CIS 4930/6930: Principles of Cyber-Physical Systems

Chapter 11 Scheduling

Hao Zheng

Department of Computer Science and Engineering
University of South Florida

Functions of an Operating System (or Microkernel)

- Memory management
- File system
- Networking
- Security
- Input and output (interrupt handling)
- Synchronization (semaphores and locks)
- Scheduling of threads or processes

Functions of an Operating System (or Microkernel)

- Memory management
- File system
- Networking
- Security
- Input and output (interrupt handling)
- Synchronization (semaphores and locks)
- Scheduling of threads or processes
  - Creation and termination of threads
  - Timing of thread activations

Real-time Systems

- A **real-time system** includes timing constraints such as:
  - Physical-time deadlines by which a task must be completed.
  - Requirements that a task must occur no earlier than a particular time.
  - A task must occur a set time after another task.
  - A task must occur periodically at some specified period.
- Tasks may be dependent on one another or simply share a processor.
- All of these cases require a careful scheduling strategy. k_6

11.1 Basics of Scheduling:
11.1.1 Scheduling Decisions

- A **scheduling decision** has three parts:
  - **Assignment**: which processor should execute the task.
  - **Ordering**: in what order each processor should execute its tasks.
  - **Timing**: the time at which each task executes.
- Decisions may be at design time or run time.
  - A **fully-static scheduler** makes all decisions at design time (no locks).
  - A **static order scheduler** defers timing to run time (may block on lock).
  - A **static assignment scheduler** defers ordering and timing to run time.
  - A **fully-dynamic scheduler** makes all decisions at run time.

11.1 Basics of Scheduling: 11.1.1 Scheduling Decisions

- A **non-preemptive scheduler** dispatches when current thread completes.
- A **preemptive scheduler** may make a scheduling decision during execution of a task
  - Upon a timer interrupt at a jiffy interval.
  - Upon an I/O interrupt.
  - When it attempts to acquire an unavailable lock, and resumed when another task releases the lock.
  - When it releases a lock, if a higher priority thread requires the lock.
  - When the current thread makes any OS call.
    - File system access
    - Network access
    - ...

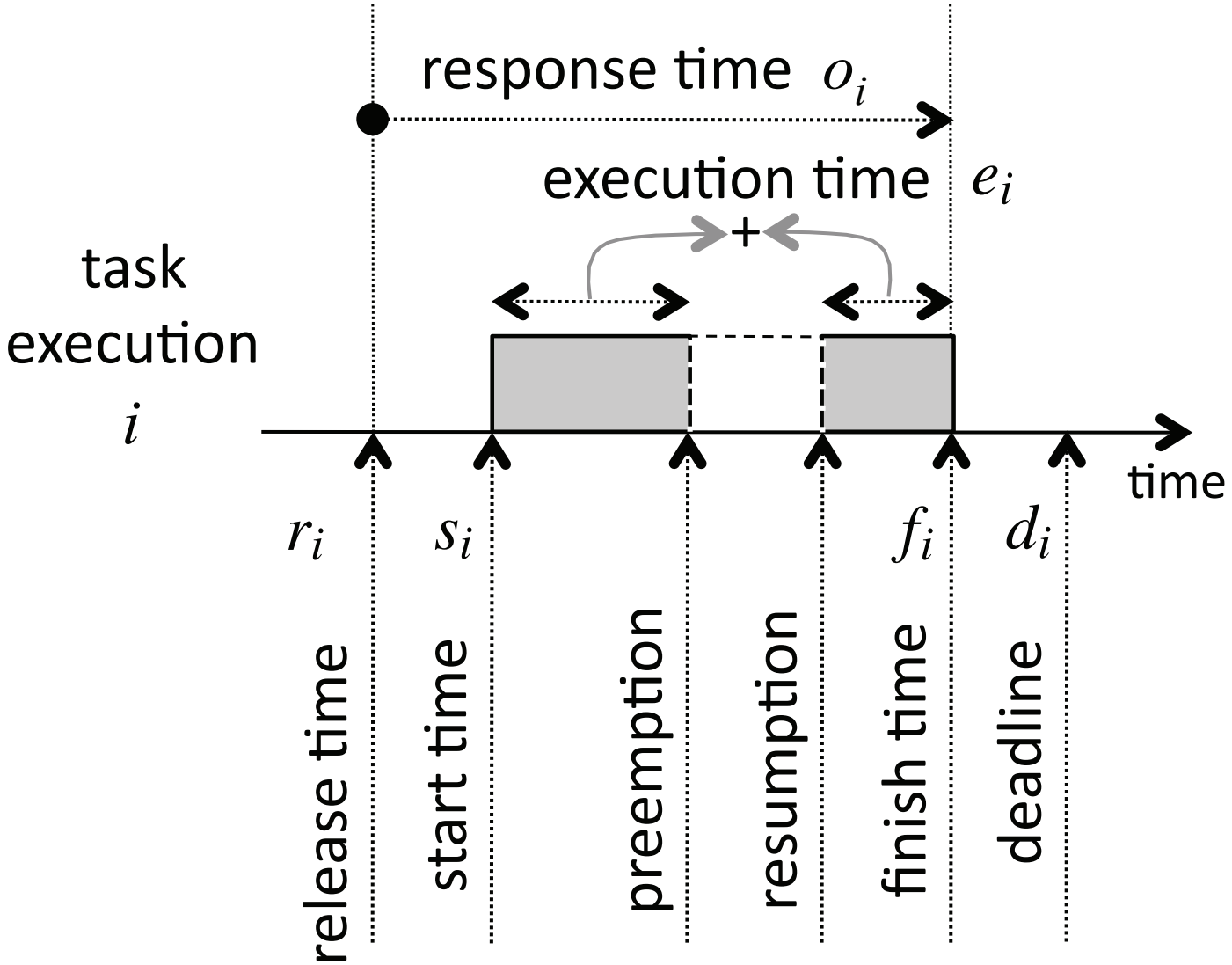
11.1.2 Task Models

- The set of assumptions is called the **task model** of the scheduler.
- Assume a finite number of tasks that may or may not terminate.
  - Real-time systems often assume that tasks terminate.
- Some schedulers can assume that
  - All tasks are known before scheduling begins, or
  - New tasks can arrive dynamically.
- Some schedulers support scenarios where each task $\tau \in T$ executes repeatedly, possibly forever, and possibly periodically.

11.1.2 Task Models

- Distinction between a task and its executions:
  - When task $\tau \in T$ executes repeatedly, the task executions are $\tau_1, \tau_2, \ldots$.
- A **sporadic** task repeats with irregular timing, but has a lower bound on the time between executions.
- If execution i must precede j, we write $i < j$ (**precedence constraint**).
- A task may require **preconditions** to be satisfied before it is **enabled**.
  - Availability of a lock may be a precondition for resumption of a task.

Task Execution Times



Deadlines and Priority

- **Hard real-time scheduling** has *hard deadlines* which are real physical constraints that are an error when missed.
- **Soft real-time scheduling** has desired deadlines which are not errors when missed.
- A **priority-based scheduler** assumes each task is assigned a number (*priority*) and chooses the enabled task with the highest priority.
- A **fixed priority** remains constant while a **dynamic priority** can change.
- A **non-preemptive priority-based scheduler** only uses the priorities to choose the next task, but never interrupts a task that is executing.
- A **preemptive priority-based scheduler** can change to a higher priority task at any time.

11.1.3 Comparing Schedulers

- A schedule is **feasible** if it meets all deadlines ($f_i \leq d_i$).
  - A scheduler that produces feasible schedules whenever possible is **optimal with respect to feasibility**.
- Schedulers also judged based on **utilization** (the percentage of time the processor is executing tasks).
  - Optimal w.r.t feasibility schedulers deliver feasible schedules whenever the utilization is $\leq 100\%$.
- **Maximum lateness** is another criterion:

$$L_{max} = \max_{i \in T}(f_i - d_i)$$

- **Total completion time** is also important:

$$M = \max_{i \in T} f_i - \min_{i \in T} r_i$$

11.1.4 Implementation of a Scheduler

- Thread data structure:
  - Copy of all state (machine registers).
  - Address at which to resume executing the thread.
  - Status of the thread (e.g. blocked on mutex).
  - Priority, worst case execution time, and other info to assist the scheduler.
- Operating System:
  - Set up periodic timer interrupts.
  - Create default thread data structures.
  - Dispatch a thread.
- Timer interrupt service routine:
  - Setup next timer interrupt.
  - Dispatch a thread.

Dispatching a Thread

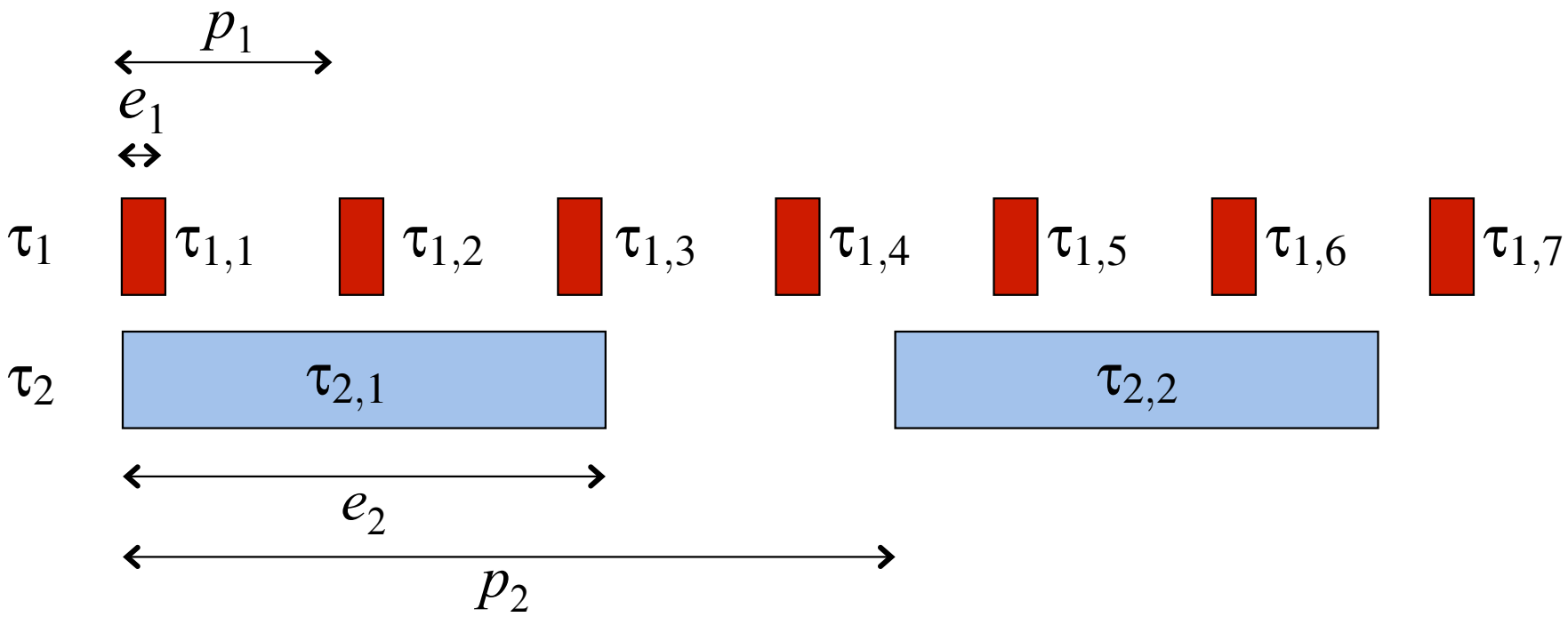
1. Disable interrupts.
2. Save state (registers) including the return address on the stack.
3. Save the stack pointer into the current thread data structure.
4. Determine which thread should execute (scheduling).
5. If the same one, enable interrupts and return.
6. Restore the stack pointer for the new thread.
7. Copy thread state into machine registers.
8. Replace program counter on the stack for the new thread.
9. Enable interrupts.
10. Return.

