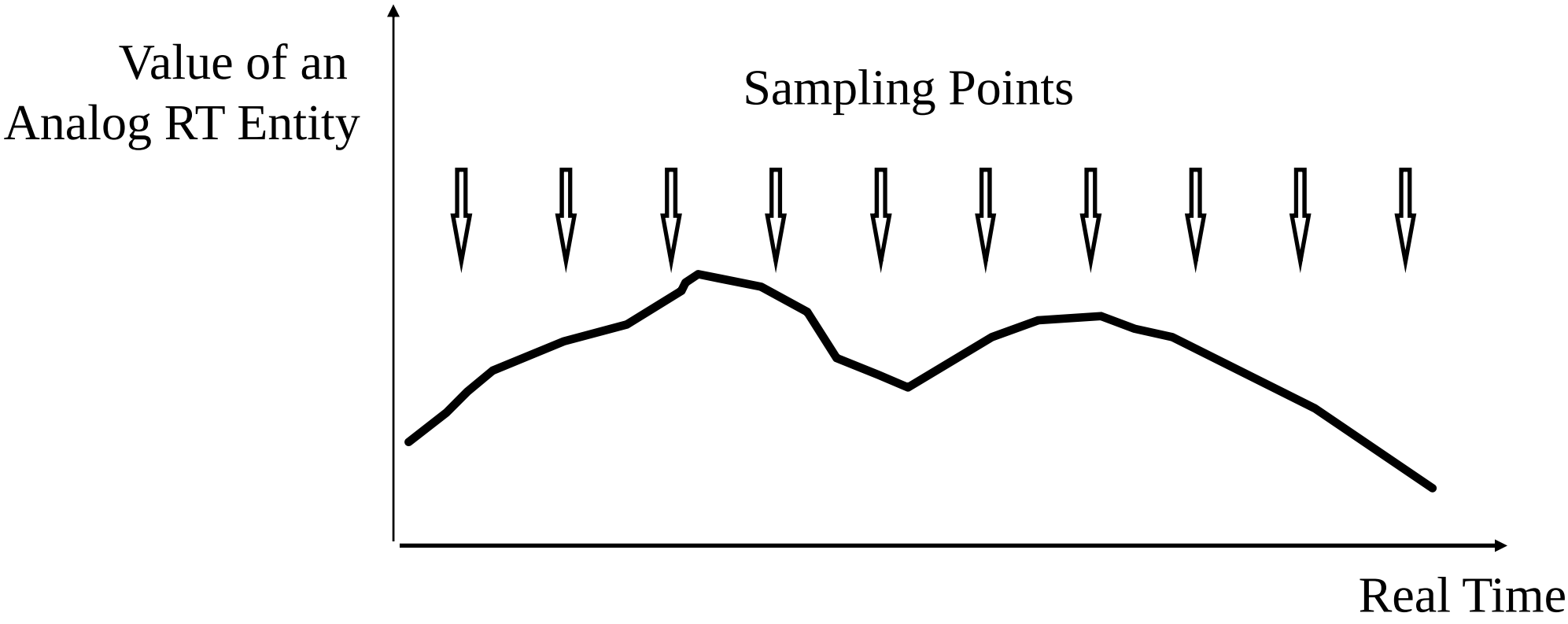
  Example: Downhill skiing

- It may demand immediate action by the computer system to react as soon as possible to this event (time as control).

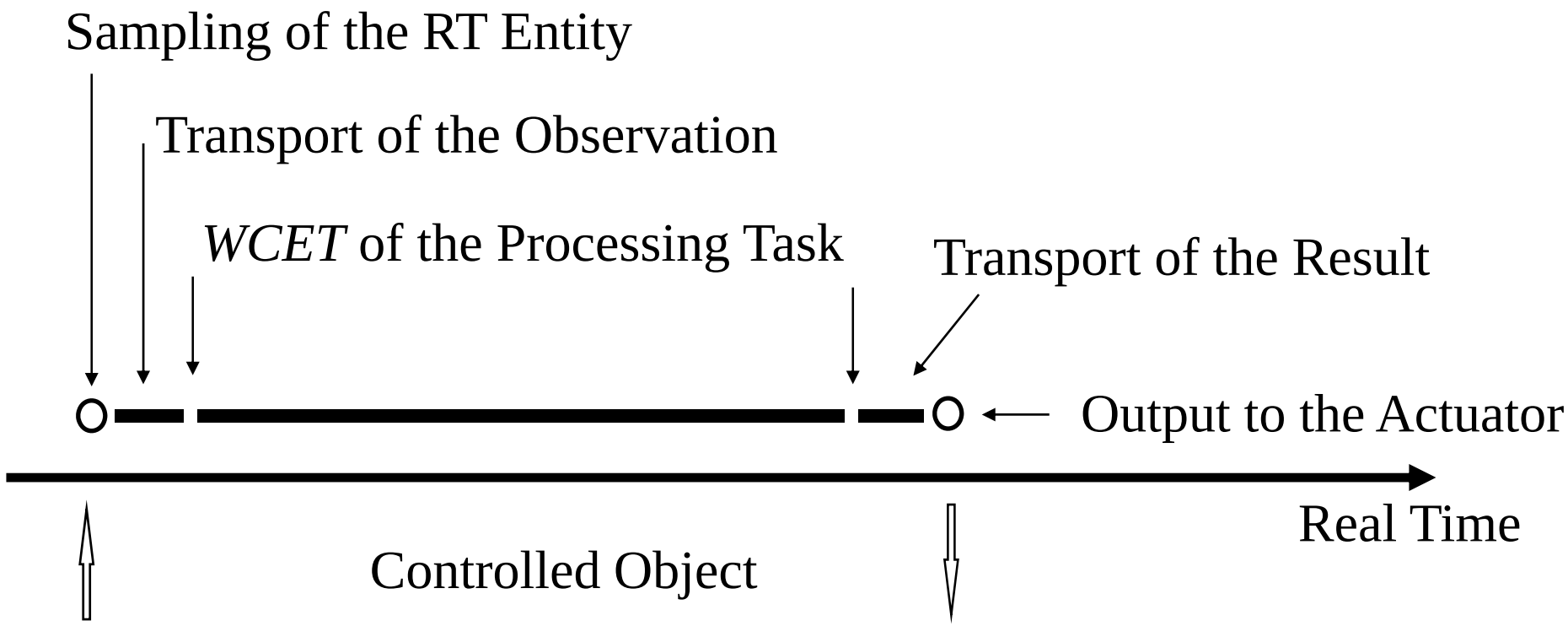
  Example: Emergency stop

It is much more demanding to implement *time as control* than to implement *time as data*!

Sampling



Timing in a Sampled System



Sampling--States versus Events

Sampling refers to the periodic interrogation of the state of a RT entity by a computer.

