IA158 Real Time Systems

Tomáš Brázdil

Organization of This Course

Sources:

- Lectures (slides, notes)
  - based on several sources (hard to obtain)
  - slides are prepared for lectures, lots of stuff on the greenboard
    ($\Rightarrow$ attend the lectures)

Organization of This Course

Sources:

- Lectures (slides, notes)
  - based on several sources (hard to obtain)
  - slides are prepared for lectures, lots of stuff on the greenboard
    ($\Rightarrow$ attend the lectures)

Homework:

- a larger homework project

Organization of This Course

Sources:

- Lectures (slides, notes)
  - based on several sources (hard to obtain)
  - slides are prepared for lectures, lots of stuff on the greenboard
    ($\Rightarrow$ attend the lectures)

Homework:

- a larger homework project

Evaluation:

- Homework project
  (have to do to be allowed to the exam)
- Oral exam

Definition 1 (Time)

Mirriam-Webster: Time is the measured or measurable period during which an action, process, or condition exists or continues.

Real-Time Systems

Definition 1 (Time)

Mirriam-Webster: Time is the measured or measurable period during which an action, process, or condition exists or continues.

Definition 2 (Real-time)

Real-time is a quantitative notion of time measured using a physical clock.

Example: After an event occurs (eg. temperature exceeds 500 degrees) the corresponding action (cooling) must take place within 100ms.

Compare with qualitative notion of time (before, after, eventually, etc.)

Real-Time Systems

Definition 1 (Time)

Mirriam-Webster: Time is the measured or measurable period during which an action, process, or condition exists or continues.

Definition 2 (Real-time)

Real-time is a quantitative notion of time measured using a physical clock.

Example: After an event occurs (eg. temperature exceeds 500 degrees) the corresponding action (cooling) must take place within 100ms.

Compare with qualitative notion of time (before, after, eventually, etc.)

Definition 3 (Real-time system)

A *real-time system* must deliver services in a timely manner.

Not necessarily fast, must satisfy some *quantitative* timing constraints

Real-time Embedded Systems

Definition 4 (Embedded system)

An *embedded system* is a computer system designed for specific control functions within a larger system, usually consisting of electronic as well as mechanical parts.

Real-time Embedded Systems

Definition 4 (Embedded system)

An *embedded system* is a computer system designed for specific control functions within a larger system, usually consisting of electronic as well as mechanical parts.

Most (not all) real-time systems are embedded

Most (not all) embedded systems are real-time

(Few) Examples of Real-time Embedded Systems

- Industrial
  - chemical plant control
  - automated assembly line (e.g. robotic assembly, inspection)

(Few) Examples of Real-time Embedded Systems

- Industrial
  - chemical plant control
  - automated assembly line (e.g. robotic assembly, inspection)
- Medical
  - pacemaker,
  - medical monitoring devices

(Few) Examples of Real-time Embedded Systems

- Industrial
  - chemical plant control
  - automated assembly line (e.g. robotic assembly, inspection)
- Medical
  - pacemaker,
  - medical monitoring devices
- Transportation systems
  - computers in cars (ABS, MPFI, cruise control, airbag ...)
  - aircraft (FMS, fly-by-wire ...)

(Few) Examples of Real-time Embedded Systems

- Industrial
  - chemical plant control
  - automated assembly line (e.g. robotic assembly, inspection)
- Medical
  - pacemaker,
  - medical monitoring devices
- Transportation systems
  - computers in cars (ABS, MPFI, cruise control, airbag ...)
  - aircraft (FMS, fly-by-wire ...)
- Military applications
  - controllers in weapons, missiles, ...
  - radar and sonar tracking

(Few) Examples of Real-time Embedded Systems

- Industrial
  - chemical plant control
  - automated assembly line (e.g. robotic assembly, inspection)
- Medical
  - pacemaker,
  - medical monitoring devices
- Transportation systems
  - computers in cars (ABS, MPFI, cruise control, airbag ...)
  - aircraft (FMS, fly-by-wire ...)
- Military applications
  - controllers in weapons, missiles, ...
  - radar and sonar tracking
- Multimedia – multimedia center, videoconferencing
- ...

(Non-)Real-time (non-)embedded systems

There are real time systems that are not embedded:

- trading systems
- ticket reservation
- multimedia (on PC)
- ...

(Non-)Real-time (non-)embedded systems

There are real time systems that are not embedded:

- trading systems
- ticket reservation
- multimedia (on PC)
- ...

There are embedded systems that are (possibly) not real-time

e.g. a weather station sends data once a day without any deadline – not really real-time system

Caveat: Aren't all systems real-time in a sense?

Characteristics of Real-Time Embedded Systems

Real-time systems often are

- safety critical
  - Serious consequences may result if services are not delivered on timely basis
  - Bugs in embedded real-time systems are often difficult to fix

... need to validate their correctness

Characteristics of Real-Time Embedded Systems

Real-time systems often are

- safety critical
  - Serious consequences may result if services are not delivered on timely basis
  - Bugs in embedded real-time systems are often difficult to fix

  ... need to validate their correctness

- concurrent
  - Real-world devices operate in parallel – better to model this parallelism by concurrent tasks in the program

  ... validation may be difficult, formal methods often needed

Characteristics of Real-Time Embedded Systems

Real-time systems often are

- safety critical
  - Serious consequences may result if services are not delivered on timely basis
  - Bugs in embedded real-time systems are often difficult to fix

  ... need to validate their correctness

- concurrent
  - Real-world devices operate in parallel – better to model this parallelism by concurrent tasks in the program

  ... validation may be difficult, formal methods often needed

- reactive
  - Interact continuously with their environment (as opposed to information processing systems)

  ... “traditional” validation methods do not apply

Validating Time Requirements and Predictability

- Given real-time requirements and an implementation on HW and SW, how to show that the requirements are met?

Validating Time Requirements and Predictability

- Given real-time requirements and an implementation on HW and SW, how to show that the requirements are met?

  ... testing might not suffice:

  Maiden flight of space shuttle, 12 April 1981: 1/67 probability that a transient overload occurs during initialization; and it actually did!

Validating Time Requirements and Predictability

- Given real-time requirements and an implementation on HW and SW, how to show that the requirements are met?

  ... testing might not suffice:

  Maiden flight of space shuttle, 12 April 1981: 1/67 probability that a transient overload occurs during initialization; and it actually did!

- We need a formal model and validation ...

Validating Time Requirements and Predictability

- Given real-time requirements and an implementation on HW and SW, how to show that the requirements are met?

  ... testing might not suffice:

  Maiden flight of space shuttle, 12 April 1981: 1/67 probability that a transient overload occurs during initialization; and it actually did!

- We need a formal model and validation ...
- ... we need **predictable** behavior!
  It is difficult to obtain
  - caches, DMA, unmaskable interrupts
  - memory management
  - scheduling anomalies
  - difficult to compute worst-case execution time
  - ...

Types of Timing Requirements

Time sharing systems: minimize average response time

The goal of scheduling in standard op. systems such as Linux and Windows

Types of Timing Requirements

Time sharing systems: minimize average response time
The goal of scheduling in standard op. systems such as Linux and Windows

Often it is **not** enough to minimize average response time!
(A man drowned crossing a stream with an average depth of 15cm.)

Types of Timing Requirements

Time sharing systems: minimize average response time
The goal of scheduling in standard op. systems such as Linux and Windows

Often it is **not** enough to minimize average response time!
(A man drowned crossing a stream with an average depth of 15cm.)

“hard” real-time tasks must be **always** finished before their deadline!

e.g. airbag in a car: whenever a collision is detected, the airbag must be deployed within 10ms

Types of Timing Requirements

Time sharing systems: minimize average response time

The goal of scheduling in standard op. systems such as Linux and Windows

Often it is **not** enough to minimize average response time!

(A man drowned crossing a stream with an average depth of 15cm.)

"hard" real-time tasks must be **always** finished before their deadline!

e.g. airbag in a car: whenever a collision is detected, the airbag must be deployed within 10ms

Not all tasks in a real-time system are critical, only the quality of service is affected by missing a deadline

Types of Timing Requirements

Time sharing systems: minimize average response time

The goal of scheduling in standard op. systems such as Linux and Windows

Often it is **not** enough to minimize average response time!

(A man drowned crossing a stream with an average depth of 15cm.)

“hard” real-time tasks must be **always** finished before their deadline!

e.g. airbag in a car: whenever a collision is detected, the airbag must be deployed within 10ms

Not all tasks in a real-time system are critical, only the quality of service is affected by missing a deadline

Most “soft” real-time tasks should finish before their deadlines.

e.g. frame rate in a videoconf. should be kept above 15fps most of the time

Types of Timing Requirements

Time sharing systems: minimize average response time

The goal of scheduling in standard op. systems such as Linux and Windows

Often it is **not** enough to minimize average response time!

(A man drowned crossing a stream with an average depth of 15cm.)

“hard” real-time tasks must be **always** finished before their deadline!

e.g. airbag in a car: whenever a collision is detected, the airbag must be deployed within 10ms

Not all tasks in a real-time system are critical, only the quality of service is affected by missing a deadline

Most “soft” real-time tasks should finish before their deadlines.

e.g. frame rate in a videoconf. should be kept above 15fps most of the time

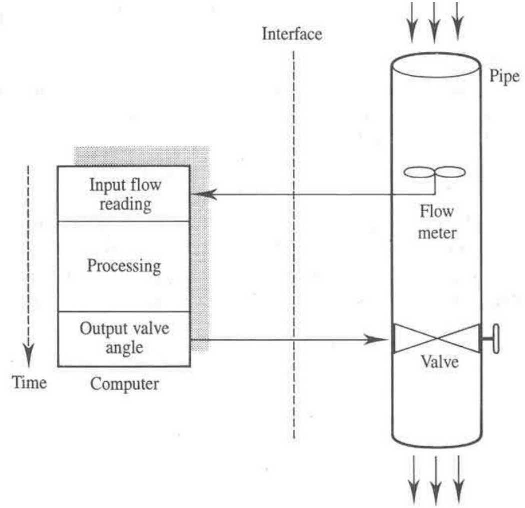
Many real-time systems combine “hard” and “soft” real-time tasks.

i.e. we optimize performance w.r.t. “soft” real-time tasks under the constraint that “hard” real-time tasks are finished before their deadlines

Examples of Real-Time Systems

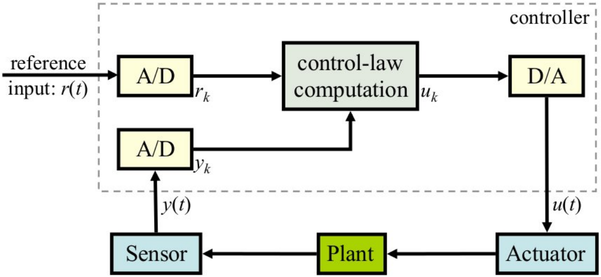
- Digital process control
  - anti-lock braking system
- Higher-level command and control
  - helicopter flight control
- Real-time databases
  - Stock trading systems

Digital Process Control



Computer controls the flow in the pipe in real-time

Digital Process Control



The controller (computer) controls the plant using the actuator (valve) based on sampled data from the sensor (flow meter)

- $y(t)$ – the measured state of the plant
- $r(t)$ – the desired state of the plant
- Calculate control output $u(t)$ as a function of $y(t), r(t)$
  e.g. $u_k = u_{k-2} + \alpha(r_k - y_k) + \beta(r_{k-1} - y_{k-1}) + \gamma(r_{k-2} - y_{k-2})$
  where α, β, γ are suitable constants

Digital Process Control

- Pseudo-code for the controller:

```
set timer to interrupt periodically with period T
foreach timer interrupt do
    analogue-to-digital conversion of y(t) to get y_k
    compute control output u_k based on r_k and y_k
    digital-to-analogue conversion of u_k to get u(t)
end
```

Digital Process Control

- Pseudo-code for the controller:

```
set timer to interrupt periodically with period T
foreach timer interrupt do
    analogue-to-digital conversion of y(t) to get y_k
    compute control output u_k based on r_k and y_k
    digital-to-analogue conversion of u_k to get u(t)
end
```

- Effective control of the plant depends on:
  - The correct reference input and control law computation
  - The accuracy of the sensor measurements
    - Resolution of the sampled data (i.e. bits per sample)
    - Frequency of interrupts (i.e. $1/T$)

Digital Process Control

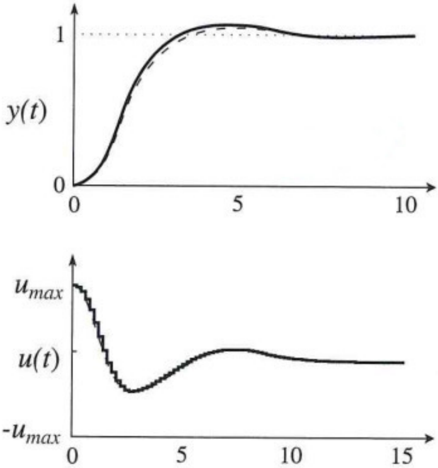
- Pseudo-code for the controller:

```
set timer to interrupt periodically with period T
foreach timer interrupt do
    analogue-to-digital conversion of y(t) to get y_k
    compute control output u_k based on r_k and y_k
    digital-to-analogue conversion of u_k to get u(t)
end
```