11.2 Rate Monotonic (RM) Scheduling

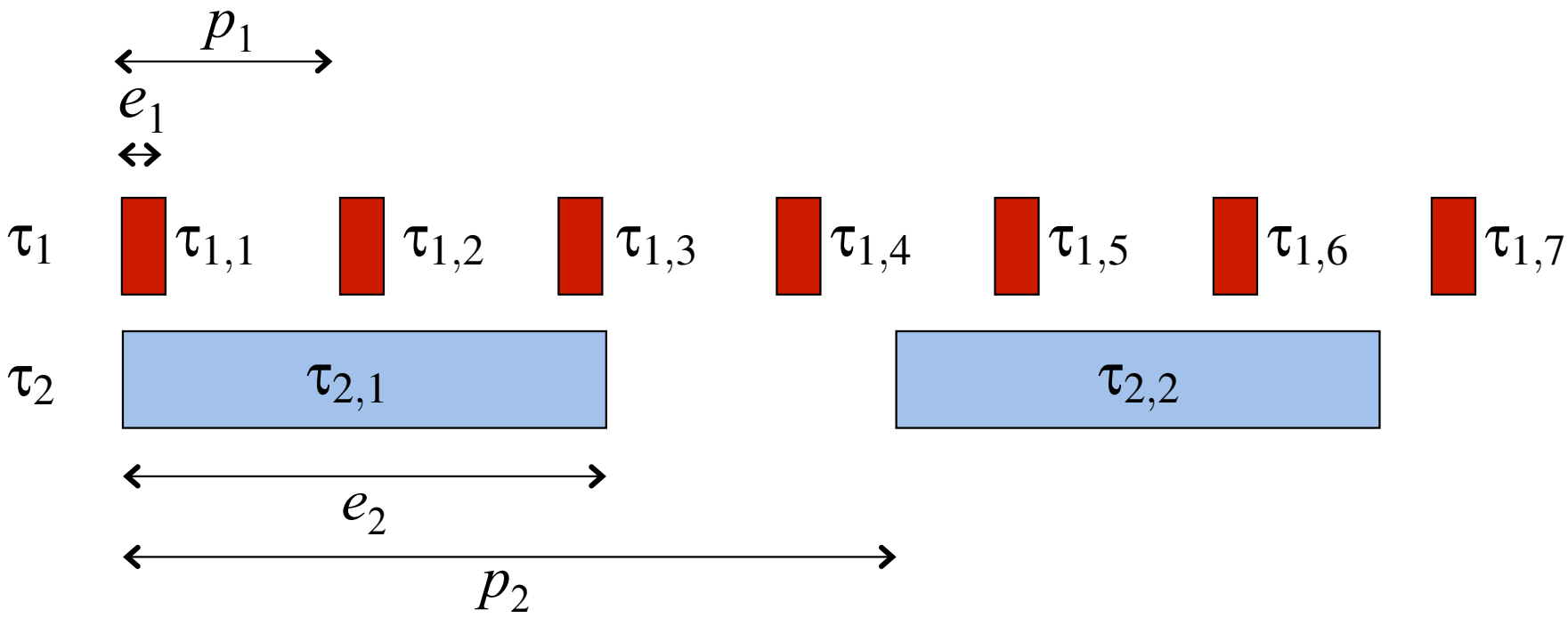
- Assume n tasks (i.e., $T = \{\tau_1, \tau_2, \ldots \tau_n\}$) invoked periodically where each task must complete in each *period*, p_i.
- **Rate monotonic schedule** gives higher priority to a task with smaller period, and it is optimal with respect to feasibility.
- Note: important assumption is that *context switch time* is negligible.

Liu and Leland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," J. ACM, 20(1), 1973.

Example: Two Periodic Tasks

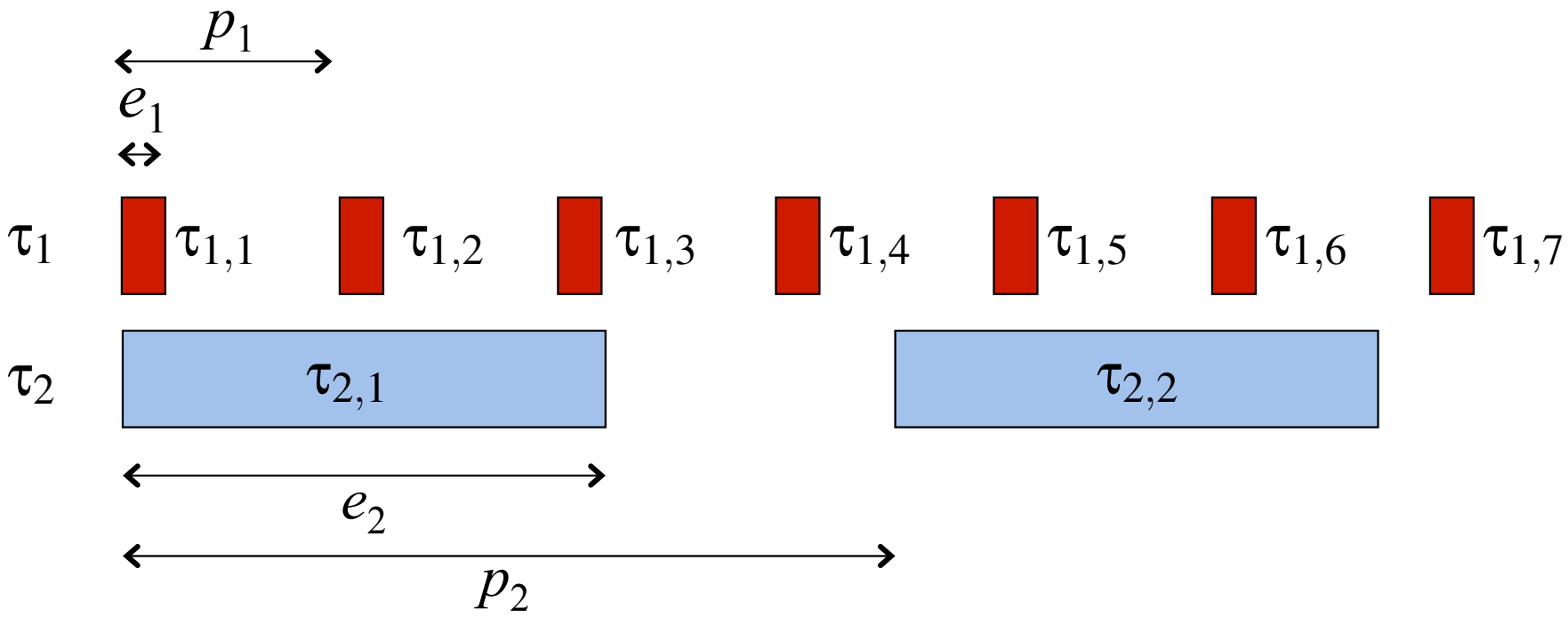


Example: Two Periodic Tasks



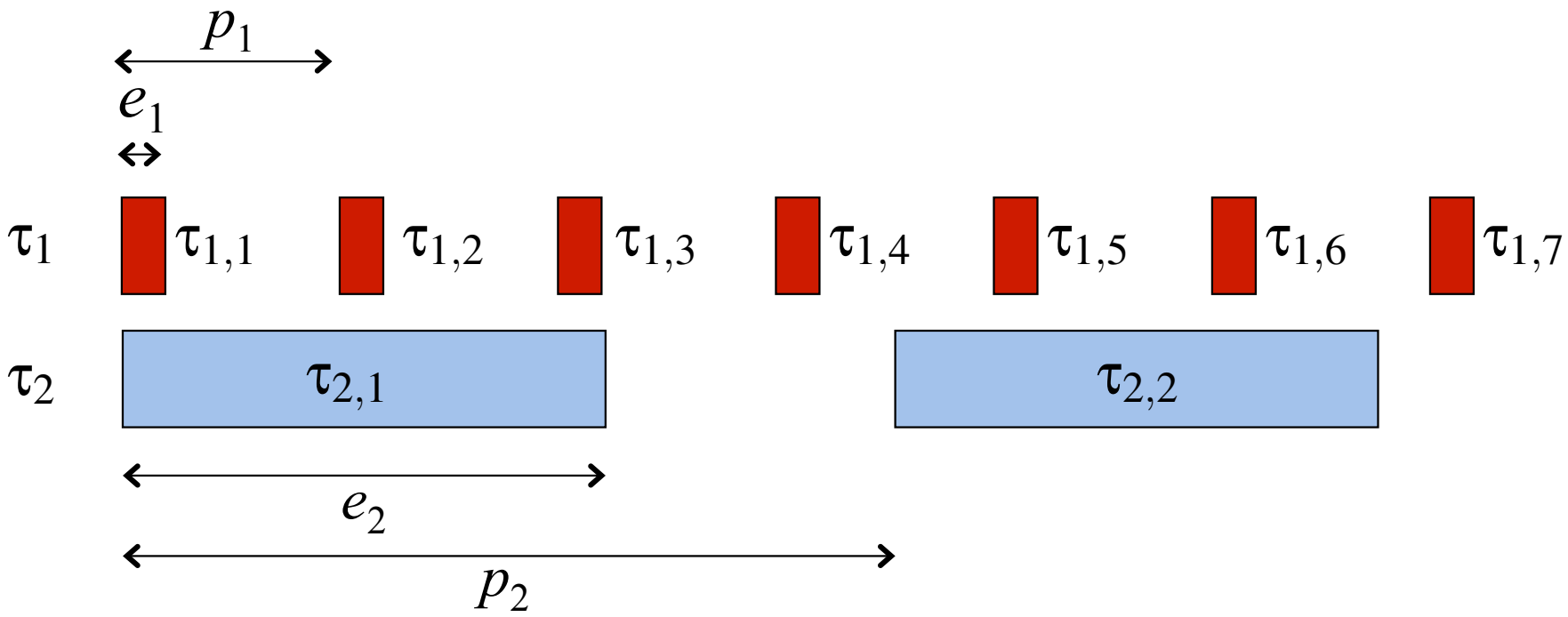
Is a non-preemptive schedule feasible?

Example: Two Periodic Tasks



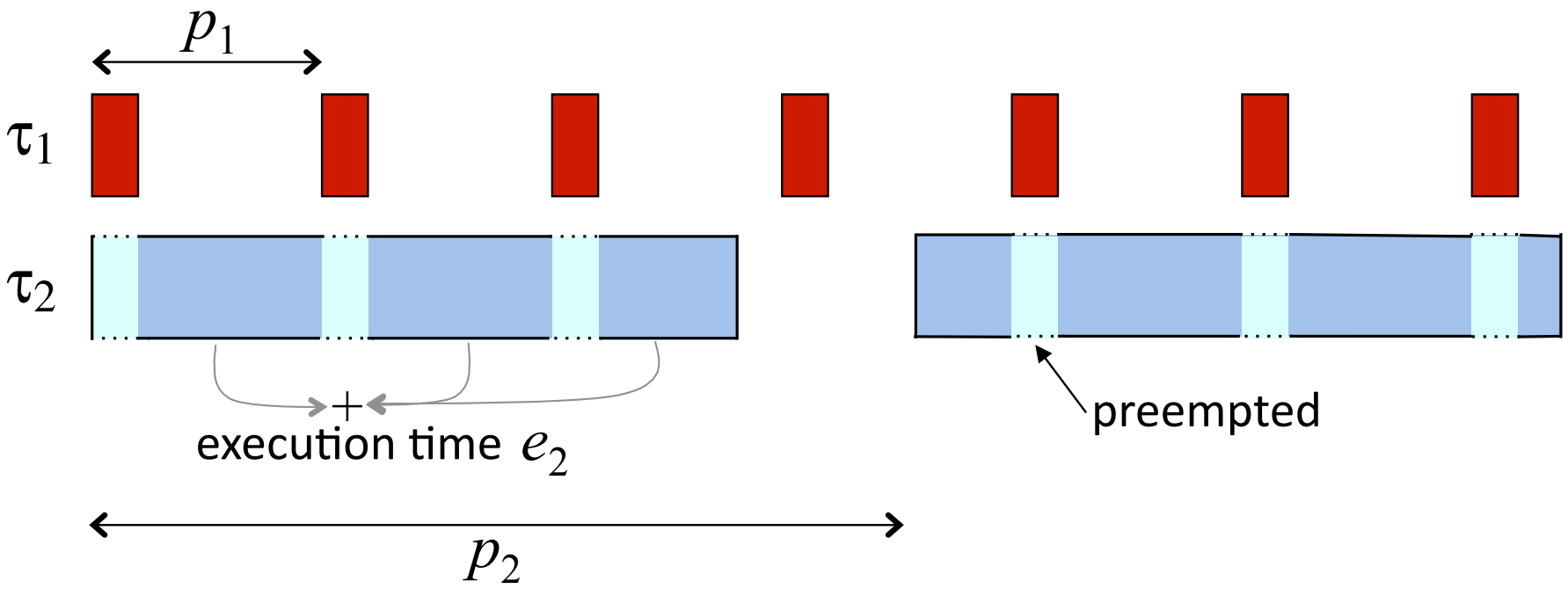
Is a non-preemptive schedule feasible? No!