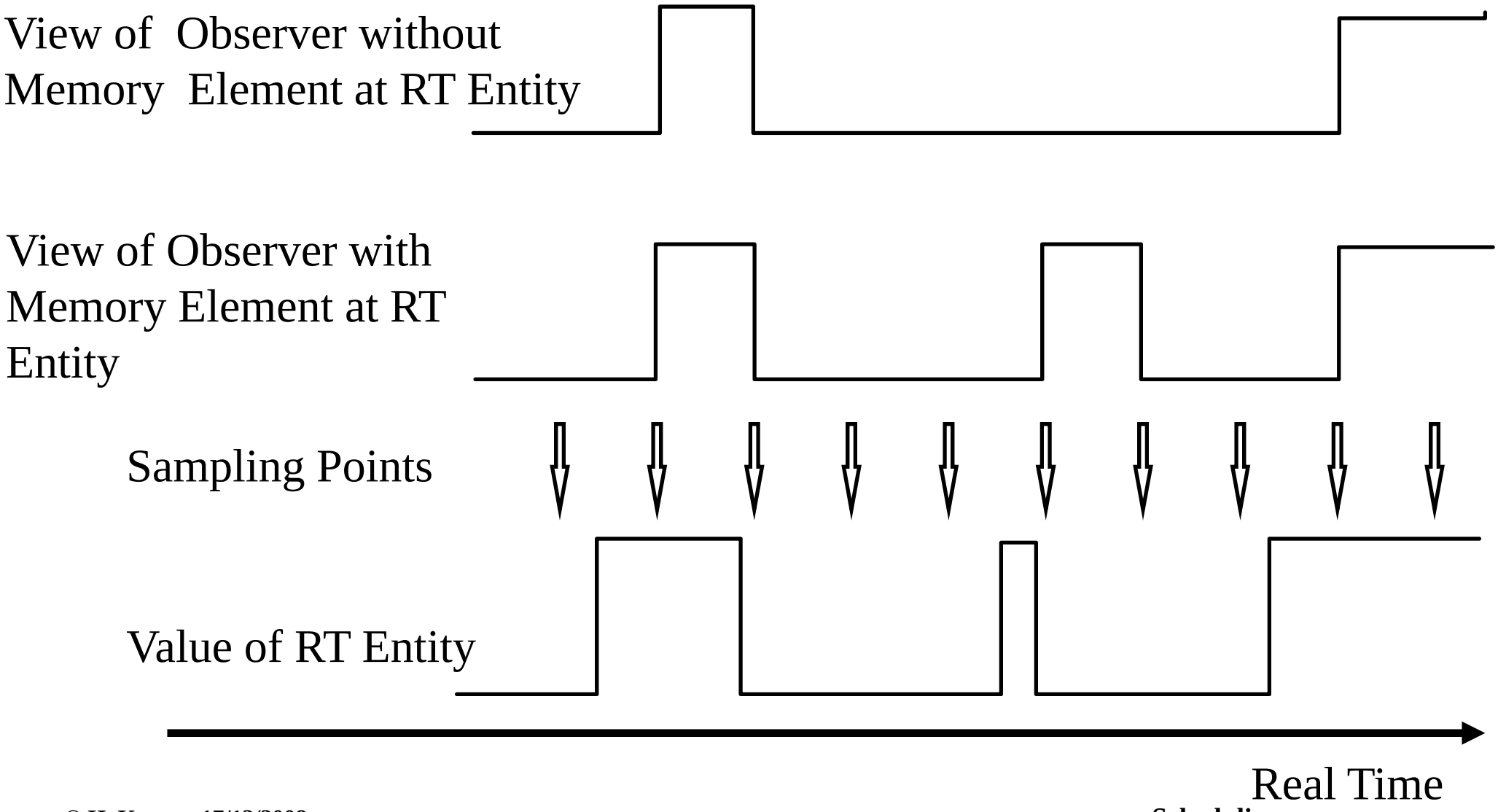
The lengthing points is called the sampling interval.

The length of the sampling interval is determined by the dynamics of the real-time entity.

States can be observed by sampling.

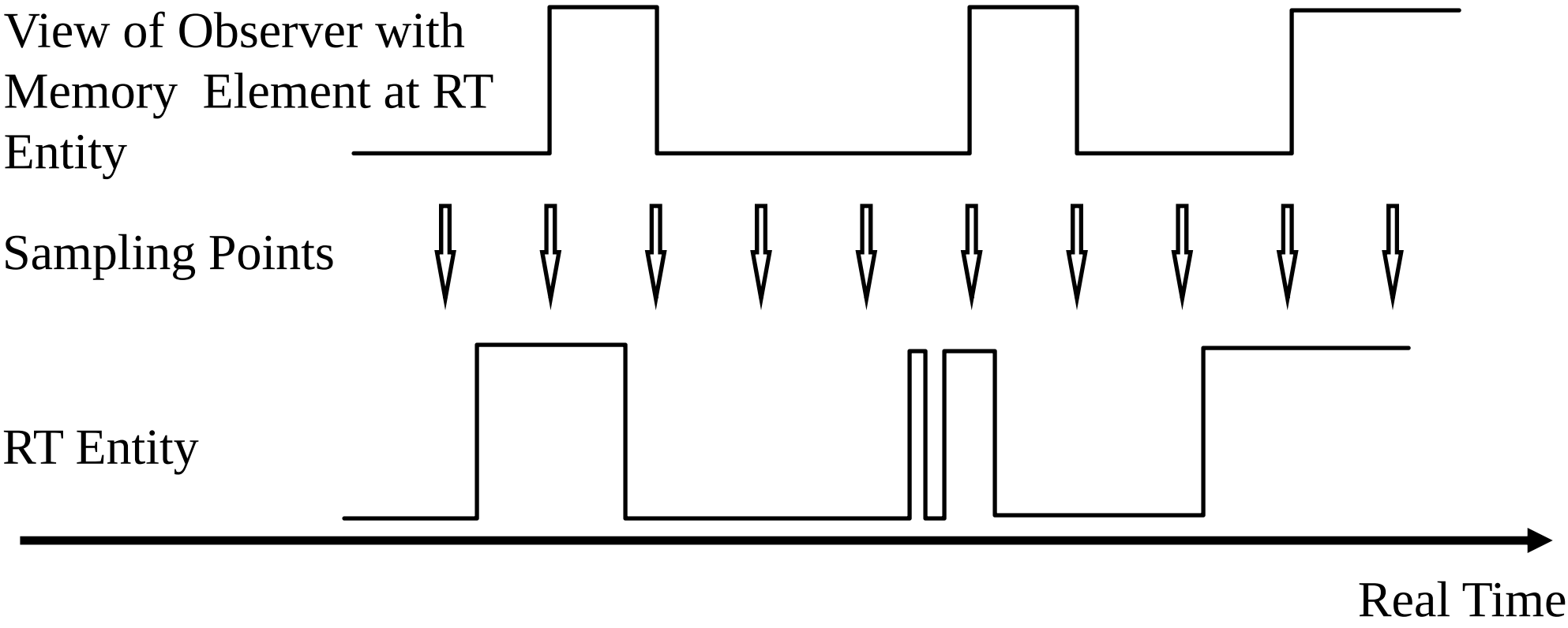
Events cannot be sampled. They have to be stored in an intermediate memory element (ME).

Sampling with and without Memory



Sampling -- Position of Memory Element

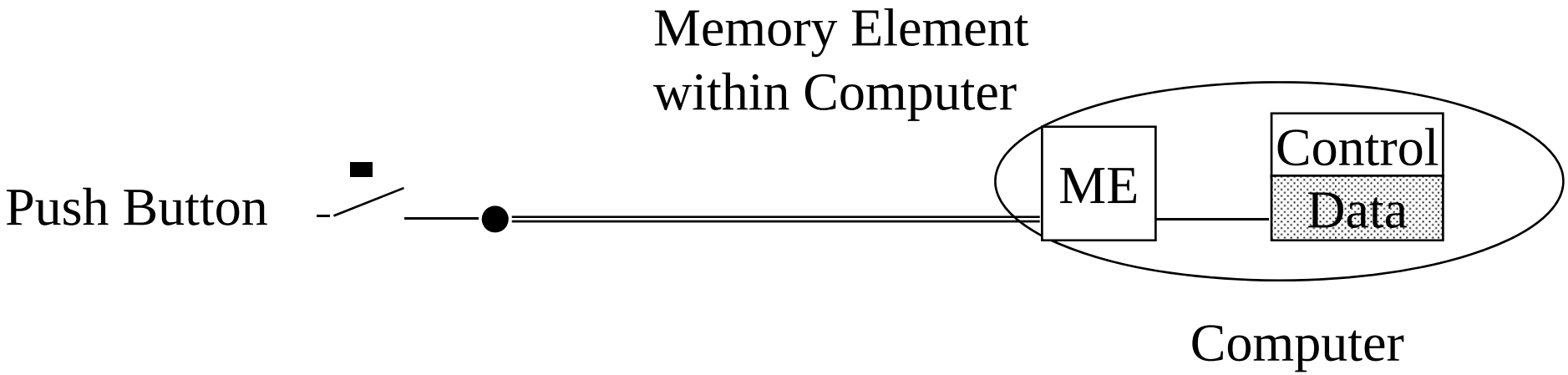
Sampling enforces *MINT*



Polling

Polling refers to the periodic interrogation of a memory element (normally within the computer interface) to determine if an event has occurred within the last polling period.

The memory element is reset by the computer after the memory element has been read.



