BLM4021 Gömülü Sistemler

Doç. Dr. Ali Can KARACA

ackaraca@yildiz.edu.tr

Yıldız Teknik Üniversitesi – Bilgisayar Mühendisliği

Lecture 12 – Scheduling in Real Time Operating Systems

- Real Time Operating Systems
- CPU Scheduling
- Pre-empitive Priority and Round Robin Scheduling
- Some Examples

Ders Materyalleri

Kaynaklar:

- Jiacun Wang, Real-Time Embedded Systems, Wiley, 2017.
- Xiacong Fan, Real-Time Embedded Systems: Design Principles and Engineering Practices, 2015.
- Stefan M. Petters, Real-Time Systems, NICTA.
- Philip Koopman, Real Time Operating Systems, Embedded System Engineering, 2016.
- Colin Perkins, Introduction to Real-Time Systems, University of Glaskow, Lecture 1.
- Kang, G. Shin, Principles of Real-Time and Embedded Systems, Lecture Notes 1-4.
- Özgür Aytekin (Collins Aerospace), Introduction to RT Operating Systems, A training for software development professionals.

Haftalık Konular

Hafta	Teorik	Laboratuvar
1	Giriş ve Uygulamalar, Mikroişlemci, Mikrodenetleyici ve Gömülü sistem kavramlarının açıklanması	Grupların oluşturulması & Kitlerin Testi
2	Bir Tasarım Örneği, Mikroişlemci, Mikrodenetleyici, DSP, FPGA, ASIC kavramları	Kitlerin gruplara dağıtımı + Raspberry Pi Kurulumu
3	16, 32 ve 64 bitlik mikrodenetleyiciler, pipeline	Raspberry Pi ile Temel Konfigürasyon
4	PIC ve MSP430 özellikleri	---- Resmi Tatil ---
5	ARM tabanlı mikrodenetleyiciler ve özellikleri	Uygulama 1 – Raspberry Pi ile Buzzer Uygulaması
6	ARM Komut setleri ve Assembly Kodları-1	Uygulama 2 – Raspberry Pi ile Ivme ve Gyro Uygulaması
7	ARM Komut setleri ve Assembly Kodları-2, Raspberry Pi vers. ve GPIO'ları	Uygulama 3 – Raspberry Pi ile Motor Kontrol Uygulaması
8	*Vize Sınavı*	
9	Haberleşme Protokolleri (SPI, I2C ve CAN)	Proje Soru-Cevap Saati - 1
10	Sensörlerden Veri Toplama, Algılayıcı, ADC ve DAC	Proje Soru-Cevap Saati - 2
11	Zamanlayıcı, PWM ve Motor Sürme	Proje kontrolü - 1
12	Gerçek Zaman Sistemler ve temel kavramlar	Proje Kontrolü - 2
13	Gerçek zaman İşletim Sistemleri	Mazeret sebepli son proje kontrollerinin yapılması
14	Nesnelerin İnterneti	
15	*Final Sınavı*	

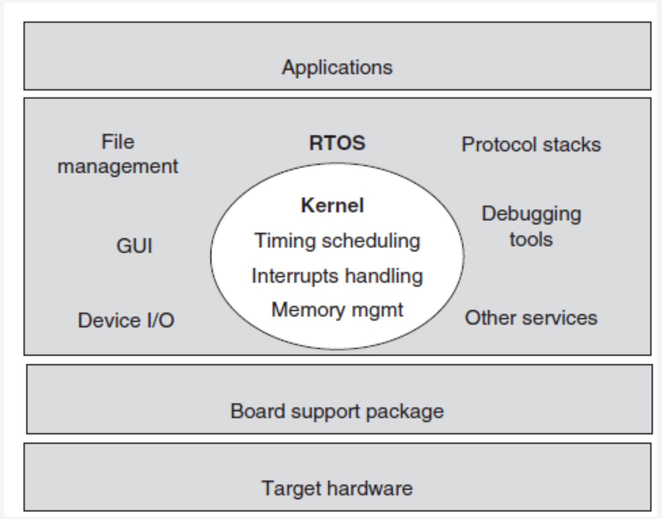
For more details -> Bologna page: http://www.bologna.yildiz.edu.tr/index.php?r=course/view&id=9463&aid=3

Recall: Real-Time Operating Systems

«The ability of the OS to provide a required level of service in a bounded response time.»

-POSIX Standard 1003.1 (Portable OS Interface specified by IEEE Computer Society.)

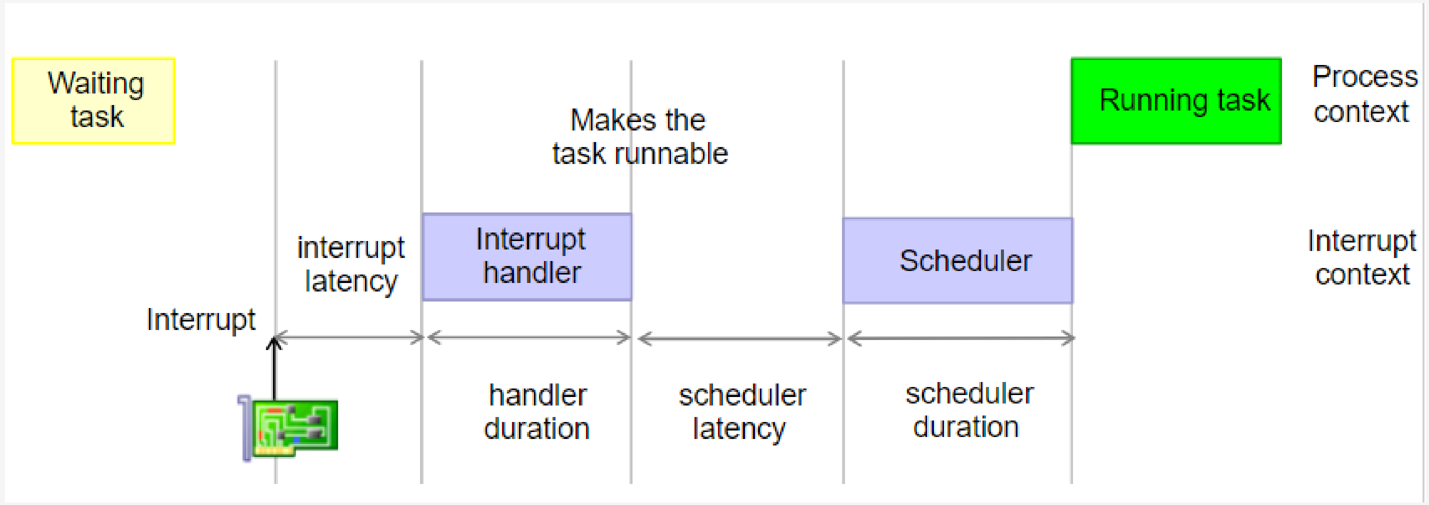
- The timing behavior of the OS must be predictable and deterministic. Upper limit is known.
- The OS manage the timing and scheduling according to priorities.
- Context switch times should be small and bounded.
- They should include RT kernels.
- High resolution timers are important.
- They should have controlled kernel size for smaller ESs.
- Sophisticated features of OSs should be removed.

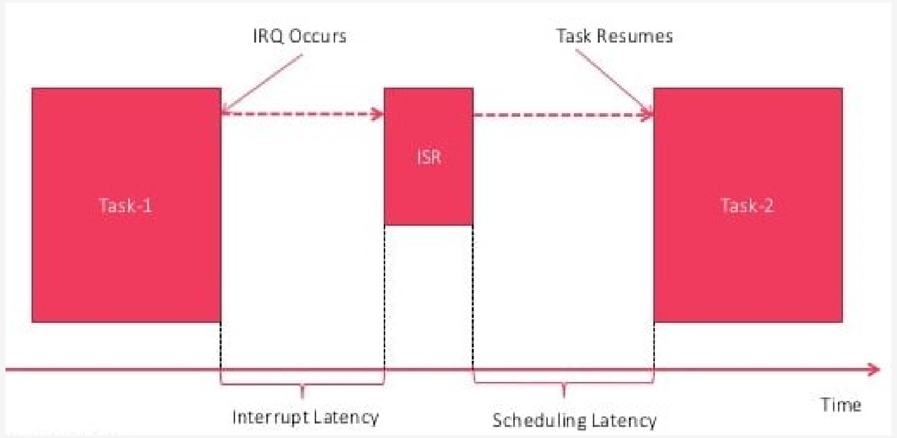


High-level view of an RTOS

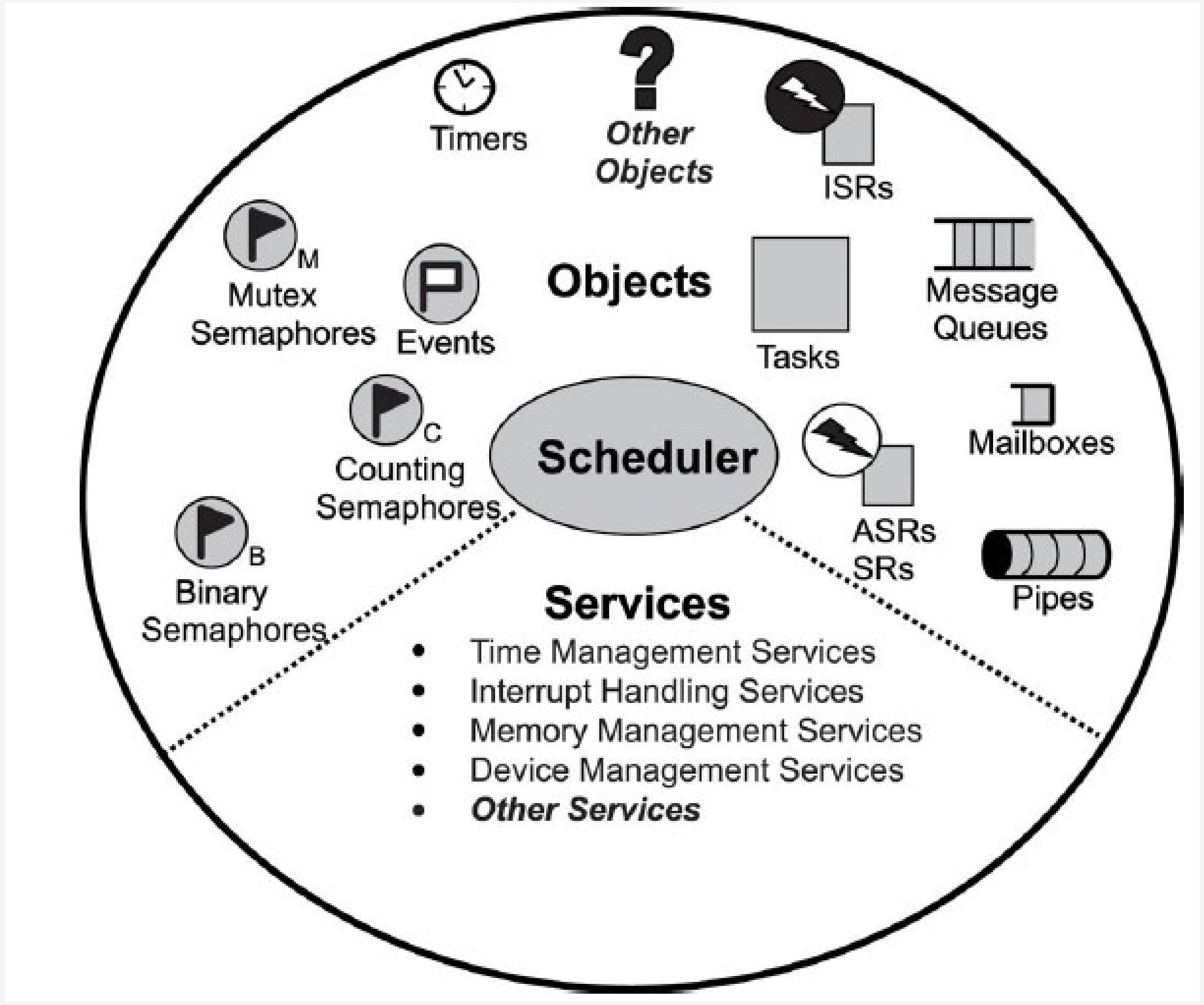
Recall: Kernel Latency = Task Latency = Event Latency