Example: Two Periodic Tasks



How about a preemptive schedule with higher priority for red task?

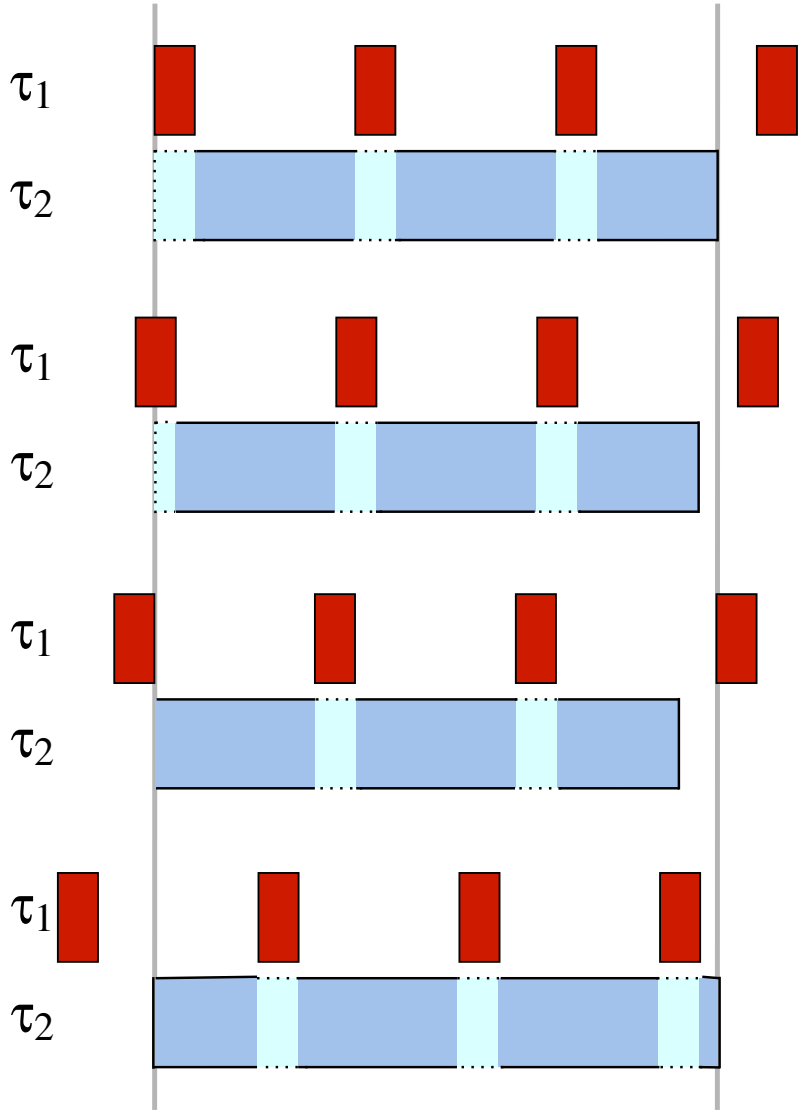
Example: Two Periodic Tasks



How about a preemptive schedule with higher priority for red task?

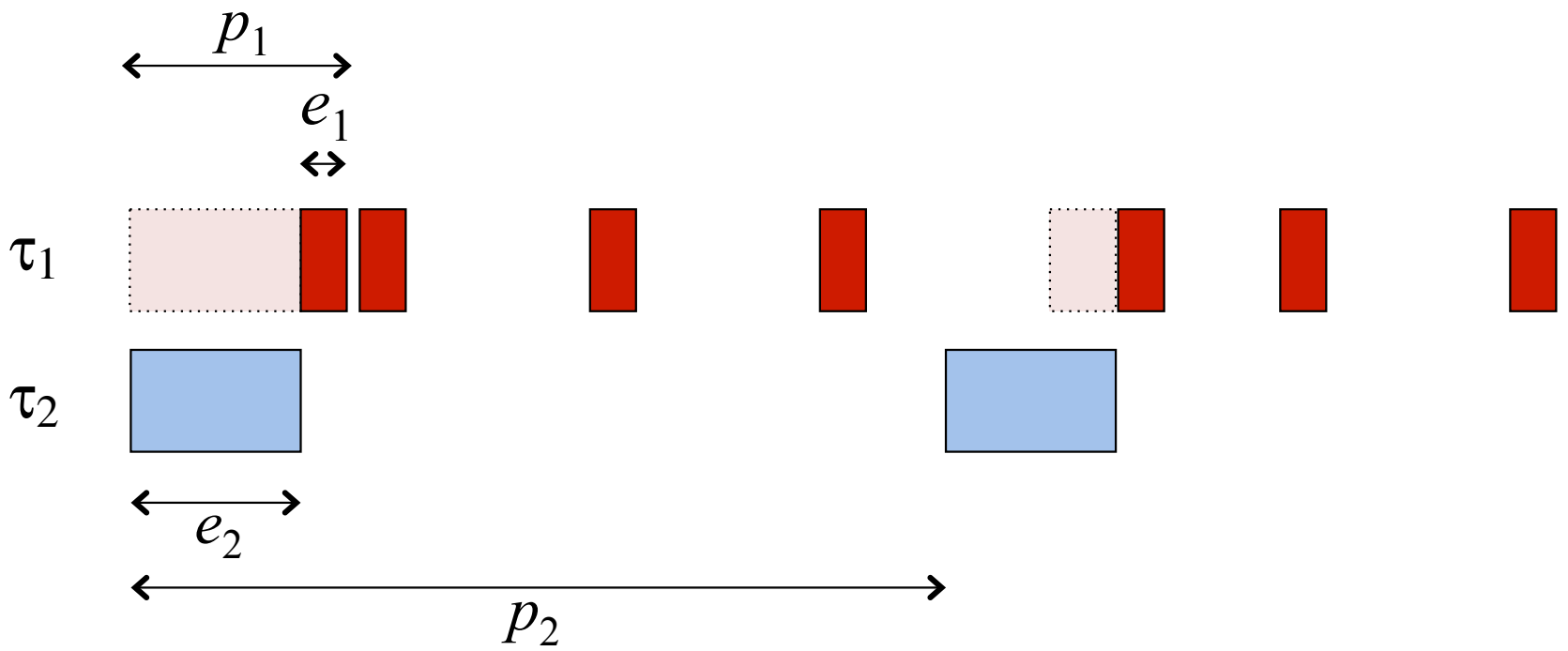
Yes!

Worst Case Response Time



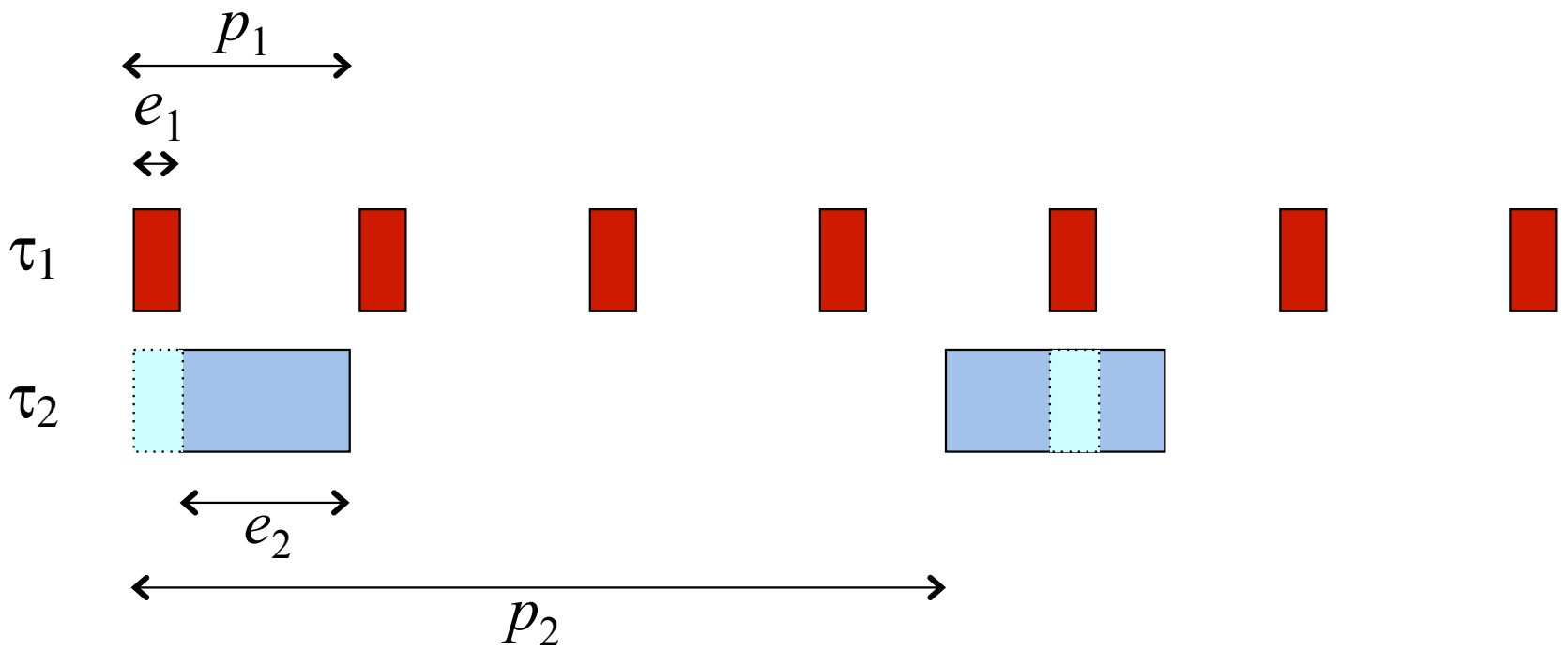
worst case response time: o_2

Non-RM Schedule Feasible



Condition for feasibility: $e_1 + e_2 \leq p_1$

RM Schedule Also Feasible



$e_1 + e_2 \leq p_1 \rightarrow$ feasible schedule

Comments

- Proof can be extended to an arbitrary number of tasks.
- Proof only gives optimality w.r.t. feasibility, not other optimality criteria.
- Practical implementation:
  - Timer interrupt at greatest common divisor of the periods.
  - Multiple timers.
- Note RM schedulers do not always achieve 100 percent utilization.

11.3 Earliest Deadline First

Jackson's **earliest due date (EDD)** Algorithm (1955)

- Given n independent one-time tasks with deadlines $d_1, \ldots, d_n$, schedule them to minimize the maximum lateness, defined as

$$L_{max} = \max_{1 \le i \le n} \{f_i - d_i\}$$

  where f_i is the finishing time of task i. Note that this is negative iff all deadlines are met.
- Earliest Due Date (EDD) algorithm: Execute them in order of non-decreasing deadlines.
  - Sort tasks such that $d_1 \le d_2 \le \ldots \le d_n$.
  - Then, executes $\tau_1, \tau_2, \ldots, \tau_n$.