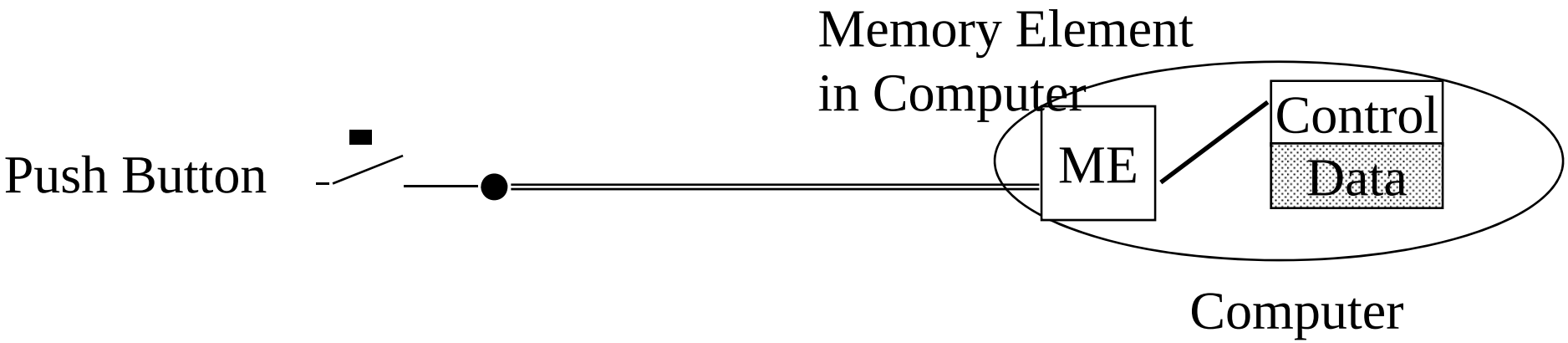
Interrupt

An interrupt is a hardware mechanism that monitors periodically (after the completion of each instruction) the state of a specified signal line (interrupt line).

If the line is active and when the interrupt is not disabled, control is transferred after completion of the currently executing instruction (from the current task) to an instruction (task) associated with the servicing of the specified interrupt.

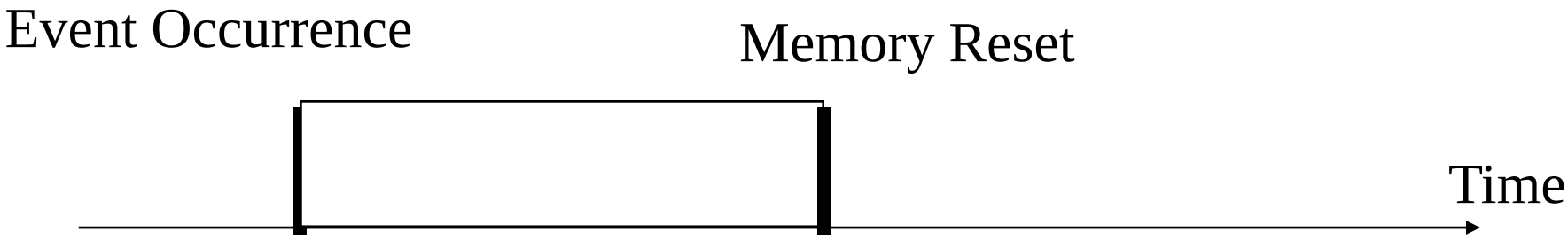
As soon as an interrupt is recognized, the state of the local “interrupt” memory is reset.

Interrupt



External event forces computer into interrupt service state.