Task Latency = Interrupt Latency + Handler Duration + Scheduler Latency + Scheduler Duration





Recall: Common Components in an RTOS

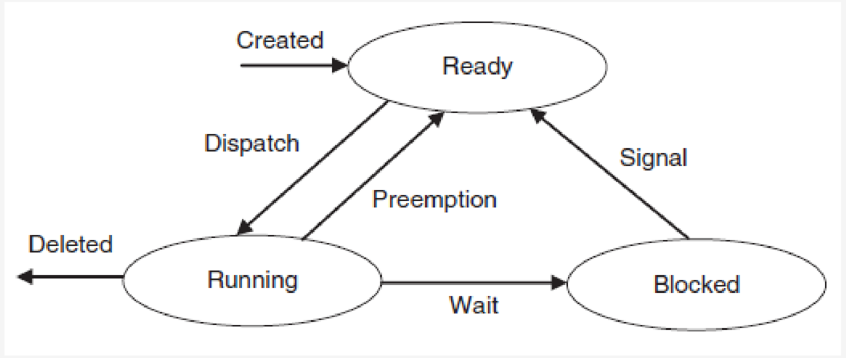


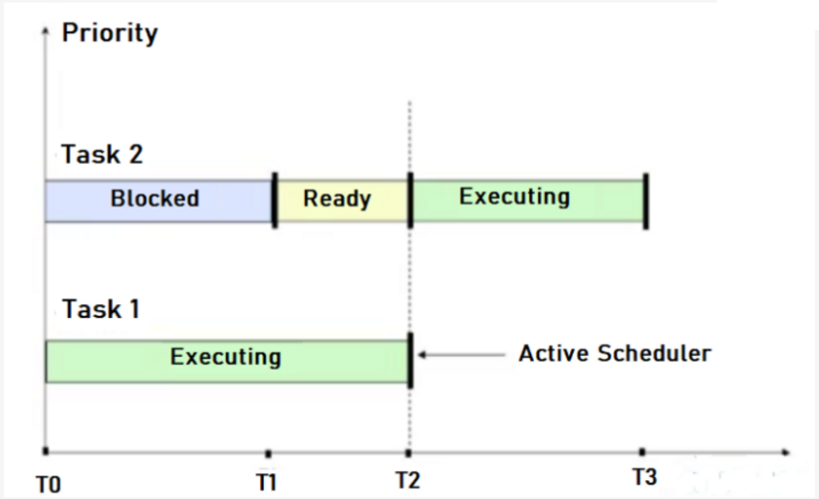
Basic Functions of RTOS:

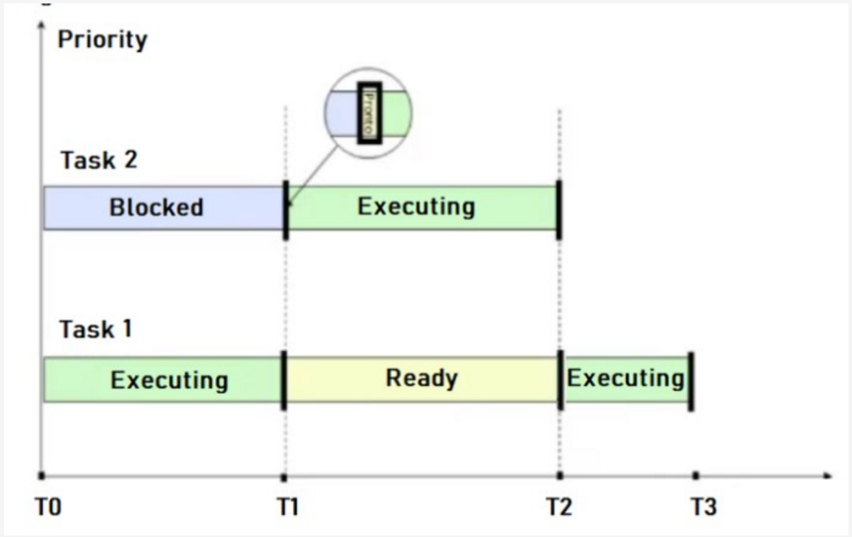
- Time management
- Task management (create, terminate)
- Interrupt handling (ISRs)
- Memory management
- Expection handling (deadlock, timout)
- Task synchronization (sem., mutex)
- Task scheduling (priorty-based)

Non-Preemptive vs Preemptive

- **Non-preemptive scheduling:**
  - The running process keeps the CPU until it voluntarily gives up the CPU
    - process exits
    - switches to blocked state
    - Transition 3 is only voluntary
- **Preemptive scheduling**:
  - The running process can be interrupted and must release the CPU (can be forced to give up CPU)







Task Models

- Periodic Tasks:

Periodic tasks are repeated once a period, for example, 200 milliseconds. They are time-driven. Periodic tasks typically arise from sensory data acquisition, control law computation, action planning, and system monitoring.

- Aperiodic Tasks:

Aperiodic tasks are one-shot tasks. They are event-driven. For example, a driver may change the vehicle's cruise speed while the cruise control system is in operation. To maintain the speed set by the driver, the system periodically takes its speed signal from a rotating driveshaft, speedometer cable, wheel speed sensor, or internal speed pulses produced electronically by the vehicle and then pulls the throttle cable with a solenoid as needed.

- Sporadic Tasks:

Sporadic tasks are also event-driven. The arrival times of sporadic task instances are not known *a priori*, but there is requirement on the minimum interarrival time. When the driver of a vehicle sees a dangerous situation in front of him and pushes the break to stop the vehicle, the speed control system has to respond to the event (a hard step on the break) within a small time window.

Credit by Jiacun Wang

Task Specification

- The release time of a task τ_i is denoted by r_i. Period T_i
- The deadline of a task is the instant of time by which its execution must be completed. The deadline of τ_i is denoted by d_i. Relative deadline -> D_i
- Task's execution time mainly depends on the complexity of the task and the speed of the processor. The execution time of τ_i is denoted by C_i.
- The response time of a task R_i is the length of time elapsed from the task is released to the execution is completed.

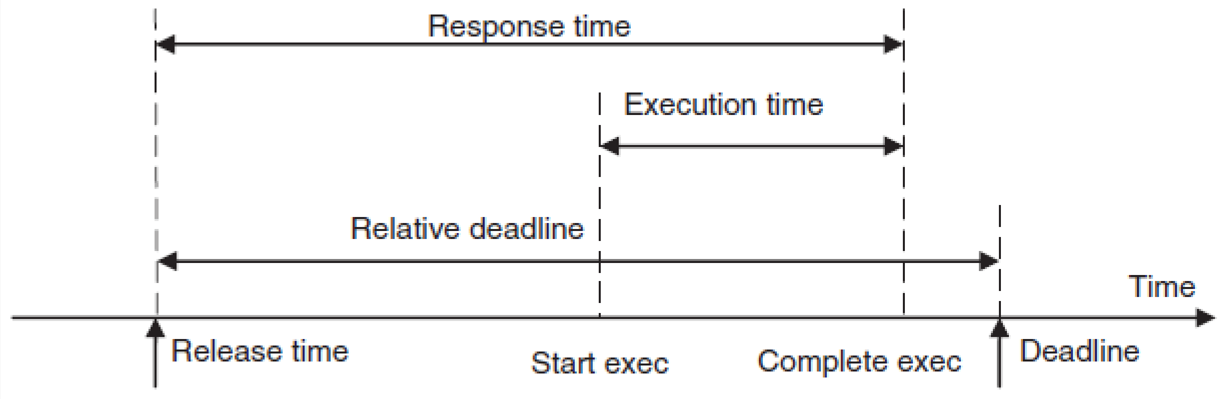
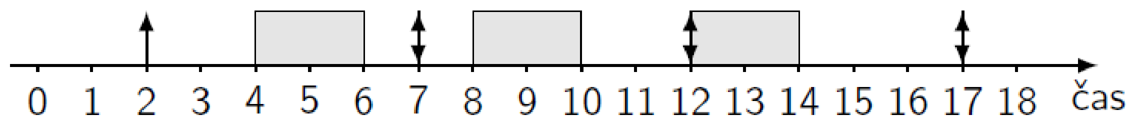


Figure 4.1 Task specification.

Periodic task τ_i with $r_i = 2,\ T_i = 5,\ C_i = 2,\ D_i = 5$ can be executed like this (continues until infinity).



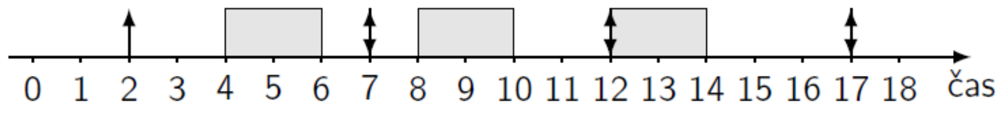
Legend: $\uparrow$ = job release time , $\downarrow$ = deadline.

(T_i, C_i, D_i)

Additional Specifications for Periodic Tasks

- Hyperperiod: Least common multiple of $\{T_1, ..., T_n\}$.
- Task utilization: $u_i = \frac{C_i}{T_i}$.
- System utilization: $U = \sum_{i=1,\dots,n} u_i$

Periodic task τ_i with $r_i = 2,\ T_i = 5,\ C_i = 2,\ D_i = 5$ can be executed like this (continues until infinity).



Legend: ↑ = job release time , ↓ = deadline.