- Effective control of the plant depends on:
  - The correct reference input and control law computation
  - The accuracy of the sensor measurements
    - Resolution of the sampled data (i.e. bits per sample)
    - Frequency of interrupts (i.e. $1/T$)
- T is the *sampling period*
  - Small T better approximates the analogue behavior
  - Large T means less processor-time demand

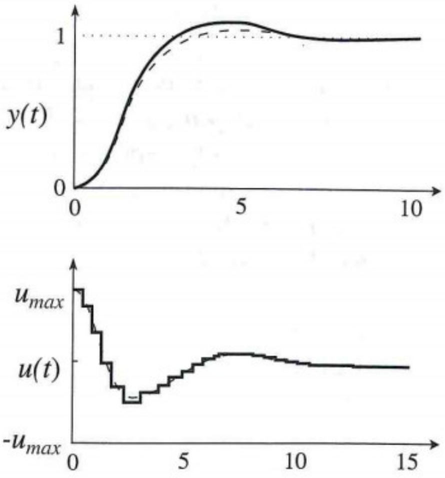
    ... but may result in unstable control

Example



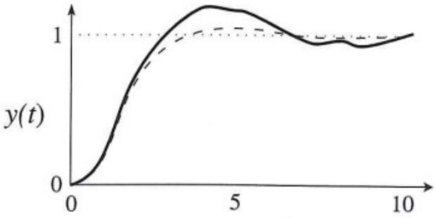
$r(t) = 1$ for $t \geq 0$

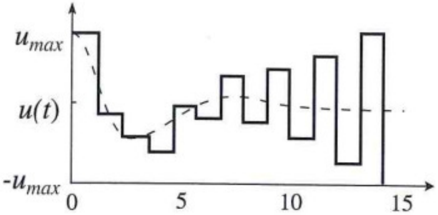
Example



$r(t) = 1$ for $t \geq 0$

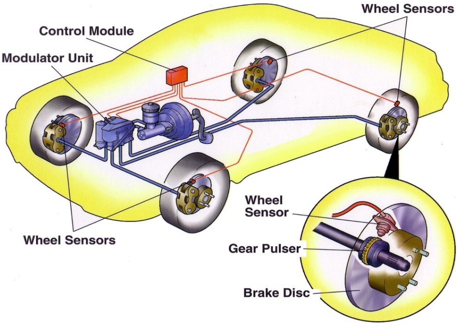
Example





$r(t) = 1$ for $t \geq 0$

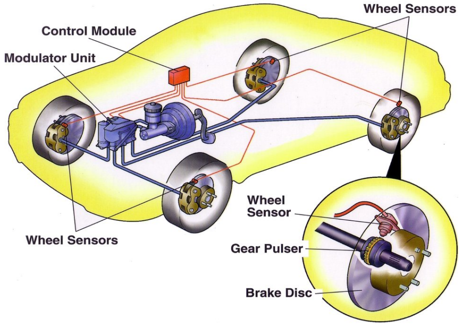
Anti-Lock Braking System



- The controller monitors the speed sensors in wheels

  Right before a wheel locks up, it experiences a rapid deceleration

Anti-Lock Braking System

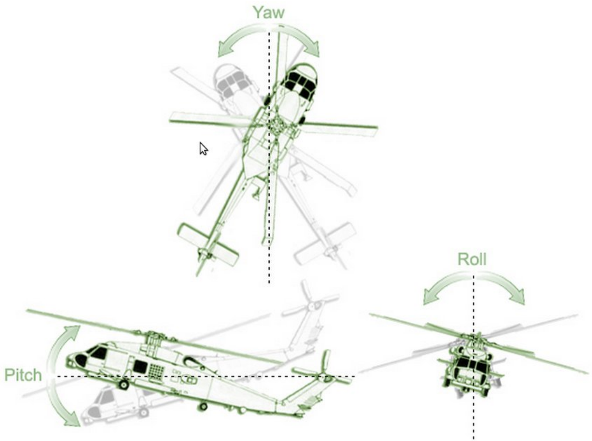


- The controller monitors the speed sensors in wheels

  Right before a wheel locks up, it experiences a rapid deceleration

- If a rapid deceleration of a wheel is observed, the controller alternately
  - reduces pressure on the corresponding brake until acceleration is observed
  - then applies brake until deceleration is observed

Multi-Rate DPC – Helicopter Flight Control



There are also three velocity components

Two control loops: pilot's control (30Hz) and stabilization (90Hz)

Multi-Rate DPC – Helicopter Flight Control

Do the following in each 1/180-second cycle:

- Validate sensor data; in the presence of failures, reconfigure the system

Multi-Rate DPC – Helicopter Flight Control

Do the following in each 1/180-second cycle:

- Validate sensor data; in the presence of failures, reconfigure the system
- Do the following 30-Hz avionics tasks, each one every six cycles:
  - keyboard input and mode selection
  - data normalization and coordinate transformation
  - tracking reference update

Multi-Rate DPC – Helicopter Flight Control

Do the following in each 1/180-second cycle:

- Validate sensor data; in the presence of failures, reconfigure the system
- Do the following 30-Hz avionics tasks, each one every six cycles:
  - keyboard input and mode selection
  - data normalization and coordinate transformation
  - tracking reference update
- Do the following 30-Hz avionics tasks, each one every six cycles:
  - control laws of the outer pitch-control loop
  - control laws of the outer roll-control loop
  - control laws of the outer yaw- and collective-control loop

Multi-Rate DPC – Helicopter Flight Control

Do the following in each 1/180-second cycle:

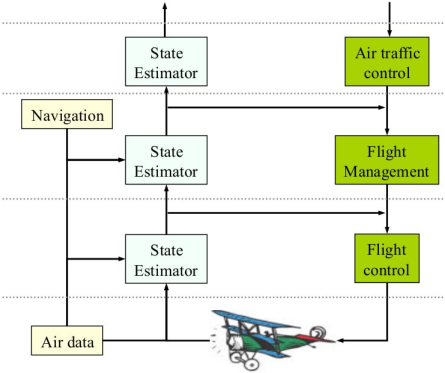
- Validate sensor data; in the presence of failures, reconfigure the system
- Do the following 30-Hz avionics tasks, each one every six cycles:
  - keyboard input and mode selection
  - data normalization and coordinate transformation
  - tracking reference update
- Do the following 30-Hz avionics tasks, each one every six cycles:
  - control laws of the outer pitch-control loop
  - control laws of the outer roll-control loop
  - control laws of the outer yaw- and collective-control loop
- Do each of the following 90-Hz computations once every two cycles, using outputs produced by 30-Hz computations and avionics tasks:
  - control laws of the inner pitch-control loop
  - control laws of the inner roll- and collective-control loop
- Compute the control laws of the inner yaw-control loop, using outputs produced by 90-Hz control-law computations as inputs

Multi-Rate DPC – Helicopter Flight Control

Do the following in each 1/180-second cycle:

- Validate sensor data; in the presence of failures, reconfigure the system
- Do the following 30-Hz avionics tasks, each one every six cycles:
  - keyboard input and mode selection
  - data normalization and coordinate transformation
  - tracking reference update
- Do the following 30-Hz avionics tasks, each one every six cycles:
  - control laws of the outer pitch-control loop
  - control laws of the outer roll-control loop
  - control laws of the outer yaw- and collective-control loop
- Do each of the following 90-Hz computations once every two cycles, using outputs produced by 30-Hz computations and avionics tasks:
  - control laws of the inner pitch-control loop
  - control laws of the inner roll- and collective-control loop
- Compute the control laws of the inner yaw-control loop, using outputs produced by 90-Hz control-law computations as inputs
- Output commands
- Carry out built-in-test
- Wait until the beginning of the next cycle

Higher-Level Command and Control



Controllers organized into a hierarchy

- At the lowest level we place the digital control systems that operate on the physical environment
- Higher level controllers monitor the behavior of lower levels
- Time-scale and complexity of decision making increases as one goes up the hierarchy (from control to planning)

Real-Time Database System

- Databases that contain perishable data, i.e. relevance of data deteriorates with time

  Air traffic control, stock price quotation systems, tracking systems, etc.

Real-Time Database System

- Databases that contain perishable data, i.e. relevance of data deteriorates with time

  Air traffic control, stock price quotation systems, tracking systems, etc.
- The temporal quality of data is quantified by *age of an image object*, i.e. the length of time since last update

Real-Time Database System

- Databases that contain perishable data, i.e. relevance of data deteriorates with time

  Air traffic control, stock price quotation systems, tracking systems, etc.
- The temporal quality of data is quantified by *age of an image object*, i.e. the length of time since last update
- temporal consistency
  - absolute = max. age is bounded by a fixed threshold
  - relative = max. difference in ages is bounded by a threshold

    e.g. planning system correlating traffic density and flow of vehicles

Applications	Size	Ave. Resp. Time	Max Resp. Time	Abs. Cons.	Rel. Cons.
Air traffic control	20,000	0.50 ms	5.00 ms	3.00 sec.	6.00 sec.
Aircraft mission	3,000	0.05 ms	1.00 ms	0.05 sec.	0.20 sec.
Spacecraft control	5,000	0.05 ms	1.00 ms	0.20 sec.	1.00 sec.
Process control		0.80 ms	5.00 sec	1.00 sec.	2.00 sec

- Users of database compete for access – various models for trading consistency with time demands exist.

Stock-Trading System

- A system for selling/buying stock at public prices

Stock-Trading System

- A system for selling/buying stock at public prices
- Prices are volatile in their movement

Stock-Trading System

- A system for selling/buying stock at public prices
- Prices are volatile in their movement
- Stop orders:
  - set upper limit on prices for buying – buy for the best available price once the limit is reached

    e.g. stock currently trading at $30 should be bought when the price rises above $35

Stock-Trading System

- A system for selling/buying stock at public prices
- Prices are volatile in their movement
- Stop orders:
  - set upper limit on prices for buying – buy for the best available price once the limit is reached

    e.g. stock currently trading at $30 should be bought when the price rises above $35
  - set lower limit on prices for selling – sell for the best available price once the limit is reached

    e.g. stock currently trading at $30 should be sold when the price sinks below $25

Stock-Trading System

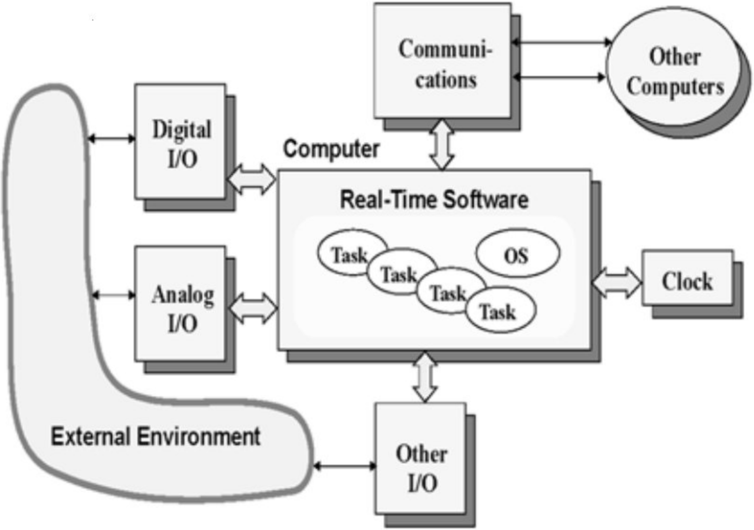
- A system for selling/buying stock at public prices
- Prices are volatile in their movement
- Stop orders:
  - set upper limit on prices for buying – buy for the best available price once the limit is reached

    e.g. stock currently trading at $30 should be bought when the price rises above $35
  - set lower limit on prices for selling – sell for the best available price once the limit is reached

    e.g. stock currently trading at $30 should be sold when the price sinks below $25
- Depending on the delay, the available price may be different from the limit

  successful stop orders depend on the timely delivery of stock trade data and the ability to trade on the changing prices in a timely manner

Structure of Real-Time (Embedded) Applications



Types of Real-Time Systems

- Purely cyclic
  - every task executes periodically; I/O operations are polled; demands in resources do not vary

  e.g. digital controllers

Types of Real-Time Systems

- Purely cyclic
  - every task executes periodically; I/O operations are polled; demands in resources do not vary

  e.g. digital controllers
- Mostly cyclic
  - most tasks execute periodically; system also responds to external events (fault recovery and external commands) asynchronously

  e.g. avionics

Types of Real-Time Systems

- Purely cyclic
  - every task executes periodically; I/O operations are polled; demands in resources do not vary

  e.g. digital controllers
- Mostly cyclic
  - most tasks execute periodically; system also responds to external events (fault recovery and external commands) asynchronously

  e.g. avionics
- Asynchronous and somewhat predictable
  - durations between consecutive executions of a task as well as demands in resources may vary considerably. These variations have either bounded range, or known statistics.

  e.g. radar signal processing, tracking

Types of Real-Time Systems

- The type of application affects how we schedule tasks and prove correctness
- It is easier to reason about applications that are more cyclic, synchronous and predictable
  - Many real-time systems are designed in this manner
  - Safe, conservative, design approach, if it works

Real-Time Systems Failures

- AT&T *long* distance calls
- Therac-25 medical accelerator disaster
- Patriot missile mistiming

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA

Handling up to 700,000 calls an hour

The problem:

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA

Handling up to 700,000 calls an hour

The problem:

- the switch in New York City neared its load limit

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA
Handling up to 700,000 calls an hour

The problem:

- the switch in New York City neared its load limit
- entered a four-second maintenance reset

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA

Handling up to 700,000 calls an hour

The problem:

- the switch in New York City neared its load limit
- entered a four-second maintenance reset
- sent “do not disturb” to neighbors

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA

Handling up to 700,000 calls an hour

The problem:

- the switch in New York City neared its load limit
- entered a four-second maintenance reset
- sent "do not disturb" to neighbors
- after the reset, the switch began to distribute calls (quickly)

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA
Handling up to 700,000 calls an hour

The problem:

- the switch in New York City neared its load limit
- entered a four-second maintenance reset
- sent “do not disturb” to neighbors
- after the reset, the switch began to distribute calls (quickly)
- then another switch received one of these calls from New York

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA

Handling up to 700,000 calls an hour

The problem:

- the switch in New York City neared its load limit
- entered a four-second maintenance reset
- sent “do not disturb” to neighbors
- after the reset, the switch began to distribute calls (quickly)
- then another switch received one of these calls from New York
- began to update its records that New York was back on line

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA
Handling up to 700,000 calls an hour

The problem:

- the switch in New York City neared its load limit
- entered a four-second maintenance reset
- sent “do not disturb” to neighbors
- after the reset, the switch began to distribute calls (quickly)
- then another switch received one of these calls from New York
- began to update its records that New York was back on line
- a second call from New York arrived less than 10 milliseconds after the first, i.e. while the first hadn’t yet been handled;
  this together with a SW bug caused maintenance reset

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA
Handling up to 700,000 calls an hour

The problem:

- the switch in New York City neared its load limit
- entered a four-second maintenance reset
- sent “do not disturb” to neighbors
- after the reset, the switch began to distribute calls (quickly)
- then another switch received one of these calls from New York
- began to update its records that New York was back on line
- a second call from New York arrived less than 10 milliseconds after the first, i.e. while the first hadn’t yet been handled;
  this together with a SW bug caused maintenance reset
- the error was propagated further ....

AT&T Long Distance Calls

114 computer-operated electronic switches scattered across USA
Handling up to 700,000 calls an hour

The problem:

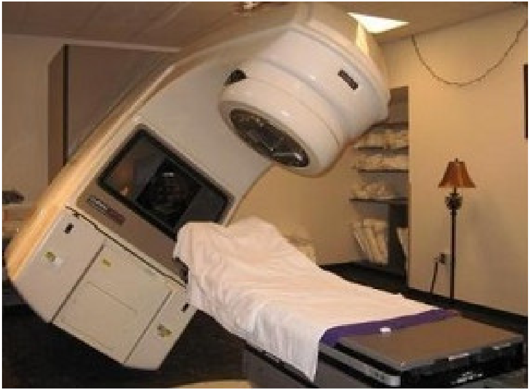
- the switch in New York City neared its load limit
- entered a four-second maintenance reset
- sent “do not disturb” to neighbors
- after the reset, the switch began to distribute calls (quickly)
- then another switch received one of these calls from New York
- began to update its records that New York was back on line
- a second call from New York arrived less than 10 milliseconds after the first, i.e. while the first hadn’t yet been handled;
  this together with a SW bug caused maintenance reset
- the error was propagated further ....

The reason for failure: The system was unable to react to closely timed messages

Therac-25 medical accelerator disaster

Therac-25 = a machine for radiotheratpy

- between 1985 and 1987 (at least) six accidents involving enormous radiation overdoses to patients
- Half of these patients died due to the overdoses



Therac-25 – the modes

1. electron mode
   - electron beam (low current)
   - various levels of energy (5 to 25-MeV)
   - scanning magnets used to spread the beam to a safe concentration

Therac-25 – the modes

1. electron mode
   - electron beam (low current)
   - various levels of energy (5 to 25-MeV)
   - scanning magnets used to spread the beam to a safe concentration
2. photon mode
   - only one level of energy (25-MeV), much larger electron-beam current
   - electron beam strikes a metal foil to produce X-rays (photons)
   - the X-ray beam is "flattened" by a device below the foil

Therac-25 – the modes

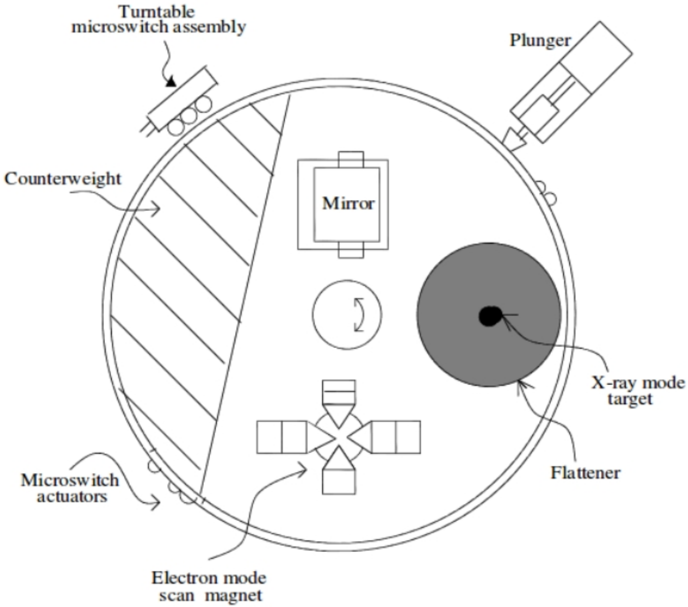
1. electron mode
   - electron beam (low current)
   - various levels of energy (5 to 25-MeV)
   - scanning magnets used to spread the beam to a safe concentration
2. photon mode
   - only one level of energy (25-MeV), much larger electron-beam current
   - electron beam strikes a metal foil to produce X-rays (photons)
   - the X-ray beam is "flattened" by a device below the foil
3. light mode – just light beam used to illuminate the field on the surface of the patient's body that will be treated

Therac-25 – the modes

1. electron mode
   - electron beam (low current)
   - various levels of energy (5 to 25-MeV)
   - scanning magnets used to spread the beam to a safe concentration
2. photon mode
   - only one level of energy (25-MeV), much larger electron-beam current
   - electron beam strikes a metal foil to produce X-rays (photons)
   - the X-ray beam is "flattened" by a device below the foil
3. light mode – just light beam used to illuminate the field on the surface of the patient’s body that will be treated

All devices placed on a turntable, supposed to be rotated to the correct position before the beam is started up

Therac-25 – turntable



The Software

The software responsible for

- Operator
  - Monitoring input and editing changes from an operator
  - Updating the screen to show current status of machine
  - Printing in response to an operator commands

The Software

The software responsible for

- Operator
  - Monitoring input and editing changes from an operator
  - Updating the screen to show current status of machine
  - Printing in response to an operator commands
- Machine
  - monitoring the machine status
  - placement of turntable
  - strength and shape of beam
  - operation of bending and scanning magnets
  - setting the machine up for the specified treatment
  - turning the beam on
  - turning the beam off (after treatment, on operator command, or if a malfunction is detected)

The Software

The software responsible for

- Operator
  - Monitoring input and editing changes from an operator
  - Updating the screen to show current status of machine
  - Printing in response to an operator commands
- Machine
  - monitoring the machine status
  - placement of turntable
  - strength and shape of beam
  - operation of bending and scanning magnets
  - setting the machine up for the specified treatment
  - turning the beam on
  - turning the beam off (after treatment, on operator command, or if a malfunction is detected)

Software running several safety critical tasks in parallel!

Insufficient hardware protection (as opposed to previous models)!!

Therac-25 – software

- The Therac-25 runs on a real-time operating system

Therac-25 – software

- The Therac-25 runs on a real-time operating system
- Four major components of software: stored data, a scheduler, a set of tasks, and interrupt services (e.g. the computer clock and handling of computer-hardware-generated errors)

Therac-25 – software

- The Therac-25 runs on a real-time operating system
- Four major components of software: stored data, a scheduler, a set of tasks, and interrupt services (e.g. the computer clock and handling of computer-hardware-generated errors)
- The software segregated the tasks above into
  - critical tasks: e.g. setup and operation of the beam
  - non-critical tasks: e.g. monitoring the keyboard

Therac-25 – software

- The Therac-25 runs on a real-time operating system
- Four major components of software: stored data, a scheduler, a set of tasks, and interrupt services (e.g. the computer clock and handling of computer-hardware-generated errors)
- The software segregated the tasks above into
  - critical tasks: e.g. setup and operation of the beam
  - non-critical tasks: e.g. monitoring the keyboard
- The scheduler directs all non-interrupt events and orders simultaneous events

Therac-25 – software

- The Therac-25 runs on a real-time operating system
- Four major components of software: stored data, a scheduler, a set of tasks, and interrupt services (e.g. the computer clock and handling of computer-hardware-generated errors)
- The software segregated the tasks above into
  - critical tasks: e.g. setup and operation of the beam
  - non-critical tasks: e.g. monitoring the keyboard
- The scheduler directs all non-interrupt events and orders simultaneous events
- Every 0.1 seconds tasks are initiated and critical tasks are executed first, with non-critical tasks taking up any remaining time

Therac-25 – software

- The Therac-25 runs on a real-time operating system
- Four major components of software: stored data, a scheduler, a set of tasks, and interrupt services (e.g. the computer clock and handling of computer-hardware-generated errors)
- The software segregated the tasks above into
    - critical tasks: e.g. setup and operation of the beam
    - non-critical tasks: e.g. monitoring the keyboard
- The scheduler directs all non-interrupt events and orders simultaneous events
- Every 0.1 seconds tasks are initiated and critical tasks are executed first, with non-critical tasks taking up any remaining time

Communication between tasks based on shared variables (without proper atomic test-and-set instructions)

What happened?

There were several accidents due to various bugs in software

What happened?

There were several accidents due to various bugs in software

One of them proceeded as follows (much simplified):

- the operator entered parameters for X-rays treatment

What happened?

There were several accidents due to various bugs in software

One of them proceeded as follows (much simplified):

- the operator entered parameters for X-rays treatment
- the machine started to set up for the treatment

What happened?

There were several accidents due to various bugs in software

One of them proceeded as follows (much simplified):

- the operator entered parameters for X-rays treatment
- the machine started to set up for the treatment
- the operator changed the mode from X-rays to electron (within the interval from 1s to 8s from the end of the original editing)

What happened?

There were several accidents due to various bugs in software

One of them proceeded as follows (much simplified):

- the operator entered parameters for X-rays treatment
- the machine started to set up for the treatment
- the operator changed the mode from X-rays to electron (within the interval from 1s to 8s from the end of the original editing)
- the patient received X-ray “treatment” with turntable in the electron position (i.e. unshielded)

What happened?

There were several accidents due to various bugs in software

One of them proceeded as follows (much simplified):

- the operator entered parameters for X-rays treatment
- the machine started to set up for the treatment
- the operator changed the mode from X-rays to electron (within the interval from 1s to 8s from the end of the original editing)
- the patient received X-ray “treatment” with turntable in the electron position (i.e. unshielded)

The cause:

- The turntable and treatment parameters were set by *different* concurrent procedures HAND and DATENT, respectively.

What happened?

There were several accidents due to various bugs in software

One of them proceeded as follows (much simplified):

- the operator entered parameters for X-rays treatment
- the machine started to set up for the treatment
- the operator changed the mode from X-rays to electron (within the interval from 1s to 8s from the end of the original editing)
- the patient received X-ray “treatment” with turntable in the electron position (i.e. unshielded)

The cause:

- The turntable and treatment parameters were set by *different* concurrent procedures HAND and DATENT, respectively.
- If the change in parameters came in the “right” time, only HAND reacted to the change.

Patriot missile mistiming

vs

Patriot missile mistiming

- Patriot – Air defense missile system

Patriot missile mistiming

- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (missile hit US army barracks, 28 persons killed)

Patriot missile mistiming

- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (missile hit US army barracks, 28 persons killed)
- The problem was caused by incorrect measurement of time

Patriot missile mistiming

- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (missile hit US army barracks, 28 persons killed)
- The problem was caused by incorrect measurement of time

Simplified principle of function:

Patriot missile mistiming

- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (missile hit US army barracks, 28 persons killed)
- The problem was caused by incorrect measurement of time

Simplified principle of function:

- Patriot's radar detects an airborne object

Patriot missile mistiming

- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (missile hit US army barracks, 28 persons killed)
- The problem was caused by incorrect measurement of time

Simplified principle of function:

- Patriot's radar detects an airborne object
- the object is identified as a scud missile (according to speed, size, etc.)

