SJSU SAN JOSÉ STATE UNIVERSITY

Charles W. Davidson College of Engineering

Department of Computer Engineering

Real-Time Embedded System Co-Design
CMPE 146 Section 1
Fall 2024

Task Scheduler

SJSU SAN JOSÉ STATE UNIVERSITY

Tasks

- Let’s loosely define a task as a group of program instructions designed to achieve a specific objective
- Tasks are typically determined and well defined when the system is being built in the design phase
- In embedded applications, quite often, execution of a task is triggered by some event
  - Typically it lasts for a finite duration before the same event occurs again
- A task generally can be organized as a function
  - In most embedded systems, there is only one program to execute
    - All the tasks are contained in the same program
    - Unlike in conventional computer systems where tasks can be organized in multiple executable program files
  - Note that a single function can be used to perform multiple tasks
  - For example:

```
process_sensor_input(Sensor_A);
process_sensor_input(Sensor_B);
process_sensor_input(Sensor_C);
```

SJSU SAN JOSÉ STATE UNIVERSITY

Benefits of Task Concept

- Effective way to build a software system with the modular concept
  - Similar to hardware design that has different functional modules
- Clear division of system development effort and functionality
  - Allows team of developers to work in parallel
- System design can be understood more easily
  - Make future enhancements easier
  - Reduces future maintenance cost
    - Embedded products can last for many years

Task Types

We generally have three types of task in embedded applications

- Periodic tasks
  - Time between two consecutive executions is fixed
    - The time (period) is known
  - For example, a weather station gets a temperature reading once every minute
- Sporadic tasks
  - Time between two consecutive executions may differ widely
    - The time is totally random
  - Can be arbitrarily short or long
  - For example, task triggered by a touchpad input device of a game console
    - The device may be untouched for days , then it is being used in rapid fashion when the user is playing game
- Aperiodic tasks
  - Time between two consecutive executions may follow a known probability distribution function
  - For example, the data packet arrival rate at a communication port may follow a Poisson distribution

Simple Task Scheduling Schemes

For simple embedded applications, a simple scheduling mechanism is probably adequate

- Round Robin
- Round Robin with Interrupts
- Queue-based

Round Robin

- Well-known principle that is widely adopted in many disciplines
  - A very simple scheduling algorithm that can be easily implemented
- All tasks are treated equally
  - Every task takes its turn sequentially to be run
  - A while loop (super loop) in main() calls the task functions sequentially
    - Each function checks for particular event(s). If no event occurs, it just exits immediately.

```
while (true)
{
    task1();
    task2();
       ⋮
    taskN();
}
```

- Suitable for simple embedded systems that can tolerate potentially slow response time
  - A task’s worst-case response time depends on sum of execution time of all other tasks

Round Robin with Interrupts

- Based on the simple round-robin scheme, add interrupt handling to provide fast response (small delay) for certain events
  - Tasks are in the foreground (or at higher level) where most normal processor execution takes place
  - Interrupt handlings are in the background (or at lower level)
- All tasks are treated equally at the higher level
  - Interrupt handlers pre-empt the round-robin loop to process the interrupt events

```
while (true)
{
      task1();
      task2();
         ⋮
      taskN();
}
```

```
void isr1()
{
      ⋮
}
```

```
void isr2()
{
      ⋮
}
```

....

```
void isrM()
{
      ⋮
}
```

- Task and ISR can work together to accomplish a more complex task that allows for a longer response time
  - ISR provides immediate response and task responses as more signals or events are collected
  - Use shared variables or data structures to communicate

Round Robin with Interrupts (cont'd)

- One example: Communication monitor handles commands from UART
  - When UART receives a character in the command string, it creates an interrupt
  - ISR quickly acknowledges UART controller and puts the received character in a shared buffer
  - A task keeps checking the buffer to see if a terminating newline is received
    - If newline received, the task handles the command and respond accordingly
    - Otherwise, it keeps checking continually
  - Without using interrupt, as in the simple round-robin scheme, some incoming characters may be lost
    - Characters can come in at such a high speed that the round-robin loop is not quick enough to invoke the handling task
- As in simple round-robin scheme, some tasks' worst-case response time remains the same, but urgent events can be handled very quickly
  - Overall system performance is better than the simple round-robin scheme
  - Extra circuitry is needed to handle the interrupt signals

Queue-based Scheduling

- We can add a queue to further improve the round-robin-with-interrupts scheme
- An ISR can invoke an upper-level task by placing the task's address in a FIFO (First-In-First-Out) queue when the task has something to do

```
while (true)
{
    if (queue_empty())
        continue;
    function = get_first_entry_from_queue();
    call function;
}
```

Task queue

T_A	T_B					

- In the previous communication monitor example, the ISR will place the handling task's address in the queue when a newline is received
- Reduces many unnecessary checking in the super loop when tasks are not ready to execute yet
  - No need to go through each task in the loop in each iteration
  - Can use low-power mode to reduce power consumption
    - Foreground super loop is only active when there is something to do