11.3 Earliest Deadline First: EDD

- Note that this does not require preemption.
- Minimizes the maximum lateness as compared to all other possible orderings.
- Does not support arrival of tasks, thus no periodic or repeating tasks.

11.3 Earliest Deadline First

- Horn's **earliest deadline first (EDF)** Algorithm (1974) extends EDD to allow tasks to arrive at any time.

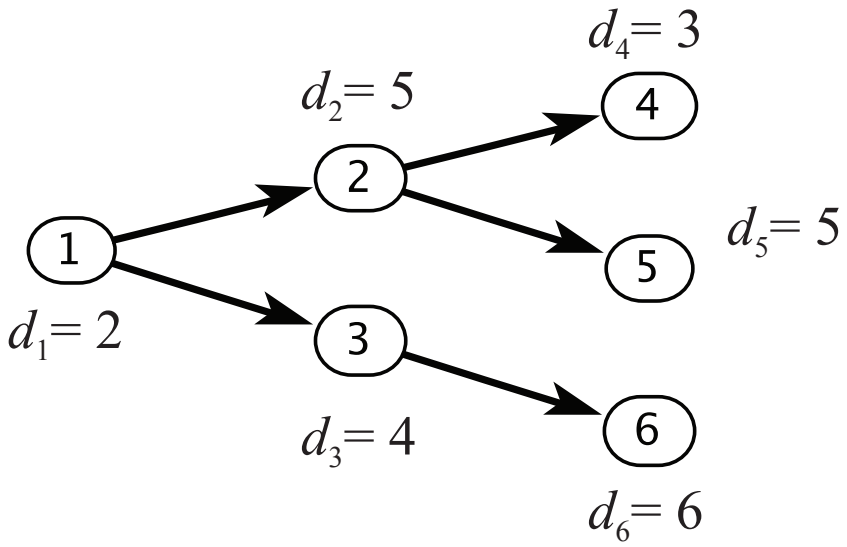
> Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all arrived tasks is optimal w.r.t. minimizing the maximum lateness.

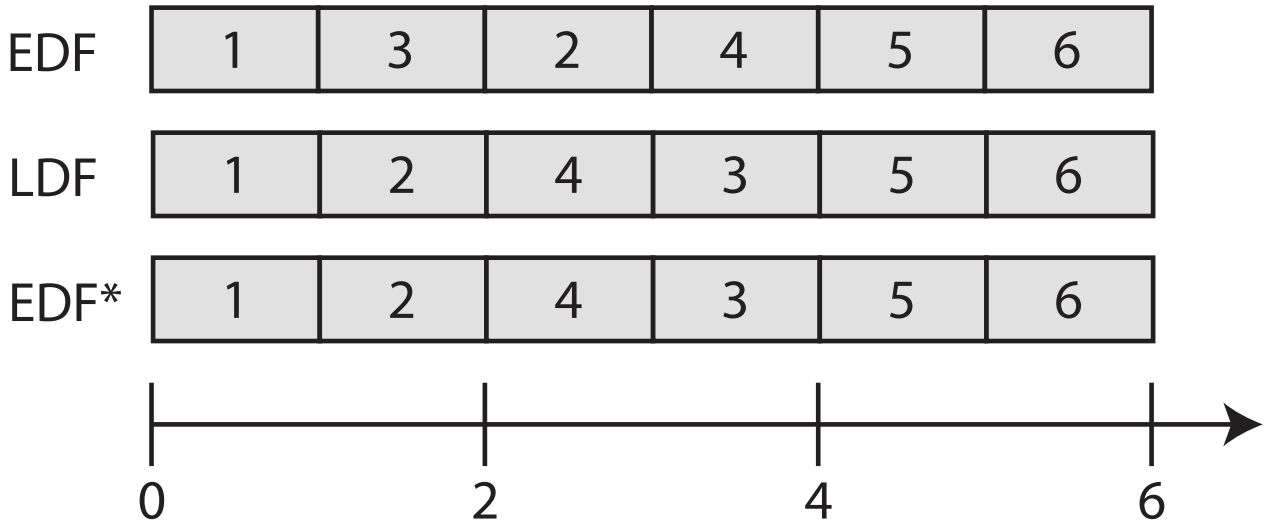
- EDF requires the scheduler to always execute the task with the earliest deadline among all arrived tasks.
- EDF has a dynamic priority making it more difficult to implement.
  - A repeated task may be assigned with different priority at different time.
- EDF can be applied to periodic tasks as well as aperiodic tasks by making deadline be the end of the period.

Comparison of EDF and RMS

- Favoring RMS:
  - Scheduling decisions are simpler (fixed vs. dynamic priorities required by EDF).
    - EDF scheduler must maintain a list of ready tasks that is sorted by priority.
- Favoring EDF:
  - Since EDF is optimal w.r.t. maximum lateness, it is also optimal w.r.t. feasibility while RMS is only optimal w.r.t. feasibility.
  - EDF can achieve full utilization where RMS fails to do that.
  - EDF results in fewer preemptions in practice, and hence less overhead for context switching.
  - Deadlines can be different from the period.

11.3.1 EDF with Precedences

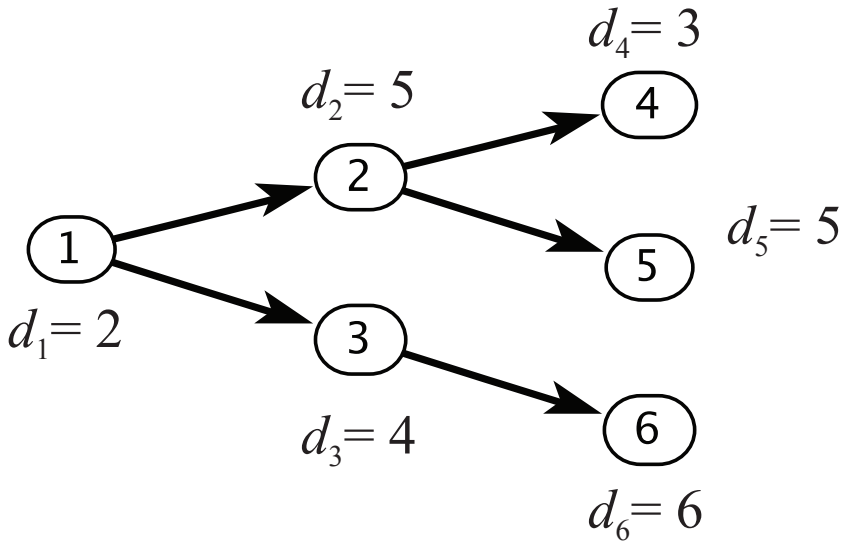


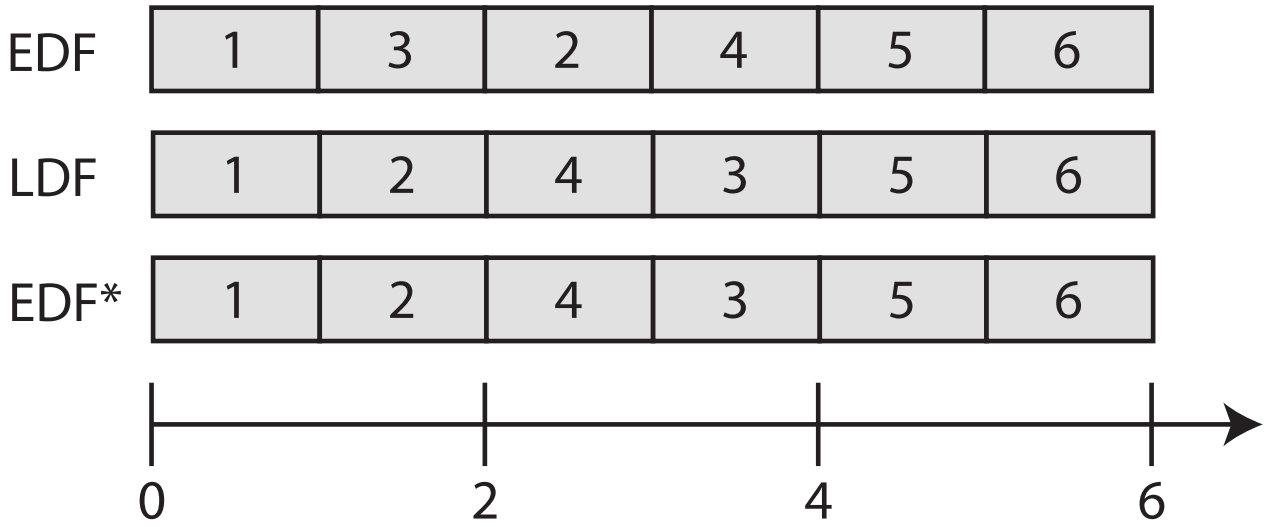


Latest Deadline First (Lawler, 1973)

- **Latest deadline first** (LDF) builds a schedule backwards.
- Given a **precedence graph**, starting from the leaf nodes, choose a node with no other dependent nodes and with the latest deadline to be scheduled last, and work backwards.
- LDF is optimal in the sense that it minimizes the maximum lateness.
- It does not require preemption while EDF does.
- However, it requires that all tasks be available and their precedences known before any task is executed.
  - Does not support arrival of tasks.

LDF with Precedences





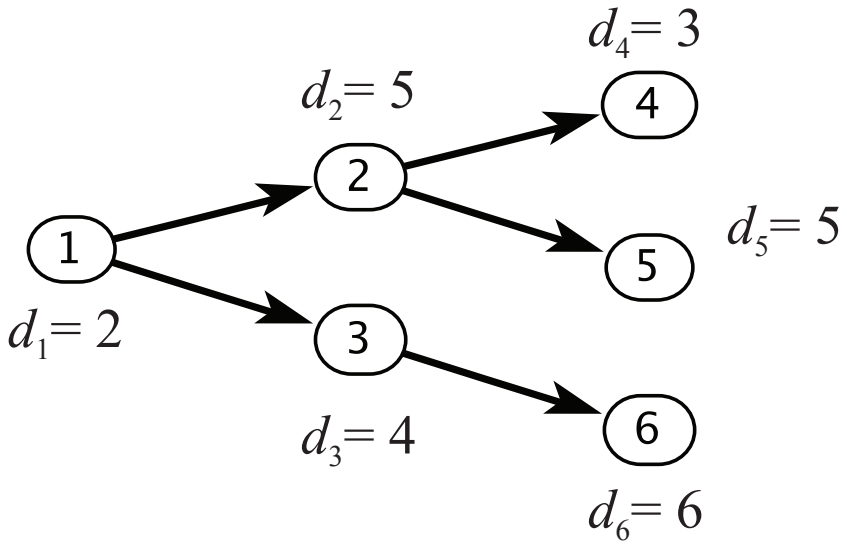
EDF with Precedences (Chetto et al., 1990)