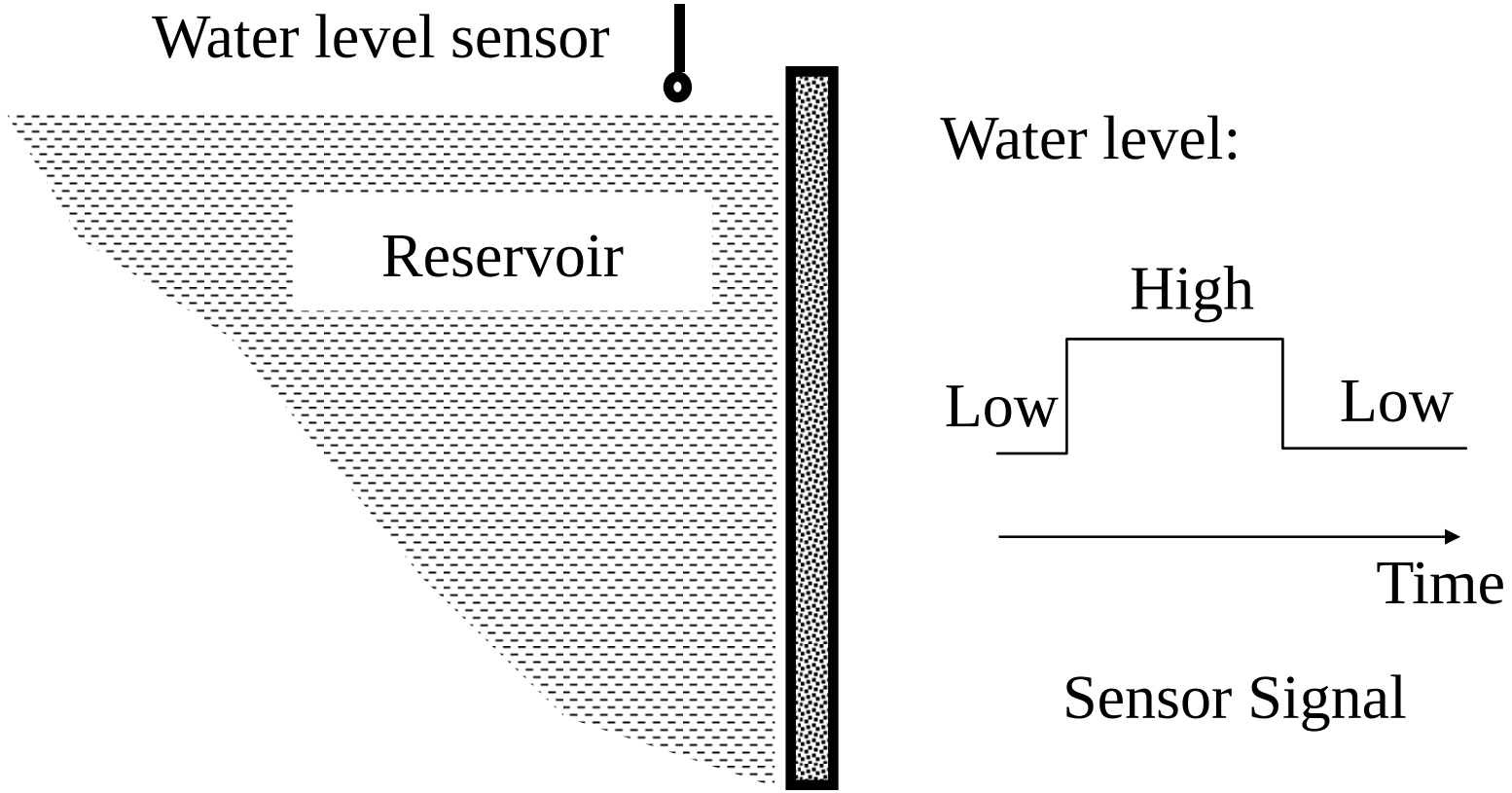
Memory Element for an Event



Sampling: after a fixed period by sensor
Polling: by CPU
Interrupt: by CPU

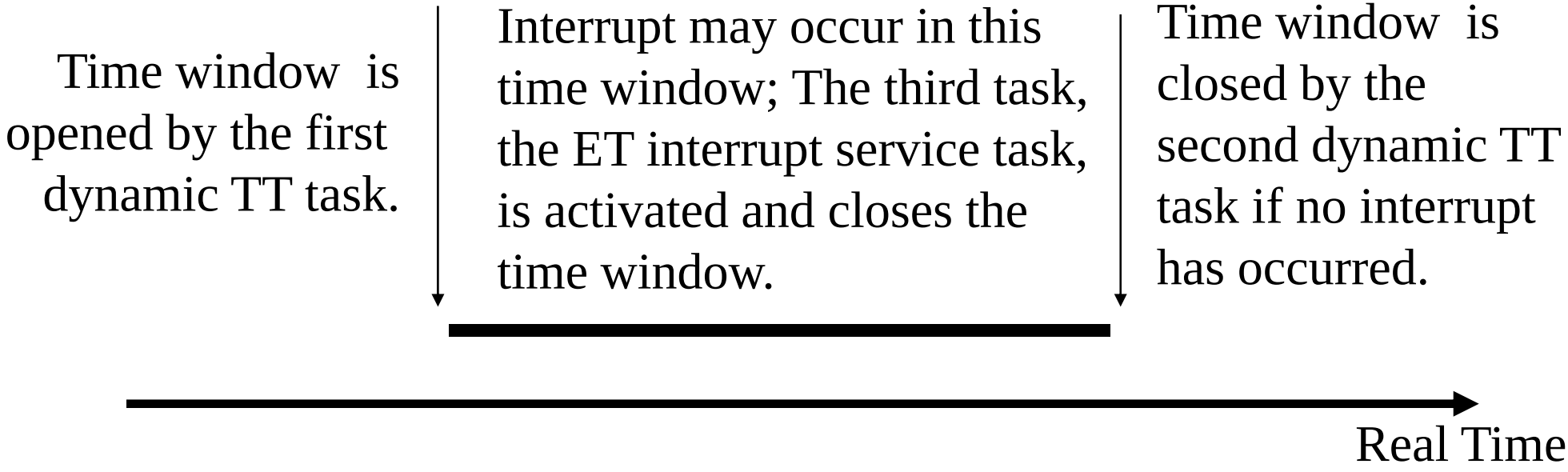
In the interval between the event occurrence and the resetting of the memory, no further events are recognized

How to find a *MINT* for Interrupts?

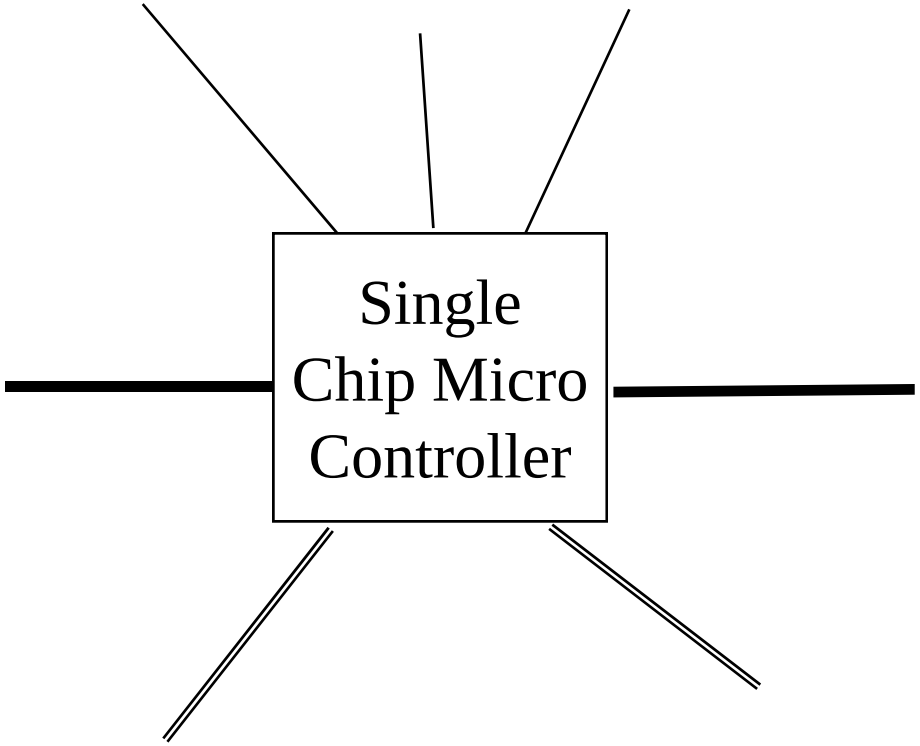


Monitor the Occurrence of Interrupts

Three tasks to handle an interrupt:



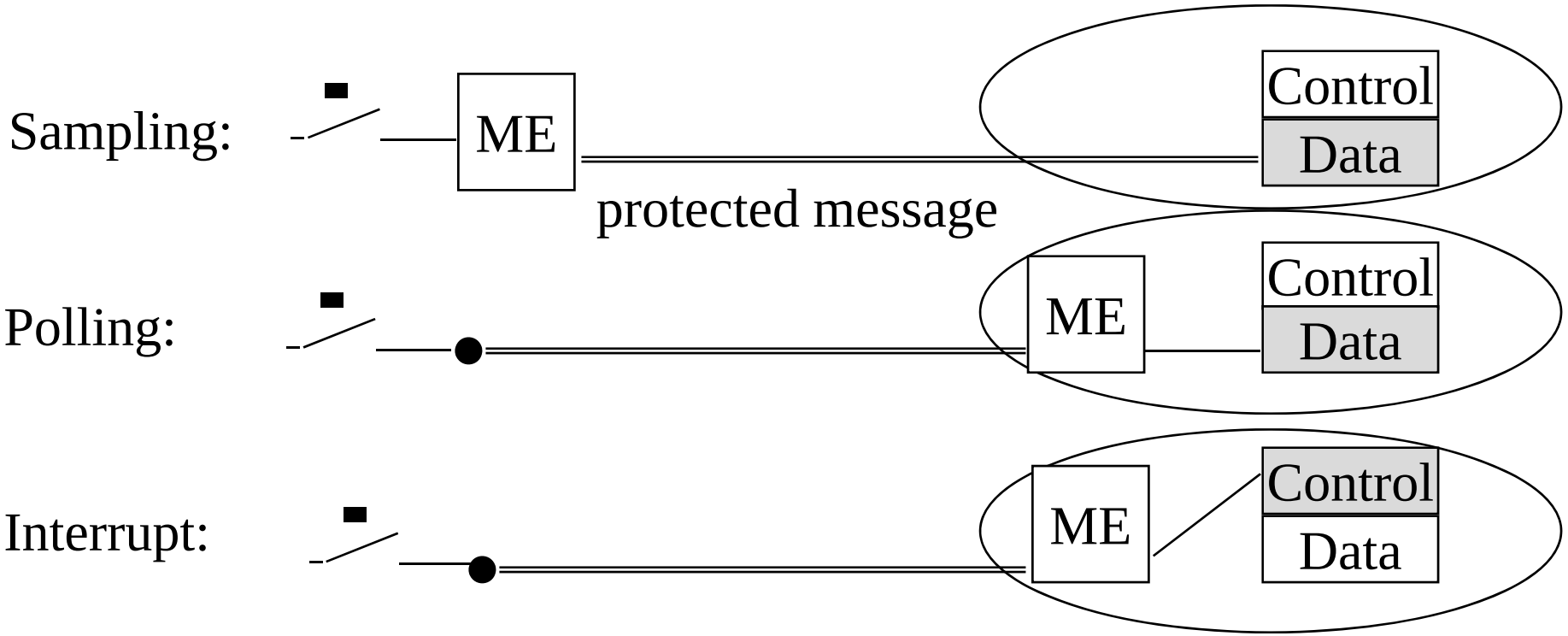
Probability of Control Error



Interrupt Line

What is an appropriate fault model for EMI transients?

Sampling, Polling, Interrupt--Failures?



What happens in the case of failure?

Sampling, Polling, Interrupt--Failure rates

Let us look at the consequences of transient transmission errors:

Sampling/Polling period: 10 msec; sampling duration: 1μsec

Probability of a 1μsec disturbance during 10 msec: 10^{-4}

Failure rates:

Sampling: 1 (data)

Polling: 10^4 (data)

Interrupt: 10^4 (control)

The probability that the sampling/polling actions overlaps with the transmission error is small!

Need for Agreement Protocols

If a RT entity is observed by two (or more) nodes of the distributed system, an agreement protocol is needed

- The same event can be time-stamped differently by two nodes (fundamental limit of time measurement)
- A real data-value can be mapped into different integer values (discrimination error).

These problems occur whenever a dense quantity is represented by discrete quantity.