Priority-based Scheduling

- Better system response can be achieved by adding a priority attribute to a task
- With additional complexity, application development would be much easier with an OS
- In POSIX (Portable Operation System Interface), threads (tasks) are schedulable entities
- The OS scheduler selects one thread to execute on each processor at some point in time
- POSIX thread is associated with a numeric value, the scheduling priority
  - Used by the scheduler to determine when and how to schedule the thread

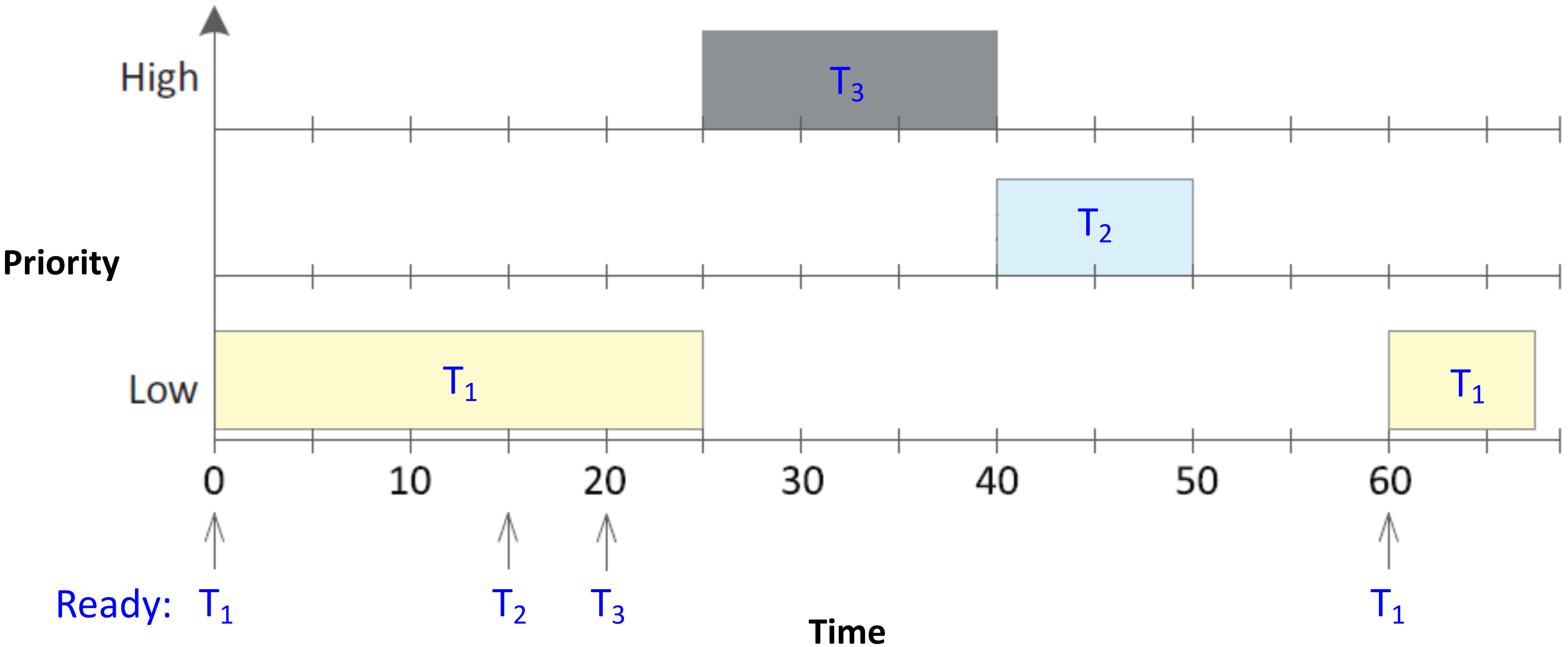
Priority-based Scheduling (cont'd)

- Three principles are used in priority-based scheduling in POSIX
  - Precedence principle
    - When no thread is running, and multiple threads are ready to be scheduled for execution, a thread with a higher priority is always scheduled prior to threads with a lower priority
  - Preemption principle
    - When a thread is currently running, and another thread with a higher priority becomes ready for execution, the higher-priority thread preempts the execution of the current thread
  - Fairness principle
    - When multiple threads, with the same priority, compete for use of the processing resources, the system's scheduling behavior is regulated by the scheduling policies

Precedence Principle

- The precedence principle is the basis of priority-based scheduling
- A <u>non-preemptive</u> scheduling approach illustrates the principle:

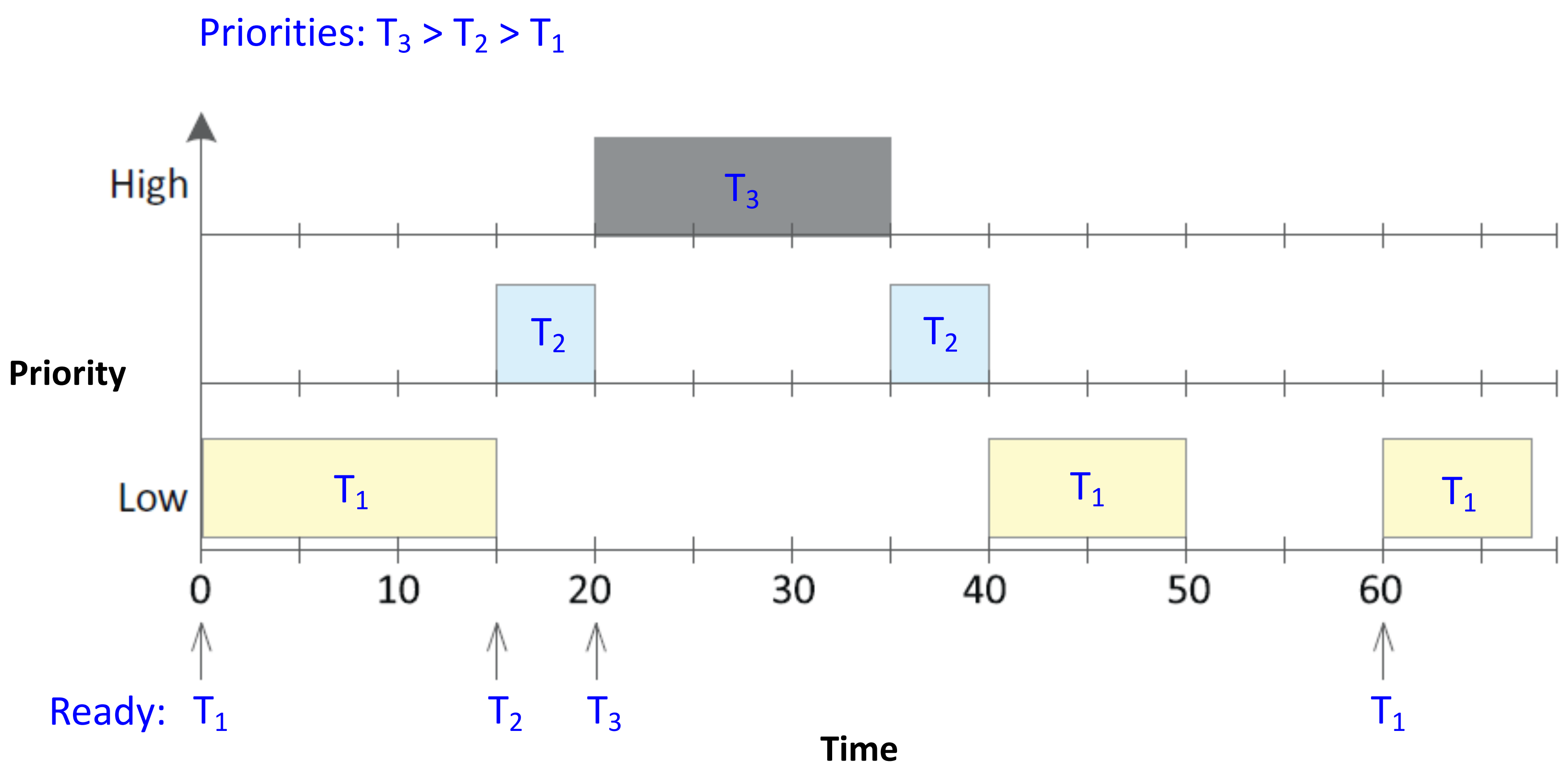
Priorities: $T_3 > T_2 > T_1$



- T_3 gets to execute before T_2 even though T_3 was ready later

Preemption Principle

- A preemptive scheduling approach illustrates the principle:

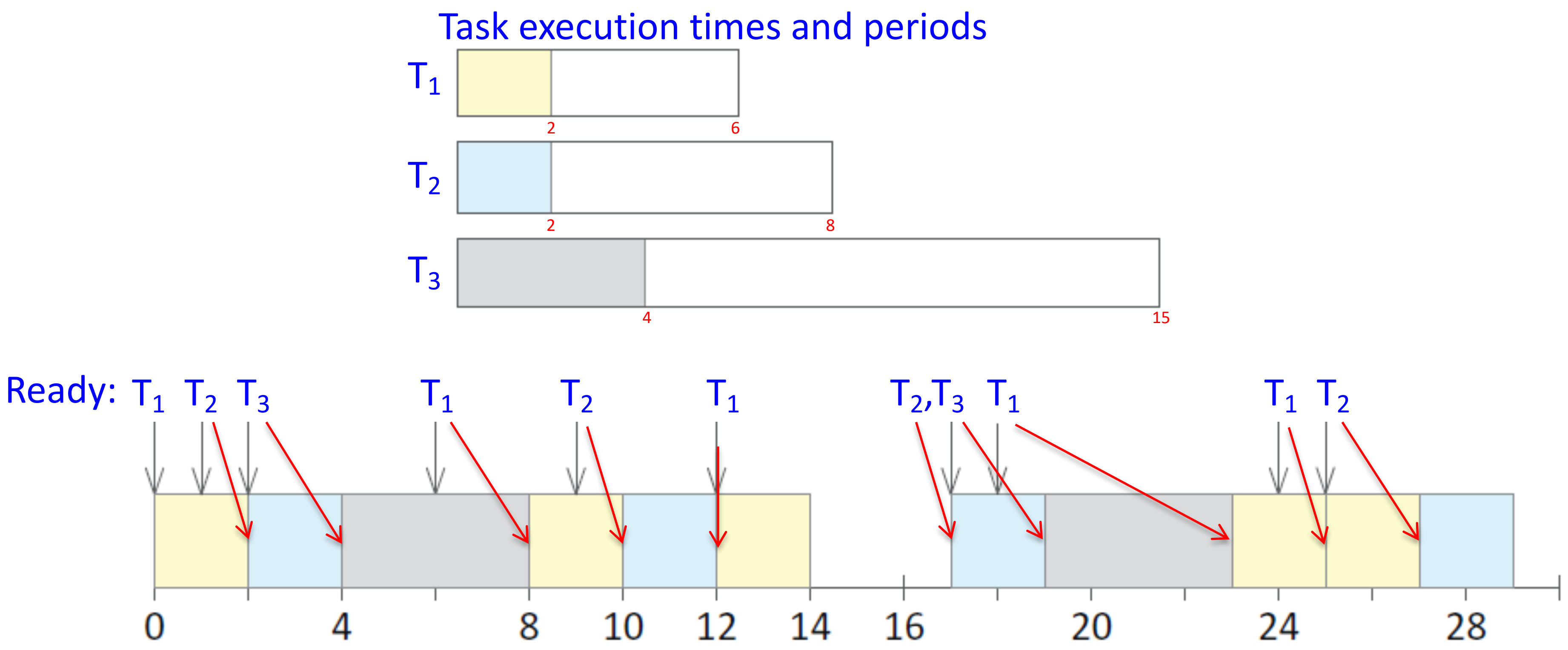


Fairness Principle

- We use scheduling policies to define the fairness rules for tasks with the same priority
- Two basic scheduling policies
  - FIFO (First-In-First-Out)
  - Round Robin

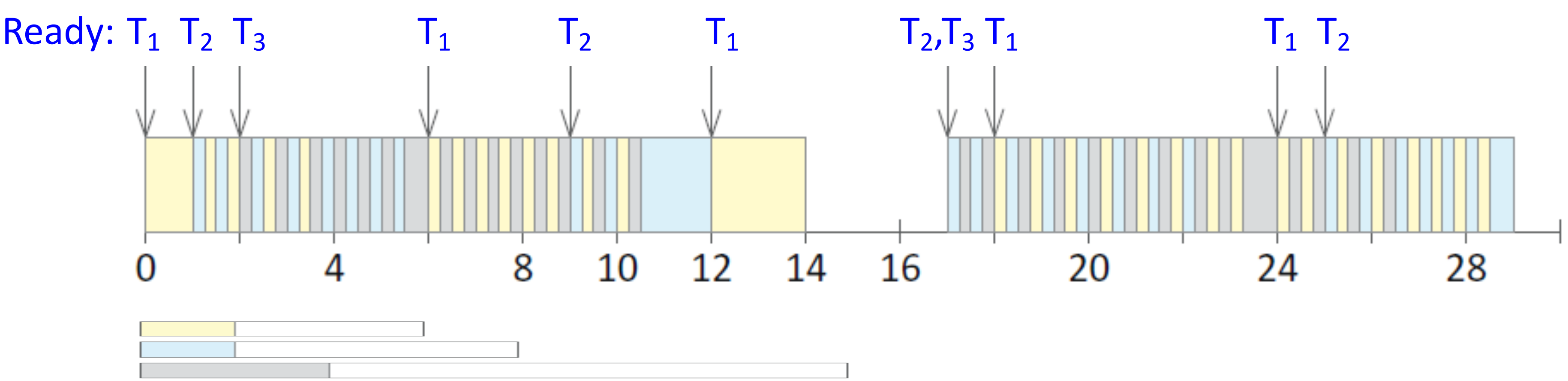
FIFO Scheduling Policy

- Applied to scheduling tasks with the same priority
- A task cannot be preempted by any other tasks with later entry time in the task-ready list
  - The task that entered the task-ready list first will run to its completion before other tasks are scheduled to run



Round-Robin Scheduling Policy

- Applied to scheduling tasks with the same priority
- Each task has a time slot (time quantum) for execution
  - A running task will be preempted when it has used up its time quantum and there are other ready tasks
- All the tasks in each task-ready list are scheduled in turn according to their orders
- A running thread, as soon as it has used up its time quantum, goes to the end of the task-ready list
- The task execution pattern from the previous illustration becomes:



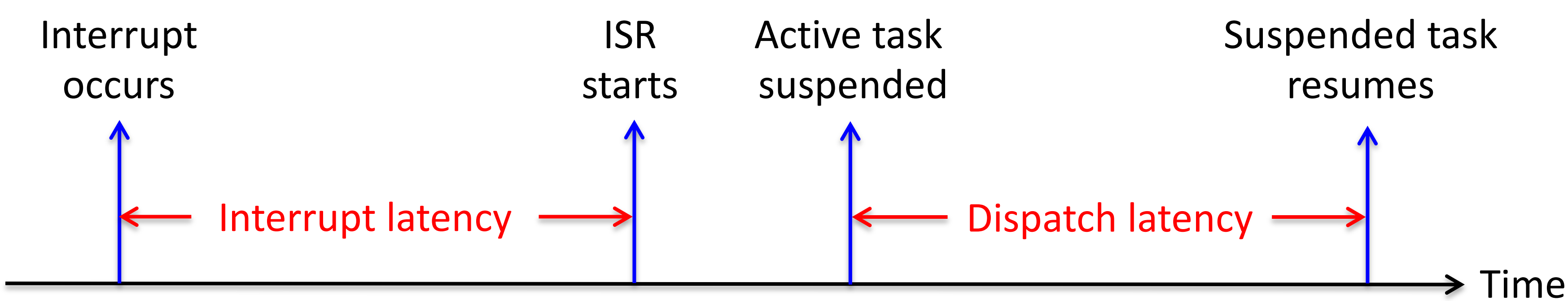
Task execution times and periods (from previous illustration)

RTOS Scheduler

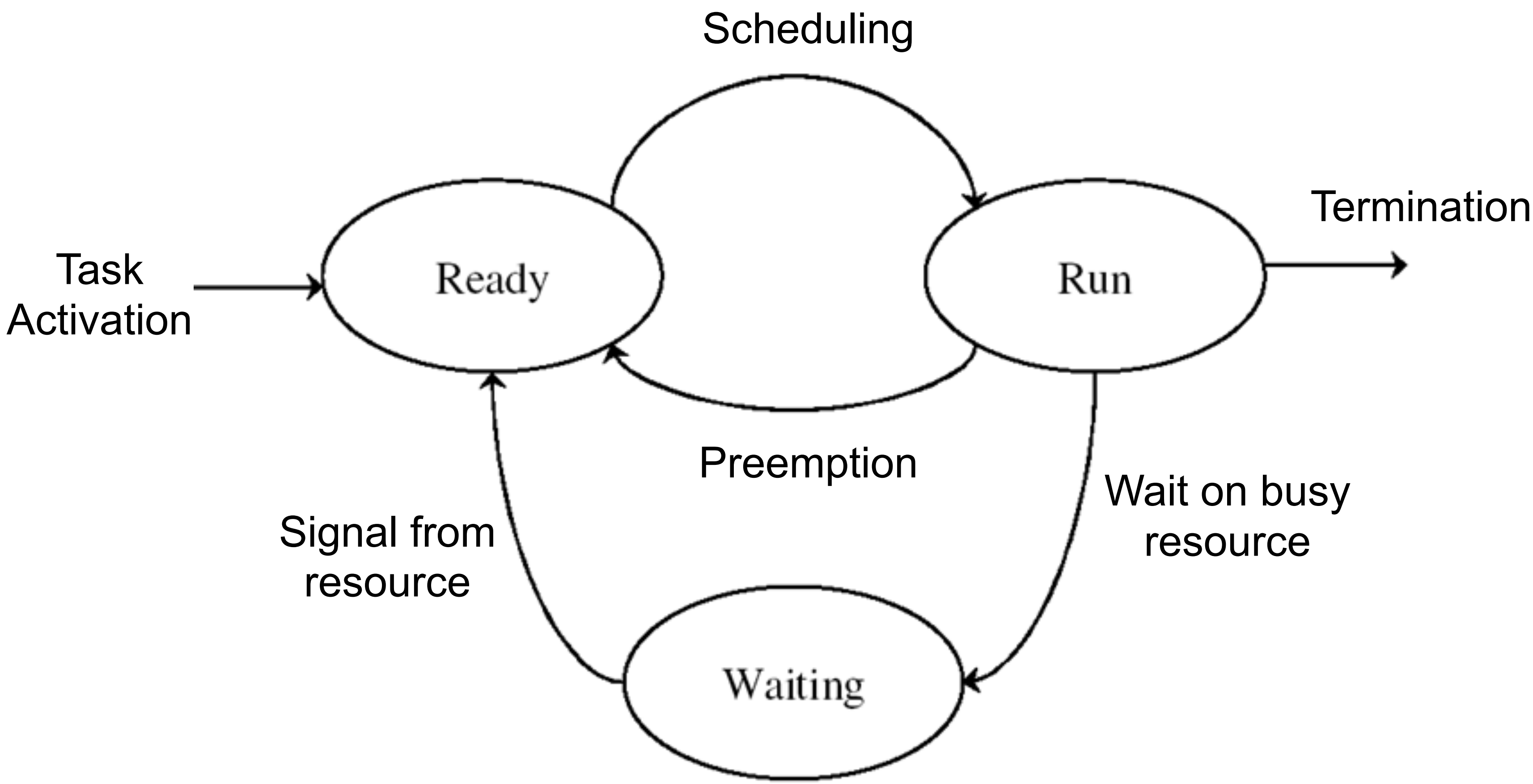
- A very important feature of a RTOS is to be able to respond immediately to an event, i.e., to run the handling task as soon as possible
- Urgency or importance of a task is usually indicated by its priority
  - Tasks need to run sooner are assigned with higher priorities
- As a result, the RTOS must support
  - Priority-based scheduler
  - Preemption of tasks
- To ensure real-time performance, a task currently running needs be preempted if a higher-priority task becomes ready to run
- Providing a preemptive, priority-based scheduler only guarantees soft real-time functionality (best-effort approach)
  - Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements
  - Making such guarantees may require additional scheduling features