Helicopter control system

Three tasks:

1. 180× per second, total computation time 1 ms
   - Read sensor data
   - Compute the control laws of the inner yaw-control loop
   - Apply the control laws results to the actuators
   - Perform internal checks
2. 90× per second, total computation time 3 ms:
   - Compute the control laws of the inner pitch-control loop
   - Compute the control laws of the inner roll- and collective-control loop
3. 30× per second, total computation time 10 ms:
   - Compute the control lows of the outer loops

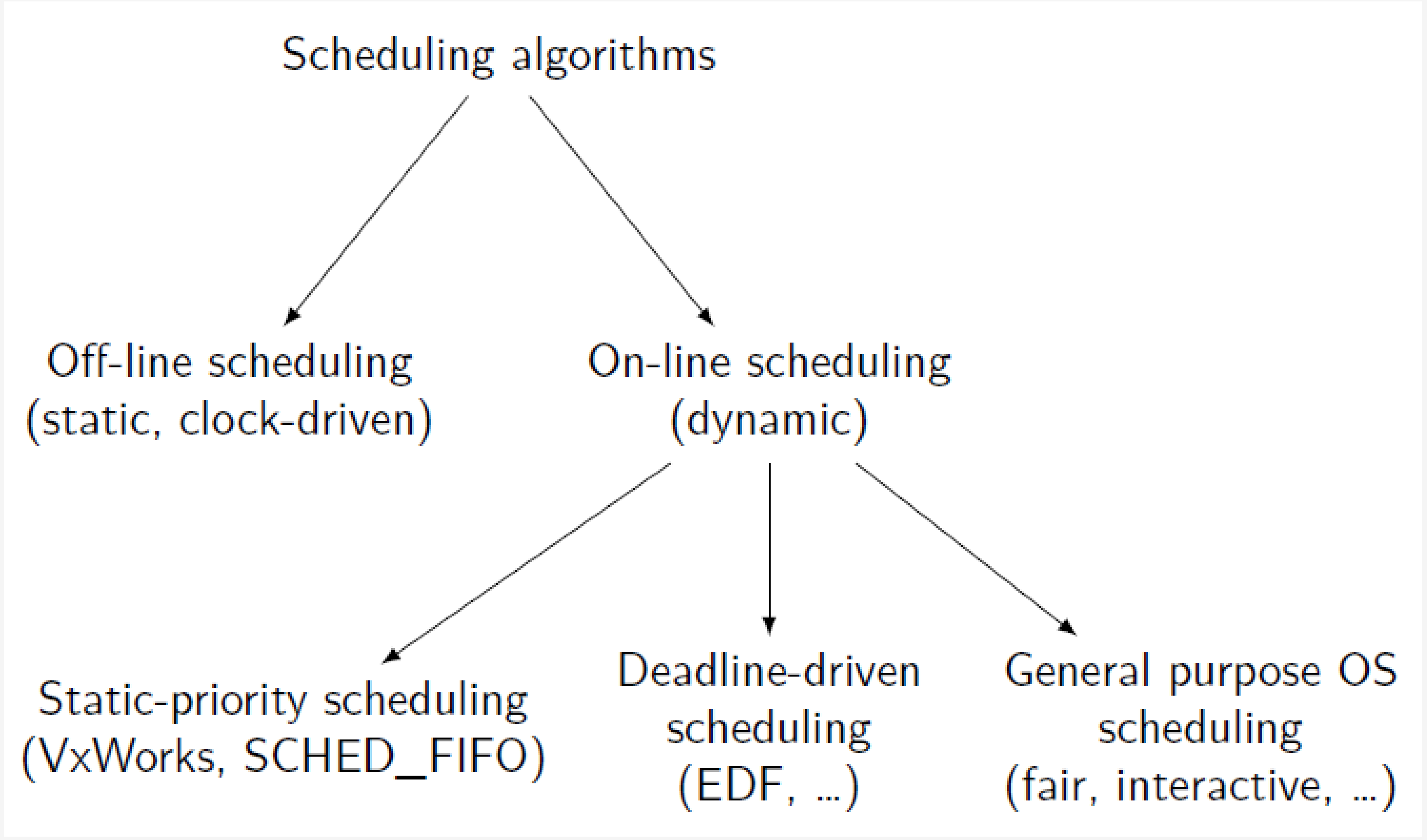
Hyperperiod: LCM {T1, T2, T3}

LCM {1/180, 1/90, 1/30} = 1/30 s = 33.33 ms

Task Utilization: $u_1 = \frac{C_1}{T_1} = \frac{10^{-3}}{1/180} = 0.18$

$u_2 = \frac{10^{-3}}{1/90} = 0.09$ $\quad u_3 = \frac{10^{-3}}{1/30} = 0.03$

Commonly-Used Real-Time Scheduling Approaches

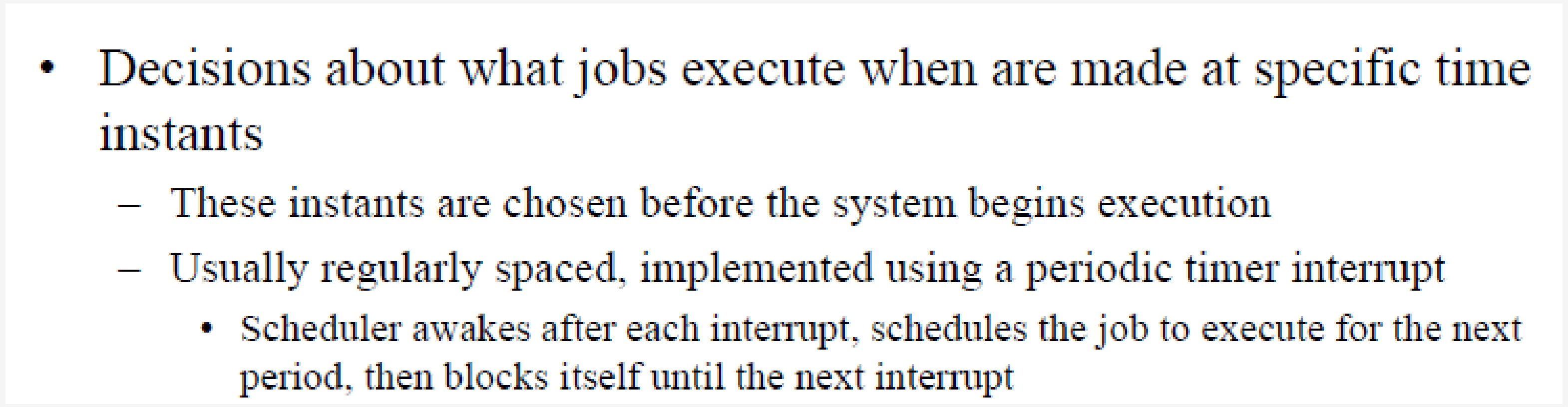

Scheduling algorithms
Off-line scheduling (static, clock-driven)
On-line scheduling (dynamic)
Static-priority scheduling (VxWorks, SCHED_FIFO)
Deadline-driven scheduling (EDF, ...)
General purpose OS scheduling (fair, interactive, ...)

A valid schedule is a feasible schedule if every task completes by its deadline.

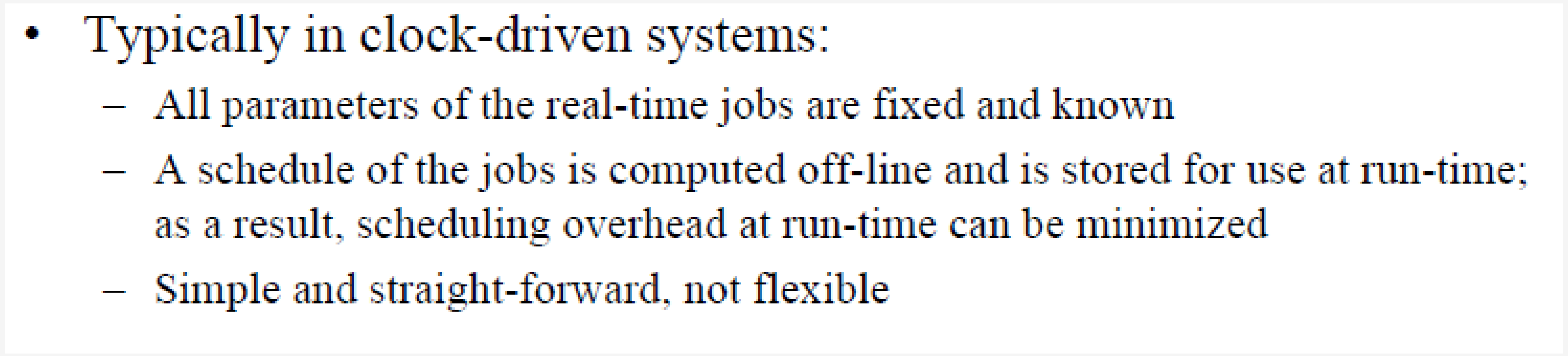
On-line vs Offline Scheduling

- ***Off-line scheduling:*** **the schedule is computed off-line and is based on the knowledge of the release times and execution times of all jobs.**
  - A system with fixed sets of functions and job characteristics does not vary or vary only slightly.
- ***On-line scheduling:*** **a scheduler makes each scheduling decision without knowledge about the jobs that will be released in future.**
  - there is no optimal on-line schedule if jobs are non-preemptive
  - when a job is released, the system can serve it, or wait for future jobs

Off-line Scheduling: Clock-Driven



- Decisions about what jobs execute when are made at specific time instants
  - These instants are chosen before the system begins execution
  - Usually regularly spaced, implemented using a periodic timer interrupt
    - Scheduler awakes after each interrupt, schedules the job to execute for the next period, then blocks itself until the next interrupt



- Typically in clock-driven systems:
  - All parameters of the real-time jobs are fixed and known
  - A schedule of the jobs is computed off-line and is stored for use at run-time; as a result, scheduling overhead at run-time can be minimized
  - Simple and straight-forward, not flexible

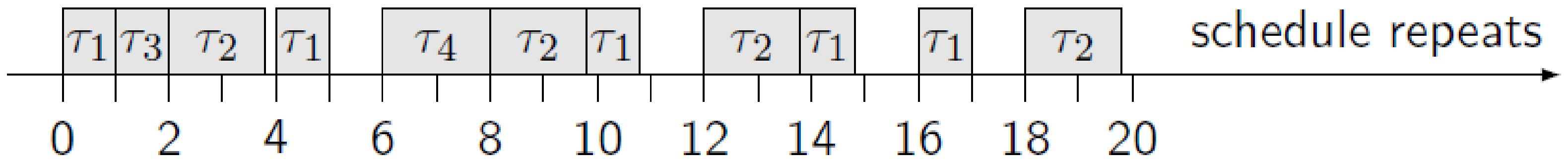
Off-line Scheduling: Clock-Driven

The scheduler will schedule periodic tasks according to a static schedule, which is computed offline and stored in a table.

If no task is to be scheduled, the special symbol × (idle) is stored there.

System of four tasks:

	τ_1	τ_2	τ_3	τ_4
Period	4	5	20	20
Execution time	1	1.8	1	2



τ_1 τ_3 τ_2 τ_1 τ_4 τ_2 τ_1 τ_2 τ_1 τ_1 τ_2
schedule repeats
0 2 4 6 8 10 12 14 16 18 20

The schedule table will look like this:

Time	0	1	2	3.8	4	5	6	8	9.8	10.8	...	18
Task	τ_1	τ_3	τ_2	$\times$	τ_1	$\times$	τ_4	τ_1	$\times$	τ_2	...	τ_2

Hyperperiod (H): LCM {4, 5, 20, 20}=20

Pseudocode of Clock-Driven Scheduler

Time	0	1	2	3.8	4	5	6	8	9.8	10.8	...	18
Task	τ_1	τ_3	τ_2	$\times$	τ_1	$\times$	τ_4	τ_1	$\times$	τ_2	...	τ_2

Scheduler – basic version without aperiodic tasks

```
/* H is hyperperiod, created by N "slices" (not necessarily of the same length).*/
Input: Schedule stored in table $(t_k, \tau(t_k))$ for $k = 0, 1, \ldots, N-1$
Procedure SCHEDULER:
    set next scheduling point $i$ and table index $k$ to 0;
    set timer to time $t_k$;
    Repeat forever
        wait for timer expiration (for example interrupt – instruction hlt, or polling)
        current task $\tau := \tau(t_k)$;                <- select current task
        $i := i + 1$;
        calculate index of next table entry as $k := i \mod N$;    <- index of the following task in table
        set timer to $\lfloor i/N \rfloor \cdot H + t_k$;        <- Increase the timer for each additional hyperperiod
        call function $\tau$;
    End
End SCHEDULER
```

We assume that τ always finishes by next timer expiration.