Agreement Protocol

An agreement protocol provides a consensus on the value of an observation and on the time when the observation occurred among a number of fault-free members of an ensemble:

- The first phase of an agreement protocol concerns the exchange of the local observations to get a globally consistent view to each of the partners
- In the second phase each partner executes the same algorithm on this global data (e.g., averaging) to come to the same conclusion--the agreed value and time

Agreement always needs an extra round of communication and thus weakens the responsiveness of a real-time system.

Agreement of Resource Controllers

As long as a number of resource controllers that observe a set of real-time entities has not agreed on the observations, there is no active redundancy possible (and therefore no need for replica determinism):

- The world interface between a sensor and the associated resource controller can be serviced without concern for replica determinism (local interrupts!!)
- It should be an explicit design goal to eliminate the h-state from the resource controller (stateless protocols)--as far as possible.
- The message interface of the resource controller to the rest of a cluster should provide agreed values only!

This problem does not exist if there is only a single observation.

Byzantine Agreement

Byzantine agreement protocols have the following requirements to tolerate the Byzantine failures of "f" nodes :

- There must be at least 3f+1 nodes in a FTU.
- Each node must be connected to all other nodes of the FTU by f+1 disjoint communication paths.
- In order to detect the malicious nodes, f+1 rounds of communication must be executed among the nodes.
- The nodes must be synchronized to within a known precision of each other.

The Scheduling Problem

Schedulability Test:

Given a set of tasks $\{T_i\}$with synchronization requirements (precedence and mutual exclusion), is there a test to determine if this set is schedulable?

Scheduling Algorithm:

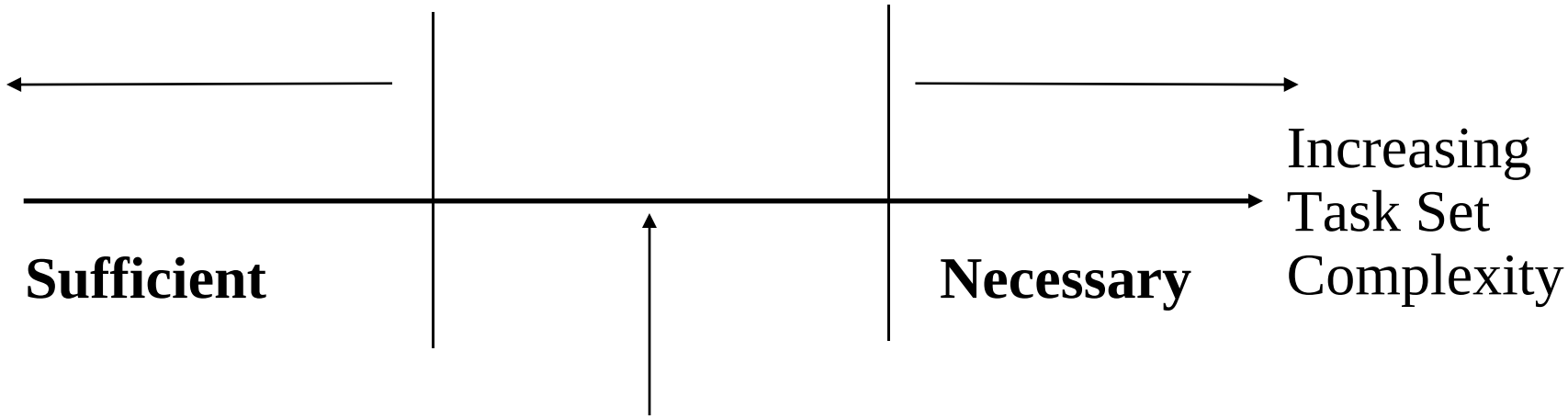
Is there an algorithm that will find a feasible schedule in bounded will it find a schedule whenever there is one?

In the general case, the effort required to solve the scheduling problem is NP-hard.

Schedulability Test

If a sufficient schedulabilty test is positive, these tasks are definitely schedulable

If a necessary schedulability test is negative, these tasks are definitely not schedulable



Exact Schedulability Test

Classification

Scheduling

- Best Effort versus Guaranteed
- Static versus Dynamic
- Preemptive versus Nonpreemptive
- Central versus distributed

Task Arrival

- Periodic
- Sporadic (with minimum interarrival time)
- aperiodic

Best Effort versus Guaranteed

Best effort

As the name implies, best effort scheduling tries to find a feasible schedule. But it can fail, because a feasible schedule might not exist or there is not enough time to find one.

Guaranteed:

There exists a schedulability test that guarantees that the given task set is schedulable. It is guaranteed that the scheduler will always find a feasible schedule, i.e. that under the given assumptions all task will meet their deadlines.

The minimum task interarrival time (*mint*) is a critical assumption that must be met by the application.

Static Scheduling

All scheduling decisions are made at compile time. The temporal task structure is fixed. At run time, the dispatcher has to perform a table lookup to decide at each point in time which task has to be executed next.

The static schedules are designed such that all precedence and mutual exclusion requirements between tasks are satisfied (*implicit synchronization*) and the run time is optimized. (No overhead for wait and signal operations).

The search for a static schedule is normally an NP hard search problem. However, optimal solutions are not required--good solutions are sufficient. The successful search finds the schedule and provides a sufficient schedulability test.

Dynamic Scheduling

All scheduling decisions are made on-line, i.e., on the basis of the ready task set.

Mutual exclusion and synchronization must be enforced by explicit synchronization constructs (e.g., wait operation) at run time.

Advantage:

- Flexibility
- Only resources claimed that are actually used

Disadvantage:

- Computational resources required for scheduling and synchronization
- Guarantees are difficult to support
- Replica determinism?

Preemptive versus Nonpreemptive

Preemptive scheduling:

Whenever a significant event occurs, the presently running task will be interrupted and a new scheduling decision will be made.

Nonpreemptive scheduling:

Whenever a task gets control of the CPU it can continue until it is finished and releases the CPU. Scheduling decisions are only made after task completion.

Nonpreemptive Scheduling:

Nonpreemptive scheduling is reasonable, when the task execution times are in the same order of magnitude as the context switching times.

The shortest response time that can be guaranteed in nonpreemptive scheduling is the longest task time plus the shortest task time.

From the point of view of replica determinism, nonpreemptive scheduling is to be preferred.

E.g., nonpreemptive scheduling is used in the fault-tolerant VOTRICS train signalling system.

Clairvoyance

A scheduler is *clairvoyant* if it knows everything about the future.

A scheduler is optimal if it can find a schedule whenever the best clairvoyant scheduler can find a schedule.

In the general case, a dynamic scheduler cannot be optimal (proof: adversary argument).

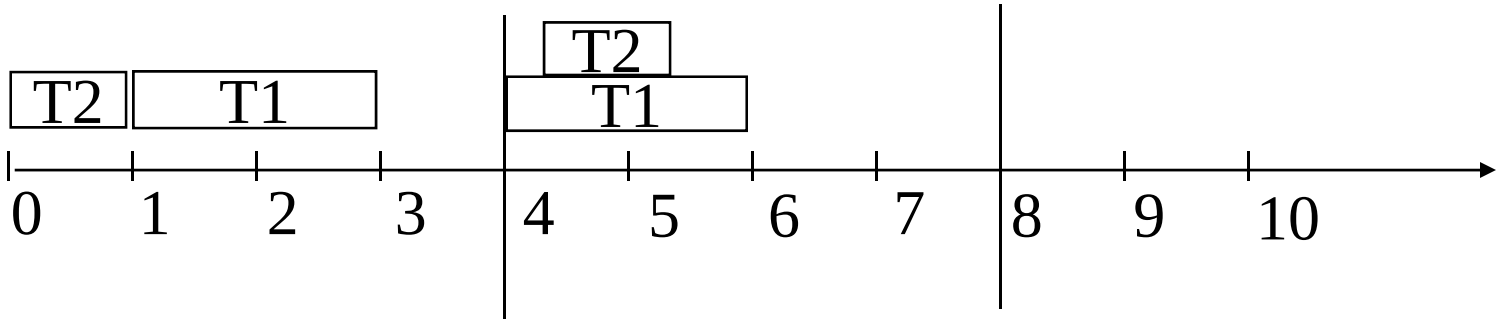
Under restricting assumptions, optimal dynamic schedulers exist.