Latencies

- When an event occurs, the system must respond to and service it as quickly as possible
- Two types of latencies affect the real-time performance under RTOS
  - Interrupt latency
    - Amount of time from the arrival of an interrupt to the start of ISR
  - Dispatch latency
    - Amount of time required for the RTOS to suspend one task and resume another



Task States


Scheduling
Termination
Task
Activation
Ready
Run
Preemption
Wait on busy
resource
Signal from
resource
Waiting

Hard Real-Time Task Scheduling

- Attributes of tasks to be scheduled
  - Period (p)
  - Deadline (d)
  - Fixed processing time (t)
- Relationship among the parameters: $0 < t \leq d \leq p$
  - Scheduler uses these relationships and assign priorities according to the periods (p) or deadlines (d)
- Three common scheduling algorithms
  - Rate-Monotonic
  - Earliest-Deadline-First
  - Deadline-Monotonic

Rate-Monotonic Scheduling

- Schedule periodic tasks using a static priority policy with preemption
- If a lower-priority task is running and a higher-priority task becomes available to run, the scheduler will preempt the lower-priority task
- Each task is assigned a priority inversely based on its period
  - The shorter the period, the higher the rate and thus the priority
    - Priority increases with rate monotonically
  - Based on the rationale that tasks need to run more often have higher priority in order to avoid missing the deadline
- If the sum of processor utilization of tasks is less than a bound given by $k(2^{1/k} - 1)$, the task set is definitely schedulable
  - where k=number of tasks and processor utilization = t_i/p_i
  - Note that the bound is rather pessimistic; exceeding it may still be schedulable
    - When k approaches infinity, utilization sum is 69%
    - In practice, the bound is about 88% on average
- Rate-monotonic scheduling is considered optimal
  - If a set of tasks cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities

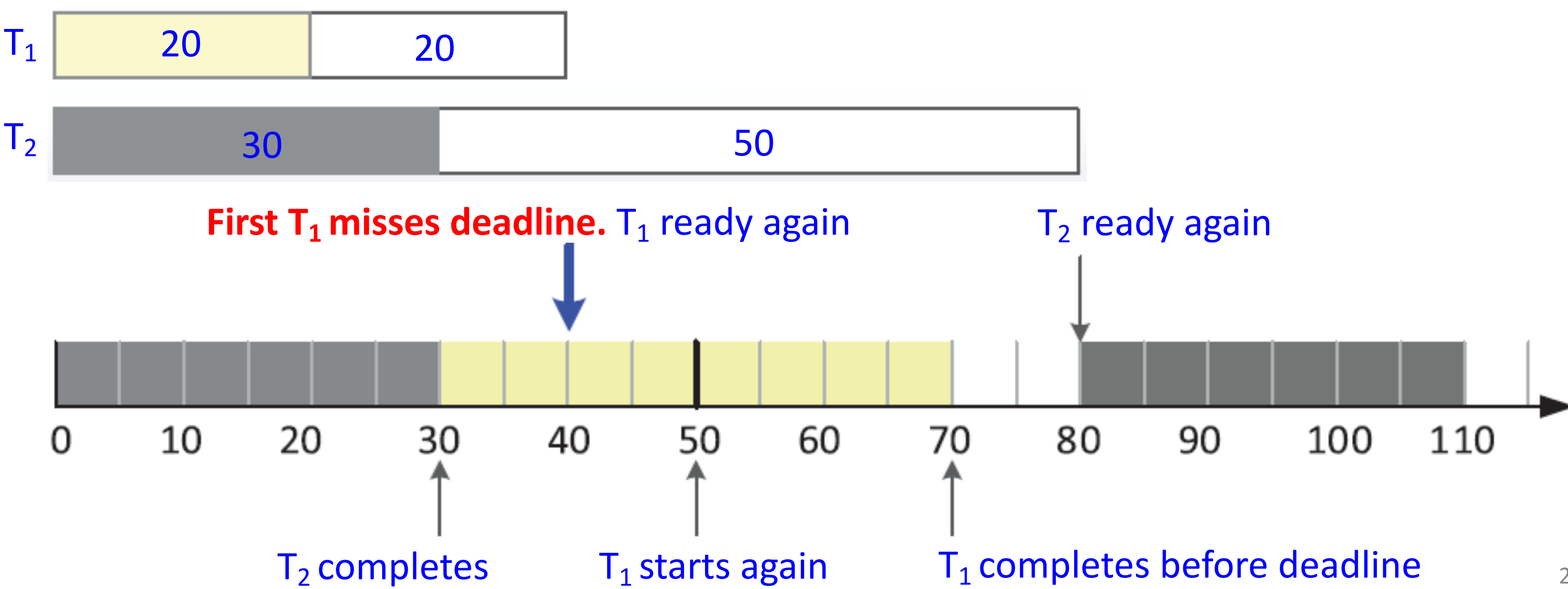
Period (p)
Deadline (d)
Fixed processing time (t)

