Real-Time Systems

IL2206 Embedded Systems

Ingo Sander

KTH Royal Institute of Technology
Stockholm, Sweden
ingo@kth.se

Outline

Outline

Real-Time

- A system is classified as real-time if it is required to complete the work on a timely basis.

$\Rightarrow$ It does not mean that a system must meet some timing deadlines!

- Often real-time systems are also safety-critical, a missing of a deadline can lead to disastrous consequences
- For safety-critical systems, it is of utmost importance that the correctness of the timely behaviour of a real-time system can be validated
- Bugs in real-time systems may be very costly to fix afterwards, especially in embedded systems

Hard, Firm and Soft Deadlines

- Real-time systems are classified dependent on the timing constraints that are imposed on them.
- Buttazzo (2011) introduces the following definitions for real-time tasks.
- A real-time task is said to be
  - hard, if producing the results after its deadline may cause catastrophic consequences on the system under control.
  - firm, if producing the results after its deadline is useless for the system, but does not cause any damage.
  - soft, if producing the results after its deadline has still some utility for the system, although causing a performance degradation.

Hard, Firm and Soft Deadlines

- Real-time systems are classified dependent on the timing constraints that are imposed on them.
- Buttazzo (2011) introduces the following definitions for real-time tasks.
- A real-time task is said to be
  - hard, if producing the results after its deadline may cause catastrophic consequences on the system under control.
  - firm, if producing the results after its deadline is useless for the system, but does not cause any damage.
  - soft, if producing the results after its deadline has still some utility for the system, although causing a performance degradation.

Definition of hard, firm, and soft real-time
There is no single definition of these terms, different definitions exist in the literature.

Hard and Soft Real-Time Systems

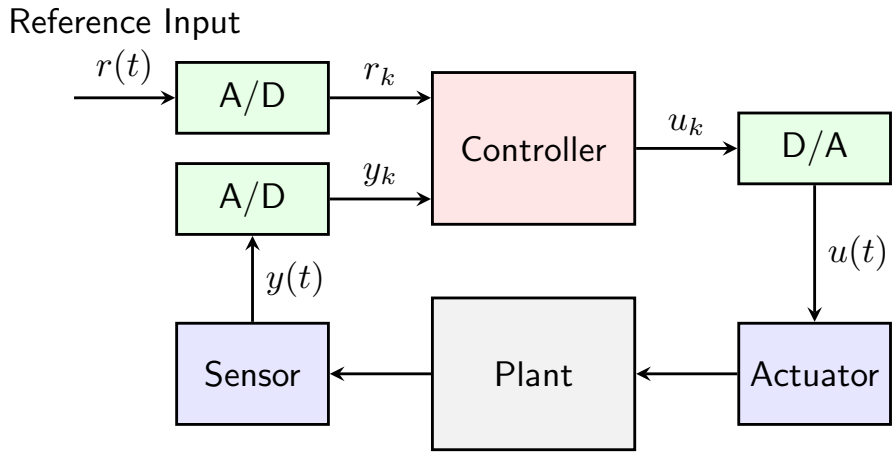
- A hard real-time system is a system containing at least one hard real-time task
  - automatically controlled train
  - airbag controller
- A soft real-time system is a system containing only soft real-time tasks
  - multimedia applications

Hard and Soft Real-Time Systems

- A hard real-time system is a system containing at least one hard real-time task
  - automatically controlled train
  - airbag controller
- A soft real-time system is a system containing only soft real-time tasks
  - multimedia applications

Hard and soft real-time systems
There is no well-defined border between hard- and soft real-time systems!

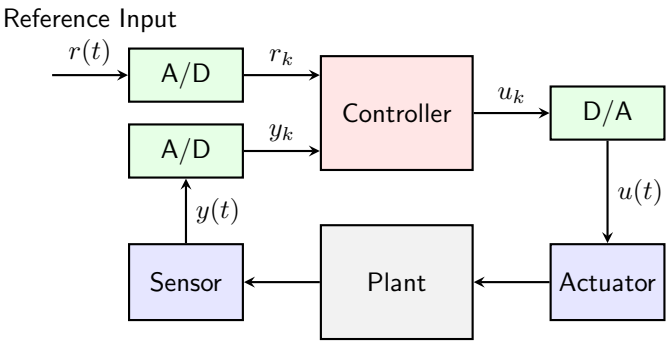
Real-Time Example: Digital Process Control



Digital controller regulates plant (plane, train, engine, patient, ...) by comparing measured state y_k with desired state r_k and calculating output results u_k to stimulate plant.

Digital Process Controller

Possible Implementation

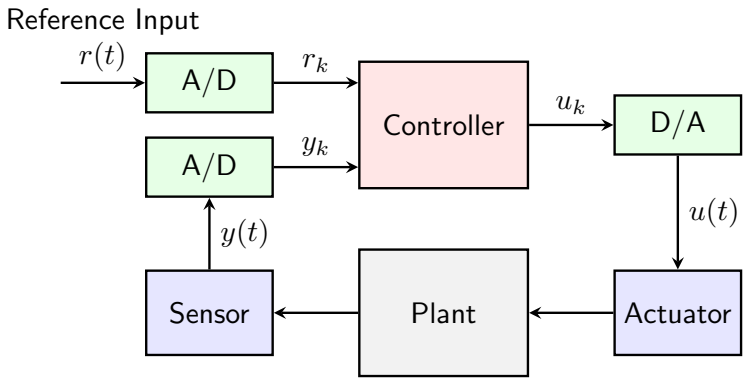


Algorithm (Pseudo-code):

Set timer to interrupt periodically with period T;
while forever **do**
    Wait for timer interrupt;
    Analogue-to-digital conversion to get y;
    Compute control output u;
    Output u and digital-to-analogue conversion of u;
end

Digital Process Control

Sampling Period



- The time T between two consecutive measurements is called the sample period
  - small T better approximates the analogue behaviour
  - large T means less processing power is needed

$\Rightarrow$ Compromise is needed!

Digital Process Control

Assumptions

For an ideal system we make the following assumptions:

- Sensor data give accurate estimates of the state variables that are monitored and controlled
- Sensor data give accurate state of the plant (sometimes values are measured indirectly)
- All parameters representing the dynamics of the plant are known

Digital Process Control

Assumptions

For an ideal system we make the following assumptions:

- Sensor data give accurate estimates of the state variables that are monitored and controlled
- Sensor data give accurate state of the plant (sometimes values are measured indirectly)
- All parameters representing the dynamics of the plant are known

Most systems are not ideal!

Digital Process Control

Role of the Control Law

If we have a non-ideal system, the control law computation will try

- to make a correct estimation of the state of the plant based on "noisy" sensor values
- compensate for the less accurate measurement by an improved model and a more complex algorithm

Discussion

Control Systems vs Information Systems

- The control law can compensate for missing and less accurate data
- In many information systems, loss or change of data is not acceptable

Discussion

Control Systems vs Information Systems

- The control law can compensate for missing and less accurate data
- In many information systems, loss or change of data is not acceptable

Different approaches may be needed for control and information real-time systems!

Real-Time Applications

Real-time applications can consist of

- purely cyclic tasks, which execute periodically
- aperiodic and sporadic jobs, which are not arriving according to a periodic scheme

It is much easier to reason about synchronous, cyclic and predictable systems

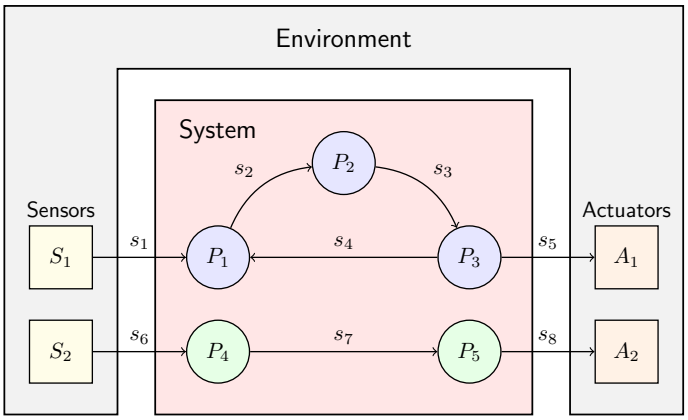
Safety and Correctness are First-Class-Citizens in Real-Time Systems

- Take a conservative design approach
- Efficiency is important, but correctness is more important!

Outline

How to design a real-time embedded system?

- Typical functionality of embedded control system:
  - System reads data from sensors
  - System computes some functionality, which might involve multiple communicating processes
  - System writes data to actuators
- Often system has to satisfy timing constraints $\Rightarrow$ real-time system



How to support development of embedded real-time system?

- Embedded systems are would benefit from parallel execution
- Difficult to describe as a sequential program
- Required to express potential and explicit parallelism in embedded system programs

How to support development of embedded real-time system?

- Embedded systems are would benefit from parallel execution
- Difficult to describe as a sequential program
- Required to express potential and explicit parallelism in embedded system programs

Idea

- Encapsulate sequential programs into concurrent (parallel) processes
- Provide mechanisms and objects for communication between processes

How to support development of embedded real-time system?

- Embedded systems are would benefit from parallel execution
- Difficult to describe as a sequential program
- Required to express potential and explicit parallelism in embedded system programs

Idea

- Encapsulate sequential programs into concurrent (parallel) processes
- Provide mechanisms and objects for communication between processes

General Principles for Concurrent Programs

- Base for many operating systems
- Base for hardware description languages

Terms: Concurrent and Parallel

Terminology and the discussion on semaphores, monitors and other programming constructs are based on the text book of Ben-Ari (2006).

- A concurrent program is a set of sequential programs that can be executed in parallel. Here, the term process is used for a sequential program that is part of a concurrent program and the term program is used for this set of sequential programs (processes).
- Traditionally the term parallel is used for systems, in which the execution of several programs overlap in time by running them on separate processors.

The term concurrent is reserved for potential parallelism, in which the executions may, but need not, overlap.