Patriot missile mistiming

- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (missile hit US army barracks, 28 persons killed)
- The problem was caused by incorrect measurement of time

Simplified principle of function:

- Patriot's radar detects an airborne object
- the object is identified as a scud missile (according to speed, size, etc.)
- the range gate computes an area in the air space where the system should next look for it

Patriot missile mistiming

- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (missile hit US army barracks, 28 persons killed)
- The problem was caused by incorrect measurement of time

Simplified principle of function:

- Patriot's radar detects an airborne object
- the object is identified as a scud missile (according to speed, size, etc.)
- the range gate computes an area in the air space where the system should next look for it
- finding the object in the calculated area confirms that it is a scud

Patriot missile mistiming

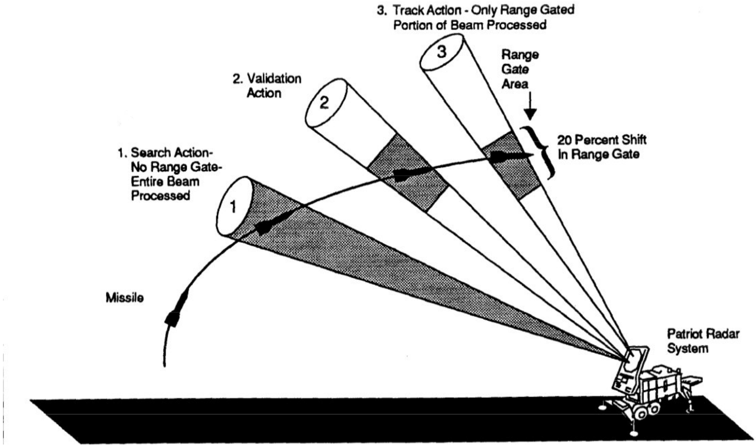
- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (missile hit US army barracks, 28 persons killed)
- The problem was caused by incorrect measurement of time

Simplified principle of function:

- Patriot's radar detects an airborne object
- the object is identified as a scud missile (according to speed, size, etc.)
- the range gate computes an area in the air space where the system should next look for it
- finding the object in the calculated area confirms that it is a scud
- then the scud is intercepted

Patriot Missile Mistiming



Patriot Missile Mistiming

Prediction of the new area:

Patriot Missile Mistiming

Prediction of the new area:

- a function of *velocity* and *time* of the last radar detection

Patriot Missile Mistiming

Prediction of the new area:

- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number

Patriot Missile Mistiming

Prediction of the new area:

- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number
- **the current time kept by incrementing whole number counter counting tenths of seconds**

Patriot Missile Mistiming

Prediction of the new area:

- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number
- **the current time kept by incrementing whole number counter counting tenths of seconds**
- computation in 24bit fixed floating point numbers

Patriot Missile Mistiming

Prediction of the new area:

- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number
- **the current time kept by incrementing whole number counter counting tenths of seconds**
- computation in 24bit fixed floating point numbers

The time converted to 24bit real number and multiplied with 1/10 represented in 24bit (i.e. the real value of 1/10 was 0.099999905)

Patriot Missile Mistiming

Prediction of the new area:

- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number
- **the current time kept by incrementing whole number counter counting tenths of seconds**
- computation in 24bit fixed floating point numbers

The time converted to 24bit real number and multiplied with 1/10 represented in 24bit (i.e. the real value of 1/10 was 0.099999905)

- the system was already running for 100 hours, i.e. the counter value was 360000, i.e. $360000 \cdot 0.099999905 = 35999.6568$

Patriot Missile Mistiming

Prediction of the new area:

- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number
- **the current time kept by incrementing whole number counter counting tenths of seconds**
- computation in 24bit fixed floating point numbers

The time converted to 24bit real number and multiplied with 1/10 represented in 24bit (i.e. the real value of 1/10 was 0.099999905)

- the system was already running for 100 hours, i.e. the counter value was 360000, i.e. $360000 \cdot 0.099999905 = 35999.6568$
- the error was 0.3432 seconds, which means 687 m off MACH 5 scud missile

Patriot Missile Mistiming

Prediction of the new area:

- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number
- **the current time kept by incrementing whole number counter counting tenths of seconds**
- computation in 24bit fixed floating point numbers

The time converted to 24bit real number and multiplied with 1/10 represented in 24bit (i.e. the real value of 1/10 was 0.099999905)

- the system was already running for 100 hours, i.e. the counter value was 360000, i.e. $360000 \cdot 0.099999905 = 35999.6568$
- the error was 0.3432 seconds, which means 687 m off MACH 5 scud missile
- the problem was not only in wrong conversion but in the fact that at some points correct conversion was used (after incomplete bug fix), so the errors did not cancel out

Patriot Missile Mistiming

Prediction of the new area:

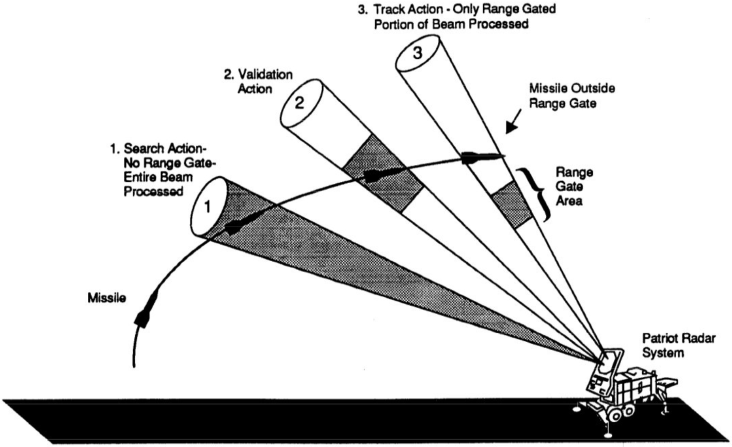
- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number
- **the current time kept by incrementing whole number counter counting tenths of seconds**
- computation in 24bit fixed floating point numbers

The time converted to 24bit real number and multiplied with 1/10 represented in 24bit (i.e. the real value of 1/10 was 0.099999905)

- the system was already running for 100 hours, i.e. the counter value was 360000, i.e. $360000 \cdot 0.099999905 = 35999.6568$
- the error was 0.3432 seconds, which means 687 m off MACH 5 scud missile
- the problem was not only in wrong conversion but in the fact that at some points correct conversion was used (after incomplete bug fix), so the errors did not cancel out

As a result, the tracking gate looked into wrong area

Patriot Missile Mistiming



Starliner

- Developed by Boeing & NASA
- Seven passengers, or a mix of crew and cargo, for missions to low-Earth orbit

Starliner

- Developed by Boeing & NASA
- Seven passengers, or a mix of crew and cargo, for missions to low-Earth orbit

- A timing issue occured on the last Orbital Flight Test on December 20, 2019

Starliner

- Developed by Boeing & NASA
- Seven passengers, or a mix of crew and cargo, for missions to low-Earth orbit

- A timing issue occured on the last Orbital Flight Test on December 20, 2019
- What is supposed to happen:
  - Atlas V leaves Starliner on a suborbital trajectory.
  - Starliner's own propulsion system takes the spacecraft into orbit and to ISS.

Starliner

- Developed by Boeing & NASA
- Seven passengers, or a mix of crew and cargo, for missions to low-Earth orbit

- A timing issue occured on the last Orbital Flight Test on December 20, 2019
- What is supposed to happen:
  - Atlas V leaves Starliner on a suborbital trajectory.
  - Starliner's own propulsion system takes the spacecraft into orbit and to ISS.
- What happened:
  - Mission Elapsed Timer (MET), or clock, on Starliner was set to the wrong time and did not trigger the engines to fire correctly.
  - Other onboard systems compensated and it reached orbit, but had depleted so much fuel there was not enough to continue the journey.

(Rough) Course Outline

- Real-time scheduling
  - Time and priority driven
  - Resource control
  - Multi-processor (a bit)

(Rough) Course Outline

- Real-time scheduling
  - Time and priority driven
  - Resource control
  - Multi-processor (a bit)
- A little bit on programming real-time systems
  - Real-time operating systems

Outline – Scheduling

The Scheduling problem:

Input:

- available processors, resources
- set of tasks/jobs

  with their requirements, deadlines, etc.

Outline – Scheduling

The Scheduling problem:

Input:

- available processors, resources
- set of tasks/jobs

  with their requirements, deadlines, etc.

Question: How to assign processors and resources to tasks/jobs so that all requirements are met?

Outline – Scheduling

The Scheduling problem:

Input:

- available processors, resources
- set of tasks/jobs

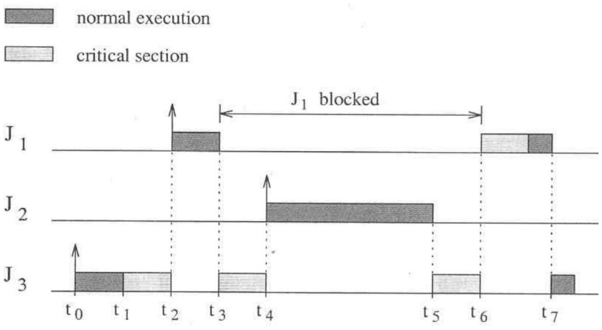
  with their requirements, deadlines, etc.

Question: How to assign processors and resources to tasks/jobs so that all requirements are met?

Example:

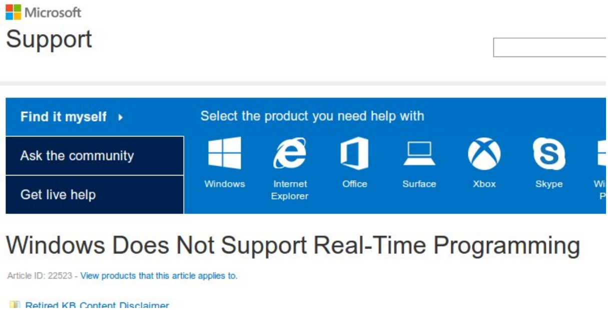
- 1 processor, one critical section shared by job 1 and job 3
- job 1: release time 1, computation time 4, deadline 8
- job 2: release time 1, computation time 2, deadline 5
- job 3: release time 0, computation time 3, deadline 4
- ...

Outline – Scheduling



- We consider a formal model of systems with parallel jobs that possibly contend for shared resources
  consider periodic as well as aperiodic jobs
- Consider various algorithms that schedule jobs to meet their timing constraints
  offline and online algorithms, RM, EDF, etc.

Outline – Programming



Basic information about RTOS and RT programming languages

- RTOS – overview
  - real-time in non-real-time operating systems
  - **implementation of theoretical concepts in freeRTOS**
- RT in programming languages – short overview

Real-Time Scheduling

Formal Model

[Some parts of this lecture are based on a real-time systems course of Colin Perkins

http://csperkins.org/teaching/rtes/index.html]

Real-Time Scheduling – Formal Model

- Introduce an abstract model of real-time systems
  - abstracts away unessential details
  - sets up consistent terminology

Real-Time Scheduling – Formal Model

- Introduce an abstract model of real-time systems
  - abstracts away unessential details
  - sets up consistent terminology

- Three components of the model
  - A workload model that describes applications supported by the system
    i.e. jobs, tasks, ...
  - A resource model that describes the system resources available to applications
    i.e. processors, passive resources, ...
  - Algorithms that define how the application uses the resources at all times
    i.e. scheduling and resource access protocols

Basic Notions

- A *job* is a unit of work that is scheduled and executed by a system

  compute a control law, transform sensor data, etc.

Basic Notions

- A *job* is a unit of work that is scheduled and executed by a system

  compute a control law, transform sensor data, etc.

- A *task* is a set of related jobs which jointly provide some system function

  check temperature periodically, keep a steady flow of water

Basic Notions

- A *job* is a unit of work that is scheduled and executed by a system

  compute a control law, transform sensor data, etc.

- A *task* is a set of related jobs which jointly provide some system function

  check temperature periodically, keep a steady flow of water

- A job executes on a *processor*

  CPU, transmission link in a network, database server, etc.

Basic Notions

- A *job* is a unit of work that is scheduled and executed by a system

  compute a control law, transform sensor data, etc.

- A *task* is a set of related jobs which jointly provide some system function

  check temperature periodically, keep a steady flow of water

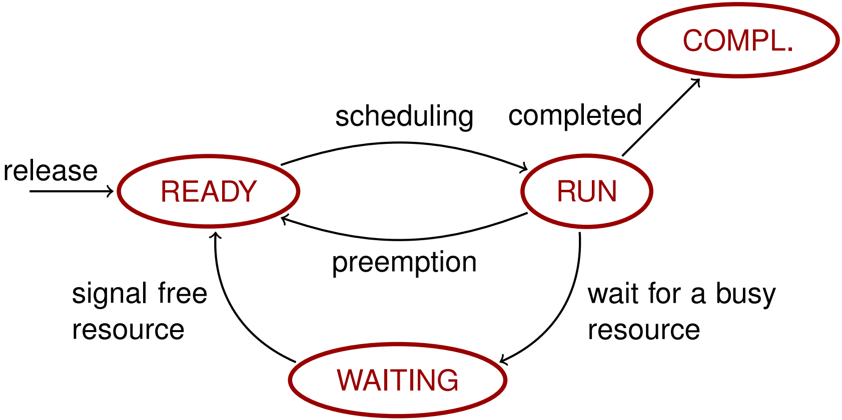
- A job executes on a *processor*

  CPU, transmission link in a network, database server, etc.

- A job may use some (shared) passive *resources*

  file, database lock, shared variable etc.

Life Cycle of a Job



Jobs – Parameters

We consider finite, or countably infinte number of jobs $J_1, J_2, \ldots$

Each job has several parameters.

Jobs – Parameters

We consider finite, or countably infinte number of jobs $J_1, J_2, \ldots$

Each job has several parameters.