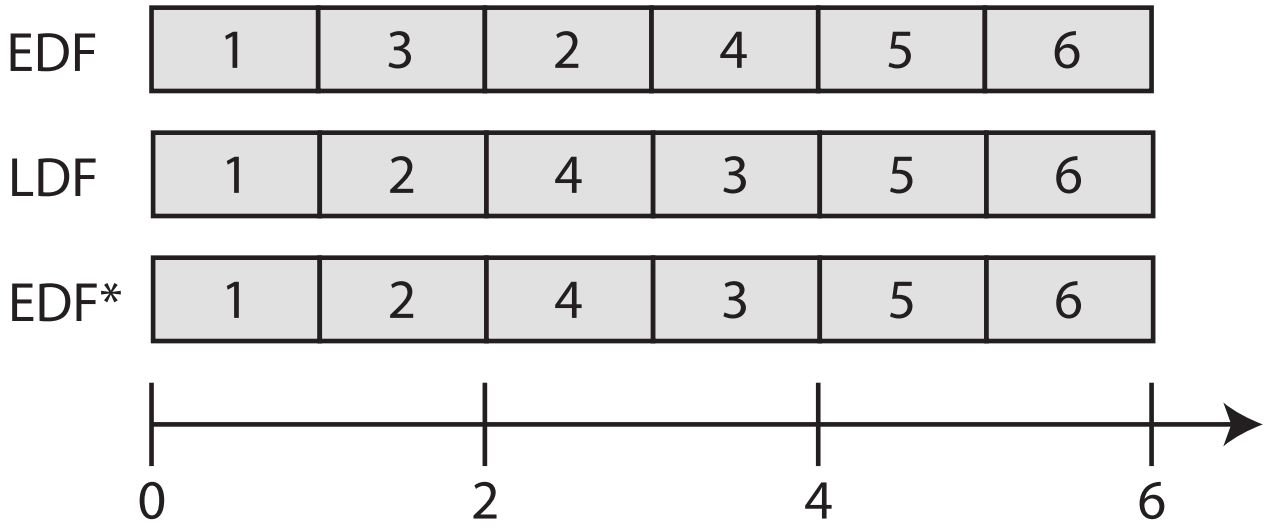
- With a preemptive scheduler, EDF can be modified to account for precedences and to allow tasks to arrive at arbitrary times.
- It adjusts the deadlines and arrival times according to the precedences.
- For a task $\tau_i \in T$, modify its deadline as follows:

$$d_i' = \min(d_i, \min_{j \in D(i)} (d_j' - e_j))$$

  where $D(i) \subset T$ are the tasks that immediately depend on i in the precedence graph.
- EDF with precedences (EDF*) is optimal in the sense of minimizing the maximum lateness.

EDF with Precedences

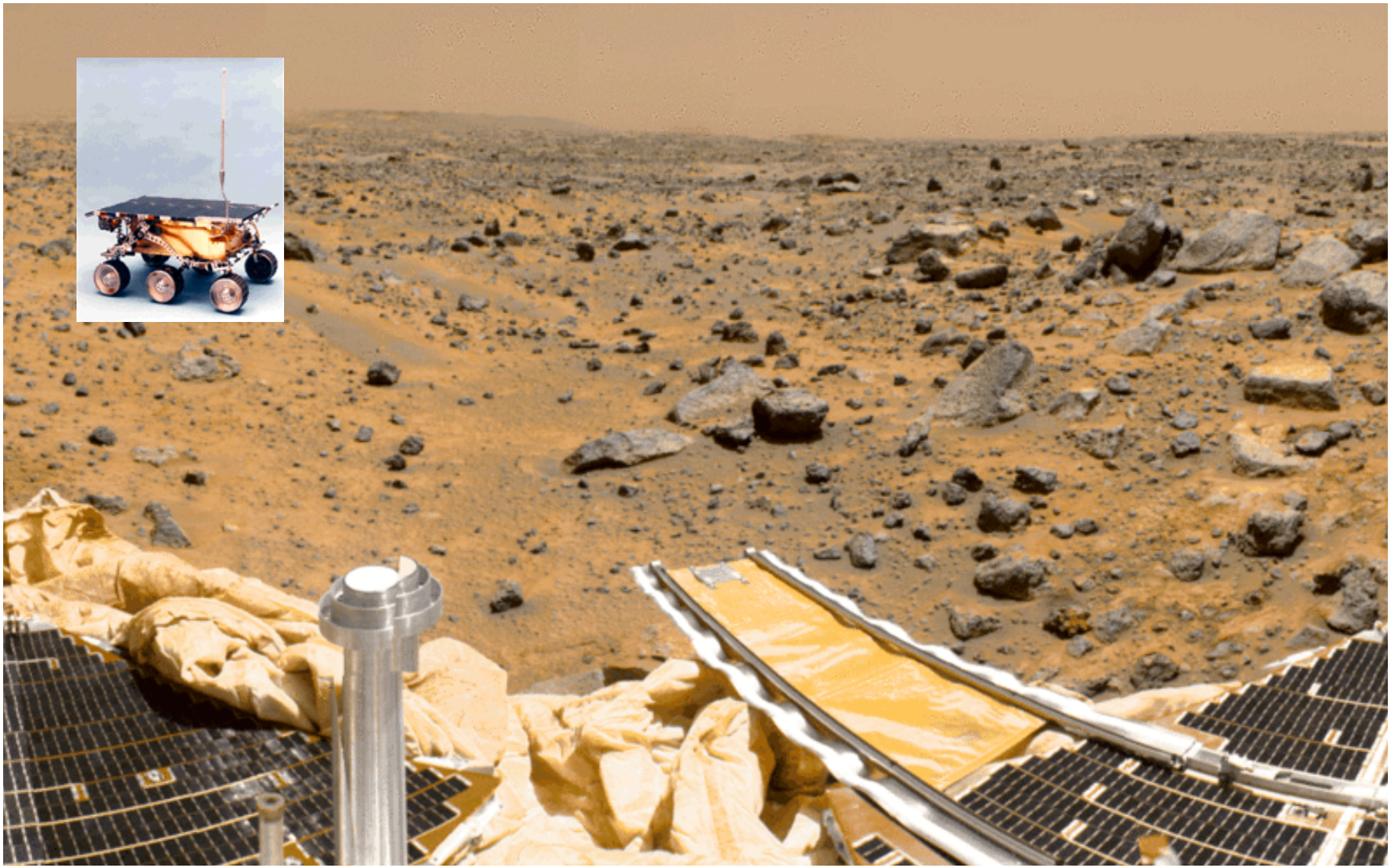




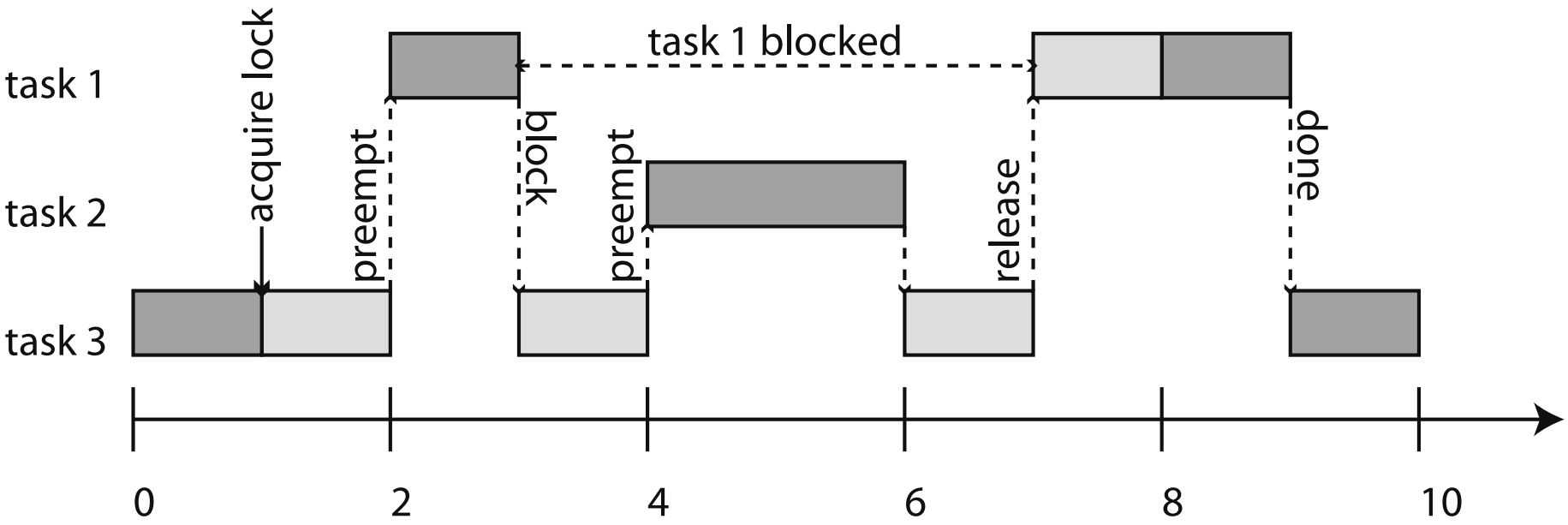
11.4 Scheduling and Mutual Exclusion

- Fixed priority scheduler always executes the enabled tasks with highest priority.
- When threads access shared resources, they need to use mutexes to ensure data integrity.
- Mutexes can also complicate scheduling.

Mars Pathfinder



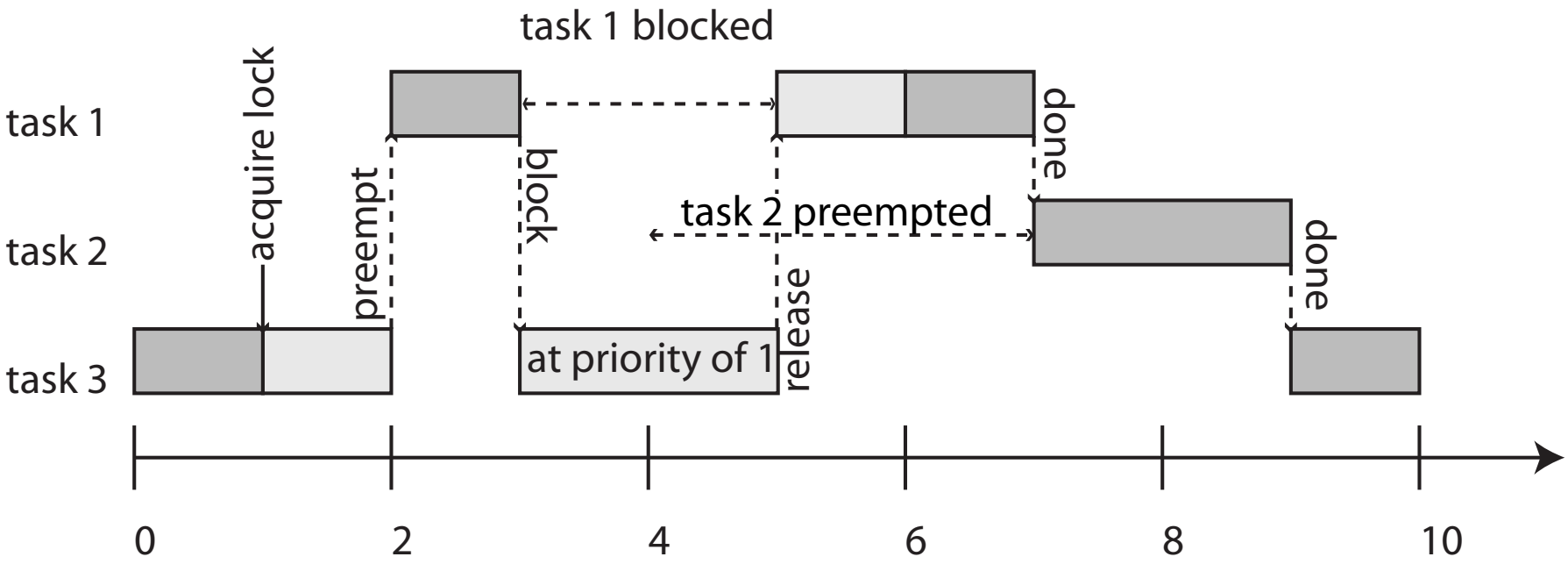
Priority Inversion: A Hazard with Mutexes



Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.