Adversary Argument

T1 : c1= 2, d1 = 4, p1 = 4, periodic

T2: c2=1, d2=1, p2=4, sporadic

T1 and T2 are mutually exclusive



Although there is a solution, an online scheduler cannot find it.

Periodic Tasks:

Assumptions for preemptive scheduling:

- All tasks are periodic
- Deadline is equal to the period
- Upper bound on CPU time is known
- Overhead of task switch can be neglected
- All tasks are independent

Rate-Monotonic Algorithm

Liu and Layland 1973:

Given a set of n tasks with WCET C_i and period T_i.

Tasks with the shortest period gets the highest

$$\sum_{i=1}^{n} (C_j / T_j) \leq n (2^{1/n} - 1)$$

then all tasks will meet their deadlines.

In case the periods are multiples of each other, then the maximum CPU utilization can approach 1.

Proof: Reasoning around critical instant.

Earliest Deadline First (EDF)

Dynamic preemptive scheduling with dynamic priorities.

Same assumptions as rate monotonic.

The tasks with the earliest deadline gets the highest (dynamic) priority.

In uniprocessor systems this is an optimal algorithm

Least Laxity (LL)

Laxity: Difference between deadline and CPU time

Dynamic preemptive scheduling with dynamic priorities.

Same assumptions as rate monotonic.

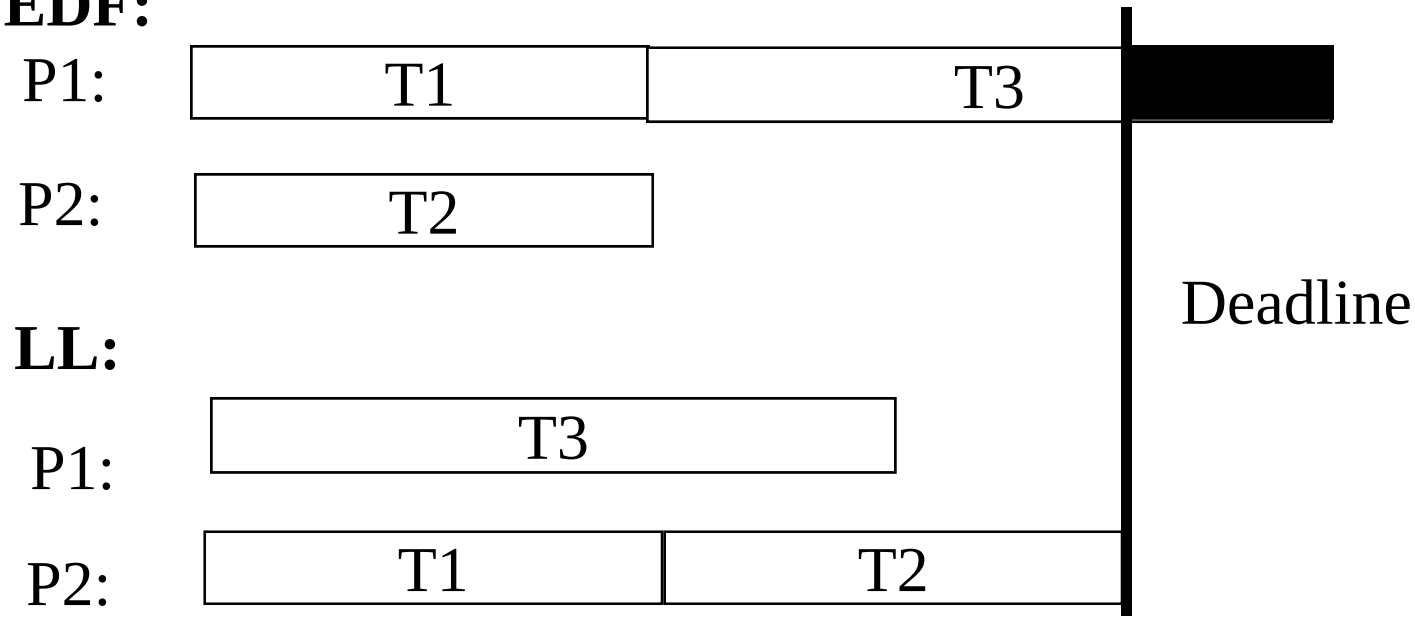
The tasks with the shortest laxity (i.e. the difference between the deadline and the CPU time) gets the highest (dynamic) priority.

In uniprocessor systems this is also an optimal algorithm

Multiprocessor Systems

Assume: T1: c1=5, d1 =10
T2: c2=5, d2=10
T3: c3=8, d3=12

EDF:

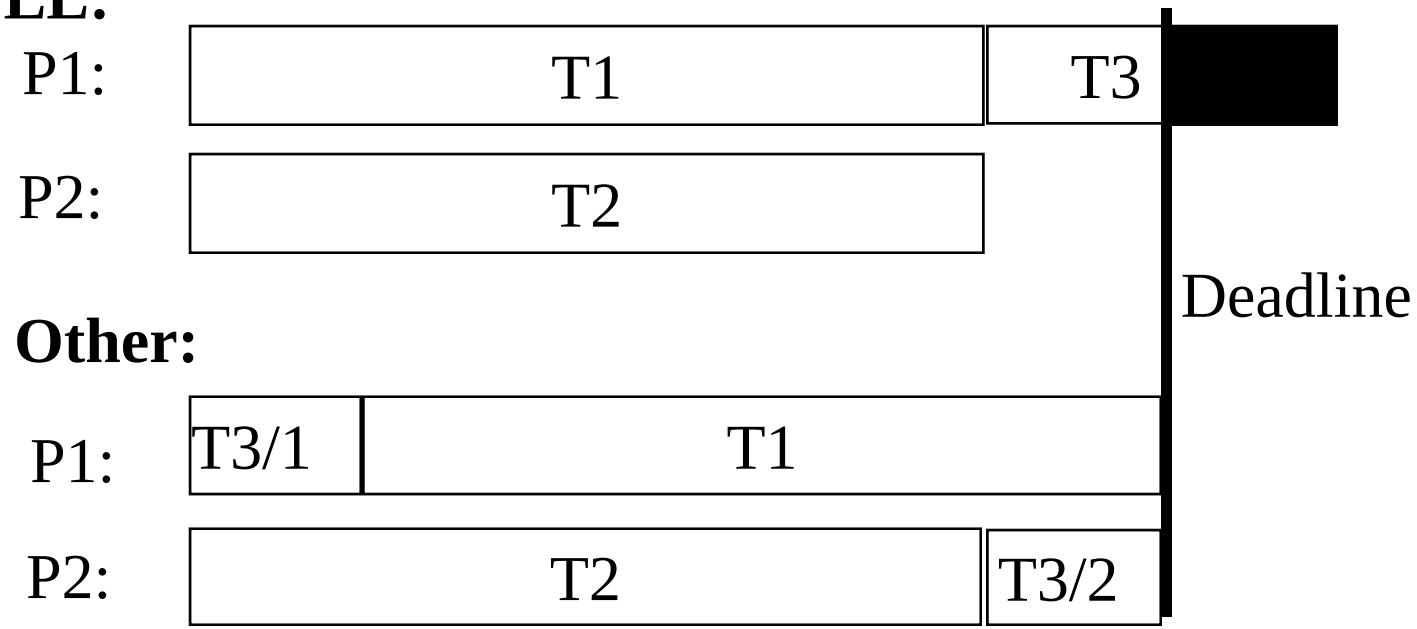


LL:

LL is not optimal in MP Systems

Assume: T1: c1=8, d1 =10
T2: c2=8, d2=10
T3: c3=4, d3=10

LL:



Other:

Scheduling of Sporadic Tasks

Transformation of the sporadic task to a quasi periodic task with the following new parameters:

new.period= Min (old.period, Laxity +1)

ci’ = ci

di’ = di

The old period denotes the minimum time between two activation requests (minimum interarrival time *mint*).

If this quasi periodic task can be scheduled, then the sporadic task will always meet its deadline.

A short laxity of a sporadic task is very demanding on the resources.

Sporadic Server Task

We can define a server task for the sporadic request that has a short latency.