Off-line Scheduling: Frame-based

Problems

- Big number of tasks $\Rightarrow$ big schedule table
- Embedded systems have limited size of memory
- Reprogramming the timer might be slow

Idea

- Divide time to constant-size frames
- Combine multiple jobs into a single frame
- Scheduling decisions are made only at frame boundaries

- Consequences:
  - There is no preemption within each frame.
  - Each job must fit into the frame
  - In addition to schedule calculation, we should check for various error conditions such as task overrun.
- Let f denote the frame size. How to select f?

Off-line Scheduling: Frame-based

1. We want big enough frames to fit every job without preempting it. This gives us

$$f \geq \max_{i=1,\ldots,n} (C_i). \tag{1}$$

2. In order to have the table small, f should divide H. Since $H = \text{lcm}(T_1, \ldots, T_n)$, f divides T_i for at least one task τ_i:

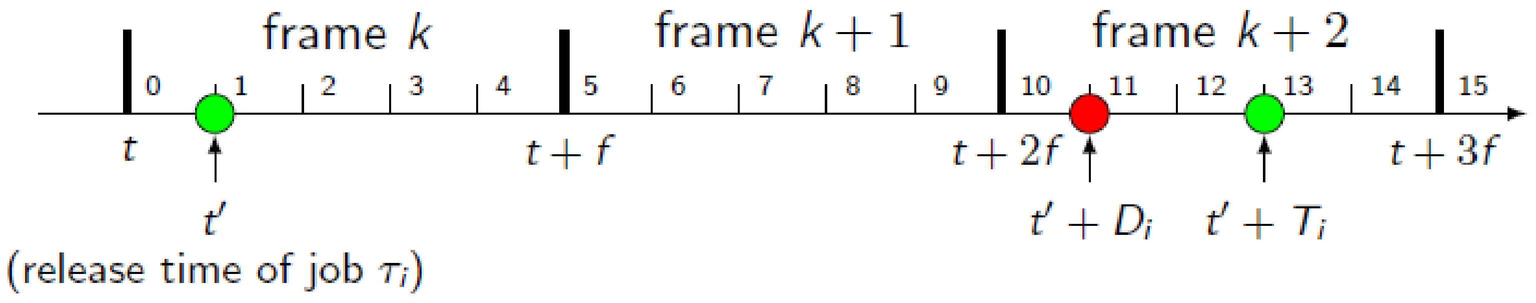
$$\left\lfloor \frac{T_i}{f} \right\rfloor = \frac{T_i}{f}. \tag{2}$$

Let $F = \frac{H}{f}$ (F is integer). Each interval H is called major cycle and each interval f minor cycle.
Each major cycle is composed from F minor cycles.

Off-line Scheduling: Frame-based

3 We want the frame size to be sufficiently small so that there is at least one whole frame between task release time and deadline. The reasons for this are:

- Our scheduler can only start jobs at frame boundaries. Jobs release in between have to wait for the boundary.
- Similarly, we want the scheduler to check whether each jobs complete by its deadline. If there is no frame boundary before the deadline, we cannot prevent overruns. For example, if the deadline falls into frame $k+2$, overrun can be checked only at the beginning of frame $k+2$.



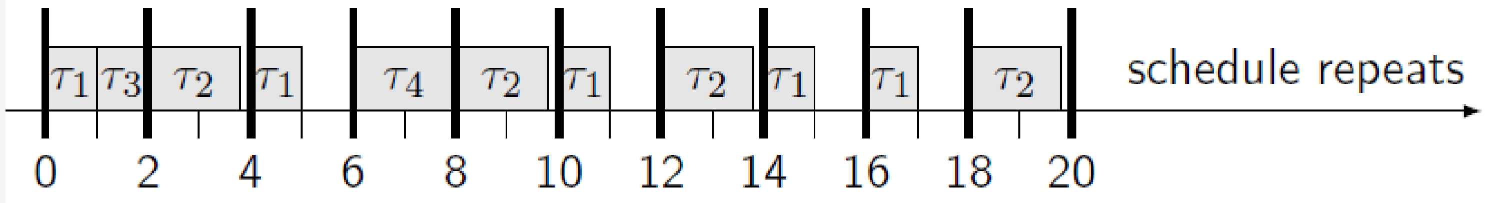
Thus:

$$2f - \gcd(T_i, f) \leq D_i. \tag{3}$$

*greatest common devisor

Off-line Scheduling: Frame-based

- Let's have a system from the last example with four tasks: $\tau_1 = (4,1)$, $\tau_2 = (5,1.8)$, $\tau_3 = (20,1)$, $\tau_4 = (20,2)$.
- From the first constraint (1): $f \geq 2$.
- Hyperperiod is 20 so second constraint (2), tells us that f can be one of 2, 4, 5, 10 a 20.
- Third constraint (3) satisfies only $f = 2$.
- Possible cyclic schedule:



1) If f = 2,

2*2 – GCD (4,2)≤ 4 -> 4 – 2 ≤ 4

2*2 – GCD (5,2)≤ 1.8 -> 4 - 1 ≤ 5

2*2 – GCD (20,2)≤ 20 -> 4 - 2 ≤ 20

2*2 – GCD (20,2)≤ 20 -> 4 - 2 ≤ 20

2) If f = 4,

2*4 – GCD (4,2)≤ 4 -> 8 – 2 ≤ 4

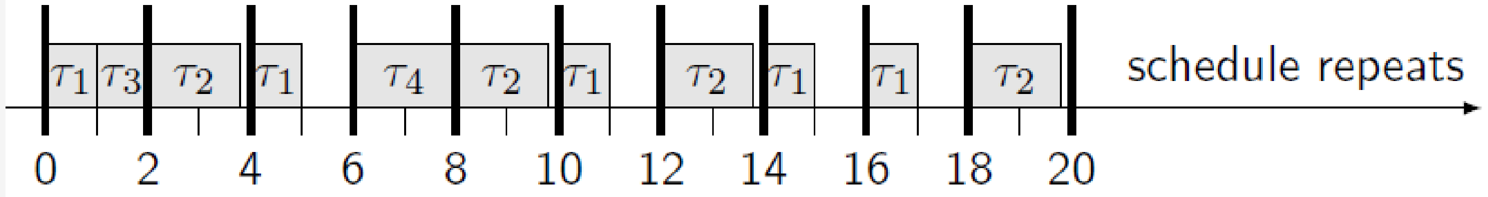
2*4 – GCD (5,2)≤ 1.8 -> 8 - 1 ≤ 5

2*4 – GCD (20,2)≤ 20 -> 8 - 2 ≤ 20

2*4 – GCD (20,2)≤ 20 -> 8 - 2 ≤ 20

Off-line Scheduling: Frame-based

- Let's have a system from the last example with four tasks: $\tau_1 = (4,1)$, $\tau_2 = (5,1.8)$, $\tau_3 = (20,1)$, $\tau_4 = (20,2)$.
- From the first constraint (1): $f \geq 2$.
- Hyperperiod is 20 so second constraint (2), tells us that f can be one of 2, 4, 5, 10 a 20.
- Third constraint (3) satisfies only $f = 2$.
- Possible cyclic schedule:



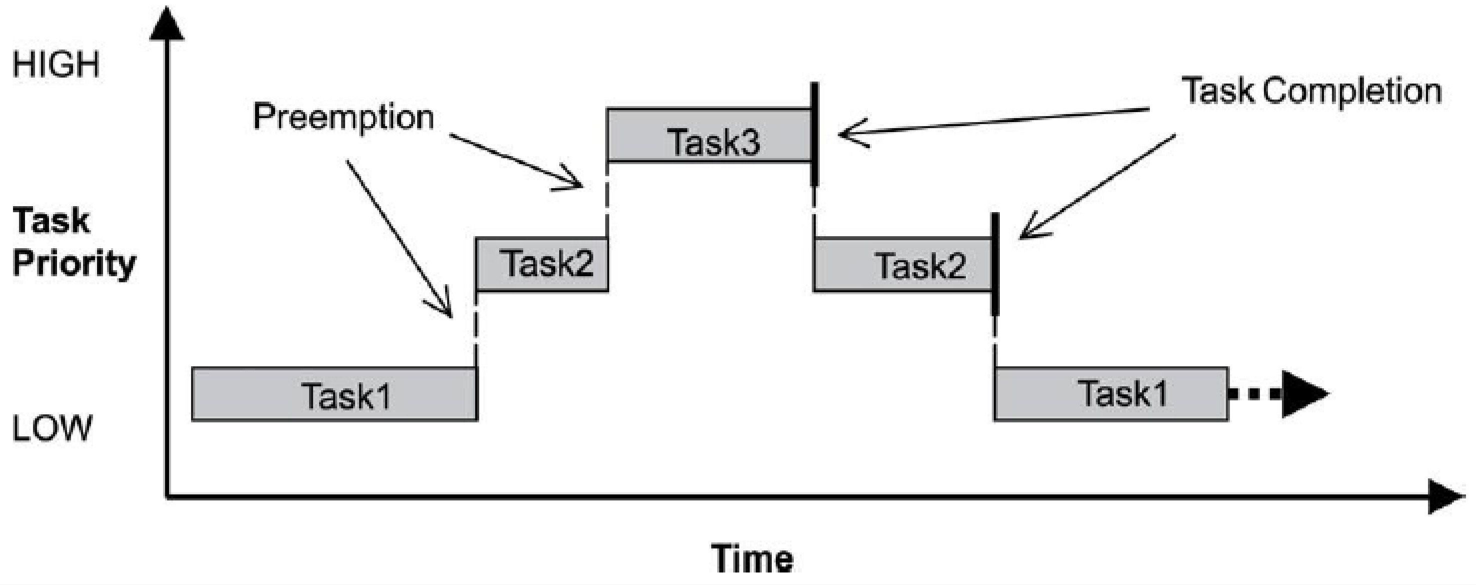
```
Input: Stored scheduling table $L(k)$ for $k$= 0, 1, …, $F$ - 1.
Task CYCLIC_SCHEDULER:
        $t$ := 0;   //current time
        $k$ := 0;   //frame number
        loop FOREVER
                accept clock interrupt;
                current Block := $L(k)$;
                $t$++;
                $k$ := $t$ mod $F$;
                execute currentTask;
                sleep until the next clock interrupt;
        end loop
end CYCLIC_SCHEDULER
```

Off-line Scheduling: Disadvantages

- Inflexible
- Not suitable for many applications.
- All parameters about jobs must be known in advance.
- Not robust during overloads.
- Difficult to expand the Schedule.
- Deep analysis is required over all tasks.