There are four types of job parameters:

- temporal
  - release time, execution time, deadlines
- functional
  - Laxity type: hard and soft real-time
  - preemptability, (criticality)
- interconnection
  - precedence constraints
- resource
  - usage of processors and passive resources

Job Parameters – Execution Time

Execution time e_i **of a job** J_i – the amount of time required to complete the execution of J_i when it executes alone and has all necessary resources

Job Parameters – Execution Time

Execution time e_i **of a job** J_i – the amount of time required to complete the execution of J_i when it executes alone and has all necessary resources

- Value of e_i depends upon complexity of the job and speed of the processor on which it executes; may change for various reasons:
  - Conditional branches
  - Caches, pipelines, etc.
  - ...
- **Execution times fall into an interval** $[e_i^-, e_i^+]$; we assume that we know this interval (WCET analysis) but not necessarily e_i

Job Parameters – Execution Time

Execution time e_i **of a job** J_i – the amount of time required to complete the execution of J_i when it executes alone and has all necessary resources

- Value of e_i depends upon complexity of the job and speed of the processor on which it executes; may change for various reasons:
  - Conditional branches
  - Caches, pipelines, etc.
  - ...
- **Execution times fall into an interval** $[e_i^-, e_i^+]$; we assume that we know this interval (WCET analysis) but not necessarily e_i

We usually validate the system using only e_i^+ for each job i.e. assume $e_i = e_i^+$

Job Parameters – Release and Response Time

Release time r_i – the instant in time when a job J_i becomes available for execution

- Release time may *jitter*, only an interval $[r_i^-, r_i^+]$ is known
- A job can be executed at any time at, or after, its release time, provided its processor and resource demands are met

Job Parameters – Release and Response Time

Release time r_i – the instant in time when a job J_i becomes available for execution

- Release time may *jitter*, only an interval $[r_i^-, r_i^+]$ is known
- A job can be executed at any time at, or after, its release time, provided its processor and resource demands are met

Completion time C_i – the instant in time when a job completes its execution

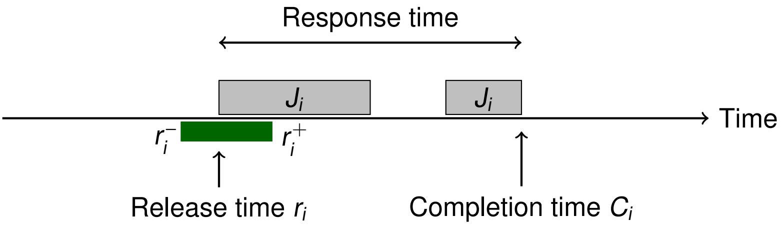
Job Parameters – Release and Response Time

Release time r_i – the instant in time when a job J_i becomes available for execution

- Release time may *jitter*, only an interval $[r_i^-, r_i^+]$ is known
- A job can be executed at any time at, or after, its release time, provided its processor and resource demands are met

Completion time C_i – the instant in time when a job completes its execution

Response time – the difference $C_i - r_i$ between the completion time and the release time



Job Parameters – Deadlines

Absolute deadline d_i – the instant in time by which a job must be completed

Job Parameters – Deadlines

Absolute deadline d_i – the instant in time by which a job must be completed

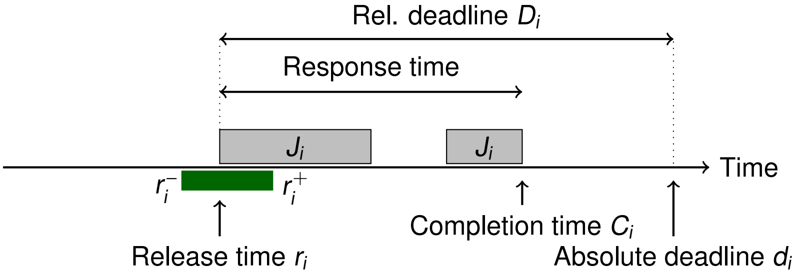
Relative deadline D_i – the maximum allowable response time i.e. $D_i = d_i - r_i$

Job Parameters – Deadlines

Absolute deadline d_i – the instant in time by which a job must be completed

Relative deadline D_i – the maximum allowable response time i.e. $D_i = d_i - r_i$

Feasible interval is the interval $(r_i, d_i]$

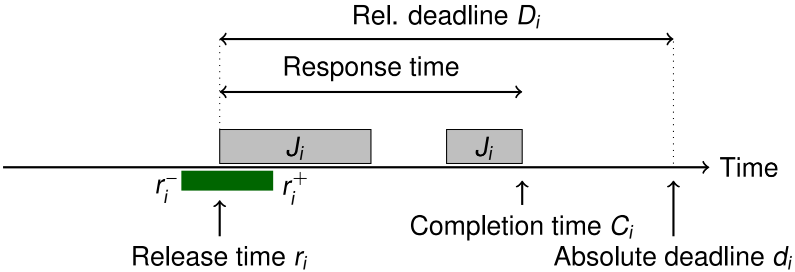


Job Parameters – Deadlines

Absolute deadline d_i – the instant in time by which a job must be completed

Relative deadline D_i – the maximum allowable response time i.e. $D_i = d_i - r_i$

Feasible interval is the interval $(r_i, d_i]$



A *timing constraint* of a job is specified using release time together with relative and absolute deadlines.

Laxity Type – Hard Real-Time

A **hard real-time constraint** specifies that a job should never miss its deadline.

Examples: Flight control, railway signaling, anti-lock brakes, etc.

Laxity Type – Hard Real-Time

A **hard real-time constraint** specifies that a job should never miss its deadline.

Examples: Flight control, railway signaling, anti-lock brakes, etc.

Several more precise definitions occur in literature:

- A timing constraint is hard if the failure to meet it is considered a fatal error

  e.g. a bomb is dropped too late and hits civilians

Laxity Type – Hard Real-Time

A **hard real-time constraint** specifies that a job should never miss its deadline.

Examples: Flight control, railway signaling, anti-lock brakes, etc.

Several more precise definitions occur in literature:

- A timing constraint is hard if the failure to meet it is considered a fatal error

  e.g. a bomb is dropped too late and hits civilians

- A timing constraint is hard if the usefulness of the results falls off abruptly (may even become negative) at the deadline

  Here the nature of abruptness allows to soften the constraint

Laxity Type – Hard Real-Time

A **hard real-time constraint** specifies that a job should never miss its deadline.

Examples: Flight control, railway signaling, anti-lock brakes, etc.

Several more precise definitions occur in literature:

- A timing constraint is hard if the failure to meet it is considered a fatal error

  e.g. a bomb is dropped too late and hits civilians

- A timing constraint is hard if the usefulness of the results falls off abruptly (may even become negative) at the deadline

  Here the nature of abruptness allows to soften the constraint

Definition 5

A *timing constraint is hard* if the user requires *formal validation* that the job meets its timing constraint.

Laxity Type – Soft Real-Time

A **soft real-time constraint** specifies that a job could occasionally miss its deadline

Examples: stock trading, multimedia, etc.

Laxity Type – Soft Real-Time

A **soft real-time constraint** specifies that a job could occasionally miss its deadline

Examples: stock trading, multimedia, etc.

Several more precise definitions occur in literature:

- A timing constraint is soft if the failure to meet it is undesirable but acceptable if the probability is low

Laxity Type – Soft Real-Time

A **soft real-time constraint** specifies that a job could occasionally miss its deadline

Examples: stock trading, multimedia, etc.

Several more precise definitions occur in literature:

- A timing constraint is soft if the failure to meet it is undesirable but acceptable if the probability is low
- A timing constraint is soft if the usefulness of the results decreases at a slower rate with *tardiness* of the job

  e.g. the probability that a response time exceeds 50 ms is less than 0.2

Laxity Type – Soft Real-Time

A **soft real-time constraint** specifies that a job could occasionally miss its deadline

Examples: stock trading, multimedia, etc.

Several more precise definitions occur in literature:

- A timing constraint is soft if the failure to meet it is undesirable but acceptable if the probability is low
- A timing constraint is soft if the usefulness of the results decreases at a slower rate with *tardiness* of the job

  e.g. the probability that a response time exceeds 50 ms is less than 0.2

Definition 6

A *timing constraint is soft* if either validation is not required, or only a demonstration that a *statistical constraint* is met suffices.

Jobs – Preemptability

Jobs may be interrupted by higher priority jobs

Jobs – Preemptability

Jobs may be interrupted by higher priority jobs

- A job is *preemptable* if its execution can be interrupted
- A job is *non-preemptable* if it must run to completion once started

  (Some preemptable jobs have periods during which they cannot be preempted)
- The *context switch time* is the time to switch between jobs

  (Most of the time we assume that this time is negligible)

Jobs – Preemptability

Jobs may be interrupted by higher priority jobs

- A job is *preemptable* if its execution can be interrupted
- A job is *non-preemptable* if it must run to completion once started

  (Some preemptable jobs have periods during which they cannot be preempted)
- The *context switch time* is the time to switch between jobs

  (Most of the time we assume that this time is negligible)

Reasons for preemptability:

- Jobs may have different levels of criticality

  e.g. brakes vs radio tunning
- Priorities may make part of scheduling algorithm

  e.g. resource access control algorithms

Jobs – Precedence Constraints

Jobs may be constrained to execute in a particular order

Jobs – Precedence Constraints

Jobs may be constrained to execute in a particular order

- This is known as a *precedence constraint*
- A job J_i is a *predecessor* of another job J_k and J_k a *successor* of J_i (denoted by $J_i < J_k$) if J_k cannot begin execution until the execution of J_i completes
- J_i is an *immediate predecessor* of J_k if $J_i < J_k$ and there is no other job J_j such that $J_i < J_j < J_k$
- J_i and J_k are *independent* when neither $J_i < J_k$ nor $J_k < J_i$

Jobs – Precedence Constraints

Jobs may be constrained to execute in a particular order

- This is known as a *precedence constraint*
- A job J_i is a *predecessor* of another job J_k and J_k a *successor* of J_i (denoted by $J_i < J_k$) if J_k cannot begin execution until the execution of J_i completes
- J_i is an *immediate predecessor* of J_k if $J_i < J_k$ and there is no other job J_j such that $J_i < J_j < J_k$
- J_i and J_k are *independent* when neither $J_i < J_k$ nor $J_k < J_i$

A job with a precedence constraint becomes ready for execution when its release time has passed and when all predecessors have completed.

Example: authentication before retrieving an information, a signal processing task in radar surveillance system precedes a tracker task

Tasks – Modeling Reactive Systems

Reactive systems – run for unlimited amount of time

Tasks – Modeling Reactive Systems

Reactive systems – run for unlimited amount of time

A system parameter: number of tasks

- may be known in advance (flight control)
- may change during computation (air traffic control)

Tasks – Modeling Reactive Systems

Reactive systems – run for unlimited amount of time

A system parameter: number of tasks

- may be known in advance (flight control)
- may change during computation (air traffic control)

We consider three types of tasks

- Periodic – jobs executed at regular intervals, hard deadlines
- Aperiodic – jobs executed in random intervals, soft deadlines
- Sporadic – jobs executed in random intervals, hard deadlines

... precise definitions later.

Processors

A processor, *P*, is an *active* component on which jobs are scheduled

Processors

A processor, P, is an *active* component on which jobs are scheduled

The general case considered in literature:

m processors $P_1, \ldots, P_m$, each P_i has its *type* and *speed*.

Processors

A processor, P, is an *active* component on which jobs are scheduled

The general case considered in literature:

m processors $P_1, \ldots, P_m$, each P_i has its *type* and *speed*.

We mostly concentrate on *single processor* scheduling

- Efficient scheduling algorithms
- In a sense subsumes multiprocessor scheduling where tasks are assigned *statically* to individual processors

  i.e. all jobs of every task are assigned to a single processor

Processors

A processor, P, is an *active* component on which jobs are scheduled

The general case considered in literature:

m processors $P_1, \ldots, P_m$, each P_i has its *type* and *speed*.

We mostly concentrate on *single processor* scheduling

- Efficient scheduling algorithms
- In a sense subsumes multiprocessor scheduling where tasks are assigned *statically* to individual processors

  i.e. all jobs of every task are assigned to a single processor

Multi-processor scheduling is a rich area of current research, we touch it only lightly (later).

Resources

A resource, R, is a *passive* entity upon which jobs may depend

Resources

A resource, R, is a *passive* entity upon which jobs may depend

In general, **we consider** n **resources** $R_1, \ldots, R_n$ **of distinct types**

Resources

A resource, R, is a *passive* entity upon which jobs may depend

In general, **we consider** n **resources** $R_1, \ldots, R_n$ **of distinct types**

Each R_i is used in a mutually exclusive manner

- A job that acquires a free resource locks the resource
- Jobs that need a busy resource have to wait until the resource is released
- Once released, the resource may be used by another job

  (i.e. it is not consumed)

(More generally, each resource may be used by k jobs concurrently, i.e., there are k units of the resource)

Resources

A resource, R, is a *passive* entity upon which jobs may depend

In general, **we consider n resources $R_1, \ldots, R_n$ of distinct types**

Each R_i is used in a mutually exclusive manner

- A job that acquires a free resource locks the resource
- Jobs that need a busy resource have to wait until the resource is released
- Once released, the resource may be used by another job
  (i.e. it is not consumed)

(More generally, each resource may be used by k jobs concurrently, i.e., there are k units of the resource)

Resource requirements of a job specify

- which resources are used by the job
- the time interval(s) during which each resource is required
  (precise definitions later)

Scheduling

Schedule assigns, in every time instant, processors and resources to jobs.