The server task is scheduled in every period, but is only executed if the sporadic request actually appears.

This will require a task set in which all the other tasks have a laxity at least of the size of the execution time of the server task.

(Remember: Up to now mutual exclusion is not permitted!!!)

Priority Inversion

Priority inversion is the phenomenon when a higher priority task is blocked by a lower priority task.

This situation can arise, if mutual exclusion (e.g. between tasks T1 and T3 in the following example) has to be enforced.

Consider the scenario:

Task 3 priority 1 (lowest) requests and gets a resource

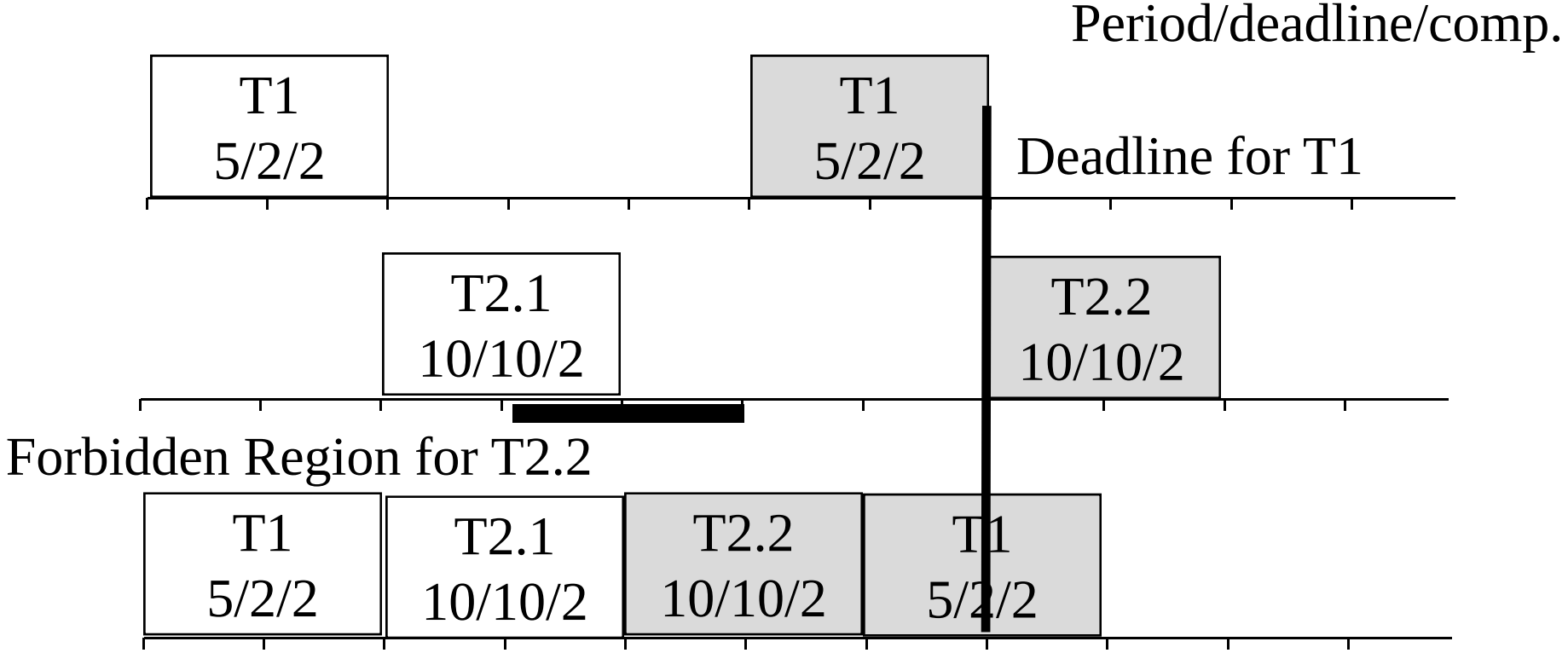
Task 1 priority 10 (highest) requests service (with the resource) and is blocked because Task3 holds the resource

Task 2 priority 5 (middle) requests service and gets the CPU

Effectively, Task 2 is preempting Task 1

Forbidden Regions

To avoid the blocking of a higher priority task by a lower priority task holding a common resource (T1, T2.2), forbidden regions for the activation of the lower priority task can be introduced off line:



Priority Inheritance

When a task T blocks one or more higher priority tasks, it ignores its original priority and gets the highest priority level of all the tasks it blocks. After exiting its critical section, task T returns to its original priority.

The basic priority inheritance protocol does not prevent deadlocks!

Priority Inheritance - Deadlock!

Consider two tasks, T1 (high priority) and T2 (low priority) with two critical regions protected by semaphores S1 and S2.

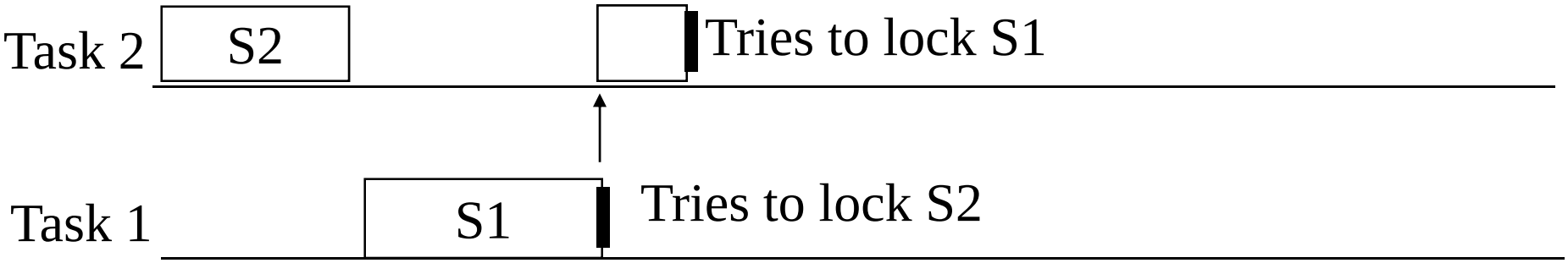
1 T2 locks S2

2 T2 is interrupted before it tries to lock S1

3 T1 preempts T2 and locks S1

4 T1 tries to lock S2 but cannot (blocked by T2)

4 T2 inherits priority from T1, but is also blocked



Priority Ceiling Protocol

The goal of the priority ceiling protocol is the prevention of deadlocks that can occur if priority inheritance is employed.

Assign a priority ceiling to each semaphore. The priority ceiling is equal to the highest priority task that may use this semaphore.

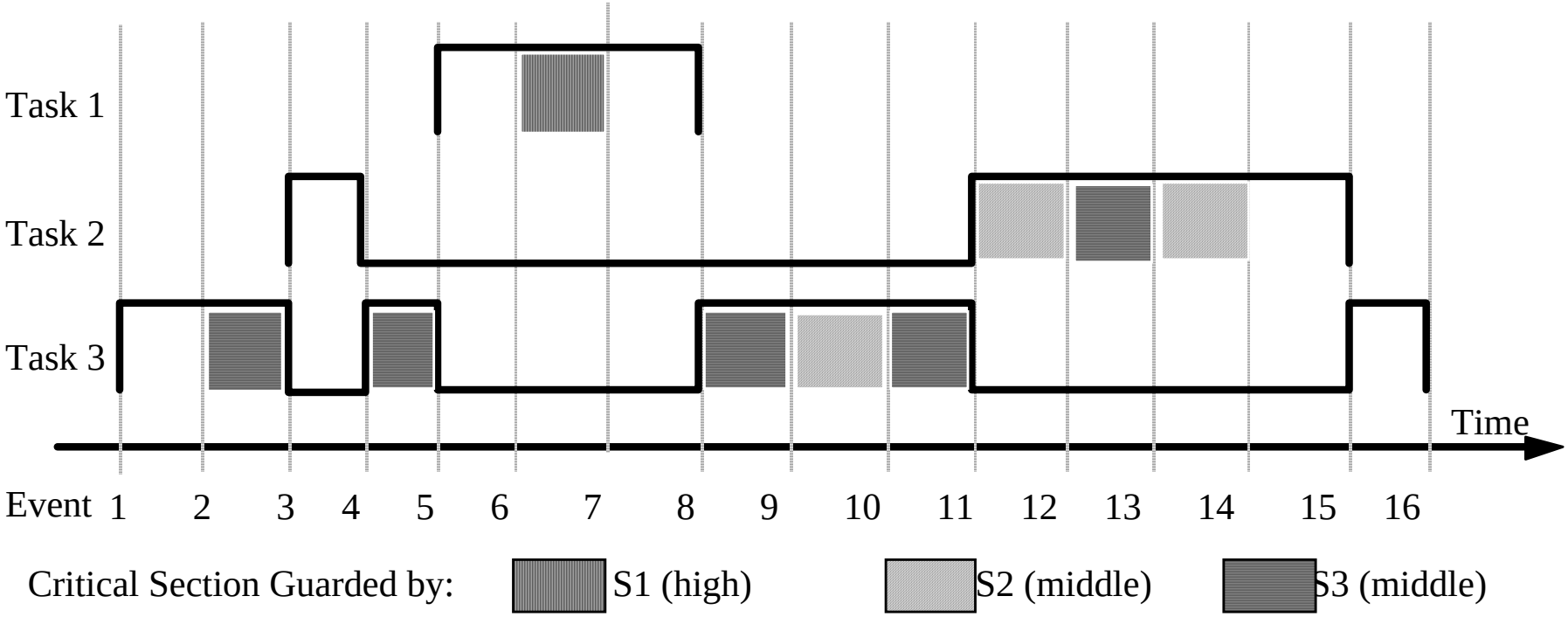
Task T is allowed to start a new critical section only if T's priority is higher than the priority ceilings of all the semaphores locked by tasks other than T.