Processes and Threads

- A software application can be organised into different sequential processes that cooperate with each other
- **Process**
  - A process is a program in execution (single thread of control)
  - A process has its own address space (instructions, data, stack)
- **Thread**
  - Thread can be seen as a lightweight process
  - Threads share the same address space
  - Many operating systems for embedded applications only support threads
    - Low overhead
    - Low hardware requirements

Static and Dynamic Systems

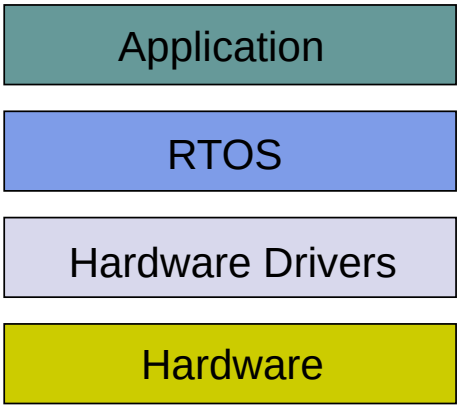
- **Static System**
  - All processes and threads in the system are known from the beginning
  - No possibility to create or destroy new processes or threads
- **Dynamic System**
  - Possibility to create or destroy new processes or threads
  - Increases the complexity of the operating system

Software Support for Real-Time Systems

- Inside a programming language
  - Ada provides concurrent tasks as part of a standardised language and also offers a real-time annex for the modelling of real-time systems
  - On top of a programming language (normally C) in form of a real-time operating system

Real-Time Operating System

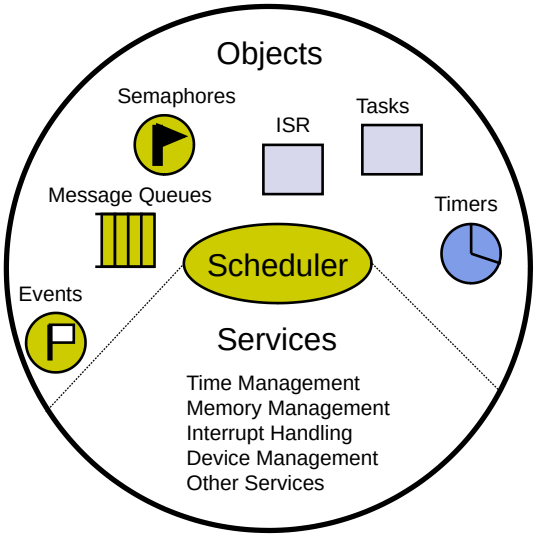
- A real-time operating system (RTOS) is a program that
  - schedules execution of tasks in a predictable manner
  - manages system resources
  - gives a base to develop multi-threaded applications with a real-time behaviour



Components in a Real-Time Operating Systems

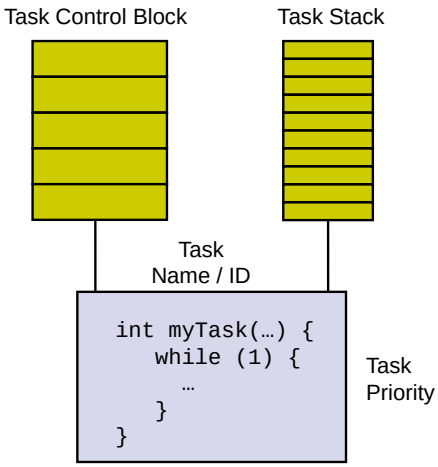
An RTOS has typically the following components

- **Scheduler**: Determines which task is running at each time instant
- **Objects**: Special constructs like tasks, semaphores or message queues
- **Services**: Operations that the kernel performs on objects



Tasks

- A task is an independent thread of execution
- Tasks are schedulable
- Tasks are defined by a distinct set of parameters and supporting data structures
- Usually tasks have different priorities
- There are usually kernel defined tasks that run on pre-defined priorities (e.g. the idle task)

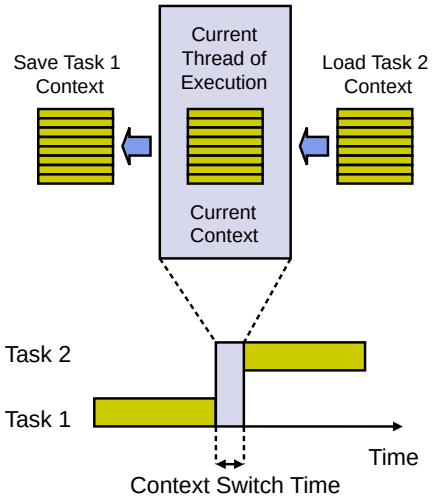


Scheduler

- The scheduler applies an algorithm that decides, which task executes at what time instant
- Scheduler implements multitasking, which "fakes" parallel execution on a single processor
- Program consists of several tasks, but only one task can run at the same time
- Timer interrupts or special events control when the scheduler is run

Context Switch

- Each task has its own context, which is determined by a task control block (TCB)
- The TCB (together with the stack) contains the state of a task
  - Program Counter
  - Registers
- When a task is executing the context of its TCB is loaded from memory and the TCB context of the replaced task is saved into memory
- Context switching is performed by the dispatcher (part of scheduler)



Cooperative Multitasking

- Scheduling is based on cooperation (non-preemptive kernel)
- Running task must explicitly give up the control of the CPU
- Interrupts can still interrupt a running task
- Drawback: Responsiveness can be very low
  - High-priority task can be blocked for a long time because running task does not give up control of CPU
  - Designer has to ensure that tasks do not keep control of CPU for a too long time

Preemptive Multitasking

- The highest priority task is always run on the CPU
- A running tasks can be preempted, if a higher priority task becomes ready
- Interrupts preempt running tasks
- Responsiveness is improved: Possible to calculate, when highest priority task will get control of CPU

Reentrant Function

- A reentrant function can be used by one or more tasks without fear for data corruption
- Reentrant functions either use local variables or protect data when global variables are used
- Example for re-entrant function

```
1  int triple(int x)
2  {
3          return 3 * x;
4  }
```

Non-Reentrant Functions in Preemptive Multitasking

Non-reentrant function `Swap(int *x, int *y)`

```
1  int Temp; /* Global Variable */
2
3  void Swap(int *x, int *y)
4  {
5          Temp = *x;
6          *x = *y;
7          *y = Temp;
8  }
```

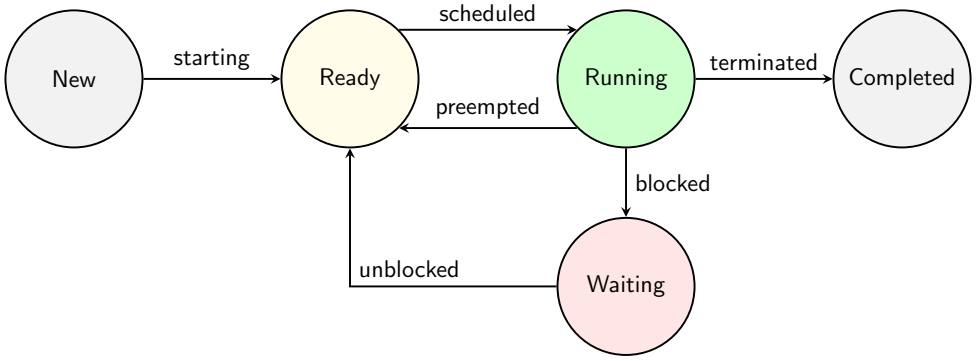
Non-reentrant code can cause serious bugs ⇒ Extra care needed!

If the running thread is preempted after `Temp = *x` in the function `Swap()`, and another thread calls the function `Swap(int *x, int *y)` and executes it, `Temp` will be changed and the original task will get a wrong value for `y`, when rescheduled.

Task States

A task can be in several states

- **New** Task has been created
- **Ready** Task is ready for execution, but has not been scheduled
- **Running** Task is scheduled
- **Waiting** Task waits for resources and is not ready for execution
- **Completed** Task has executed its final statement



Interprocess Communication Mechanisms

Operating systems and concurrent programming languages offer programming constructs for the communication between processes.
In order to develop an efficient embedded system these constructs have to be well understood.

Examples

Semaphore, Mailbox, Message Queue, Event Flag, Monitor, Protected Object, Rendezvous, Channel

In this lecture we concentrate on semaphores and message queues.

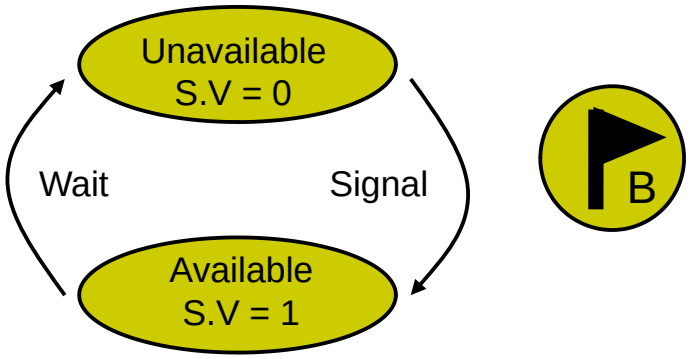
Semaphores

- A semaphore can be used for process synchronisation or to ensure mutual exclusion
- A semaphore S can be viewed as a record with two fields.
  - $S.V$ A non-negative integer
  - $S.L$ A set of processes
- A semaphore is initialised with a value $k \geq 0$ for $S.V$ and with the empty set $\emptyset$ for $S.L$.
- A process p can use two statements that are defined on a semaphore: `wait(S)` and `signal(S)`[1].
- A semaphore, where $S.V$ can take arbitrary non-negative values is called general semaphore, a semaphore, where $S.V$ can only take the values 0 and 1 is called binary semaphore.

[1] Originally Dijkstra used `P(S)` for `wait(S)` and `V(S)` for `signal(S)`.