More formally, a schedule is a function

$$\sigma : \{J_1, \ldots\} \times \mathbb{R}_0^+ \to \mathcal{P}(\{P_1, \ldots, P_m, R_1, \ldots, R_n\})$$

so that for every $t \in \mathbb{R}_0^+$ there are rational $0 \leq t_1 \leq t < t_2$ such that $\sigma(J_i, \cdot)$ is constant on $[t_1, t_2)$.

(We also assume that there is the least time quantum in which scheduler does not change its decisions, i.e. each of the intervals $[t_1, t_2)$ is larger than a fixed $\varepsilon > 0$.)

Valid and Feasible Schedule

A schedule is *valid* if it satisfies the following conditions:

- Every processor is assigned to at most one job at any time
- Every job is assigned to at most one processor at any time
- No job is scheduled before its release time
- The total amount of processor time assigned to a given job is equal to its actual execution time
- *All the precedence and resource usage constraints are satisfied*

Valid and Feasible Schedule

A schedule is *valid* if it satisfies the following conditions:

- Every processor is assigned to at most one job at any time
- Every job is assigned to at most one processor at any time
- No job is scheduled before its release time
- The total amount of processor time assigned to a given job is equal to its actual execution time
- *All the precedence and resource usage constraints are satisfied*

A schedule is *feasible* if *all jobs with hard real-time constraints* complete before their deadlines

Valid and Feasible Schedule

A schedule is *valid* if it satisfies the following conditions:

- Every processor is assigned to at most one job at any time
- Every job is assigned to at most one processor at any time
- No job is scheduled before its release time
- The total amount of processor time assigned to a given job is equal to its actual execution time
- *All the precedence and resource usage constraints are satisfied*

A schedule is *feasible* if *all jobs with hard real-time constraints* complete before their deadlines

A set of jobs is *schedulable* if there is a feasible schedule for the set.

Scheduling – Algorithms

Scheduling algorithm computes a schedule for a set of jobs

A set of jobs is *schedulable according to a scheduling algorithm* if the algorithm produces a feasible schedule

Scheduling – Algorithms

Scheduling algorithm computes a schedule for a set of jobs

A set of jobs is *schedulable according to a scheduling algorithm* if the algorithm produces a feasible schedule

Scheduling – Algorithms

Scheduling algorithm computes a schedule for a set of jobs

A set of jobs is *schedulable according to a scheduling algorithm* if the algorithm produces a feasible schedule

Definition 7

A scheduling algorithm is optimal if it always produces a feasible schedule whenever such a schedule exists.

Real-Time Scheduling

Individual Jobs

Scheduling of Individual Jobs

We start with scheduling of finite sets of jobs $\{J_1, \ldots, J_m\}$ for execution on **single processor** systems.

Scheduling of Individual Jobs

We start with scheduling of finite sets of jobs $\{J_1, \ldots, J_m\}$ for execution on **single processor** systems.

Each J_i has a release time r_i, an execution time e_i and an absolute deadline d_i.

We assume hard real-time constraints.

The question: Is there an optimal scheduling algorithm?

Scheduling of Individual Jobs

We start with scheduling of finite sets of jobs $\{J_1, \ldots, J_m\}$ for execution on **single processor** systems.

Each J_i has a release time r_i, an execution time e_i and an absolute deadline d_i.

We assume hard real-time constraints.

The question: Is there an optimal scheduling algorithm?

We proceed in the direction of growing generality:

1. No resources, independent, synchronized (i.e. $r_i = 0$ for all i)
2. No resources, independent but not synchronized
3. No resources but possibly dependent
4. The general case

No resources, Independent, Synchronized

	J_1	J_2	J_3	J_4	J_5
e_i	1	1	1	3	2
d_i	3	10	7	8	5

Is there a feasible schedule?

No resources, Independent, Synchronized

	J_1	J_2	J_3	J_4	J_5
e_i	1	1	1	3	2
d_i	3	10	7	8	5

Is there a feasible schedule?

Note: Preemption does not help in synchronized case.

No resources, Independent, Synchronized

	J_1	J_2	J_3	J_4	J_5
e_i	1	1	1	3	2
d_i	3	10	7	8	5

Is there a feasible schedule?

Note: Preemption does not help in synchronized case.

Theorem 8

If there are no resource contentions, then executing independent jobs in the order of non-decreasing deadline (EDD) produces a feasible schedule (if it exists).

No resources, Independent, Synchronized

	J_1	J_2	J_3	J_4	J_5
e_i	1	1	1	3	2
d_i	3	10	7	8	5

Is there a feasible schedule?

Note: Preemption does not help in synchronized case.

Theorem 8

If there are no resource contentions, then executing independent jobs in the order of non-decreasing deadline (EDD) produces a feasible schedule (if it exists).

Proof.

Let σ be a schedule. **Inversion** is a pair (J_a, J_b) such that J_a precedes J_b in σ but $d_b < d_a$.

Note that σ is EDD iff it does not contain any inversion.

Proof cont.

Assume $k > 0$ inversions in σ.

Let (J_a, J_b) be an inversion such that J_a is scheduled right before J_b.
There is always at least one such inversion (homework).

Let $t_a < t_b$ be the time instants when J_a, J_b start to be executed in σ.
Recall: C_a, C_b are completion times of J_a, J_b, and e_a, e_b are execution times.

Note that $C_a \leq d_a$ and that $C_b \leq d_b < d_a$.

Proof cont.

Assume $k > 0$ inversions in σ.

Let (J_a, J_b) be an inversion such that J_a is scheduled right before J_b.
There is always at least one such inversion (homework).

Let $t_a < t_b$ be the time instants when J_a, J_b start to be executed in σ.
Recall: C_a, C_b are completion times of J_a, J_b, and e_a, e_b are execution times.

Note that $C_a \leq d_a$ and that $C_b \leq d_b < d_a$.

Define a new schedule σ' in which:

- All jobs except J_a, J_b are scheduled as in σ,
- J_b starts at t_a,
- J_a starts at $t_a + e_b$.

Observe that σ' is still feasible:

- J_b is completed at $t_a + e_b < t_a + e_b + e_a = t_b + e_b = C_b \leq d_b$
- J_a is completed at $t_a + e_b + e_a = C_b \leq d_b < d_a$

Note that σ' has $k - 1$ inversions. By repeating the above procedure k times, we obtain an EDD schedule. □

No resources, Independent, Synchronized

Is there any simple schedulability test?

$\{J_1, \ldots, J_n\}$ where $d_1 \leq \cdots \leq d_n$ is schedulable iff
$\forall i \in \{1, \ldots, n\} \ : \ \sum_{k=1}^{i} e_k \leq d_i$

No resources, Independent (No Synchro)

	J_1	J_2	J_3
r_i	0	0	2
e_i	1	2	2
d_i	2	5	4

- find a (feasible) schedule (with and without preemption)
- determine response time of each job in your schedule

No resources, Independent (No Synchro)

	J_1	J_2	J_3
r_i	0	0	2
e_i	1	2	2
d_i	2	5	4

- find a (feasible) schedule (with and without preemption)
- determine response time of each job in your schedule

Preemption makes a difference.

No resources, Independent (No Synchro)

Earliest Deadline First (EDF) scheduling:

At any time instant, a job with the earliest absolute deadline is executed

Here EDF works in the preemptive case but not in the non-preemptive one.

	J_1	J_2
r_i	0	1
e_i	4	2
d_i	7	5

No Resources, Independent (No Synchro)

Theorem 9

If there are no resource contentions, jobs are independent and preemption is allowed, the EDF algorithm finds a feasible schedule (if it exists).

Proof.

We show that any feasible schedule σ can be transformed in finitely many steps to EDF schedule which is feasible.

No Resources, Independent (No Synchro)

Theorem 9

If there are no resource contentions, jobs are independent and preemption is allowed, the EDF algorithm finds a feasible schedule (if it exists).

Proof.

We show that any feasible schedule σ can be transformed in finitely many steps to EDF schedule which is feasible.

Let σ be a feasible schedule but not EDF. Assume, w.l.o.g., that for every $k \in \mathbb{N}$ at most one job is executed in the interval $[k, k+1)$ and that all release times and deadlines are in $\mathbb{N}$.

(Otherwise rescale by the least common multiple.)

No Resources, Independent (No Synchro)

Proof cont.

We say that σ violates EDF at k if there are two jobs J_a and J_b that satisfy:

- J_a and J_b are ready for execution at k
- J_a is executed in $[k, k+1)$
- $d_b < d_a$

No Resources, Independent (No Synchro)

Proof cont.

We say that σ violates EDF at k if there are two jobs J_a and J_b that satisfy:

- J_a and J_b are ready for execution at k
- J_a is executed in $[k, k+1)$
- $d_b < d_a$

Let $k \in \mathbb{N}$ be the *least* time instant such that σ violates EDF at k as witnessed by jobs J_a and J_b.
Assume, w.l.o.g. that J_b has the minimum deadline among all jobs ready for execution at k.

There is $k < \ell < d_b$ such that J_b is executed in $[\ell, \ell+1)$.

No Resources, Independent (No Synchro)

Proof cont.

We say that σ violates EDF at k if there are two jobs J_a and J_b that satisfy:

- J_a and J_b are ready for execution at k
- J_a is executed in $[k, k+1)$
- $d_b < d_a$

Let $k \in \mathbb{N}$ be the *least* time instant such that σ violates EDF at k as witnessed by jobs J_a and J_b.
Assume, w.l.o.g. that J_b has the minimum deadline among all jobs ready for execution at k.

There is $k < \ell < d_b$ such that J_b is executed in $[\ell, \ell+1)$.

Let us define a new schedule σ' which is the same as σ except:

- executes J_b in $[k, k+1)$
- executes J_a in $[\ell, \ell+1)$

Then σ' is feasible and does not violate EDF at any $k' \leq k$.

Finitely many steps transform any feasible schedule to EDF. □

No resources, Independent (No Synchro)

The **non-preemptive** case is NP-hard.

No resources, Independent (No Synchro)

The **non-preemptive** case is NP-hard.

Heuristics are needed, such as the **Spring algorithm**, that usually work in much more general setting (with resources etc.)

No resources, Independent (No Synchro)

The **non-preemptive** case is NP-hard.

Heuristics are needed, such as the **Spring algorithm**, that usually work in much more general setting (with resources etc.)

Use the notion of *partial schedule* where only a subset of tasks has been scheduled.

Exhaustive search through partial schedules

- start with an empty schedule

No resources, Independent (No Synchro)

The **non-preemptive** case is NP-hard.

Heuristics are needed, such as the **Spring algorithm**, that usually work in much more general setting (with resources etc.)

Use the notion of *partial schedule* where only a subset of tasks has been scheduled.

Exhaustive search through partial schedules

- start with an empty schedule
- in every step either
  - add a job which maximizes a *heuristic function H* among jobs that have not yet been tried in this partial schedule
  - or backtrack if there is no such a job

No resources, Independent (No Synchro)

The **non-preemptive** case is NP-hard.

Heuristics are needed, such as the **Spring algorithm**, that usually work in much more general setting (with resources etc.)

Use the notion of *partial schedule* where only a subset of tasks has been scheduled.

Exhaustive search through partial schedules

- start with an empty schedule
- in every step either
  - add a job which maximizes a *heuristic function H* among jobs that have not yet been tried in this partial schedule
  - or backtrack if there is no such a job
- After failure, backtrack to previous partial schedule

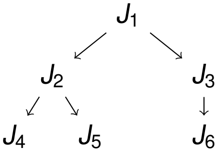
Heuristic function identifies plausible jobs to be scheduled (earliest release, earliest deadline, etc.)

No Resources, Dependent (No Synchro)

Example:

	J_1	J_2	J_3	J_4	J_5	J_6
e_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6

Dependencies:



Does EDF work?

No resources, Dependent (No Synchro)

Theorem 10

Assume that there are no resource contentions and jobs are preemptable. There is a polynomial time algorithm which decides whether a feasible schedule exists and if yes, then computes one.

Idea: Reduce to independent jobs by changing release times and deadlines. Then use EDF.

No resources, Dependent (No Synchro)

Theorem 10

Assume that there are no resource contentions and jobs are preemptable. There is a polynomial time algorithm which decides whether a feasible schedule exists and if yes, then computes one.

Idea: Reduce to independent jobs by changing release times and deadlines. Then use EDF.

Observe that if $J_i < J_k$ then replacing

- r_k with $\max\{r_k, r_i + e_i\}$
  (J_k cannot be scheduled for execution before $r_i + e_i$ because J_i cannot be finished before $r_i + e_i$)
- d_i with $\min\{d_i, d_k - e_k\}$
  (J_i must be finished before $d_k - e_k$ so that J_k can be finished before d_k)

does not change feasibility.

Replace systematically according to the precedence relation.

No Resources, Dependent (No Synchro)

Define r_k^*, d_k^* systematically as follows:

- Pick J_k whose all predecessors have been processed and compute $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$. Repeat for all jobs.
- Pick J_k whose all successors have been processed and compute $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$. Repeat for all jobs.

No Resources, Dependent (No Synchro)

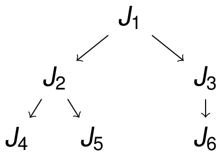
Define r_k^*, d_k^* systematically as follows:

- Pick J_k whose all predecessors have been processed and compute $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$. Repeat for all jobs.
- Pick J_k whose all successors have been processed and compute $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$. Repeat for all jobs.