Preemptive Priority-based Scheduling

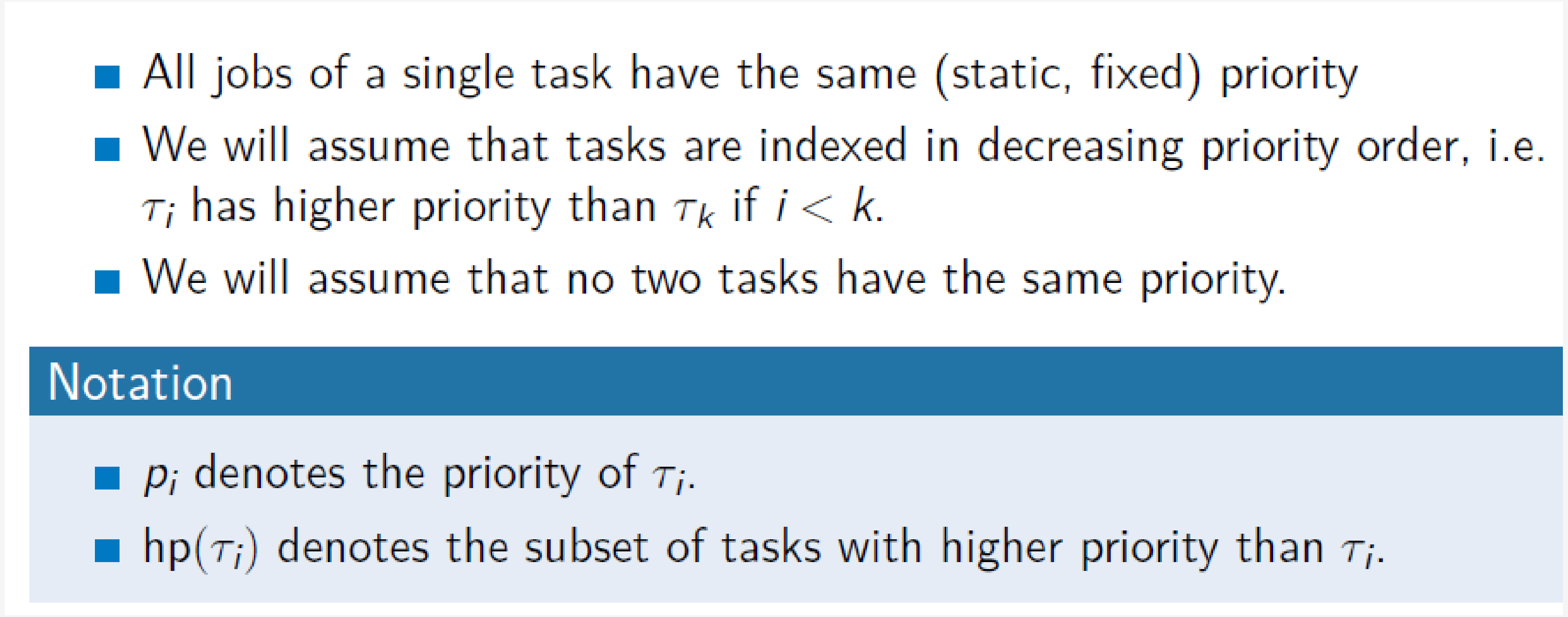
- All real-time kernels use preemptive priority-based scheduling by default.
- The highest priority READY thread is the one which gets the CPU



Online: Static Priority Scheduling = Fixed Priorty

It is online scheduling, and decisions are made at runtime. Priority is assigned to each task. Priority assignment can be done statically or dynamically while the system is running.

A scheduling algorithm that assigns priority to tasks statically is called a *static-priority* or *fixed-priority* algorithm, and an algorithm that assigns priority to tasks dynamically is said to be *dynamic-priority* algorithm.



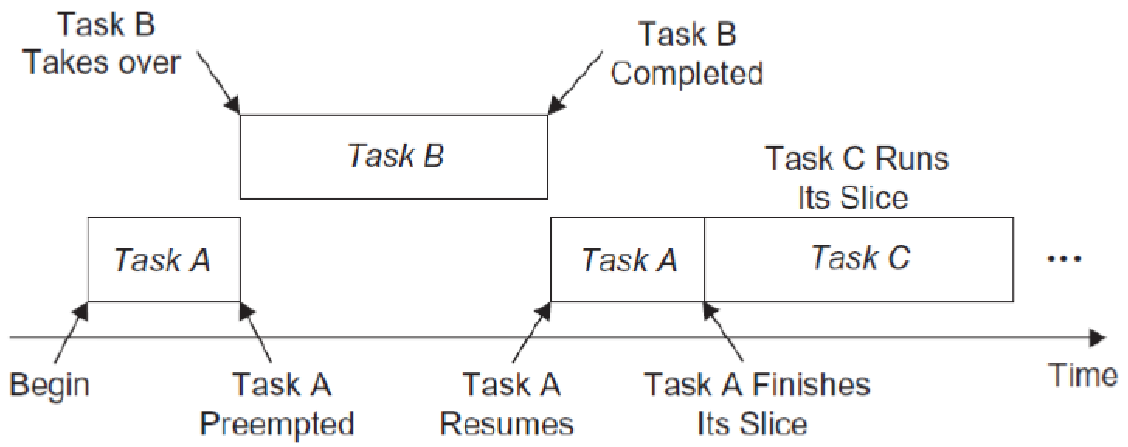
- All jobs of a single task have the same (static, fixed) priority
- We will assume that tasks are indexed in decreasing priority order, i.e. τ_i has higher priority than τ_k if $i < k$.
- We will assume that no two tasks have the same priority.

Notation

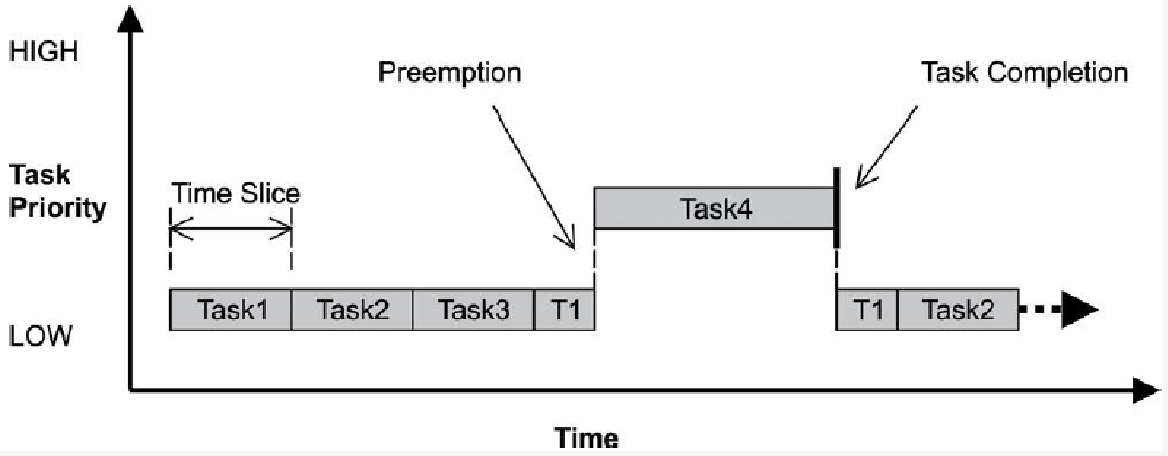
- p_i denotes the priority of τ_i.
- $\text{hp}(\tau_i)$ denotes the subset of tasks with higher priority than τ_i.

Round Robin (under Fixed Priority Group)

- Preemptive, priority-based scheduling can be augmented with round-robin scheduling which uses time slicing to achieve equal allocation of the CPU for tasks of the <u>same priority</u>



- Each executable task is assigned a fixed time quantum called a time slice
- A fixed rate clock is used to initiate an interrupt at a rate corresponding to the time slice.



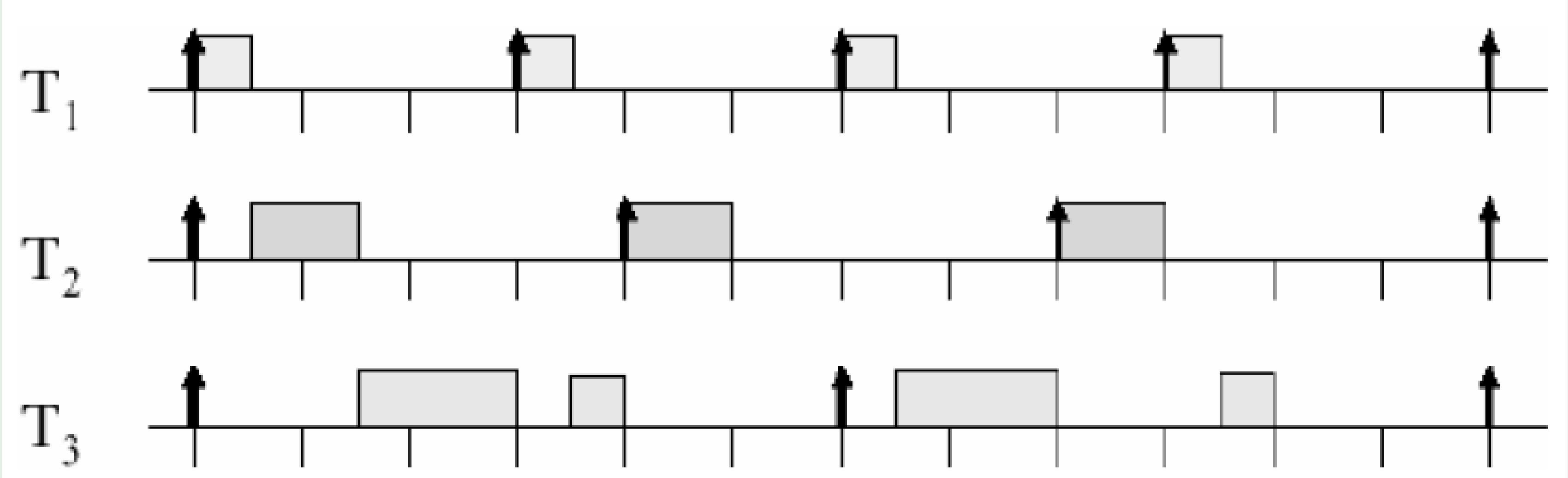
Rate-Monotonic (under Fixed Priority Group)

Rate-Monotonic priority assignment

The less period T_i the higher priority p_i.
For every two tasks τ_i and τ_j: $T_i < T_j \Rightarrow p_i > p_j$.

Example (RM schedule)

Three tasks (T, C): $\tau_1 = (3, 0.5)$, $\tau_2 = (4, 1)$ a $\tau_3 = 6, 2$.