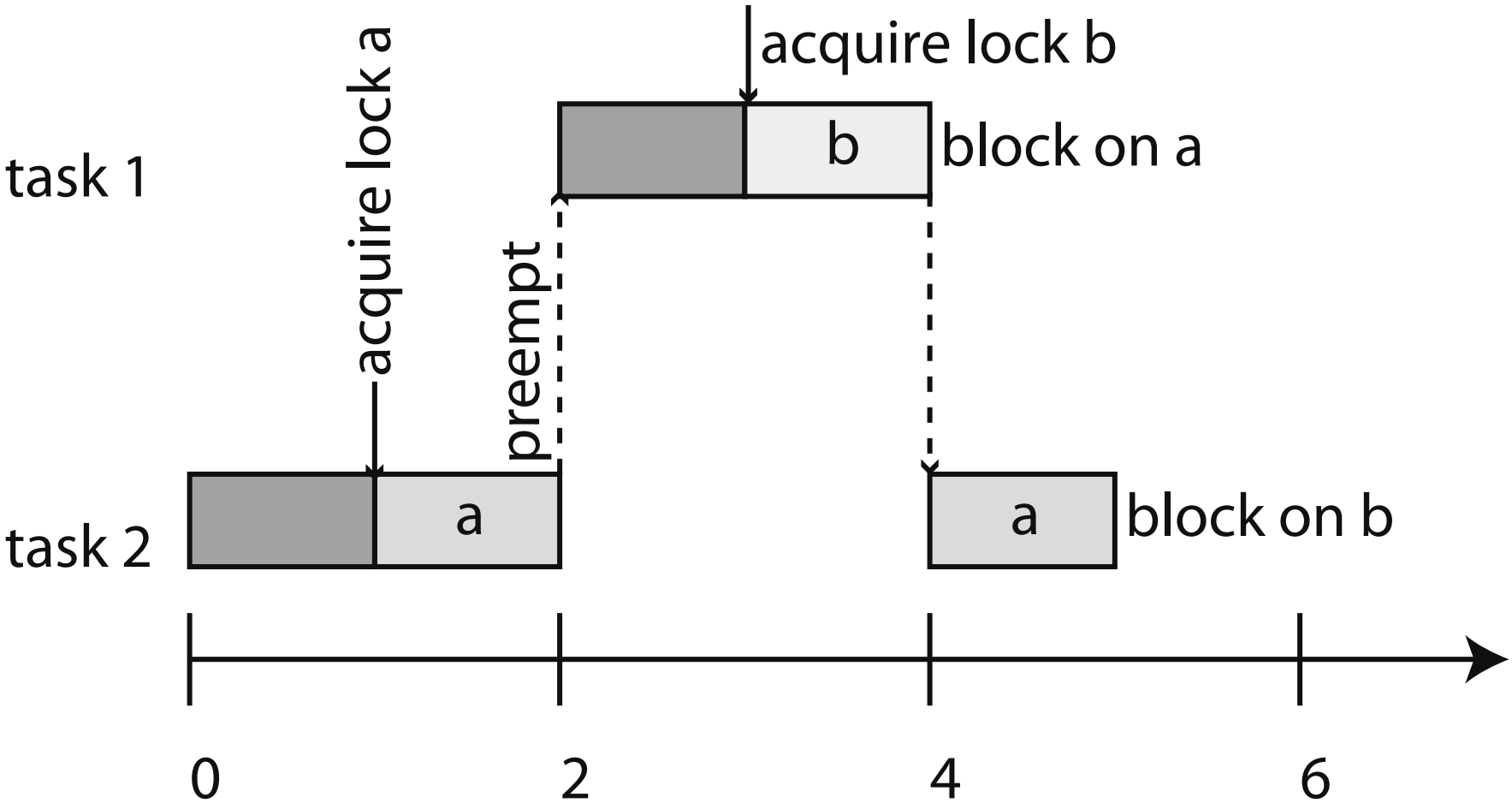
11.4.2 Priority Inheritance Protocol (PIP)

- When a task blocks attempting to acquire a lock, the task holding the lock inherits the priority of the blocked task.



11.4.3 Priority Ceiling Protocol

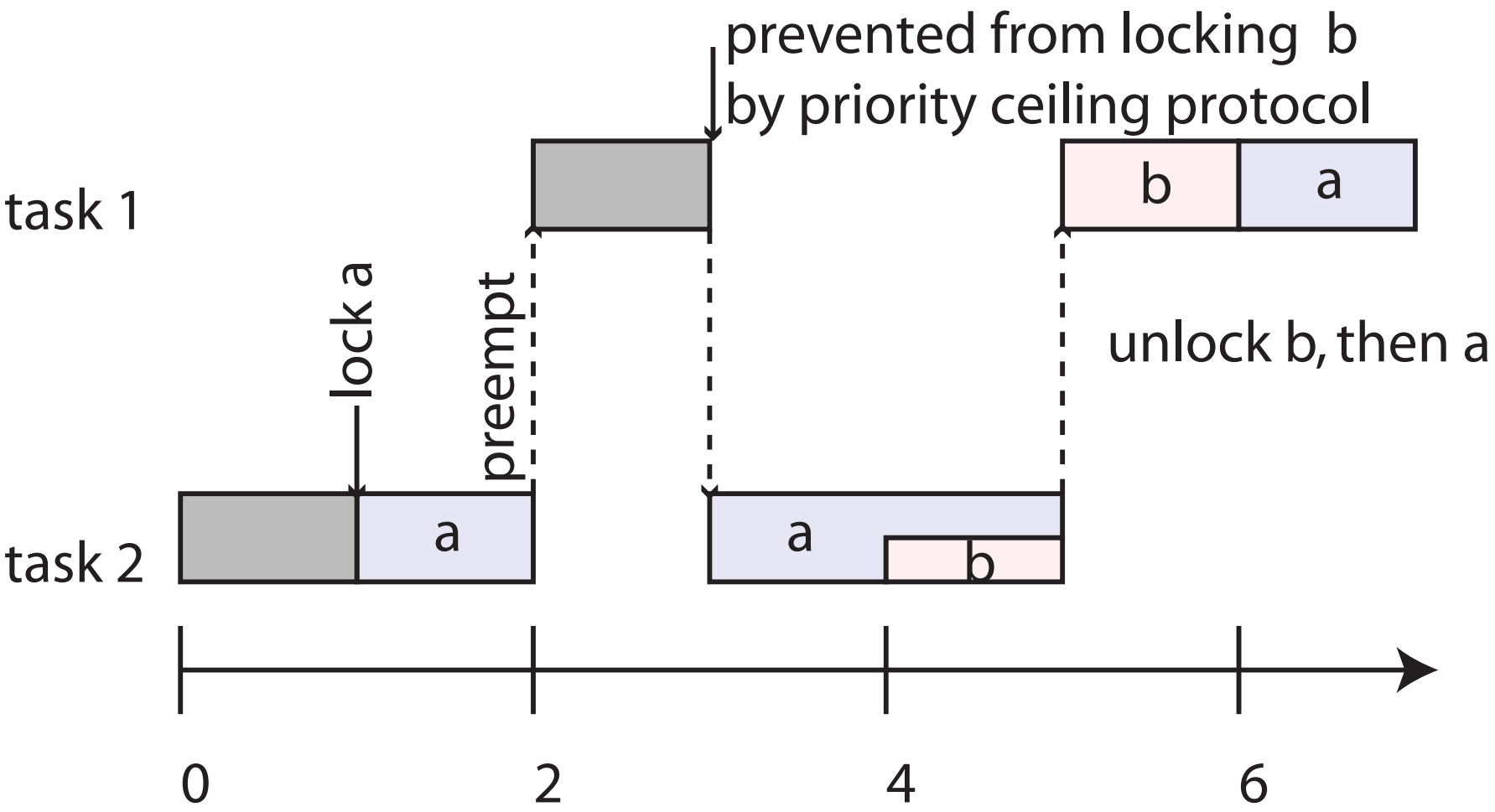
Priorities can be used to remove certain kinds of deadlocks.



11.4.3 Priority Ceiling Protocol (PCP)

- Every lock or semaphore is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it.
  - Can one automatically compute the priority ceiling?
- A task τ can acquire a lock only if the task's priority is strictly higher than the priority ceilings of all locks currently held by other tasks.
  - Intuition: task τ will not later try to acquire these locks held by other tasks.
  - Locks that are not held by any task don't affect the task.
- This prevents deadlocks.
- There are extensions supporting dynamic priorities and dynamic creations of locks.

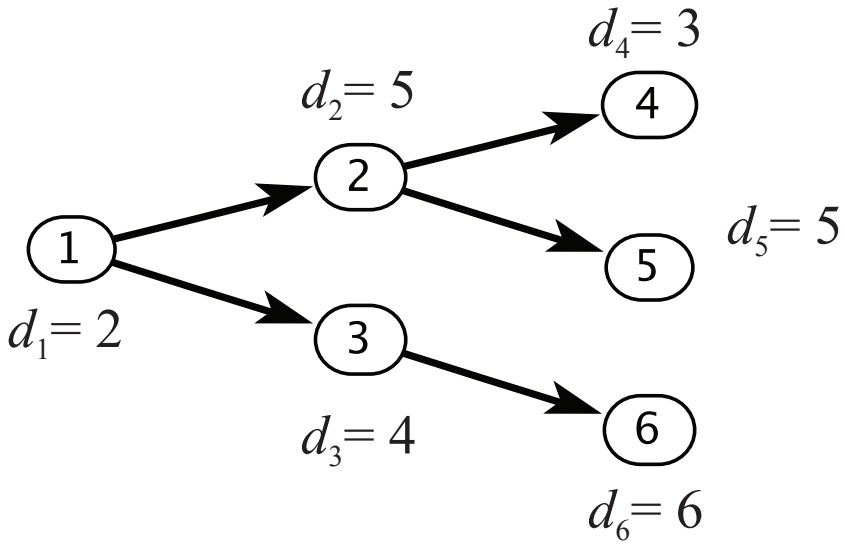
11.4.3 Priority Ceiling Protocol (PCP)