Sha et al. IEEE Trans. Comp. Sept. 1990, pp. 1175-1185

Priority Ceiling Example/1

Command Sequence Executed by Task:

T1: . ., P(S1), . ., V(S1), . . . (highest priority)
T2: . ., P(S2), . ., P(S3), . ., V(S3), . ., V(S2), . . (middle priority)
T3: . ., P(S3), . ., P(S2), . ., V(S2), . ., V(S3), . . (lowest priority)



Priority Ceiling Example

Event	Action
1	T3 begins execution.
2	T3 locks S3.
3	T2 is started and preempts T3.
4	T2 becomes blocked when trying to access S2 since the priority of T2 is not higher than the priority ceiling of the locked S3. T3 resumes the execution of its critical section at the inherited priority of T2.
5	T1 is initiated and preempts T3.
6	T1 locks the semaphore S1. The priority of T1 is higher than the priority ceiling of all locked semaphores.
7	T1 unlocks semaphore S1.
8	T1 finishes its execution. T3 contintues with the inherited priority of T2.
9	T3 locks semaphore S2.
10	T3 unlocks S2.
11	T3 unlocks S3 and returns to its lowest priority. At this point T2 can lock S2.
12	T2 locks S3.
13	T2 unlocks S3.
14	T2 unlocks S2.
15	T2 completes. T3 resumes its operation.
16	T3 completes.

Schedulability Test for Priority Ceiling

A set of n periodic tasks with periods T_i and computation times C_i using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm if the following n conditions are satisfied for all tasks i:

$$\forall i, 1 \le i \le n, \sum_{j=1}^{i} (C_j / T_j) + B_i / T_i \le i\,(2^{1/i} - 1)$$

The first i terms in the above inequality constitute the effect of preemptions from all higher priority tasks and task i's own execution time. B_i represent the worst case blocking time of task i due to all lower priority tasks.

Search Tree for Static Scheduling

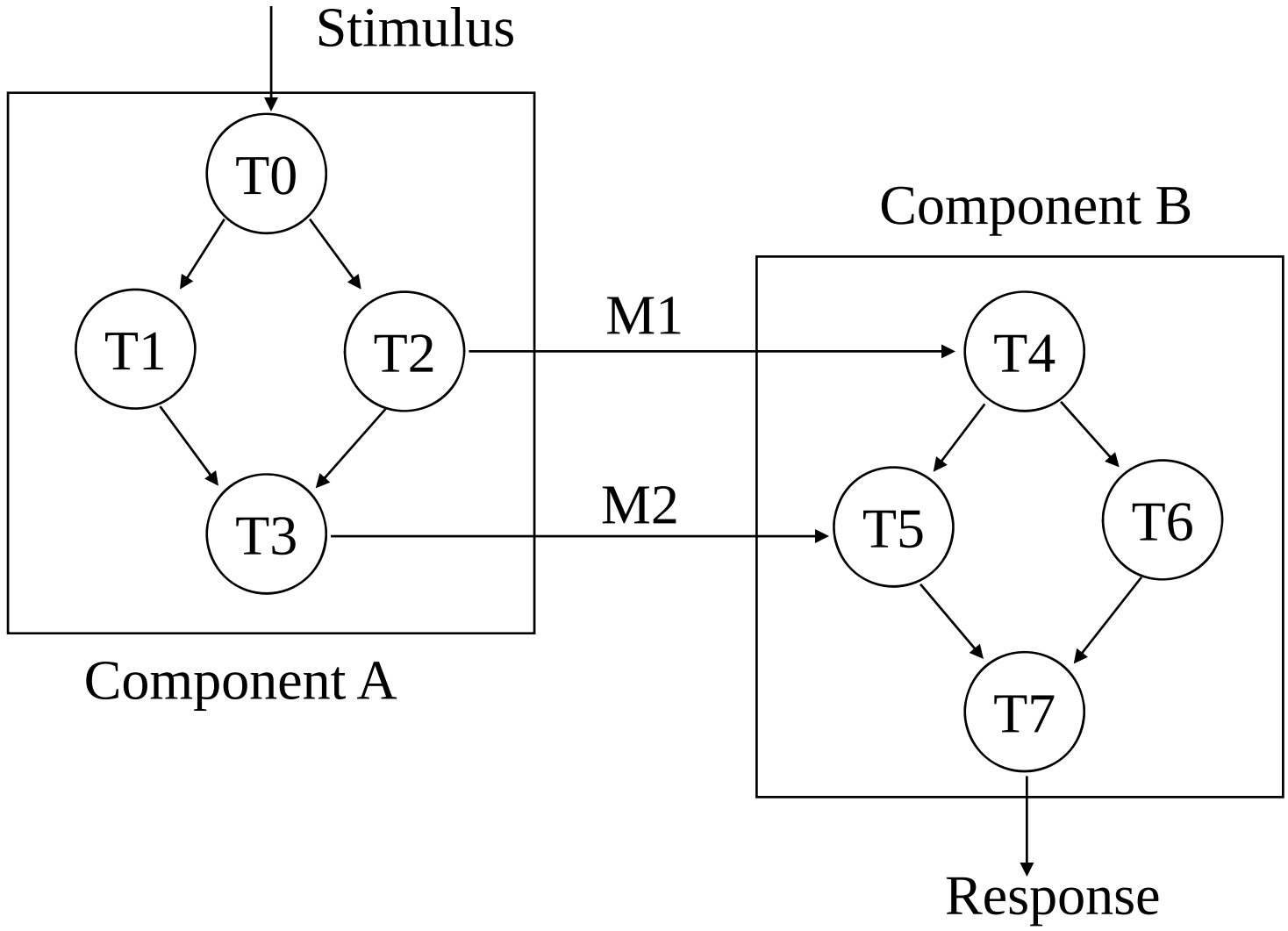
The goal of the static scheduler is to find a path through the search tree that

- meets all deadlines
- observes all constraints (mutual exclusion, etc.)

and to generate the dispatcher table for the TT operating system. The search tree for a static schedule is constructed out of the task precedence graph.

A heuristic function that estimates the value of the time needed to complete the precedence graph is the TUR (Time until Response). If the actual time needed for completion is always longer than TUR, then branches of the search tree can be pruned if the time needed ro reach the branch plus TUR is later than the deadline.

Static Scheduling--Precedence Graph



Search Tree--Example

Time Slot			
1		T0	
2	T1		T2
3	T2		M1 & T1
4	M1&T3		T3 & T4
5	M2&T4		M2 & T6
6	T5	T6	T5
7	T6	T5	T7
8	T7	T7	