Binary Semaphore: States

- A binary semaphore has either a value of
  - S.V = 0 (unavailable) or
  - S.V = 1 (available)
- A binary semaphore is a shared resource ⇒ Any process can release it!



wait(S) and signal(S)

`wait(S)`

if $S.V > 0$ **then**
  $S.V \leftarrow S.V - 1$
else
  $S.L \leftarrow S.L \cup \mathtt{p}$
  $\mathtt{p.state} \leftarrow \mathtt{blocked}$
end if

If the value of $S.V$ is

- zero, the process p is blocked and is added to the set of processes waiting for S. The process p is blocked on the semaphore S.
- non-zero, decrement $S.V$. The process p continues execution.

`signal(S)`

if $S.L = \emptyset$ **then**
  $S.V \leftarrow S.V + 1$
else
  $S.L \leftarrow S.L - \{\mathtt{q}\}$
  $\mathtt{q.state} \leftarrow \mathtt{ready}$
end if

If $S.L$ is

- empty, increment the value of $S.V$
- not empty, unblock an arbitrary process q in the waiting list $S.L$. The process q is unblocked and is put into the state `ready`.

Critical Section Problem

Problem definition

- Each of N processes is executing in an infinite loop a sequence of statements that can be divided into two subsequences
  - the critical section
  - the non-critical section.
- The following properties shall be fulfilled

  Mutual exclusion Statements from the critical section of two or more processes must not be interleaved

  Freedom from deadlock If one or more processes are trying to enter their critical section, then one of them must eventually succeed

  Freedom from (individual) starvation If any process tries to enter the critical section, then that process must eventually succeed

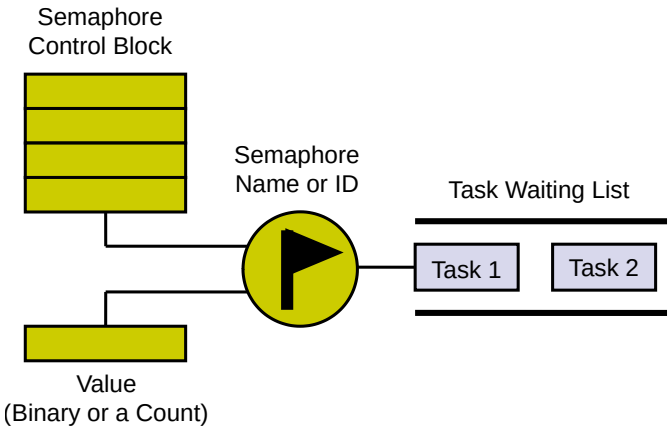
Semaphores: Critical Section Problem

- The critical section problem is difficult to solve on a bare machine without support for interprocess communication mechanisms.
- The critical section problem for two processes can be solved with a binary semaphore S that is initialised to $S = \{1, \emptyset\}$.

P_1	P_2
`loop forever`	`loop forever`
`  non-critical section`	`  non-critical section`
`  wait(S)`	`  wait(S)`
`  critical section`	`  critical section`
`  signal(S)`	`  signal(S)`

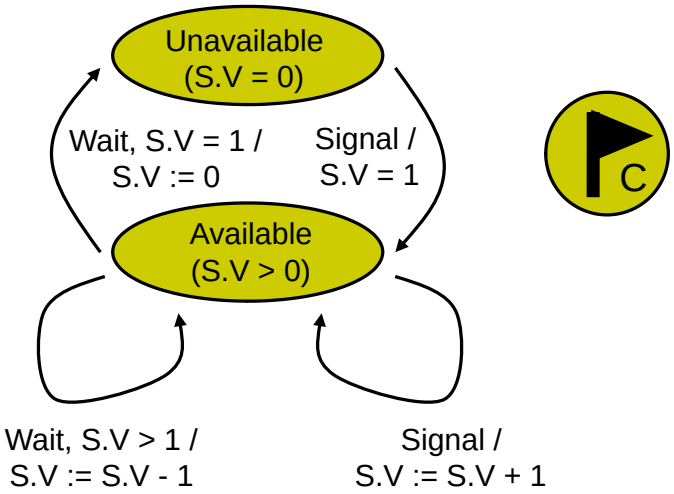
Creation of a semaphore

- When a semaphore is first created, the kernel creates
  - an associated semaphore control block (SCB)
  - a unique ID
  - an initial value
  - a task-waiting list



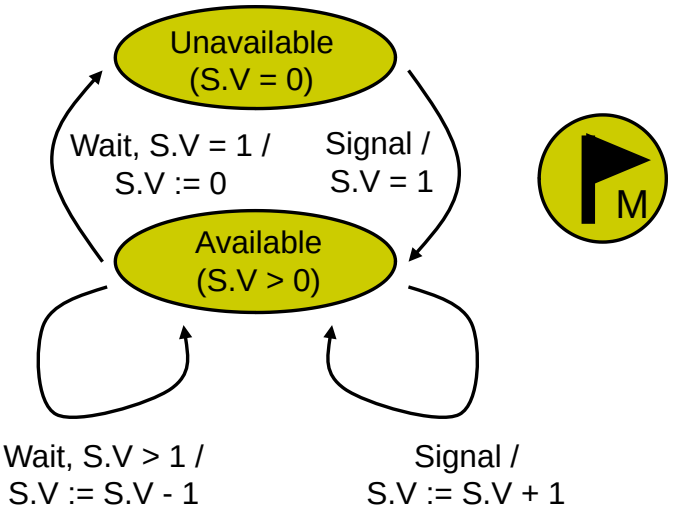
Counting Semaphore: States

- A counting semaphore uses a count to be able to give more than one task access
  - S.V = 0 (unavailable), or
  - S.V > 0 (available)
- The count can be bounded (initial value gives maximum number) unbounded (no maximum value)
- A counting semaphore is a shared resource ⇒ Any task can release it!



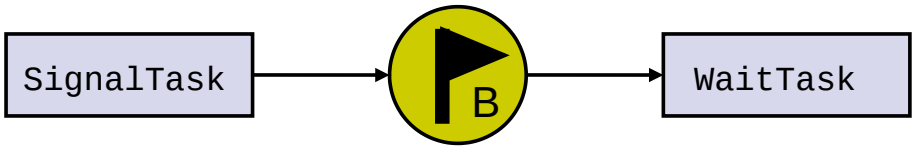
Mutual exclusion semaphore (Mutex)

- A mutex is a special binary semaphore that supports ownership and often recursive access (recursive mutex)
- Only the task that has locked a mutex can release it
- Mutex is created in unlocked state



Synchronisation with Binary Semaphore

- Semaphores can be used to synchronise tasks
- The example below shows how a binary semaphore can be used to coordinate the execution of control
- The task WaitTask has to wait until the task SignalTask releases the binary semaphore



Semaphores: A Low-Level Mechanism

- Semaphores can be implemented with very low-overhead
- Conceptually very difficult to analyse systems with multiple semaphores, which are accessed by several tasks

Be very restrictive with semaphores

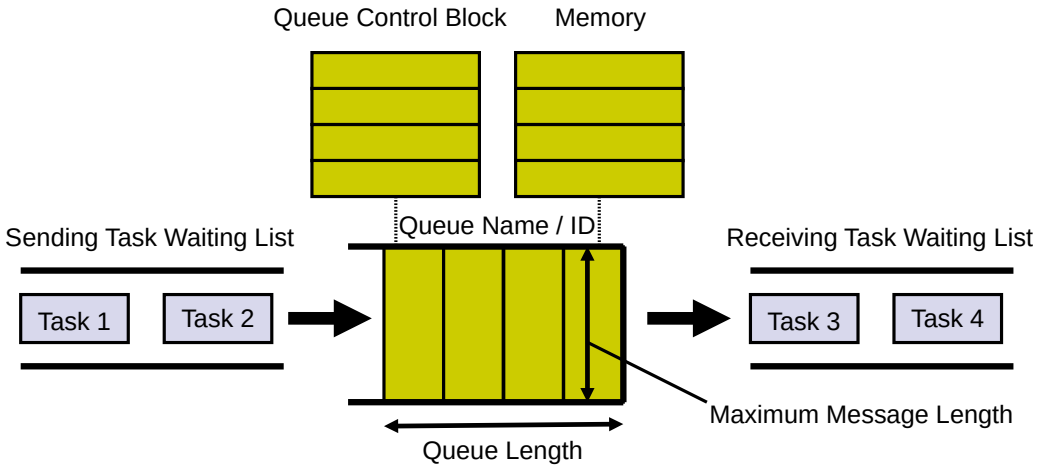
Very difficult to debug a system, where a signal statement is missing.

Message Queue

- A message queue is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronise with data
- A message queue holds temporarily messages from a sender until a receiver fetches them
- Temporary buffering decouples sending from receiving
- A message queue with a single buffer is often called a mailbox

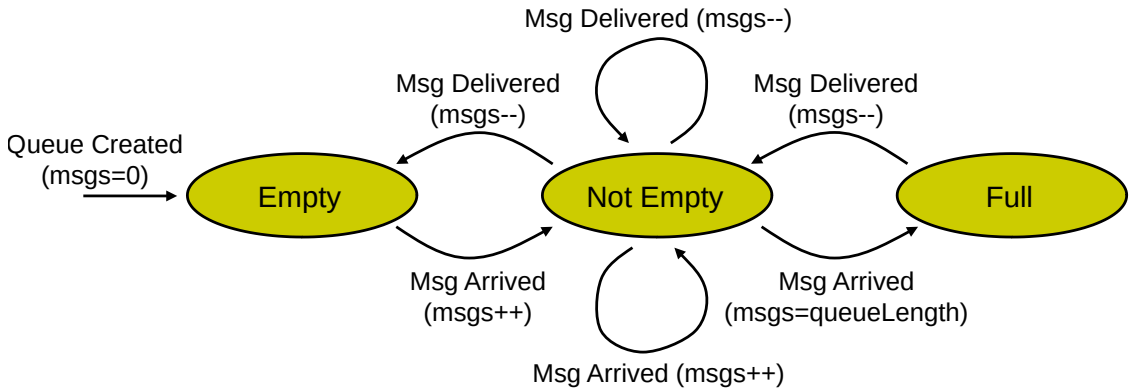
Message Queue: Data Structure

- When a message queue is created, it is assigned a queue control block, a queue name, an ID, memory buffers, a queue length, a maximum message length, and one or more task-waiting lists