Another Example:

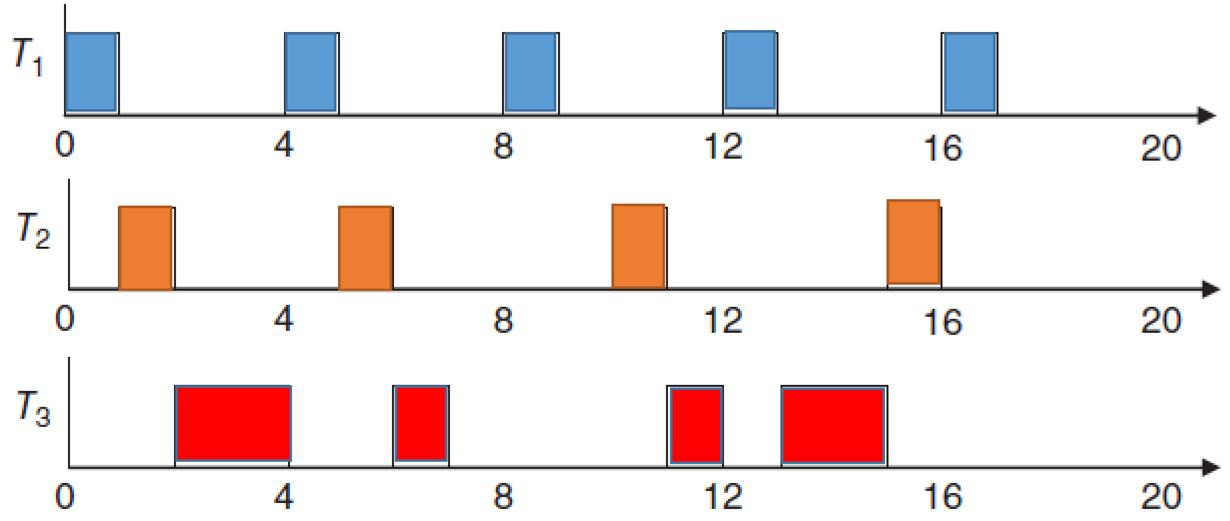
Task	Period	Priority
τ_1	25	0
τ_2	60	2
τ_3	42	1
τ_4	105	4
τ_5	75	3

Example: Rate-Monotonic

Example 4.9 ***Task missing deadline by RM***
Figure 4.13 shows that when we schedule three periodic tasks

$$\tau 1 = (4, 1),\ \tau 2 = (5, 2),\ \tau 3 = (10, 3.1).$$ -> (Period, Execution Time)

according to the RM algorithm, the first instance of T_3 misses its deadline – it has remaining 0.1 units of execution time that is not completed by its deadline 10.



Priorities -> $\tau 1$, $\tau 2$, $\tau 3$

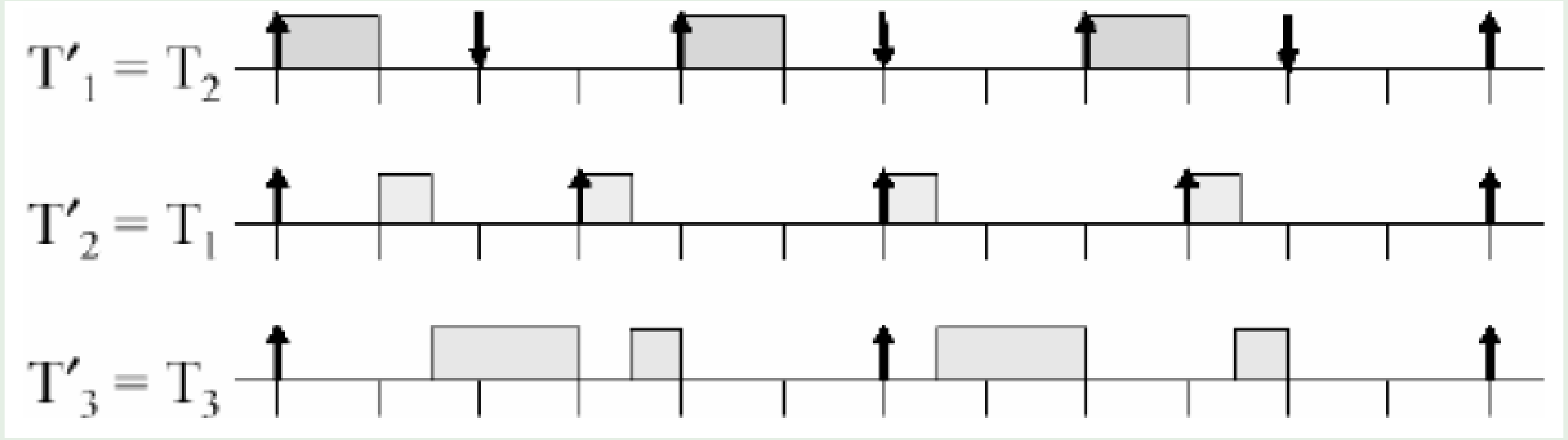
Deadline-Monotonic (under Fixed Priority Group)

Deadline-Monotonic priority assignment

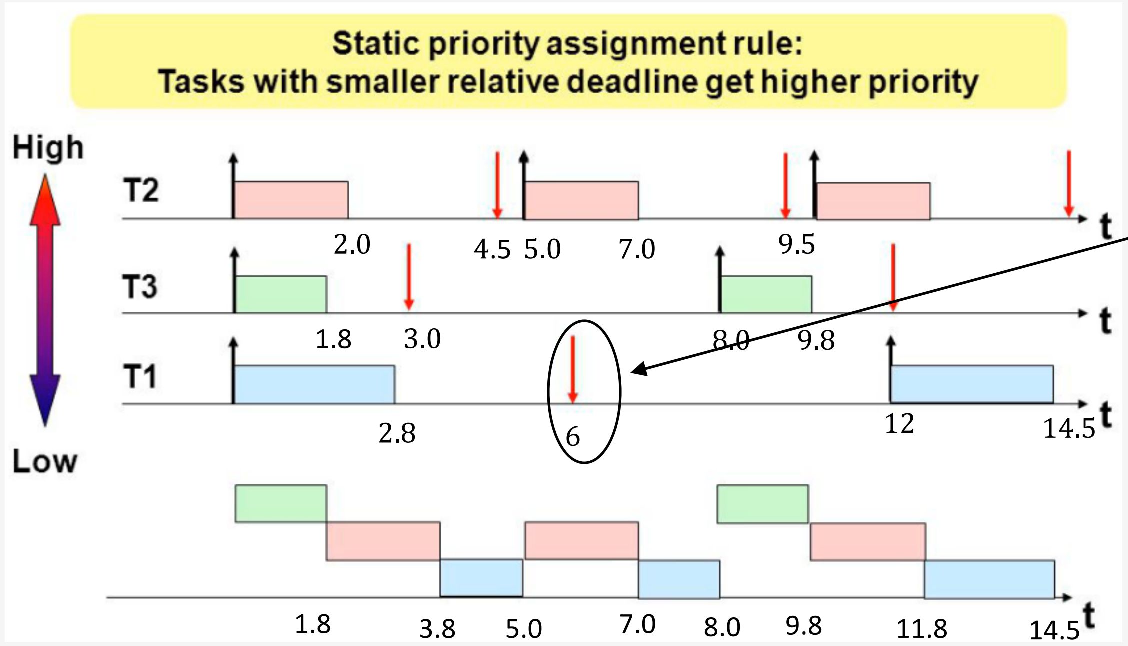
The earlier deadline D_i, the higher priority p_i.
Pro each two tasks τ_i a τ_j: $D_i < D_j \Rightarrow p_i > p_j$.

Example (DM schedule)

Let's change the RM example by tightening deadline of $\tau_2 = (T, C, D)$:
$\tau_1 = (3, 0.5)$, $\tau_2 = (4, 1, 2)$ a $\tau_3 = 6, 2$.



Deadline Monotonic



Missed the deadline

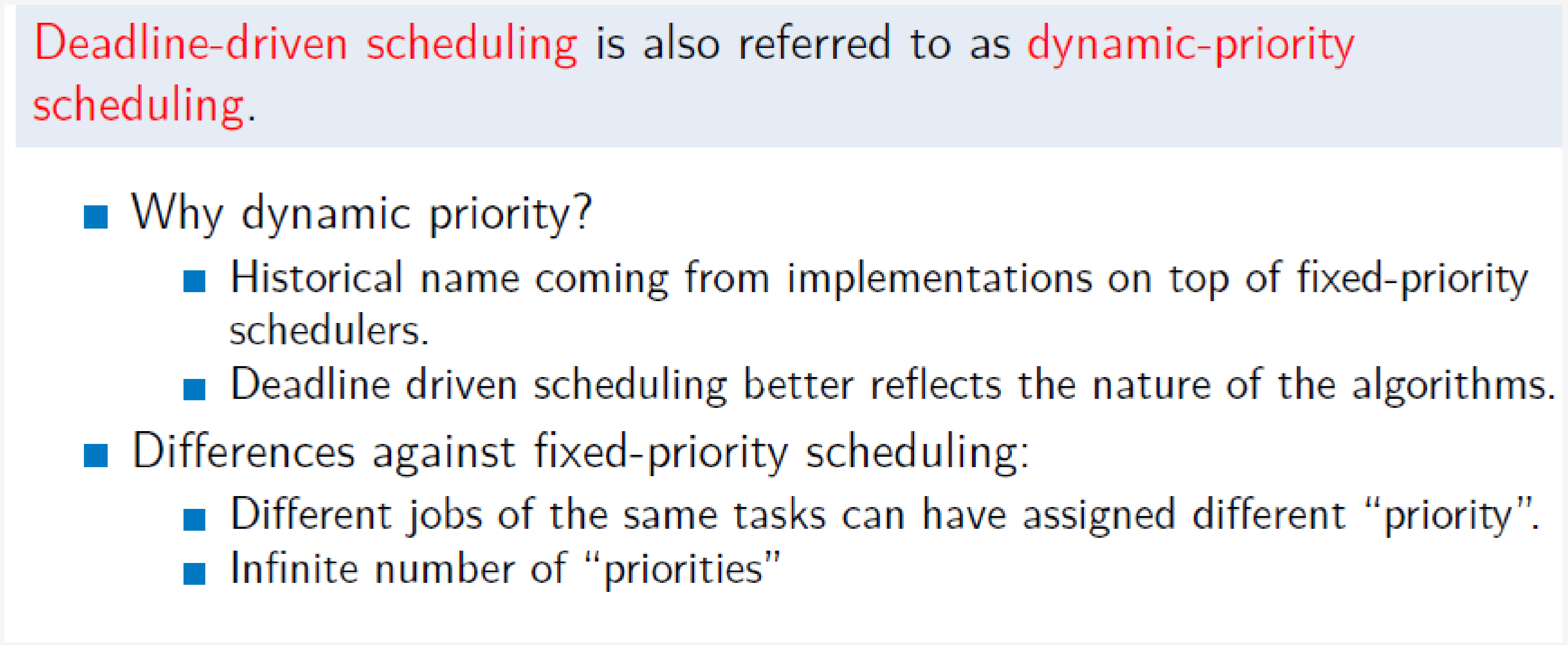
T, C, D

Task1	$\tau_1 = (12, 2.8, 6)$
Task2	$\tau_2 = (5, 2, 4.5)$
Task3	$\tau_3 = (8, 1.8, 3.0)$

Online: Deadline-Driven Scheduling = Dynamic-Priority

It is online scheduling, and decisions are made at runtime. Priority is assigned to each task. Priority assignment can be done statically or dynamically while the system is running.