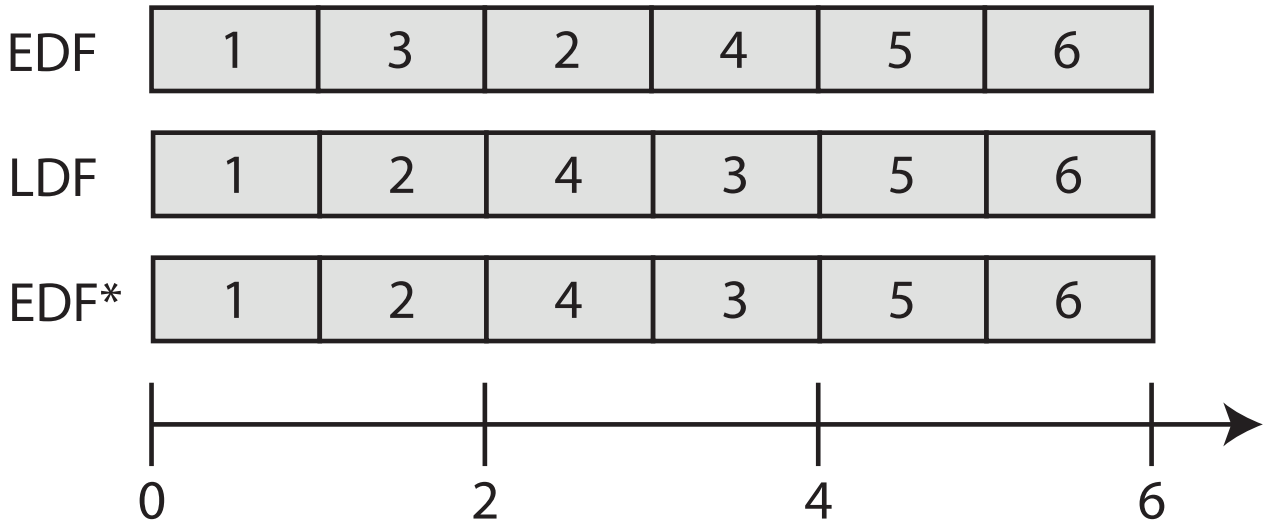


11.5 Multiprocessor Scheduling

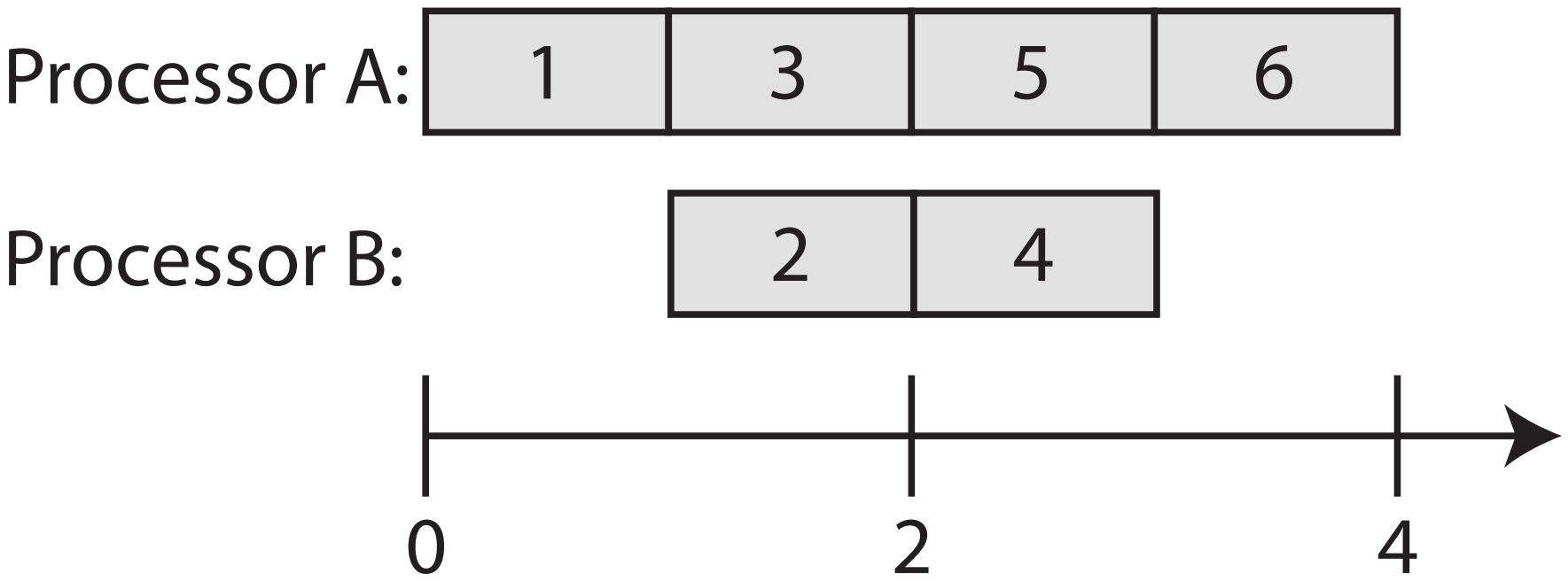
- Scheduling a fixed finite set of tasks with precedence on a finite number of processors with goal to minimize execution time is NP-hard.
- **Hu level scheduling** algorithm assigns priority to task τ based on the *level*.
- Level is the greatest sum of execution times of tasks on a path in the precedence graph from τ to another task with no dependents.
- It is a **critical path** method that is not optimal, but approximates an optimal solution for most graphs.
- **List scheduler** assigns tasks to processors based on priorities.

Precedence Example Revisited





A Two Processor Parallel Schedule

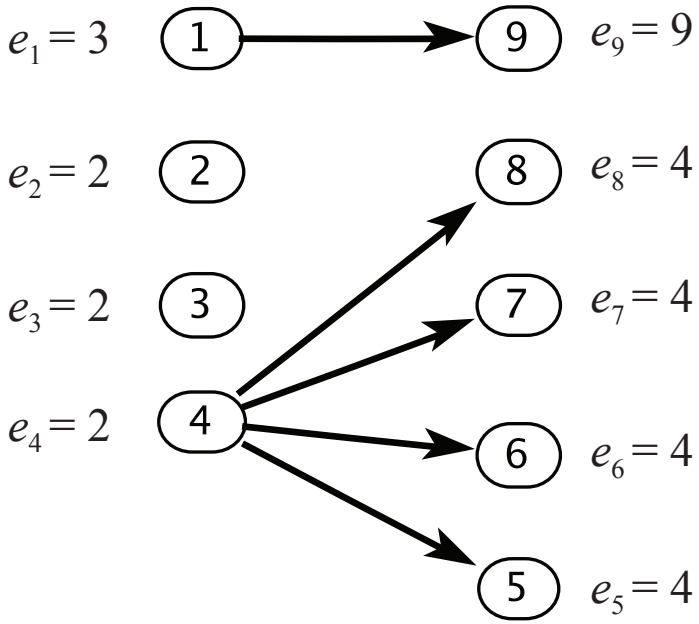


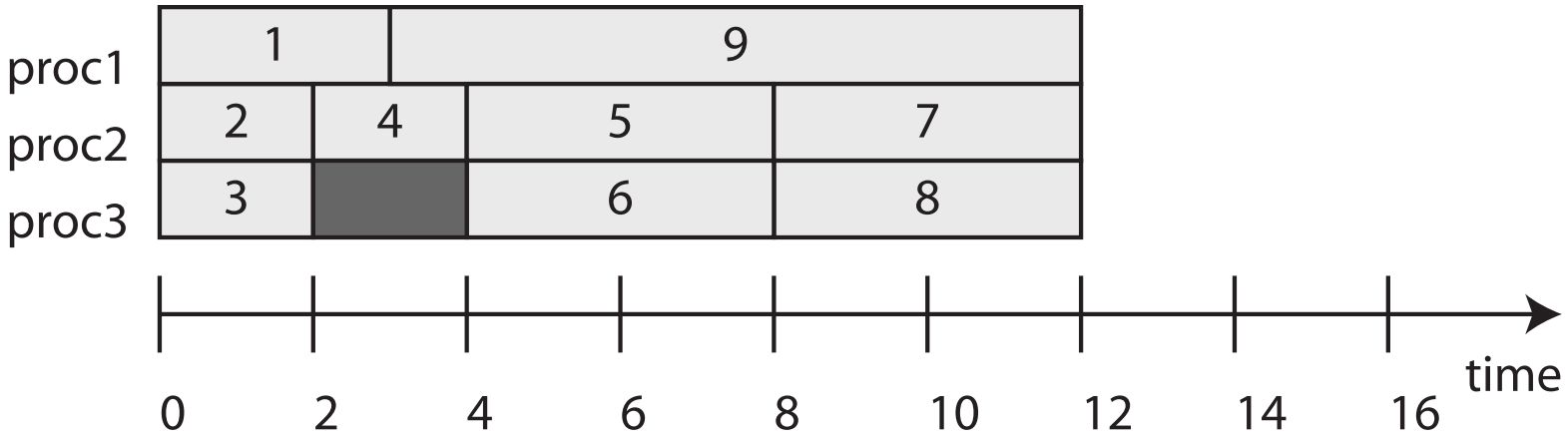
11.5.1 Scheduling Anomalies and Brittleness

- All thread scheduling algorithms are *brittle* (i.e., small changes can have big, unexpected consequences).
- Let us consider a schedule for a multiprocessor (or multicore).
- **Theorem (Richard Graham, 1976)**:

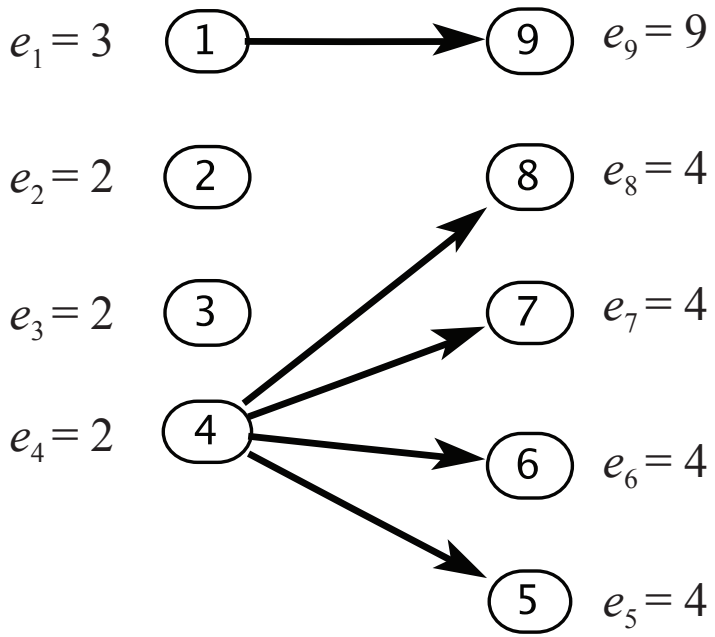
  If a task set with fixed priorities, execution times, and precedence constraints is scheduled according to priorities on a fixed number of processors, then increasing the number of processors, reducing execution times, or weakening precedence constraints can increase the schedule length.