Message Queue: States

- The message queue is operated by means of an FSM with 3 states
- If the queue is full, the sending task will not be successful and has to wait before sending its message (sending task waiting list)
- If the queue is empty, the receiving task has to wait for its messages (receiving task waiting list)

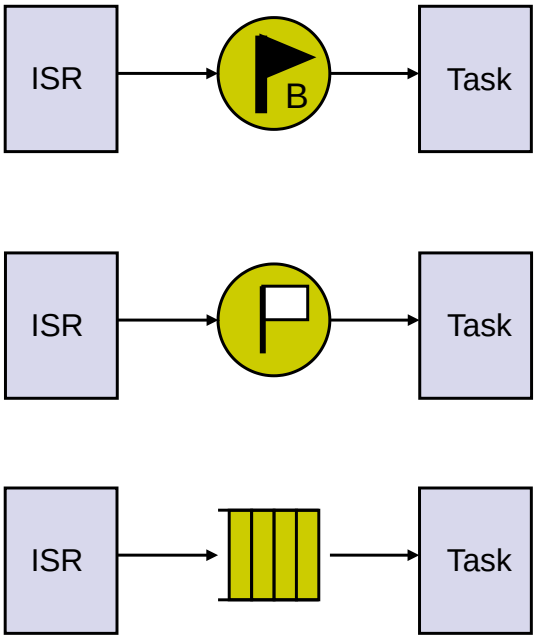


Sending and Receiving Messages

- **Sending Messages**: Messages can be sent to the queue in different ways:
  - non-blocking (ISR and tasks)
    - sending task does not wait, if the send call is successful
    - if the queue is full the send call returns an error
  - block with a timeout (tasks only)
    - send call waits if the queue is full for a defined period
  - block forever (task only)
    - send call wait until message can be send
- **Receiving Messages**: Messages can be received from the queue in different ways:
  - non-blocking
  - block with a timeout
  - block forever
- Blocking occurs when the message queue is empty

ISR to Task Synchronisation

- Interrupt service routines can be synchronised with tasks in several ways
- During the interrupt control is passed to the ISR who for instance releases a semaphore, sets an event flag or sends a message
- When the task continues execution it can take appropriate action

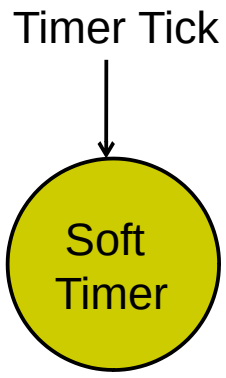


Timer

- Timers are an integral part of an embedded system
- Hardware timers are available as peripheral components to an embedded processor core
- Many real-time operating systems provide soft timers as services to the user
- Usually multiple soft timers can run at the same time

Soft Timer

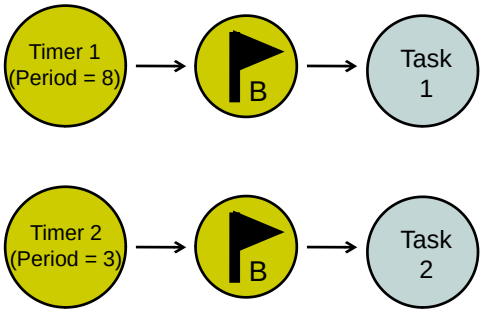
- Soft timer needs to be triggered by an external signal (Timer Tick)
- The resolution of the soft timer depends on the frequency of the external timer tick
- A soft timer can usually be operated in two modes
  - Periodic: Timer runs for some period and restarts
  - One-Shot: Timer runs once for a defined period
- Action can be defined to be executed, when period is expired (callback function)



Using Soft Timers for Periodic Tasks

- Soft timers can be used to implement periodic tasks
  - Periodic timer signals a semaphore every time interval
  - Task waits for semaphore after operation
- **NOTE**: Computation time cannot be controlled by the real-time operating system!

- Task $\tau_1(8, 4)$:
  - Period $T_1 = 8$,
  - Computation Time: $C_1 = 4$
- Task $\tau_2(3, 1)$:
  - Period $T_2 = 3$,
  - Computation Time: $C_1 = 1$



Outline

Real-Time System Model

The classic real-time theory based on the seminal paper of Liu and Layland (1973) uses

1. a workload model describing the applications in the system
2. a resource model describing the resources available to the system
3. scheduling algorithms defining how the applications uses the resources at all times

Tasks and Jobs

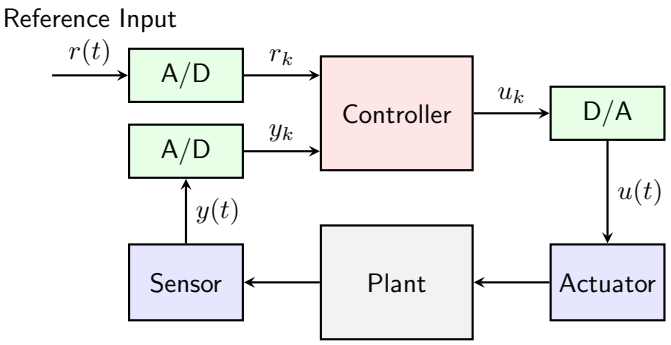
Task A task is a set of related jobs which jointly provide a system function

Job A job or task instance is a unit of work that is scheduled and executed by the system

A task that implements a digital process control may periodically execute jobs that perform the control law computation

Digital Process Controller

Possible Implementation

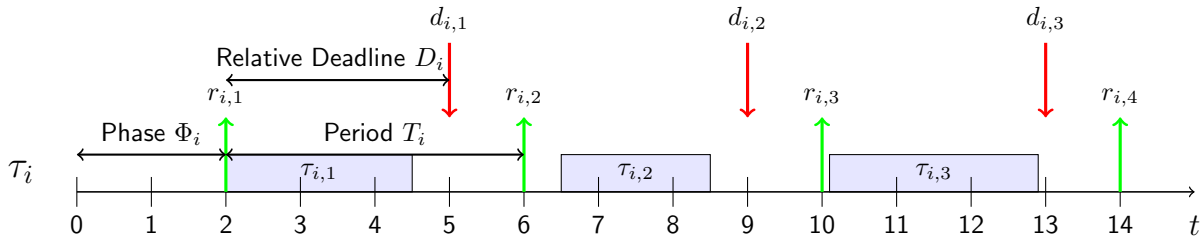


Algorithm (Pseudo-code):

Set timer to interrupt periodically with period T;
while forever **do**
  Wait for timer interrupt;
  Analogue-to-digital conversion to get y;
  Compute control output u;
  Output u and digital-to-analogue conversion of u;
end

Tasks and Jobs

Digital Process Control



- After an initial time, the phase Φ_i, the first job $\tau_{i,1}$ of task τ_i is released at the release time $r_{i,1} = \Phi_i$.
- A job $\tau_{i,j}$ has to compute its result within the task τ_i's relative deadline D_i.
- The absolute deadline of a job $\tau_{i,j}$ is $r_{i,j} + D_i$.
- A new job $\tau_{i,j}$ of of a task τ_i is released periodically at the time $\Phi_i + (j-1)T_i$, where T_i is the period of the task τ_i.

Jobs

- The basic components of any real-time application are jobs
- The operating system treats jobs as a unit of work and allocates processors and resources
- A job or task instance $J_{i,k} = \tau_{i,k}$ of a task τ_i is characterised by several parameters
  - release time
  - execution time
  - deadlines
  - preemptivity
  - resource requirements
  - soft or hard real time
  - ...

Real-Time Theory: Notations

- Unfortunately, different authors have used different symbols for a few parameters to specify the real-time model.
- The course will mainly use the notation used in (Buttazzo, 2011), but will also introduce the notation used in (Liu, 2000).
- Students should be able to use both notation styles!

Real-Time Theory: Notations

- A task i is denoted τ_i in (Buttazzo, 2011) and T_i in (Liu, 2000).
- A *job* k of a task i is denoted $\tau_{i,k}$ in (Buttazzo, 2011) and $T_{i,k}$ in (Liu, 2000).
- The period of a task i is denoted T_i in (Buttazzo, 2011) and P_i in (Liu, 2000).
- The computation time or execution time of a job within task i is denoted C_i (Buttazzo, 2011) and e_i in (Liu, 2000). It is the amount of time required to complete the job, when it executes alone and has all resources it needs.

Computation Time

Especially for hard-real time systems C_i typically denotes the maximum execution time, the so called worst case execution time (WCET) will be used to be able to give save bounds.

Real-Time Theory: Notations

- The phase of a task i is denoted Φ_i.
- The release time r_i or arrival time a_i is the time when the task i becomes ready for execution.
- The absolute deadline is the time, when the k-th job $\tau_{i,k}$ in task i needs to be completed.
- The relative deadline D_i is the difference between the absolute deadline $d_{i,k}$ and the release time $r_{i,k}$ for a job $\tau_{i,k}$ within a task τ_i. The relative line is the same for all jobs in a task τ_i.
- The start time $s_{i,k}$ of a job $\tau_{i,k}$ in task i is the time at which job $\tau_{i,k}$ starts to execute.
- The finishing time or completion time $f_{i,k}$ of a job $\tau_{i,k}$ in task i is the time at which job $\tau_{i,k}$ completes it execution.
- The response time of a job $\tau_{i,k}$ is defined as $f_{i,k} - s_{i,k}$ and denoted $R_{i,k}$ in (Buttazzo, 2011) and $w_{i,k}$ in (Liu, 2000).

Periodic, Aperiodic and Sporadic Tasks

- A periodic task executes periodically, hard deadline is assumed
- An aperiodic task executes jobs on demand, jobs have a soft deadline
- A sporadic task executes jobs on demand, jobs have a hard deadline

Periodic Task - Notation

- A periodic task τ_i is defined as a tuple $\tau_i(\phi_i, T_i, C_i, D_i)$.
- The notation $\tau_i(\phi_i, T_i, C_i)$ implies that $D_i = T_i$.
- The notation $\tau_i(T_i, C_i)$ implies that $\Phi_i = 0$ and $D_i = T_i$.