Rate-Monotonic Scheduling (cont'd-1)

- Example: Tasks 1 and 2 with the following periods and processing times
  - T_1: $p_1 = d_1 = 40$, $t_1 = 20$
  - T_2: $p_2 = d_2 = 80$, $t_2 = 30$

Period (p)
Deadline (d)
Fixed processing time (t)

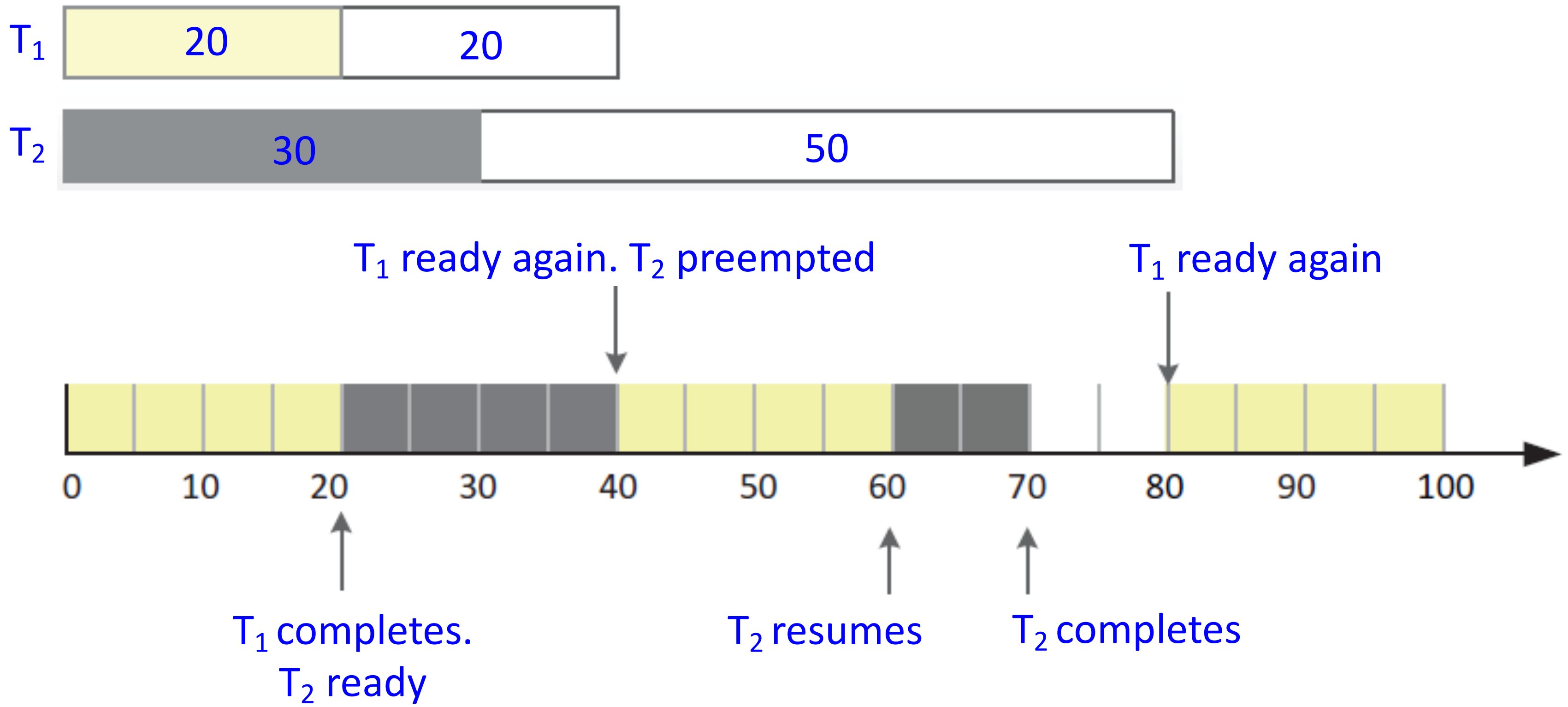
- Both tasks are ready at Time 0
- Scenario 1: Set Priority(T_1) < Priority(T_2) — not rate-monotonic
- Result: T_1 misses deadline



Rate-Monotonic Scheduling (cont'd-2)

Period (p)
Deadline (d)
Fixed processing time (t)