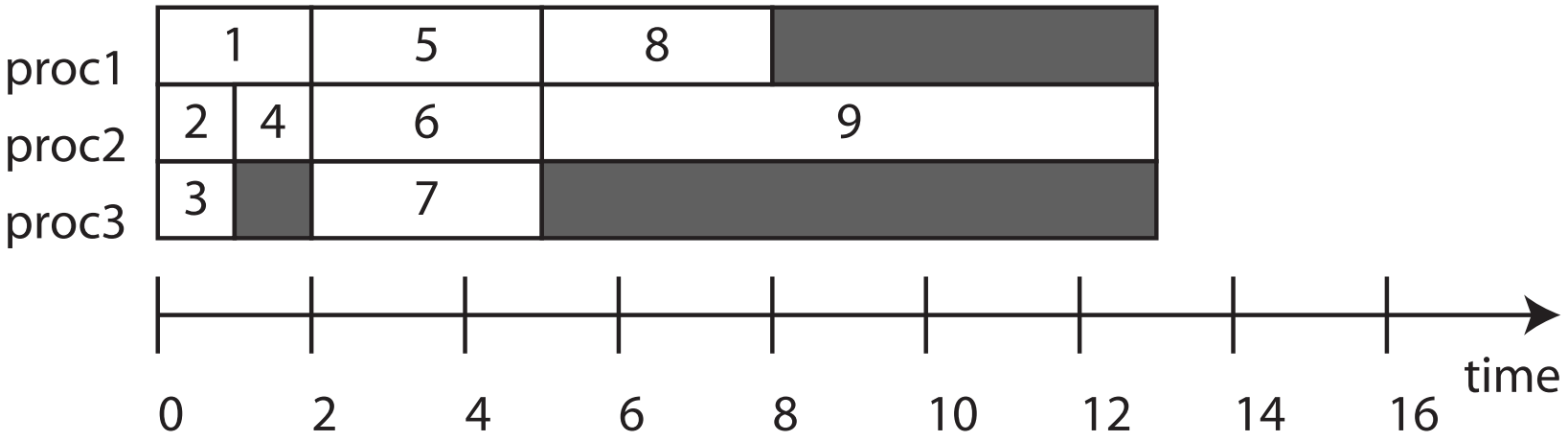
Richard's Anomalies



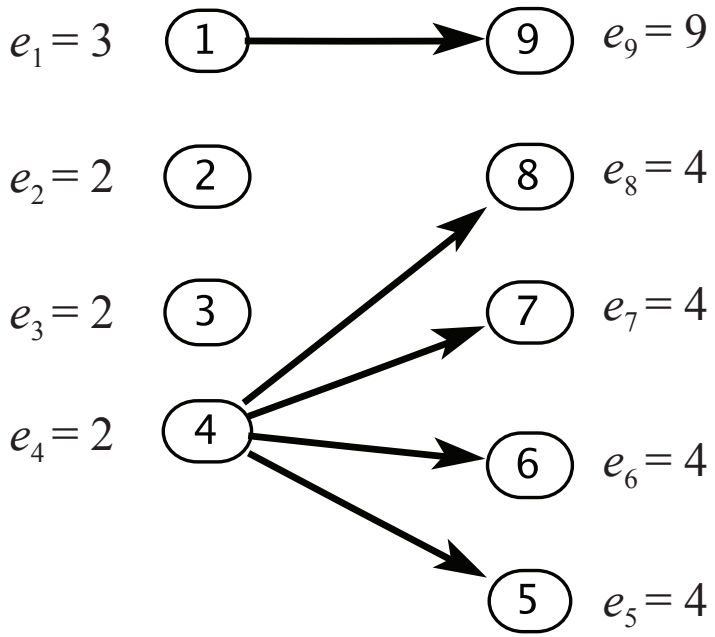


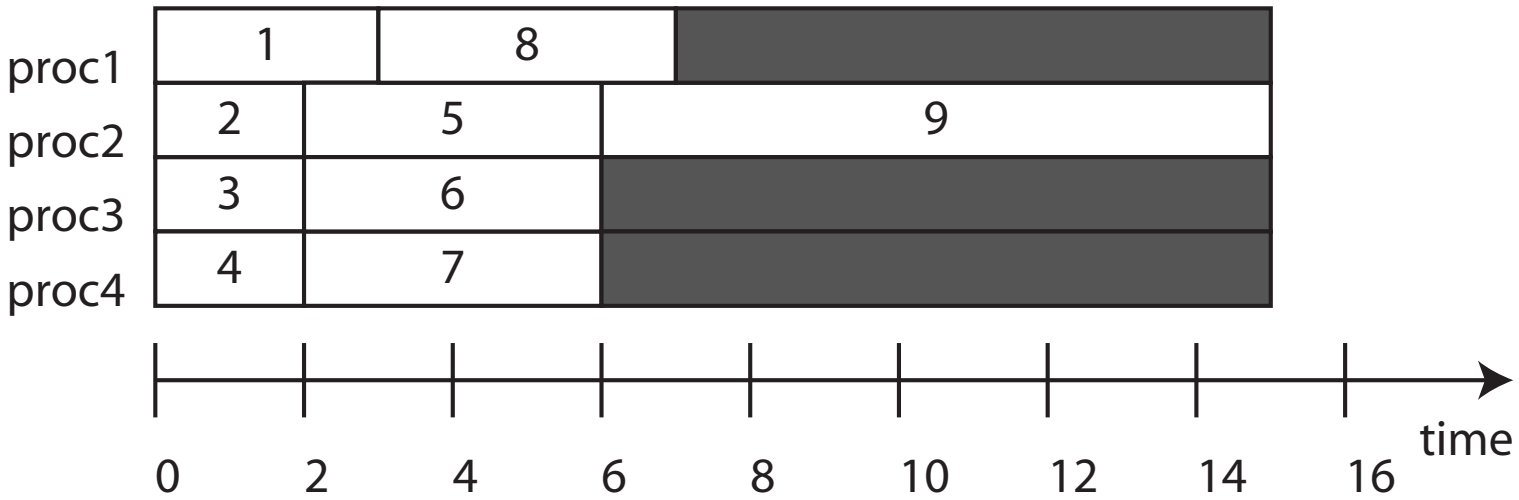
Richard's Anomalies: Smaller Computation Time





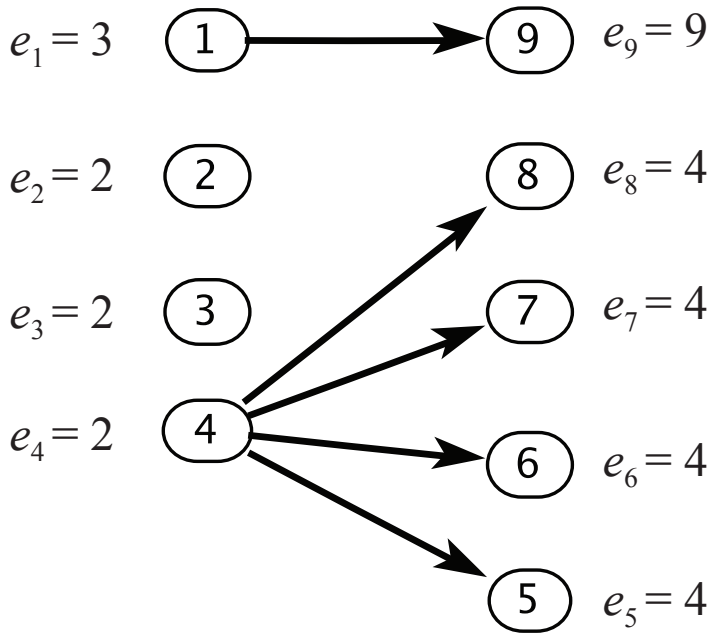
Richard's Anomalies: More Processors

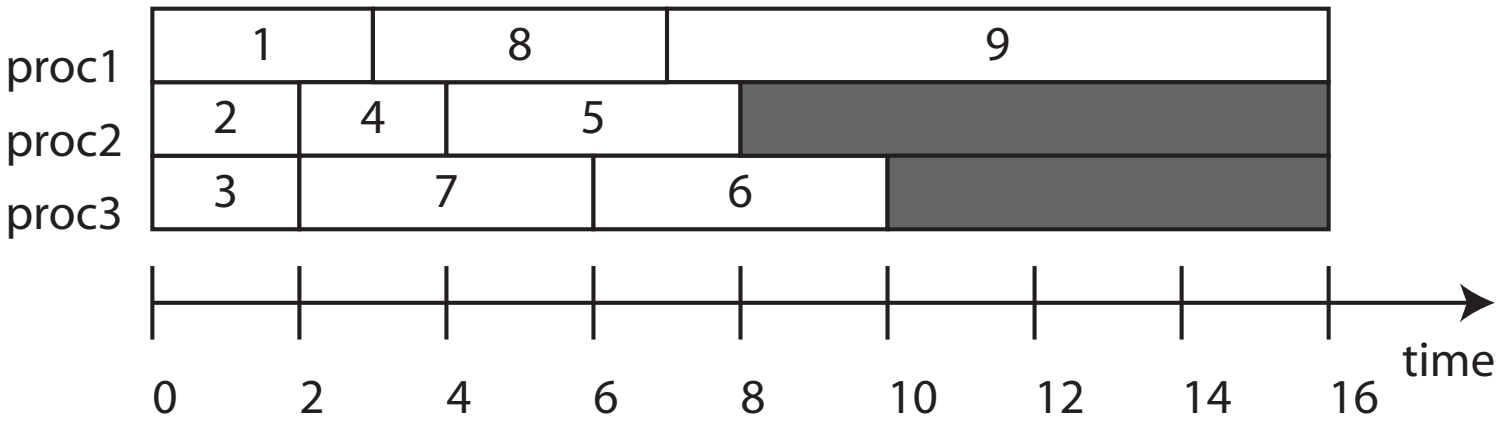




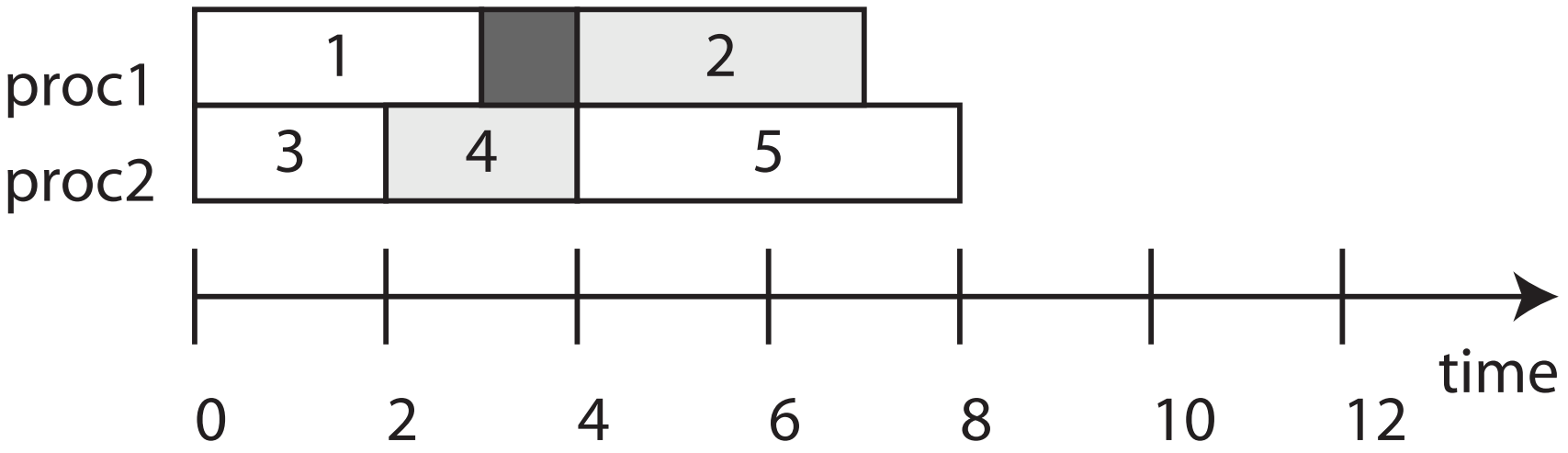
Richard's Anomalies: Less Precedences

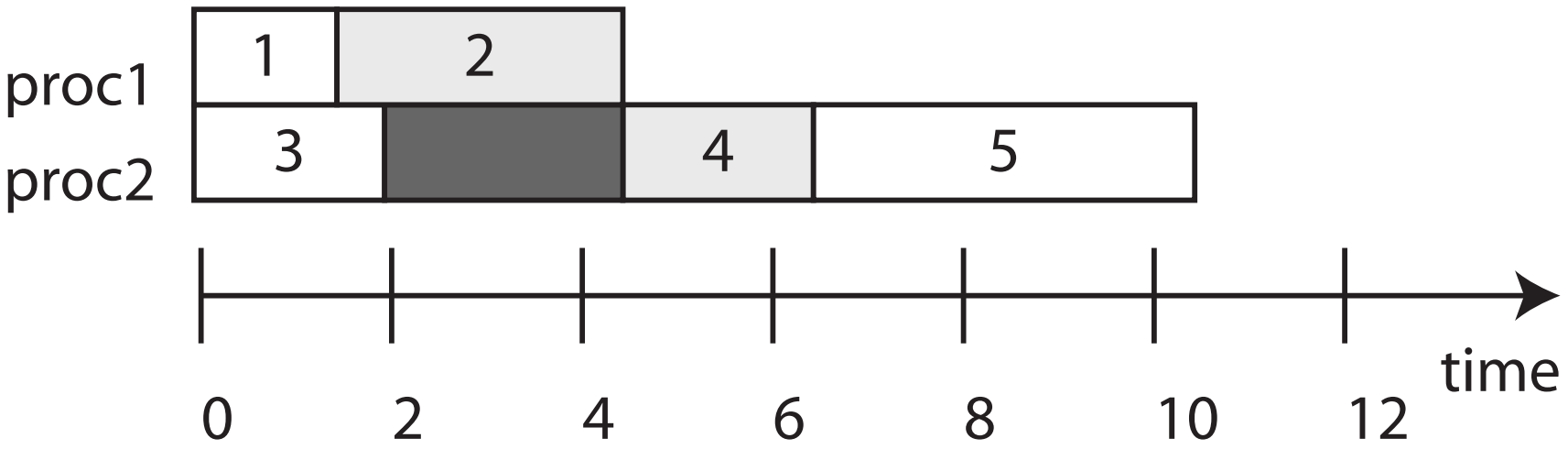
Suppose precedences between task 4 and tasks 7 and 8 are removed.





Richard's Anomalies with Mutexes





Concluding Remarks

- Scheduling is inherently difficult
  - SW execution time hardly to predict accurately.
  - The actual dynamic behavior can be quite different.
- Timing behavior under all known task scheduling strategies is brittle.
- Small changes can have big (and unexpected) consequences.
- Unfortunately, since execution times are so hard to predict, such brittleness can result in unexpected system failures.