Periodic Task - Notation

- A periodic task τ_i is defined as a tuple $\tau_i(\phi_i, T_i, C_i, D_i)$.
- The notation $\tau_i(\phi_i, T_i, C_i)$ implies that $D_i = T_i$.
- The notation $\tau_i(T_i, C_i)$ implies that $\Phi_i = 0$ and $D_i = T_i$.

Different Notations exist!
Liu (2000) uses the following notation for a task $T_i = (\phi_i, p_i, e_i, D_i)$, where p_i is the period and e_i the computation time.

Outline

Task Assignment and Scheduling

- **Problem**
  - A set of tasks has to be implemented on one or more processors
  - Each task is accompanied by information (e.g. computation time, periodicity) and constraints (e.g. release times)
- **Objective**
  - To find a schedule that is feasible, i.e. a schedule where all requirements are met!

Scheduling Algorithms

- In order to find a schedule, different algorithms for dynamic schedulers have been proposed
- These algorithms use priorities in order to decide, which task is scheduled onto which processor at what time instant
- An optimal algorithm produces a feasible schedule, if it exists
- We only look at algorithm for single processors with single thread of execution ⇒ Task assignment is trivial

Dynamic Schedulers

RTOSs use dynamic scheduling algorithms, which perform scheduling at run-time. It is also possible to use static scheduling, where the schedule is calculated before system start and stored in tables in memory.

Assumptions

- Tasks are periodic and are characterised by $\tau_i(T_i, C_i)$ where
  - each job in task $\tau_i(T_i, C_i)$ has computation time C_i;
  - jobs in task $\tau_i(T_i, C_i)$ are released periodically with a period T_i at the beginning of period;
  - period and relative deadline are equal, $T_i = D_i$;
  - there is no initial phase, $\Phi_i = 0$
- A job has to be completed before its deadline
- Tasks may be preempted
- No dependency between tasks
- Scheduler can run at any time instance
- Context-switch time is zero
- Tasks are scheduled on a single-thread processor

Assumptions are Idealisations

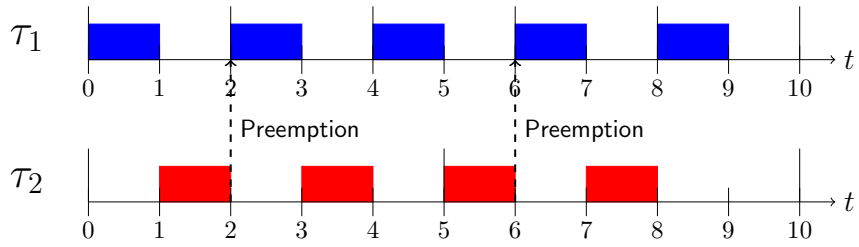
Many practical factors are not taken into account

Priority-Driven Scheduling - First Example

- Priority-driven scheduling is a dynamic scheduling method, where the scheduler always schedules the task that is ready and has the highest priority
- Example:
  - Task Set: $\tau_1(2, 1)$, $\tau_2(5, 2)$
  - Assumption: Task τ_1 has a higher priority than τ_2: $P_1 > P_2$

Priority-Driven Scheduling - First Example

- Priority-driven scheduling is a dynamic scheduling method, where the scheduler always schedules the task that is ready and has the highest priority
- Example:
  - Task Set: $\tau_1(2,1)$, $\tau_2(5,2)$
  - Assumption: Task τ_1 has a higher priority than τ_2: $P_1 > P_2$



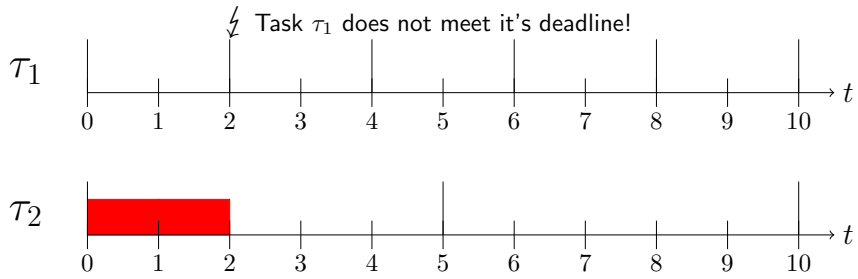
  - Schedule needs to be simulated for one hyperperiod $H = lcm(T_1, T_2) = 10$.
  - Both tasks meet their deadlines $\Rightarrow$ Feasible schedule exists!

Priority-Driven Scheduling - First Example

- Example: Change of Priorities
  - Task Set: $\tau_1(2,1)$, $\tau_2(5,2)$
  - Assumption: Task τ_2 has a higher priority than τ_1: $P_2 > P_1$

Priority-Driven Scheduling - First Example

- Example: Change of Priorities
  - Task Set: $\tau_1(2,1)$, $\tau_2(5,2)$
  - Assumption: Task τ_2 has a higher priority than τ_1: $P_2 > P_1$



  - Task τ_1 misses its first deadline at $t = 2 \Rightarrow$ No feasible schedule!

Rate-Monotonic Scheduling

- The rate-monotonic algorithm assigns fixed task priorities in reverse-order of period length
  - The shorter the period, the higher the priority
  - Idea: Tasks that require more frequent action should receive higher priority
- The rate-monotonic algorithm uses preemption
- The rate-monotonic algorithm uses fixed (static) priorities $\Rightarrow$ all jobs in a task have the same priority

The algorithm is called rate-monotonic algorithm (RMA) and the schedule produced by the algorithm rate-monotonic schedule (RMS).

Rate-Monotonic Analysis

The rate-monotonic analysis uses several assumptions:

- All process run on single CPU
- Zero context switch time
- No data dependencies between tasks
- Task computation time is constant
- Deadline is at end of period
- Ready task with highest-priority will be scheduled

Rate-Monotonic Scheduling

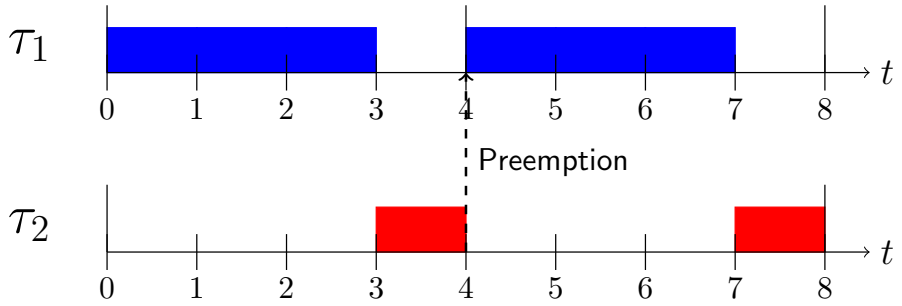
First Example

- Task Set: $\tau_1(4,3)$, $\tau_2(8,2)$
- Task τ_1 has shorter period $\Rightarrow$ Priorities: $P_1 > P_2$

Rate-Monotonic Scheduling

First Example

- Task Set: $\tau_1(4,3)$, $\tau_2(8,2)$
- Task τ_1 has shorter period $\Rightarrow$ Priorities: $P_1 > P_2$



- Both tasks meet their deadlines $\Rightarrow$ Feasible schedule exists!

Rate-Monotonic Scheduling

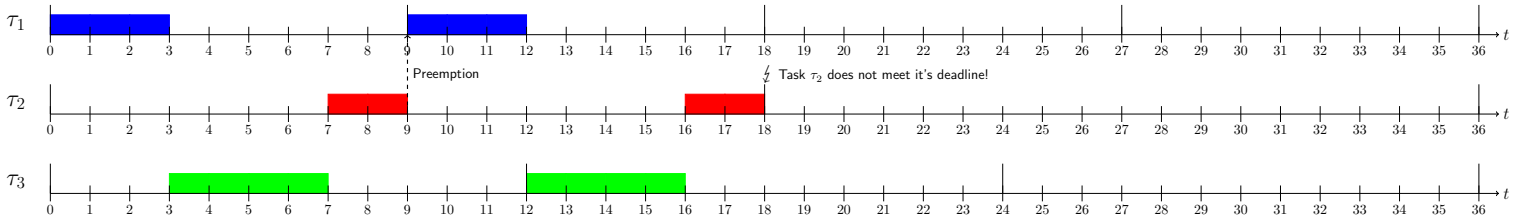
Second Example

- Task Set: $\tau_1(9,3)$, $\tau_2(18,5)$, $\tau_2(12,4)$
- Priorities based on length of period: $P_1 > P_3 > P_2$

Rate-Monotonic Scheduling

Second Example

- Task Set: $\tau_1(9, 3)$, $\tau_2(18, 5)$, $\tau_2(12, 4)$
- Priorities based on length of period: $P_1 > P_3 > P_2$



- τ_2 cannot meet its deadline at time 18 $\Rightarrow$ No feasible schedule

Earliest Deadline First (EDF) Algorithm

- Assigns job priorities based on deadlines
- Earlier deadline gives higher priority
- Dynamic priorities $\Rightarrow$ Different jobs in a task can have different priorities
- Requires additional knowledge of deadlines

Earliest Deadline First

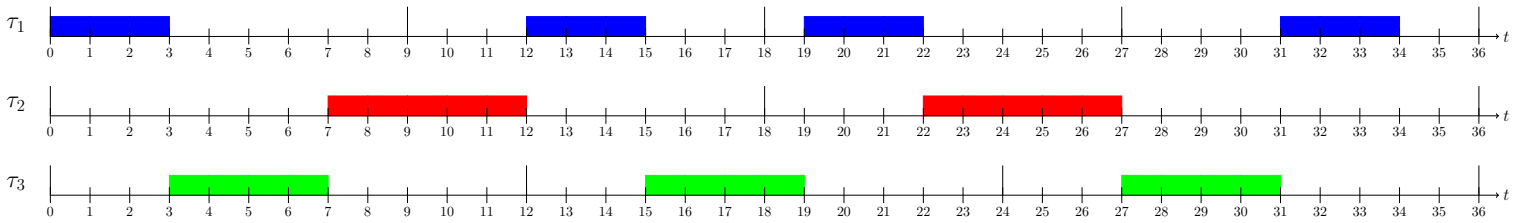
Example

- Task Set: $\tau_1(9,3)$, $\tau_2(18,5)$, $\tau_2(12,4)$
- Dynamic priorities based on the next deadline

Earliest Deadline First

Example

- Task Set: $\tau_1(9,3)$, $\tau_2(18,5)$, $\tau_2(12,4)$
- Dynamic priorities based on the next deadline



- All tasks meet their deadlines ⇒ Feasible schedule exists!