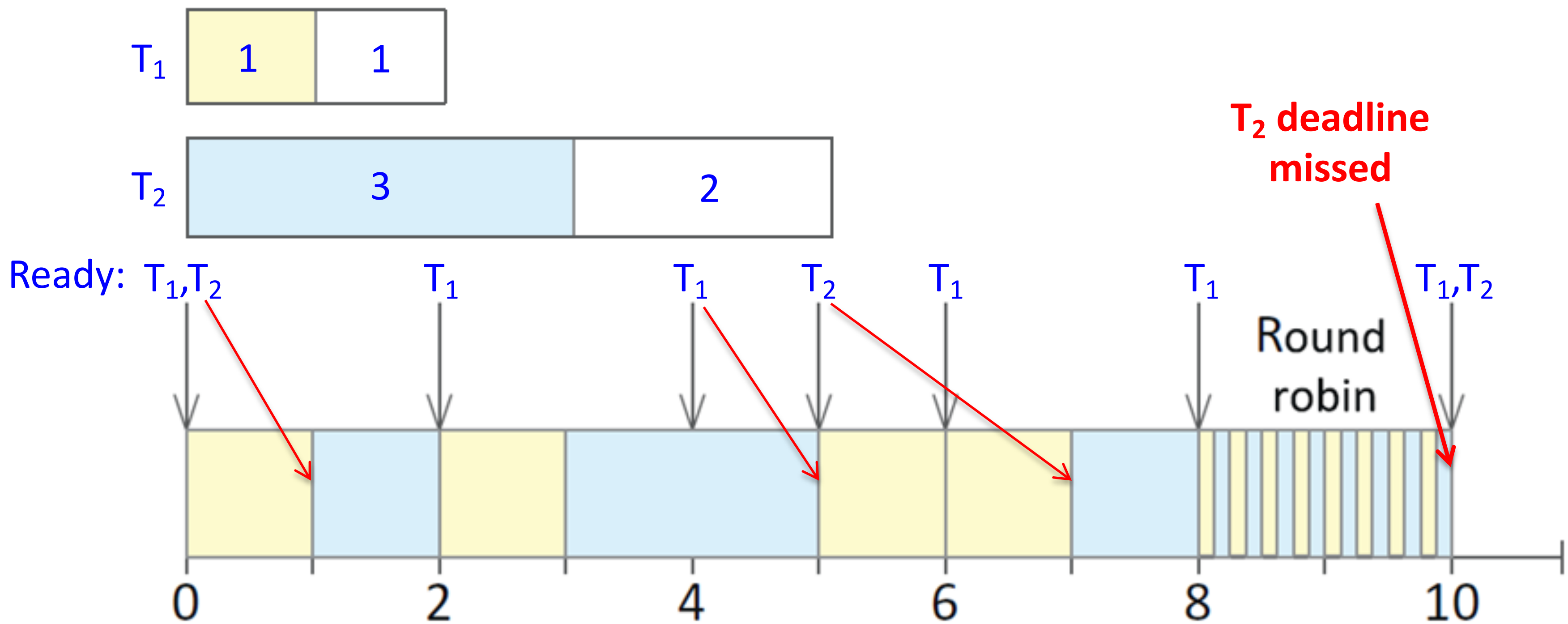
- Scenario 2: Set Priority(T_1) > Priority(T_2) — rate-monotonic
- Test bound: $k(2^{1/k} - 1) = 2(2^{1/2} - 1) = 0.83$
- Sum of processor utilizations = $t_1/p_1 + t_2/p_2 = 20/40 + 30/80 = 0.88$
- Result: No missing deadlines



Earliest-Deadline-First (EDF) Scheduling

- A dynamic-priority scheduling algorithm
- Requires that a task with an earlier deadline has a higher priority
- Example: Tasks 1 and 2 with the following periods and processing times
  - T_1: $p_1 = d_1 = 2$, $t_1 = 1$
  - T_2: $p_2 = d_2 = 5$, $t_2 = 3$

Period (p)
Deadline (d)
Fixed processing time (t)



T_1 deadlines: 2 4 6 6 8 10

T_2 deadlines: 5 5 5 5 5 10 10 10 10

Deadline-Monotonic Scheduling

- A static-priority scheduling algorithm
- Priorities are assigned to tasks according to their deadlines known at design time
  - A higher priority is assigned to a task with an earlier deadline
    - Priority increases with 1/deadline monotonically
  - Unlike the EDF approach, task priorities are fixed
- Example: Tasks 1, 2 and 3 with the following deadlines and processing times
  - T_1: $p_1 = 60$, $t_1 = 25$, $d_1 = 50$
  - T_2: $p_2 = 60$, $t_2 = 10$, $d_2 = 40$
  - T_3: $p_3 = 60$, $t_3 = 15$, $d_3 = 60$

Period (p)
Deadline (d)
Fixed processing time (t)

- According to their deadlines, $d_2 < d_1 < d_3$
  - Task priorities: $T_2 > T_1 > T_3$

Deadline-Monotonic Scheduling (cont'd)

- Task priorities: $T_2 > T_1 > T_3$