A scheduling algorithm that assigns priority to tasks statically is called a *static-priority* or *fixed-priority* algorithm, and an algorithm that assigns priority to tasks dynamically is said to be *dynamic-priority* algorithm.



Deadline-driven scheduling is also referred to as dynamic-priority scheduling.

- Why dynamic priority?
  - Historical name coming from implementations on top of fixed-priority schedulers.
  - Deadline driven scheduling better reflects the nature of the algorithms.
- Differences against fixed-priority scheduling:
  - Different jobs of the same tasks can have assigned different “priority”.
  - Infinite number of “priorities”

Earliest Deadline First (EDF)

EDF is one of the most popular dynamic scheduling algorithm.

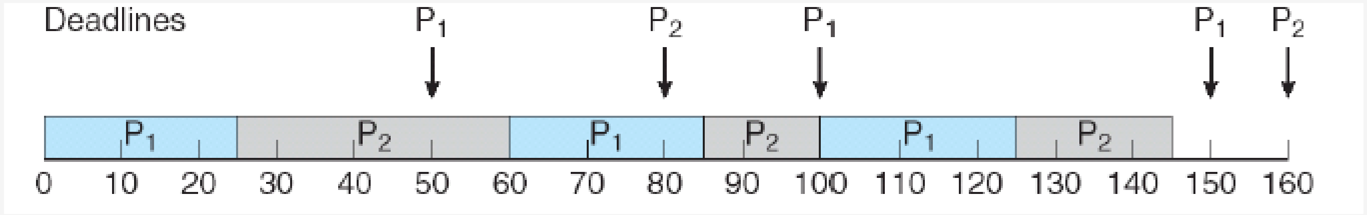
A task's priority is not fixed. It is decided at runtime based on how close it is to its absolute deadline.

Priorities are assigned according to deadlines:

the earlier the deadline, the higher the priority;
the later the deadline, the lower the priority

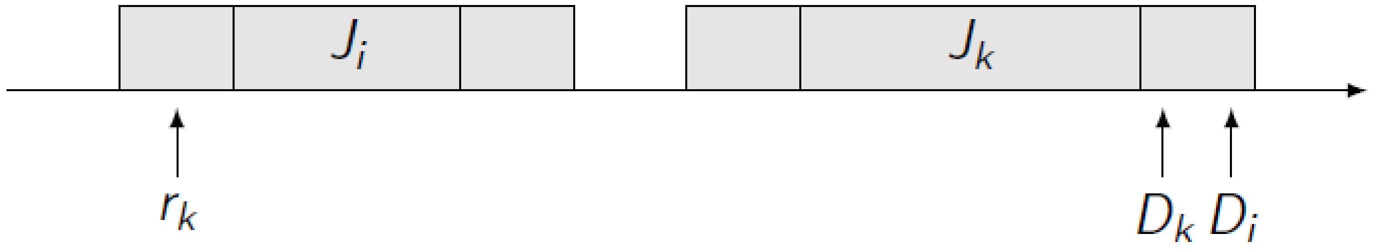
- Does not require the knowledge of execution time
- Optimal if single processor and preempitve

- Scheduler selects a job with earliest deadline
  (the job with earliest deadline has the "highest priority")

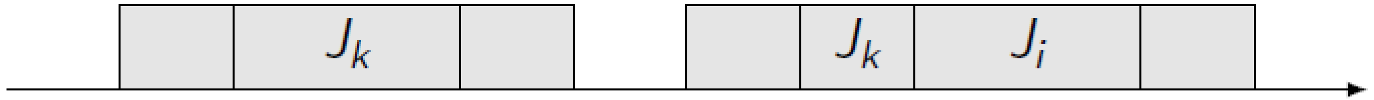


Earliest Deadline First (EDF)

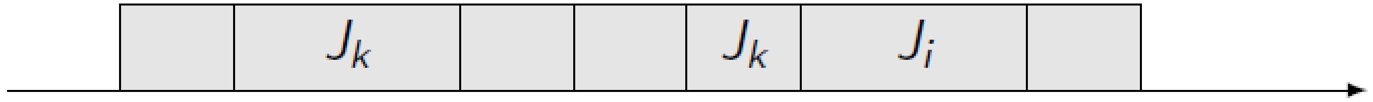
- Assume that parts of two jobs J_i a J_k are scheduled in non-EDF order:



- This can be easily resolved by swapping the jobs:



- These gaps can be easily eliminated by shifting the schedule ahead of time:



Example: Earliest Deadline First (EDF)

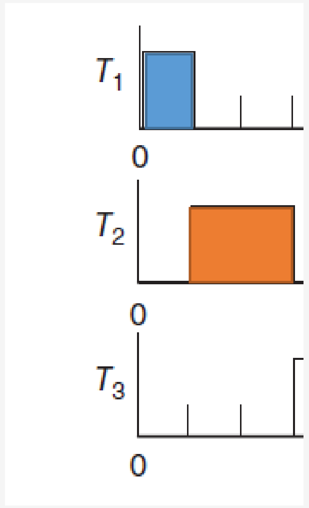
$T_1 = (4, 1), T_2 = (5, 2), T_3 = (10, 3.1)$

three independent, preemptable periodic tasks

At 0, the first instance in each task is released. Because the instance in T_1 has the earliest deadline, it executes. It is done at 1.

At 1, the instance in T_2 has higher priority than the instance in T_3 and executes. It is done at 3.

At 3, T_3 is the only task waiting to run, and so it executes.



Example: Earliest Deadline First (EDF)

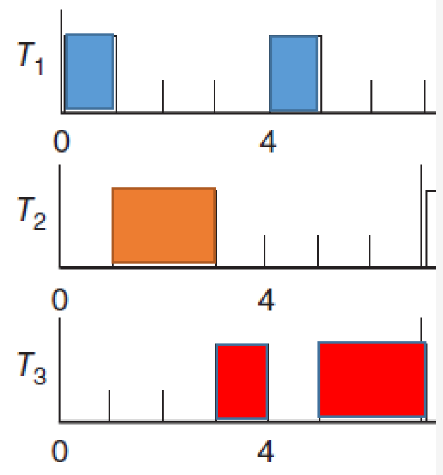
$T_1 = (4, 1), T_2 = (5, 2), T_3 = (10, 3.1)$

three independent, preemptable periodic tasks

At 4, the second instance in T_1 is released. Its deadline is at 8, earlier than the deadline of the T_3 instance, which is at 10. So, the T_1 instance preempts T_3 and executes. It is done at 5.

At 5, the second instance in T_2 is released. Its deadline is at 10, same as the deadline of the T_3 instance. Since T_3 arrives earlier, T_3 executes and is completed at 7.1, which means that the first instance of T_3 meets its deadline!

At 7.1, T_2 is the only task waiting to run, and so it executes.



Example: Earliest Deadline First (EDF)

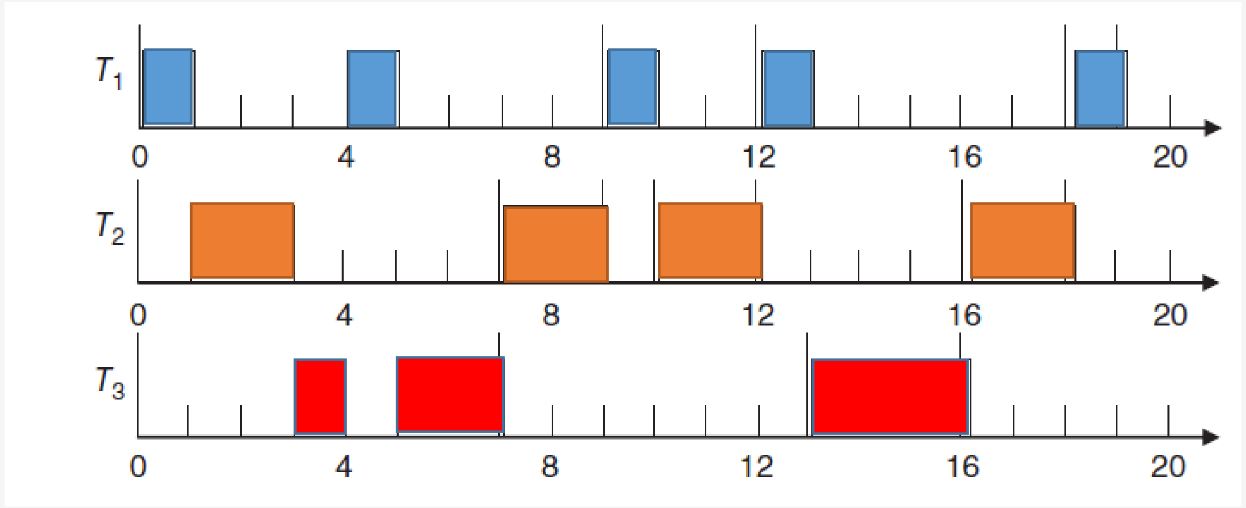
$T_1 = (4, 1), T_2 = (5, 2), T_3 = (10, 3.1)$

three independent, preemptable periodic tasks

At 8, the third instance in T_1 arrives. Its deadline is at 12, later than the deadline of the T_2 instance in execution, which still has 1.1 units to complete. So, T_2 continues to run and completes at 9.1.

At 9.1, T_1 is the only task waiting to run, and so it executes.

At 10, an instance in T_2 is released with deadline 15, and meanwhile, an instance in T_3 is released with deadline at 20. So, the T_1 instance continues to run. Both T_2 and T_3 are waiting.



Least Slack Time (LST)

LST is another of the most popular dynamic scheduling algorithm.