Optimality of EDF Algorithm

- The EDF algorithm is optimal under the following condition:
  - If preemption is allowed, the algorithm will always produce a feasible schedule of a set **J** of jobs with arbitrary release times and deadlines on one processor, if and only if **J** has a feasible schedule.
- EDF is not optimal, if jobs are non-preemptable or more than one processor is used

Fixed- vs. Dynamic-Priority Algorithms (1/2)

Fixed- and dynamic-priority scheduling algorithms have different properties.

- EDF outperforms RMA with respect to schedulable utilisation
- The behaviour of dynamic algorithms is much more difficult to predict
  - difficult to predict, which task misses the deadline due to overload
  - late job that misses may cause other jobs to be late
  - good management required for such unstable systems

Fixed- vs. Dynamic-Priority Algorithms (2/2)

Fixed-priority scheduling algorithms are easier to implement

- RM algorithm has an efficient implementation
  - scan processes and choose the highest-priority active process
  - supported by many real-time operating systems, light-weight implementation

EDF requires a more complex implementation

- on each timer interrupt
  - compute time to deadline for each process that is ready
  - choose process closest to deadline
  - expensive to use in practice for small RTOS

Outline

Schedulability Tests

- It is of critical importance that the correctness of a schedule can be validated
  - Simulating schedules is time consuming and error prone
  - Other faster methods are needed
- A schedulability test is a mechanism that proofs that all deadlines are met, when scheduling with a particular algorithm
- A schedulability test is useful also for acceptance of new on-line-jobs

Schedulable Utilisation

Definition: Schedulable Utilisation U_{ALG}

A scheduling algorithm can feasibly schedule any set of periodic tasks on a processor if the total utilisation of the tasks is equal to or less than the schedulable utilisation U_{ALG} of that algorithm!

Schedulable Utilisation

- The utilisation of a periodic task τ_i is defined as $u_i = \frac{C_i}{T_i}$.
- The total utilisation U of a set of tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ is $U = \sum_{i=1}^{n} u_i$
- An algorithm is optimal, if $U_{ALG} = 1$
- Schedulability tests can be developed based on U_{ALG}

Schedulable Utilisation

Given as Theorem 6.1 in (Liu, 2000)

A system of independent preemptable tasks with relative deadlines equal to its periods can be feasibly scheduled on one processor if and only if its total utilisation is equal or less than 1.

Schedulable Utilisation: EDF

- EDF is an optimal algorithms. Thus $U_{EDF} = 1$ for the systems, where $T_i = D_i$ for tasks.
- Also $U_{EDF} = 1$, if $D_i \geq T_i$.
- These results are independent of the phase ϕ_i.

If there is at least one task, where $D_i < T_i$, in a system with $U \leq 1$ there might not be a feasible schedule any longer. Test must be extended!

Schedulable Utilisation: EDF

Density

The schedulability test can be extended by looking at the density of a task.

- Density of task τ_i is defined as $\delta_i = \frac{C_i}{\min(D_i, T_i)}$
- Density of a system is $\Delta = \sum \delta_i$

Given as Theorem 6.2 in (Liu, 2000)

A system Γ of independent, preemptable tasks can feasibly be scheduled, if its density is equal or less to 1.

The theorem is a sufficient, but not necessary condition. The system may nevertheless be feasible when its density is greater than 1.

Schedulable Utilisation: RMA

Given as Theorem 6.11 in (Liu, 2000)

A system of n independent, preemptable periodic tasks with relative deadlines equal to their respective periods can be feasibly scheduled on a processor according to the rate-monotonic algorithm if its total utilisation U is less than or equal to $U_{RM}(n) = n(2^{\frac{1}{n}} - 1)$.

Schedulable Utilisation: RMA

n	U_{RM}
1	1.000
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
∞	$0.693 = ln2$

$U \leq U_{RM}(n)$ is a sufficient, but not necessary condition.

Optimality of RMA

- The RM algorithm cannot be optimal, since they assign fixed priorities. Not all systems with $U \leq 1$ can be scheduled by RM.
- But RM is optimal for simply periodic systems.

Simply Periodic Systems

A system of periodic tasks is simply periodic, if for every pair of tasks T_i and T_k in the system and $T_i < T_k$, T_k is an integer multiple of T_i.

Optimality of RMA

Given as Theorem 6.11 in (Liu, 2000)

A system of simply periodic, independent, preemptable tasks whose deadlines are equal or larger than their periods is schedulable on one processor according to the RM algorithm, if and only if its total utilisation is equal or less than 1

Example: $\Gamma = \{\tau_1(2,1), \tau_2(4,1), \tau_3(8,2)\}$ is schedulable according to the theorem.

Schedulable Utilisation for Rate-Monotonic Analysis

Given a set of n periodic tasks Γ that shall be scheduled with the rate-monotonic algorithm on a single processor,

- a feasible schedule exists, if
  - $U(\Gamma) \leq U_{\mathrm{RM}}(n)$, or
  - $U_{\mathrm{RM}}(n) \leq U(\Gamma) \leq 1$ when the task set Γ forms a simply periodic system;
- a feasible schedule does not exist if $U(\Gamma) > 1$;
- a feasible might exist, if $U_{\mathrm{RM}}(n) \leq U(\Gamma) \leq 1$ when the task set Γ does not form a simply periodic system.

Schedulable Utilisation for Rate-Monotonic Analysis

Given a set of n periodic tasks Γ that shall be scheduled with the rate-monotonic algorithm on a single processor,

- a feasible schedule exists, if
  - $U(\Gamma) \leq U_{\mathrm{RM}}(n)$, or
  - $U_{\mathrm{RM}}(n) \leq U(\Gamma) \leq 1$ when the task set Γ forms a simply periodic system;
- a feasible schedule does not exist if $U(\Gamma) > 1$;
- a feasible might exist, if $U_{\mathrm{RM}}(n) \leq U(\Gamma) \leq 1$ when the task set Γ does not form a simply periodic system.

For the last case, where $U_{\mathrm{RM}}(n) \leq U(\Gamma) \leq 1$ and where the task set Γ does not form a simply periodic system, either task simulation can be used to decide, if there is a feasible schedule, or the schedulability test must be extended.

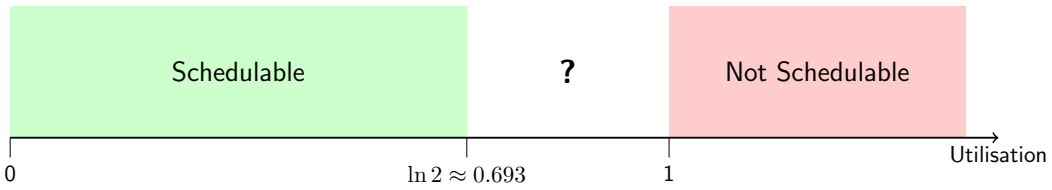
Outline

Need for Better Analysis Methods

We do not know, if a set of tasks Γ is schedulable, if $U_{RM} < U(\Gamma) \leq 1$.

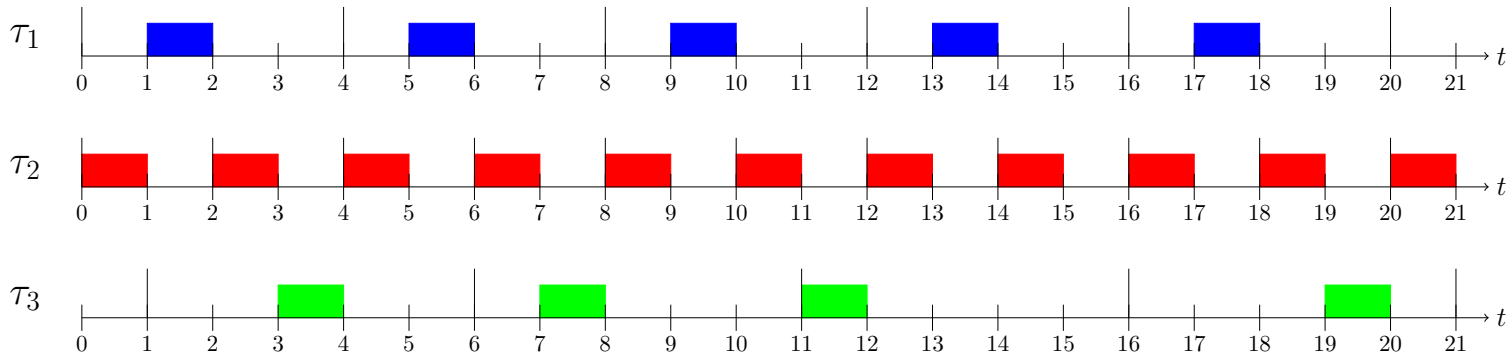
Possible solutions:

1. Simulate the schedule for the set of tasks (can be difficult)
2. Improve formal analysis, since fixed-priority algorithms are predictable
   - Identify the worst case
   - Analyse the worst case and draw conclusions