Example:

	J_1	J_2	J_3	J_4	J_5	J_6
e_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6

Dependencies:



No Resources, Dependent (No Synchro)

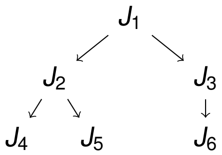
Define r_k^*, d_k^* systematically as follows:

- Pick J_k whose all predecessors have been processed and compute $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$. Repeat for all jobs.
- Pick J_k whose all successors have been processed and compute $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$. Repeat for all jobs.

Example:

	J_1	J_2	J_3	J_4	J_5	J_6
e_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6

Dependencies:



Do you need the precedence constraints?

No Resources, Dependent (No Synchro)

Define r_k^*, d_k^* systematically as follows:

- Pick J_k whose all predecessors have been processed and compute $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$. Repeat for all jobs.
- Pick J_k whose all successors have been processed and compute $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$. Repeat for all jobs.

No Resources, Dependent (No Synchro)

Define r_k^*, d_k^* systematically as follows:

- Pick J_k whose all predecessors have been processed and compute $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$. Repeat for all jobs.
- Pick J_k whose all successors have been processed and compute $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$. Repeat for all jobs.

This gives a new set of jobs $J_1^*, \ldots, J_m^*$ where each J_k^* has the release time r_k^* and the absolute deadline d_k^*.

We impose **no precedence constraints** on $J_1^*, \ldots, J_m^*$.

No Resources, Dependent (No Synchro)

Define r_k^*, d_k^* systematically as follows:

- Pick J_k whose all predecessors have been processed and compute $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$. Repeat for all jobs.
- Pick J_k whose all successors have been processed and compute $d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$. Repeat for all jobs.

This gives a new set of jobs $J_1^*, \ldots, J_m^*$ where each J_k^* has the release time r_k^* and the absolute deadline d_k^*.

We impose **no precedence constraints** on $J_1^*, \ldots, J_m^*$.

Lemma 11

$\{J_1, \ldots, J_m\}$ *is feasible iff* $\{J_1^*, \ldots, J_m^*\}$ *is feasible. If EDF schedule is feasible on* $\{J_1^*, \ldots, J_m^*\}$*, then the same schedule is feasible on* $\{J_1, \ldots, J_m\}$*.*

The same schedule means that whenever J_i^* is scheduled at time t, then J_i is scheduled at time t.

No Resources, Dependent (No Synchro)

Recall: $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$ and
$d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$

Proof of Lemma 11.

$\Rightarrow$: It is easy to show that in *no feasible schedule* on $\{J_1, \ldots, J_m\}$ any job J_k can be executed before r_k^* and completed after d_k^* (otherwise, precedence constraints would be violated).

No Resources, Dependent (No Synchro)

Recall: $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$ and
$d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$

Proof of Lemma 11.

$\Rightarrow$: It is easy to show that in *no feasible schedule* on $\{J_1, \ldots, J_m\}$ any job J_k can be executed before r_k^* and completed after d_k^* (otherwise, precedence constraints would be violated).

$\Leftarrow$: Assume that EDF σ is feasible on $\{J_1^*, \ldots, J_m^*\}$. Let us use σ on $\{J_1, \ldots, J_m\}$.

I.e. J_i is executed iff J_i^* is executed.

No Resources, Dependent (No Synchro)

Recall: $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$ and
$d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$

Proof of Lemma 11.

$\Rightarrow$: It is easy to show that in *no feasible schedule* on $\{J_1, \ldots, J_m\}$ any job J_k can be executed before r_k^* and completed after d_k^* (otherwise, precedence constraints would be violated).

$\Leftarrow$: Assume that EDF σ is feasible on $\{J_1^*, \ldots, J_m^*\}$. Let us use σ on $\{J_1, \ldots, J_m\}$.

I.e. J_i is executed iff J_i^* is executed.

Timing constraints of $\{J_1, \ldots, J_m\}$ are satisfied since $r_k \leq r_k^*$ and $d_k \geq d_k^*$ for all k.

No Resources, Dependent (No Synchro)

Recall: $r_k^* := \max\{r_k, \max_{J_i < J_k} r_i^* + e_i\}$ and
$d_k^* := \min\{d_k, \min_{J_k < J_i} d_i^* - e_i\}$

Proof of Lemma 11.

$\Rightarrow$: It is easy to show that in *no feasible schedule* on $\{J_1, \ldots, J_m\}$ any job J_k can be executed before r_k^* and completed after d_k^* (otherwise, precedence constraints would be violated).

$\Leftarrow$: Assume that EDF σ is feasible on $\{J_1^*, \ldots, J_m^*\}$. Let us use σ on $\{J_1, \ldots, J_m\}$.

I.e. J_i is executed iff J_i^* is executed.

Timing constraints of $\{J_1, \ldots, J_m\}$ are satisfied since $r_k \leq r_k^*$ and $d_k \geq d_k^*$ for all k.

Precedence constraints: Assume that $J_s < J_t$. Then J_s^* executes completely before J_t^* since $r_s^* < r_s^* + e_s \leq r_t^*$ and $d_s^* \leq d_t^* - e_t < d_t^*$ and σ is EDF on $\{J_1^* \ldots, J_m^*\}$.

Resources, Dependent, Not Synchronized

Even the preemptive case is NP-hard

- reduce the non-preemptive case without resources to the preemptive with resources
- Use a common resource R.
  - Whenever a job starts its execution it locks the resource R.
  - Whenever a job finishes its execution it releases the resourse R.

Could be solved using heuristics, e.g. the Spring algorithm.

Real-Time Scheduling

Scheduling of Reactive Systems

[Some parts of this lecture are based on a real-time systems course of Colin Perkins

http://csperkins.org/teaching/rtes/index.html]

Reminder of Basic Notions

- Jobs are executed on processors and need resources
- Parameters of jobs
  - temporal:
    - release time – r_i
    - execution time – e_i
    - absolute deadline – d_i
    - derived params: relative deadline (D_i), completion time, response time, ...
  - functional:
    - laxity type: hard vs soft
    - preemptability
  - interconnection
    - precedence constraints (independence)
  - resource
    - what resources and when are used by the job
- Tasks = sets of jobs

Reminder of Basic Notions

- Schedule assigns, in every time instant, processors and resources to jobs
- valid schedule = correct (common sense)
- Feasible schedule = valid and all hard real-time jobs meet deadlines
- Set of jobs is schedulable if there is a feasible schedule for it

- Scheduling algorithm computes a schedule for a set of jobs
- Scheduling algorithm is optimal if it always produces a feasible schedule whenever such a schedule exists, and if a cost function is given, minimizes the cost

Scheduling Reactive Systems

We have considered scheduling of individual jobs

From this point on we concentrate on reactive systems

i.e. systems that run for unlimited amount of time

Scheduling Reactive Systems

We have considered scheduling of individual jobs

From this point on we concentrate on reactive systems

i.e. systems that run for unlimited amount of time

Recall that a task is a set of related jobs that jointly provide some system function.

Scheduling Reactive Systems

We have considered scheduling of individual jobs

From this point on we concentrate on reactive systems

i.e. systems that run for unlimited amount of time

Recall that a task is a set of related jobs that jointly provide some system function.

- We consider various types of tasks
  - Periodic
  - Aperiodic
  - Sporadic

Scheduling Reactive Systems

We have considered scheduling of individual jobs

From this point on we concentrate on reactive systems

i.e. systems that run for unlimited amount of time

Recall that a task is a set of related jobs that jointly provide some system function.

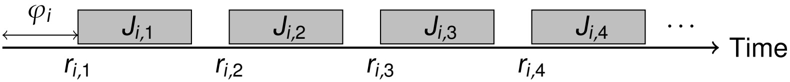
- We consider various types of tasks
  - Periodic
  - Aperiodic
  - Sporadic

- Differ in execution time patterns for jobs in the tasks

- Must be modeled differently
  - Differing scheduling algorithms
  - Differing impact on system performance
  - Differing constraints on scheduling

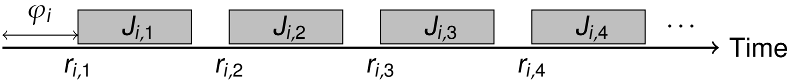
Periodic Tasks

- A set of jobs that are executed repeatedly at regular time intervals can be modeled as a *periodic task*



Periodic Tasks

- A set of jobs that are executed repeatedly at regular time intervals can be modeled as a *periodic task*



- Each periodic task T_i is a sequence of jobs $J_{i,1}, J_{i,2}, \ldots J_{i,n}, \ldots$
  - The *phase* φ_i of a task T_i is *the release time* $r_{i,1}$ *of the first job* $J_{i,1}$ in the task T_i ;
    tasks are *in phase* if their phases are equal
  - The *period* p_i of a task T_i is *the minimum length of all time intervals between release times of consecutive jobs* in T_i
  - The *execution time* e_i of a task T_i is *the maximum execution time of all jobs* in T_i
  - The *relative deadline* D_i is relative deadline of all jobs in T_i

(The period and execution time of every periodic task in the system are known with reasonable accuracy at all times)

Periodic Tasks – Notation

The 4-tuple $T_i = (\varphi_i, p_i, e_i, D_i)$ refers to a periodic task T_i with phase φ_i, period p_i, execution time e_i, and relative deadline D_i

Periodic Tasks – Notation

The 4-tuple $T_i = (\varphi_i, p_i, e_i, D_i)$ refers to a periodic task T_i with phase φ_i, period p_i, execution time e_i, and relative deadline D_i

For example: jobs of $T_1 = (1, 10, 3, 6)$ are

- released at times 1, 11, 21, . . .,
- execute for 3 time units,
- have to be finished in 6 time units (the first by 7, the second by 17, ...)

Periodic Tasks – Notation

The 4-tuple $T_i = (\varphi_i, p_i, e_i, D_i)$ refers to a periodic task T_i with phase φ_i, period p_i, execution time e_i, and relative deadline D_i

For example: jobs of $T_1 = (1, 10, 3, 6)$ are

- released at times 1, 11, 21, ...,
- execute for 3 time units,
- have to be finished in 6 time units (the first by 7, the second by 17, ...)

Default phase of T_i is $\varphi_i = 0$ and default relative deadline is $d_i = p_i$

$T_2 = (10, 3, 6)$ satisfies $\varphi = 0$, $p_i = 10$, $e_i = 3$, $D_i = 6$, i.e. jobs of T_2 are

- released at times 0, 10, 20, ...,
- execute for 3 time units,
- have to be finished in 6 time units (the first by 6, the second by 16, ...)

Periodic Tasks – Notation

The 4-tuple $T_i = (\varphi_i, p_i, e_i, D_i)$ refers to a periodic task T_i with phase φ_i, period p_i, execution time e_i, and relative deadline D_i

For example: jobs of $T_1 = (1, 10, 3, 6)$ are

- released at times 1, 11, 21, $\ldots$,
- execute for 3 time units,
- have to be finished in 6 time units (the first by 7, the second by 17, ...)

Default phase of T_i is $\varphi_i = 0$ and default relative deadline is $d_i = p_i$

$T_2 = (10, 3, 6)$ satisfies $\varphi = 0$, $p_i = 10$, $e_i = 3$, $D_i = 6$, i.e. jobs of T_2 are

- released at times 0, 10, 20, $\ldots$,
- execute for 3 time units,
- have to be finished in 6 time units (the first by 6, the second by 16, ...)

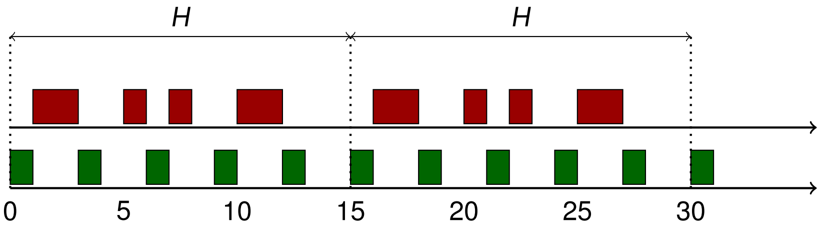
$T_3 = (10, 3)$ satisfies $\varphi = 0$, $p_i = 10$, $e_i = 3$, $D_i = 10$, i.e. jobs of T_3 are

- released at times 0, 10, 20, $\ldots$,
- execute for 3 time units,
- have to be finished in 10 time units (the first by 10, the second by 20, ...)

Periodic Tasks – Hyperperiod

The *hyper-period H* of a set of periodic tasks is the least common multiple of their periods

If tasks are in phase, then H is the time instant after which the pattern of job release/execution times starts to repeat



Aperiodic and Sporadic Tasks

- Many real-time systems are required to respond to external events

Aperiodic and Sporadic Tasks

- Many real-time systems are required to respond to external events
- The tasks resulting from such events are *sporadic* and *aperiodic* tasks
  - Sporadic tasks – hard deadlines of jobs
    e.g. autopilot on/off in aircraft

    The usual goal is to decide, whether a newly released job can be feasibly scheduled with the remaining jobs in the system
  - Aperiodic tasks – soft deadlines of jobs
    e.g. sensitivity adjustment of radar surveilance system

    The usual goal is to minimize the average response time
    For rigorous analysis we typically assume that the inter-arrival times between aperiodic jobs are distributed according to a known distribution.