When a task has deadline di, execution time Ci and release time Ri,

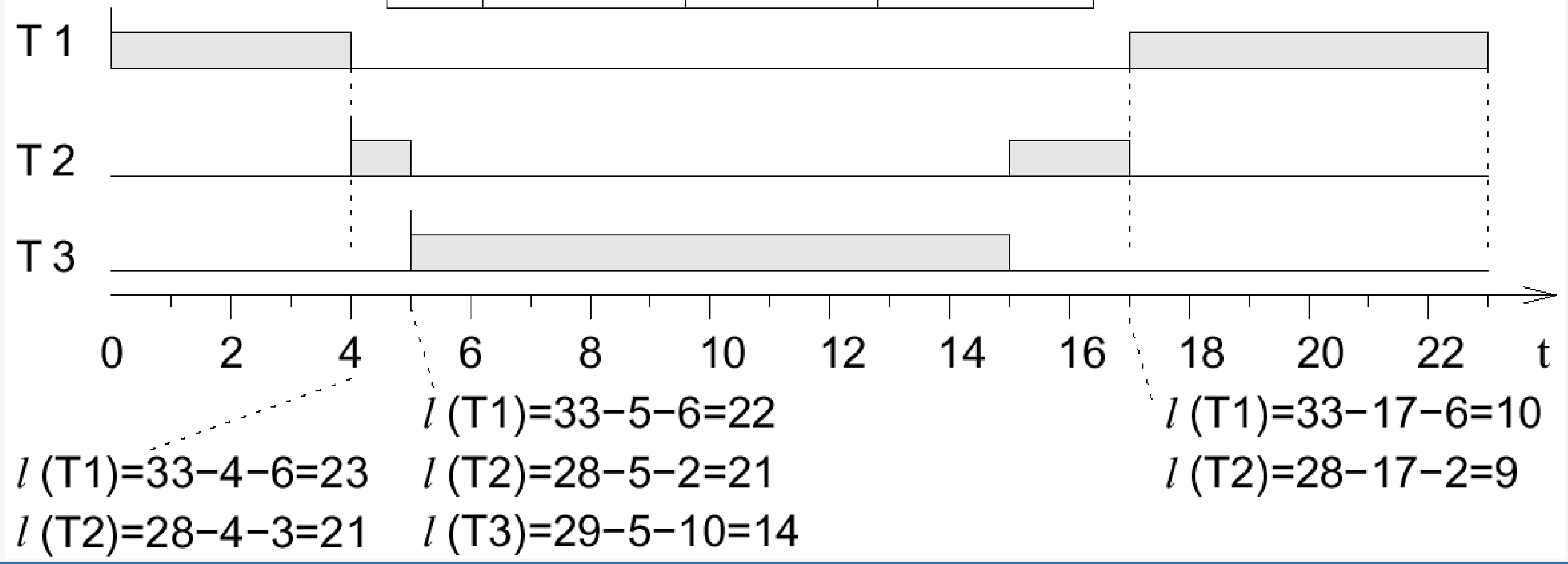
- Remaining execution time : $t_{REM} = C_i - (t - R_i)$
- Slack time: $t_{Slack} = d_i - t - t_{REM}$

- Assign priority to jobs based on slack time
- The smaller the slack time, the higher the priorty.
- More complex, requires knowledge of execution times and deadlines

Least Slack Time (LST)

- Remainin execution time : $t_{REM} = C_i - (t - R_i)$
- Slack time (laxity): $l = t_{Slack} = d_i - t - t_{REM}$

	R arrival	C duration	d deadline
T 1	0	10	33
T 2	4	3	28
T 3	5	10	29



l (T1)=33−4−6=23
l (T2)=28−4−3=21

l (T1)=33−5−6=22
l (T2)=28−5−2=21
l (T3)=29−5−10=14

l (T1)=33−17−6=10
l (T2)=28−17−2=9

Requires calling the scheduler periodically, and to recompute the laxity.

When t=4,

l (T1) = $d_1 - t - t_{REM} = 23$

Problems about Priority Based Scheduling

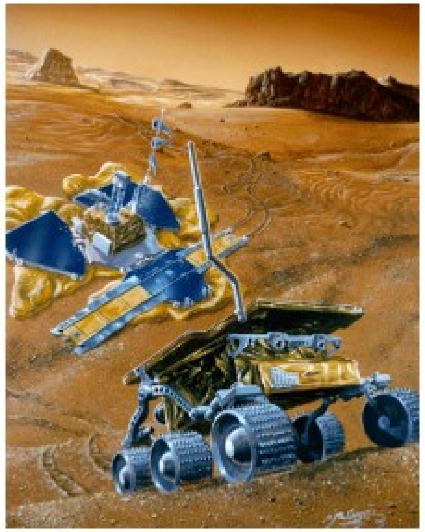
Mars Pathfinder Incident

- Landing on July 4, 1997
- "…experiences software glitches…"
- Pathfinder experiences repeated RESETs after starting gathering of meteorological data.

- RESETs generated by watchdog process.
- Timing overruns caused by priority inversion.

- Resources:

  `research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html`

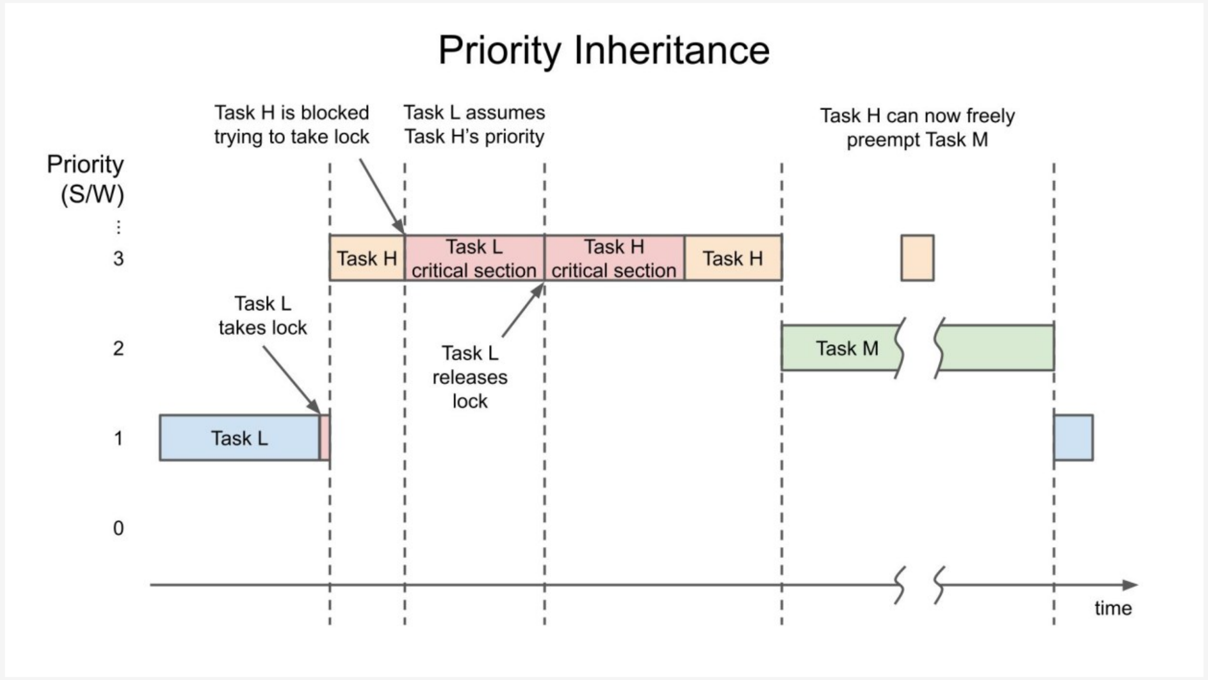


- "Faster, better, cheaper" had NASA and JPL using "shrink-wrap" hardware (IBM RS6000) and software (Wind River VxWorks RTOS).

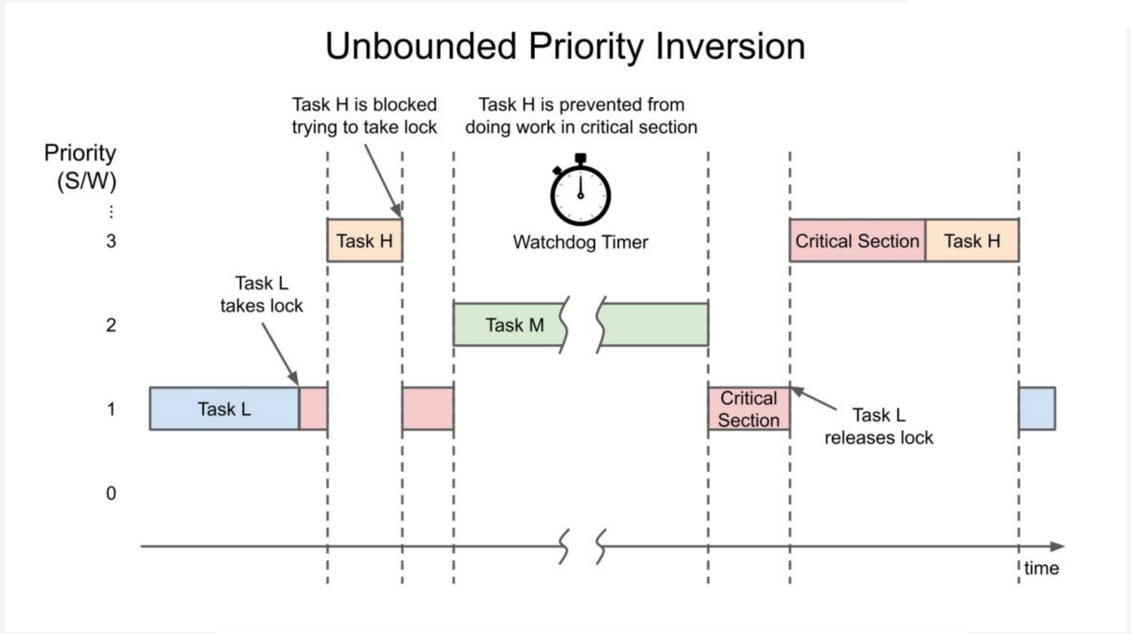
- Logging designed into VxWorks enabled NASA and Wind River to reproduce the failure on Earth. This reproduction made the priority inversion obvious.

Problems about Priority Based Scheduling

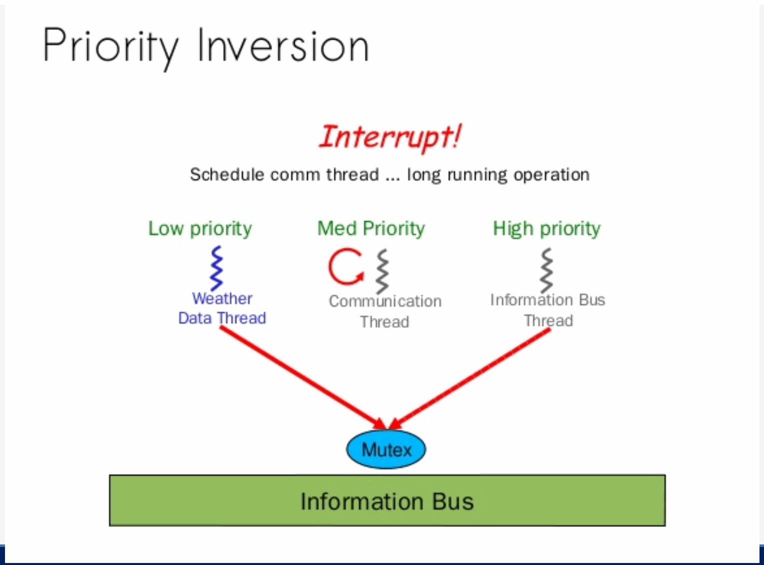
Solution:



Priority Inheritance
Task H is blocked trying to take lock
Task L assumes Task H's priority
Task H can now freely preempt Task M
Priority (S/W)
3
2
1
0
Task H
Task L critical section
Task H critical section
Task H
Task L takes lock
Task L releases lock
Task M
Task L
time



Unbounded Priority Inversion
Task H is blocked trying to take lock
Task H is prevented from doing work in critical section
Priority (S/W)
3
2
1
0
Task H
Watchdog Timer
Critical Section
Task H
Task L takes lock
Task M
Task L
Critical Section
Task L releases lock
time



Priority Inversion
Interrupt!
Schedule comm thread ... long running operation
Low priority
Med Priority
High priority
Weather Data Thread
Communication Thread
Information Bus Thread
Mutex
Information Bus