Worst Case Response Time for a Task

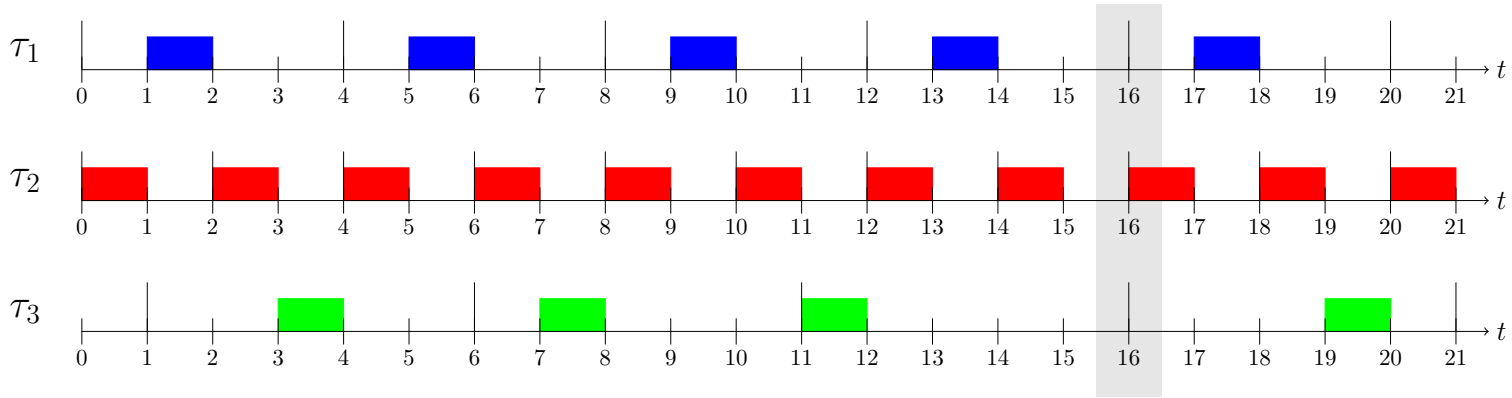
$\tau_1 = (4, 1), \tau_2 = (2, 1), \tau_3 = (1, 5, 1, 5)$



What is the worst case for task τ_3?

Worst Case Response Time for a Task

$\tau_1 = (4, 1), \tau_2 = (2, 1), \tau_3 = (1, 5, 1, 5)$



For τ_3 the worst case, which generates the longest response time is $t = 16$, where both tasks with higher priority τ_1 and τ_2 are released. This time instant is called a critical instant.

Critical Instant

Theorem (Liu 6.5)

In a fixed-priority system where every job completes before the next job in the same task is released, a critical instant of any task τ_i occurs when one of its job $\tau_{i,c}$ is released at the same time with a job in every higher-priority task, that is $a_{i,c} = a_{k,l_k}$ for some l_k for every $k = 1, 2, \ldots, i-1$.

Schedulability Test

1. Find critical instant when system is most loaded and has worst response time
2. Use time demand analysis to determine if the system is schedulable at the critical time instant
3. Prove that, if a fixed-priority system is schedulable at the critical instant, it is always schedulable (not done here)

Time Demand Analysis

Time demand analysis uses critical instants in order to conduct a schedulability test

1. Compute the total demand for processor time by a job released at a critical instant of the task and by each higher-priority tasks
2. Check then if the demand can be met before the deadline of the job

$$w_i(t) = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{T_k} \right\rceil C_k, \text{ for } 0 < t \leq T_i$$

Workload Analysis

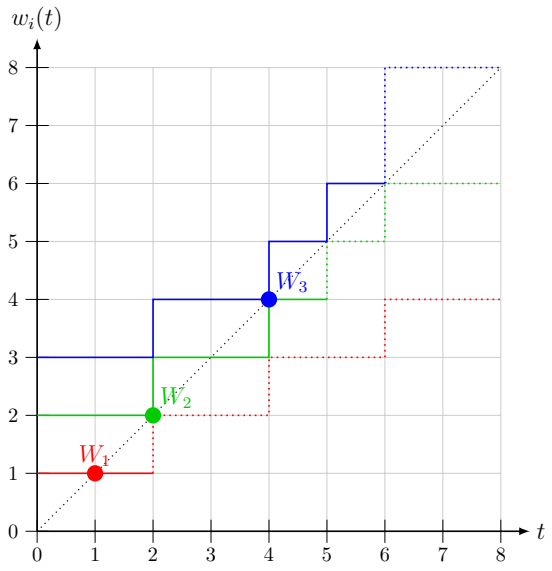
Another term for time demand analysis is workload analysis as used by Buttazzo (2011).

Time Demand Analysis

- Compare time demand $w_i(t)$ with available time t
  - If $w_i(t) \leq t$ for some $t \leq D_i$ where $D_i \leq T_i$, all jobs in τ_i can complete before its deadline
  - If $w_i(t) > t$ for all $0 \leq t \leq Di$ where $D_i \leq T_i$, this job in τ_i cannot complete by its deadline
  - ! This is valid under the assumption that the critical instant occurs, but this must not be the case (can be proved by simulation or analysis)
- Check time demand for all tasks
- If all tasks meet their time demand at their critical instant, the system is schedulable

Time Demand Analysis - RM

Set of tasks: $\tau_1 = (2, 1), \tau_2 = (5, 1), \tau_3 = (6, 1)$



- Worst Case Response Times at $W_1 = 1, W_2 = 2, W_3 = 4$
- All tasks meet their deadlines!

Response Time Analysis

- Consider tasks in descending order
- Focus on job $\tau_{i,j}$ in τ_i where we assume that the release time t_0 is a critical instant of τ_i
- The longest response time of a task τ_i is then comprised of C_i and the possible interference from higher priority tasks.

Response Time

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$$

- A task set is schedulable if $\forall \tau_i \in \Gamma$: $R_i \leq D_i$

Response Time Analysis

- No simple solution exists for the response time equation, as R_i appears on both sides.
- However, only points in the interval $[0, D_i]$ need to be checked.
- The value can be computed iteratively, where $R_i^{(k)}$ is the k^{th} estimate of R_i

$$R_i^{(k+1)} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot C_j$$

- The computation stops if $R_i > D_i$ or $R_i^{(k)} = R_i^{(k+1)}$

$$R_i \leq D_i$$

Response Time Analysis - RM

Set of tasks: $\tau_1 = (2, 1), \tau_2 = (5, 1), \tau_3 = (6, 1)$

- Response Time τ_1:
  - $R_1^{(0)} = C_1 = 1$
- Response Time τ_2:
  - $R_2^{(0)} = C_2 = 1$
  - $R_2^{(1)} = C_2 + \left\lceil \frac{R_2^{(0)}}{T_2} \right\rceil \cdot C_1 = 1 + \lceil \frac{1}{5} \rceil \cdot 1 = 2$
  - $R_2^{(2)} = C_2 + \left\lceil \frac{R_2^{(1)}}{T_2} \right\rceil \cdot C_1 = 1 + \lceil \frac{2}{5} \rceil \cdot 1 = 2$
- Response Time τ_3:
  - ...

Response Time Analysis

- Schedulability test based on the time demand analysis or response time analysis is more complex than the schedulability test based on utilisation, but is more general
- Can be extended to cover other cases

Alternative Approach

- Simulate behaviour of tasks at their critical instants up to the largest period of the tasks
- ! Gives same results as time demand analysis

Practical Factors

So far we have assumed

- every job is preemptable at any time
- released jobs never suspend themselves
- scheduling and context switch overhead is negligible
- scheduler is event-driven and acts immediately upon event occurrences
- tasks have distinct priorities
- priorities in a fixed-priority system never change

In practice these assumptions are often not valid! New schedulability tests based on time demand and response time analysis can be used to take practical factors into account.

Practical Factors: Blocking Time

An additional term for the blocking time B_i can be used in the time demand or response time analysis function. The term B_i reflects the blocking time, which a high priority job can experience when a low priority job executes a non-preemptable section, for instance a critical section protected by a semaphore.

- Time Demand Analysis: $w_i(t) = B_i + C_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{T_k} \right\rceil C_k, for 0 < t \leq T_i$
- Response Time Analysis: $R_i = B_i + C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j$

Outline

Resource Access Protocols

- Previous discussion assumed independent tasks
- In practice
  - tasks communicate via communication objects like semaphores, message queues or protected objects with each other
  - $\Rightarrow$ tasks are not independent

Resources

Definition: Resource (Buttazzo, 2011)

- A resource S is any software structure that can be used by a task to advance its execution[a].
- A resource
  - that is dedicated to a particular task is called a private resource;
  - that can be used by more tasks is called a shared resource;
  - that is protected against concurrent access is called a exclusive resource.

[a] The symbol S is used, because the symbol R is already occupied to denote the response time. Also, Buttazzo (2011) uses the semaphore as example for a resource.

- Resources can have several units, e.g. counting semaphore
- Following discussion only deals with exclusive resources with one unit
- Exclusive resources are allocated to tasks on a non-preemptive basis and used in a mutually exclusive manner

Mutually Exclusive Resource Access

Assumptions

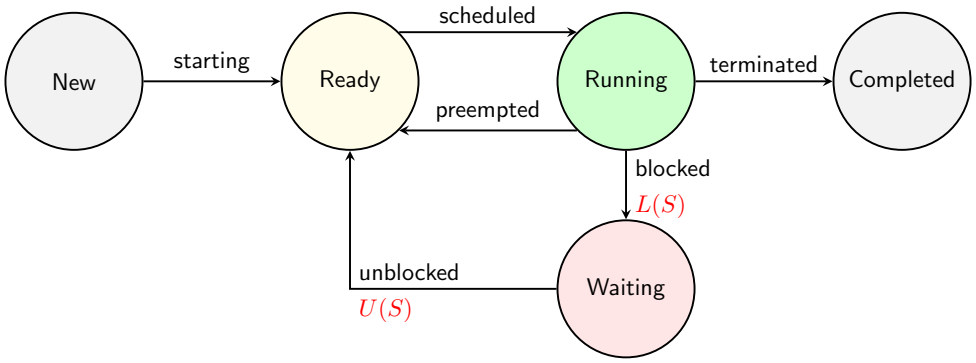
- A lock-based concurrency control mechanism assumed to be used to enforce mutual exclusive access to resources
- When a task wants to use a resource S_i, it executes a command `lock` $L(S_i)$ to request the resource.
- When a task no longer needs the resource S_i, it releases the resource by executing a command unlock signal $U(S_i)$.

Correspondence to critical sections protected by a semaphore

A critical section protected by a semaphore `S` is modelled as an exclusive resource S

- The command `wait(S)` is modelled by a `lock` command $L(S)$
- The command `signal(S)` is modelled by an `unlock` command $U(S)$

Mutually Exclusive Resource Access



- When a lock request $L(S)$ fails, the requesting job is blocked and loses the processor
- An unlock command $U(S)$ frees the resource and might cause that another task is unblocked
- A task stays blocked until the scheduler grants the resources the task is waiting for

Critical Section

- A segment of a task tau_i that begins with a lock $L(S_k)$ and ends with a matching unlock $U(S_k)$ is called a critical section, and is denoted by $z_{i,k}$.
- The longest critical section of τ_i to a resource S_k is denoted by $Z_{i,k}$.
- The duration of $Z_{i,k}$ is denoted by $\delta_{i,k}$

```
-- Critical Section
lock(S_k)
-- Access to exclusive resource S_k
unlock(S_k)
```

Nested Critical Sections

- In nested critical sections, resources are always released in last-in-first-out order ⇒ properly nested critical sections
- $z_{i,h} \subset z_{i,k}$ indicates that the critical section $z_{i,h}$ is entirely contained in $z_{i,k}$.
- A critical section that is not included in other critical sections is called an outermost critical section
- Assumption: All nested critical sections are properly nested

Properly Nested Critical Section

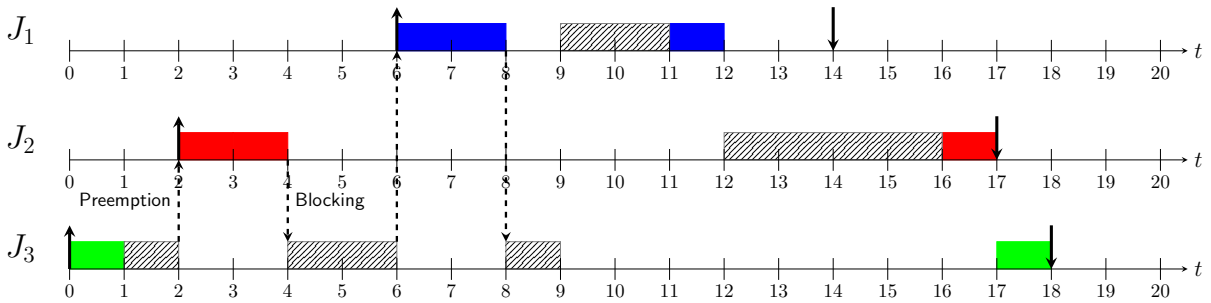
```
lock(S_1)
...
lock(S_2)
..
unlock(S_2)
...
unlock(S_1)
```

Not Properly Nested Critical Section

```
lock(S_2)
...
lock(S_1)
..
unlock(S_2)
...
unlock(S_1)
```

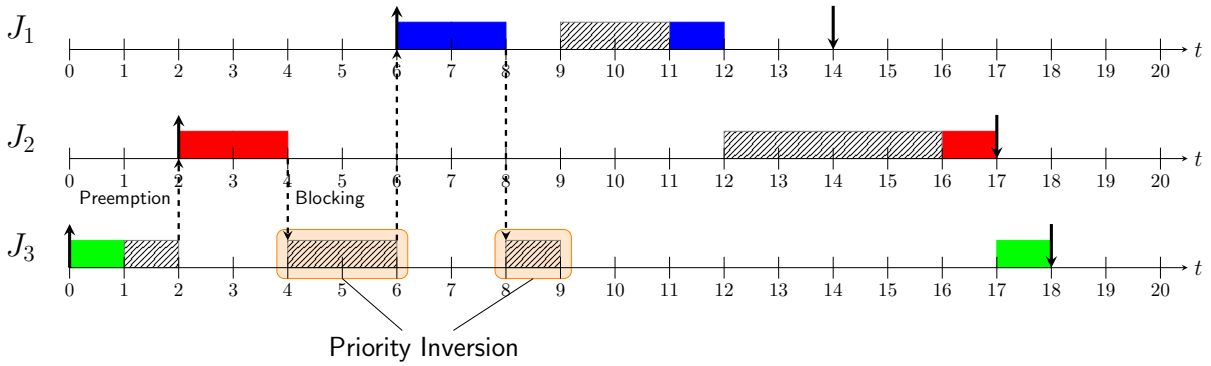

Priority Inversion due to Resource Conflicts

- Jobs J_1, J_2, J_3 have feasible intervals (6,14], (2,17], (0,18].
- Jobs J_1, J_2, J_3 are scheduled using EDF algorithm
- Jobs access critical sections with exclusive resource S_k for the following time units: $\delta_{1,k} = 2$, $\delta_{2,k} = 4$, $\delta_{3,k} = 4$.



Priority Inversion due to Resource Conflicts

- Priority inversion occurs, when a low-priority job executes while a ready higher-priority job waits

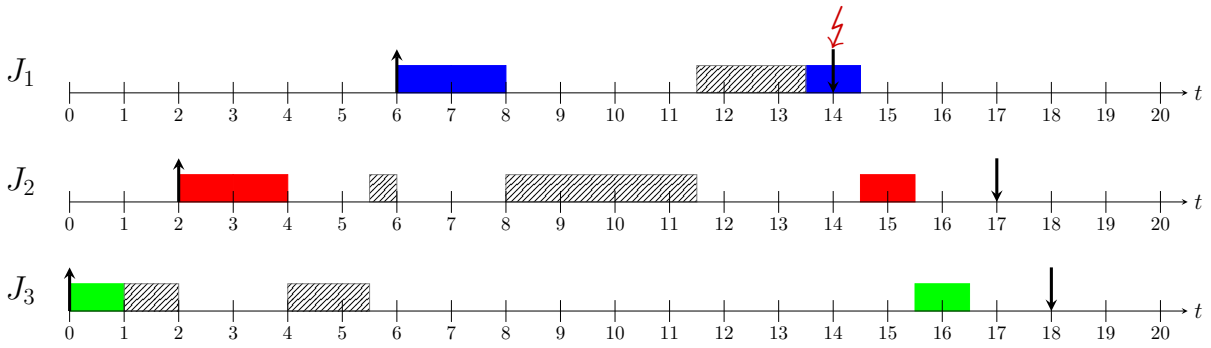


Timing Anomalies

- Timing Anomalies can occur due to priority inversion
- Assume that critical section of job J_3 is reduced to $\delta_{3,k} = 2.5$ instead of $\delta_{3,k} = 4$.

Timing Anomalies

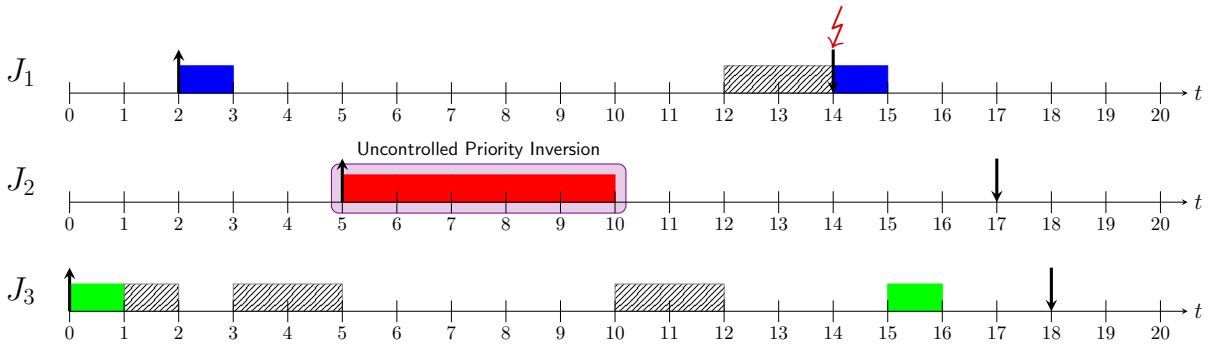
- Timing Anomalies can occur due to priority inversion
- Assume that critical section of job J_3 is reduced to $\delta_{3,k} = 2.5$ instead of $\delta_{3,k} = 4$.



Job J_1 misses its deadline!

Uncontrolled Priority Inversion

- Without suitable protocols, the duration of priority inversion can become unbounded.
- A job J_1 who wants to execute its critical section $z_{1,k}$ can be blocked by a job J_n that holds the resource S_k.
- The duration of of $\delta_{i,k}$ can be prolonged by execution of several jobs J_j with a lower priority than the blocked job and a higher priority than the blocking job.
- This phenomenon is called uncontrolled priority inversion.

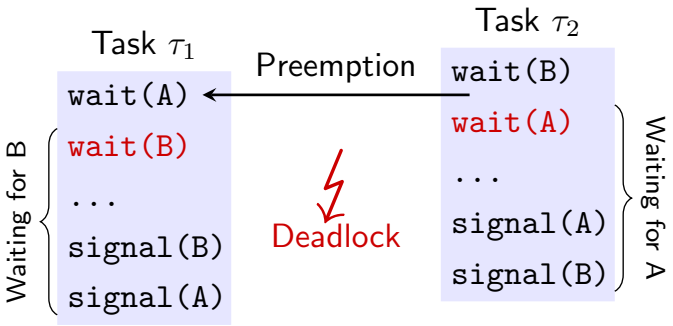


Protocols for Resource Access Control

- Protocols are needed to handle priority inversion in a controlled way and to avoid deadlock

Deadlock

- Deadlock can occur, if tasks block each other from execution



Disciplined approach required to avoid deadlocks

- If a deadlock occurs the program cannot proceed!
- Difficult to test for deadlock, because deadlock situation is difficult to create.

Do you want to know more about the topic?

Do you want to know more about the topic?

Interested in Real-Time Systems?
IL2212 Embedded Software continues the discussion!

References

Mordechai Ben-Ari. Principles of Concurrent and Distributed Programming. Addison-Wesley, 2006.

Giorgio C. Buttazzo. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer, 3rd edition, 2011.

C.L. Liu and James. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM, 20(1):46–71, January 1973.

Jane W. S. Liu. Real-Time Systems. Prentice Hall, 2000.