Scheduling – Classification of Algorithms

- Off-line vs Online
  - Off-line – sched. algorithm is executed on the whole task set before activation
  - Online – schedule is updated at runtime every time a new task enters the system

Scheduling – Classification of Algorithms

- Off-line vs Online
  - Off-line – sched. algorithm is executed on the whole task set before activation
  - Online – schedule is updated at runtime every time a new task enters the system
- Optimal vs Heuristic
  - Optimal – algorithm computes a feasible schedule and minimizes cost of soft real-time jobs
  - Heuristic – algorithm is guided by heuristic function; tends towards optimal schedule, may not give one

The main division is on

- Clock-Driven
- Priority-Driven

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 jobs to execute for the next period, then blocks itself until the next interrupt

    E.g. the helicopter example with the interrupt every 1/180 th of a second

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 jobs to execute for the next period, then blocks itself until the next interrupt

    E.g. the helicopter example with the interrupt every 1/180 th of a second
- 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 runtime; thus scheduling overhead at run-time can be minimized
  - Simple and straight-forward, not flexible

Scheduling – Priority-Driven

- Assign priorities to jobs, based on some algorithm
- Make scheduling decisions based on the priorities, when events such as releases and job completions occur
  - Priority scheduling algorithms are *event-driven*
  - Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed

(The assignment of jobs to priority queues, along with rules such as whether preemption is allowed, completely defines a priority-driven alg.)

Scheduling – Priority-Driven

- Assign priorities to jobs, based on some algorithm
- Make scheduling decisions based on the priorities, when events such as releases and job completions occur
  - Priority scheduling algorithms are *event-driven*
  - Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed

  (The assignment of jobs to priority queues, along with rules such as whether preemption is allowed, completely defines a priority-driven alg.)
- Priority-driven algs. make *locally optimal* scheduling decisions
  - Locally optimal scheduling is often *not* globally optimal
  - Priority-driven algorithms *never* intentionally leave idle processors

Scheduling – Priority-Driven

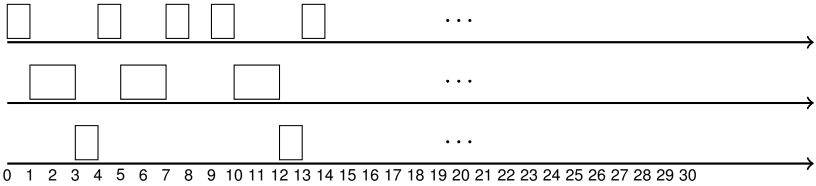
- Assign priorities to jobs, based on some algorithm
- Make scheduling decisions based on the priorities, when events such as releases and job completions occur
  - Priority scheduling algorithms are *event-driven*
  - Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed

  (The assignment of jobs to priority queues, along with rules such as whether preemption is allowed, completely defines a priority-driven alg.)
- Priority-driven algs. make *locally optimal* scheduling decisions
  - Locally optimal scheduling is often *not* globally optimal
  - Priority-driven algorithms *never* intentionally leave idle processors
- Typically in priority-driven systems:
  - Some parameters do not have to be fixed or known
  - A schedule is computed online; usually results in larger scheduling overhead as opposed to clock-driven scheduling
  - Flexible – easy to add/remove tasks or modify parameters

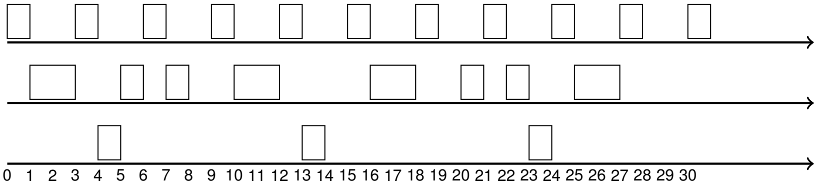
Clock-Driven & Priority-Driven Example

	T_1	T_2	T_3
p_i	3	5	10
e_i	1	2	1

Clock-Driven:



Priority-driven: $T_1 \succ T_2 \succ T_3$



Real-Time Scheduling

Scheduling of Reactive Systems

Clock-Driven Scheduling

Current Assumptions

- Fixed number, n, of periodic tasks $T_1, \ldots, T_n$

Current Assumptions

- Fixed number, n, of periodic tasks $T_1, \ldots, T_n$
- Parameters of periodic tasks are known a priori
  - Execution time $e_{i,k}$ of each job $J_{i,k}$ in a task T_i is fixed
  - For a job $J_{i,k}$ in a task T_i we have
    - $r_{i,1} = \varphi_i = 0$ (i.e., synchronized)
    - $r_{i,k} = r_{i,k-1} + p_i$

Current Assumptions

- Fixed number, n, of periodic tasks $T_1, \ldots, T_n$
- Parameters of periodic tasks are known a priori
  - Execution time $e_{i,k}$ of each job $J_{i,k}$ in a task T_i is fixed
  - For a job $J_{i,k}$ in a task T_i we have
    - $r_{i,1} = \varphi_i = 0$ (i.e., synchronized)
    - $r_{i,k} = r_{i,k-1} + p_i$
- We allow aperiodic tasks
  - assume that the system maintains a single queue for jobs of aperiodic tasks
  - Whenever the processor is available for aperiodic tasks, the job at the head of this queue is executed
- We treat sporadic tasks later

Abuse of notation: Periodic, aperiodic, sporadic jobs are jobs of periodic, aperiodic, sporadic tasks, respectively.

Static, Clock-Driven Scheduler

- Construct a *static schedule* offline
  - The schedule specifies exactly when each job executes
  - The amount of time allocated to every job is equal to its execution time
  - The schedule repeats each hyperperiod

    i.e. it suffices to compute the schedule up to hyperperiod

Static, Clock-Driven Scheduler

- Construct a *static schedule* offline
  - The schedule specifies exactly when each job executes
  - The amount of time allocated to every job is equal to its execution time
  - The schedule repeats each hyperperiod

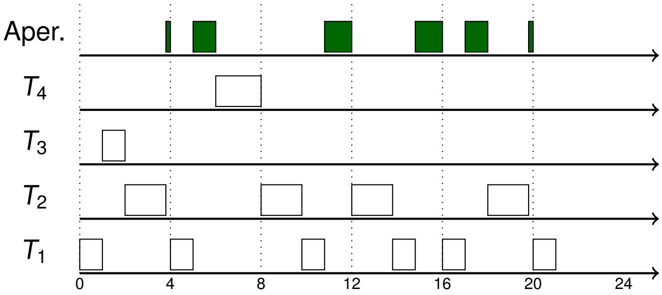
    i.e. it suffices to compute the schedule up to hyperperiod
- Can use complex algorithms offline
  - Runtime of the scheduling algorithm is not relevant
  - Can compute a schedule that optimizes some characteristics of the system

    e.g. a schedule where the idle periods are nearly periodic (useful to accommodate aperiodic jobs)

Example

$T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$, $T_4 = (20, 2)$

Hyperperiod $H = 20$



Implementation of Static Scheduler

- Store pre-computed schedule as a table
  - Each entry $(t_k, T(t_k))$ gives
    - a decision time t_k
    - scheduling decision $T(t_k)$ which is either a task to be executed, or idle (denoted by I)

Implementation of Static Scheduler

- Store pre-computed schedule as a table
  - Each entry $(t_k, T(t_k))$ gives
    - a decision time t_k
    - scheduling decision $T(t_k)$ which is either a task to be executed, or idle (denoted by I)
- The system creates all tasks that are to be executed:
  - Allocates memory for the code and data
  - Brings the code into memory

Implementation of Static Scheduler

- Store pre-computed schedule as a table
  - Each entry $(t_k, T(t_k))$ gives
    - a decision time t_k
    - scheduling decision $T(t_k)$ which is either a task to be executed, or idle (denoted by I)
- The system creates all tasks that are to be executed:
  - Allocates memory for the code and data
  - Brings the code into memory
- Scheduler sets the hardware timer to interrupt at the first decision time $t_0 = 0$

Implementation of Static Scheduler

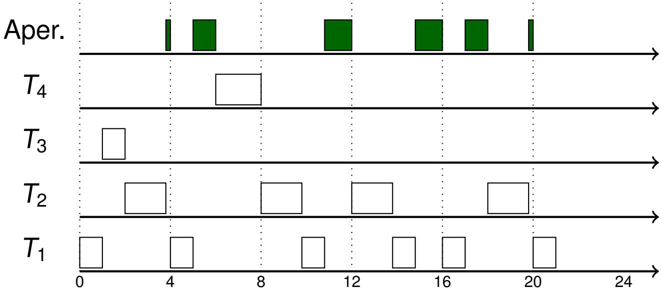
- Store pre-computed schedule as a table
  - Each entry $(t_k, T(t_k))$ gives
    - a decision time t_k
    - scheduling decision $T(t_k)$ which is either a task to be executed, or idle (denoted by I)
- The system creates all tasks that are to be executed:
  - Allocates memory for the code and data
  - Brings the code into memory
- Scheduler sets the hardware timer to interrupt at the first decision time $t_0 = 0$
- On receipt of an interrupt at t_k:
  - Scheduler sets the timer interrupt to t_{k+1}
  - If previous task overrunning, handle failure
  - If $T(t_k) = I$ and aperiodic job waiting, start executing it
  - Otherwise, start executing the next job in $T(t_k)$

k	t_k	$T(t_k)$
0	0.0	T_1
1	1.0	T_3
2	2.0	T_2
3	3.8	I
4	4.0	T_1
5	5.0	I
6	6.0	T_4
7	8.0	T_2
8	9.8	T_1
9	10.8	I
10	12.0	T_2
11	13.8	T_1
12	14.8	I
13	17.0	T_1
14	17.0	I
15	18.0	T_2
16	19.8	I

Example

$T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$, $T_4 = (20, 2)$

Hyperperiod $H = 20$



t_k	0.0	1.0	2.0	3.8	4.0	5.0	6.0	$\cdots$
$T(t_k)$	T_1	T_3	T_2	I	T_1	I	T_4	$\cdots$

Frame Based Scheduling

- Arbitrary table-driven cyclic schedules flexible, but inefficient
  - Relies on accurate timer interrupts, based on execution times of tasks
  - High scheduling overhead

Frame Based Scheduling

- Arbitrary table-driven cyclic schedules flexible, but inefficient
  - Relies on accurate timer interrupts, based on execution times of tasks
  - High scheduling overhead
- Easier to implement if a structure is imposed
  - Make scheduling decisions at periodic intervals *(frames)* of length *f*
  - Execute a fixed list of jobs within each frame; **no preemption within frames**

Frame Based Scheduling

- Arbitrary table-driven cyclic schedules flexible, but inefficient
  - Relies on accurate timer interrupts, based on execution times of tasks
  - High scheduling overhead
- Easier to implement if a structure is imposed
  - Make scheduling decisions at periodic intervals *(frames)* of length *f*
  - Execute a fixed list of jobs within each frame; **no preemption within frames**
- Gives two benefits:
  - Scheduler can easily check for overruns and missed deadlines at the end of each frame.
  - Can use a periodic clock interrupt, rather than programmable timer.

Frame Based Scheduling – Cyclic Executive

- Modify previous table-driven scheduler to be frame based
- Table that drives the scheduler has F entries, where $F = H/f$
  - The k-th entry $L(k)$ lists the names of the jobs that are to be scheduled in frame k ($L(k)$ is called *scheduling block*)
  - Each job is implemented by a procedure

Frame Based Scheduling – Cyclic Executive

- Modify previous table-driven scheduler to be frame based
- Table that drives the scheduler has F entries, where $F = H/f$
  - The k-th entry $L(k)$ lists the names of the jobs that are to be scheduled in frame k ($L(k)$ is called *scheduling block*)
  - Each job is implemented by a procedure
- Cyclic executive executed by the clock interrupt that signals the start of a frame:
  - If an aperiodic job is executing, preempts it; if a periodic overruns, handles the overrun
  - Determines the appropriate scheduling block for this frame
  - Executes the jobs in the scheduling block
  - Executes jobs from the head of the aperiodic job queue for the remainder of the frame

Frame Based Scheduling – Cyclic Executive

- Modify previous table-driven scheduler to be frame based
- Table that drives the scheduler has F entries, where $F = H/f$
  - The k-th entry $L(k)$ lists the names of the jobs that are to be scheduled in frame k ($L(k)$ is called *scheduling block*)
  - Each job is implemented by a procedure
- Cyclic executive executed by the clock interrupt that signals the start of a frame:
  - If an aperiodic job is executing, preempts it; if a periodic overruns, handles the overrun
  - Determines the appropriate scheduling block for this frame
  - Executes the jobs in the scheduling block
  - Executes jobs from the head of the aperiodic job queue for the remainder of the frame
- Less overhead than pure table driven cyclic scheduler, since only interrupted on frame boundaries, rather than on each job

Frame Based Scheduling – Frame Size

How to choose the frame length?
(Assume that periods are in $\mathbb{N}$ and choose frame sizes in $\mathbb{N}$.)

1. Necessary condition for avoiding preemption of jobs is

   $$f \geq \max_i e_i$$

   (i.e. we want each job to have a chance to finish within a frame)

Frame Based Scheduling – Frame Size

How to choose the frame length?
(Assume that periods are in $\mathbb{N}$ and choose frame sizes in $\mathbb{N}$.)

1. Necessary condition for avoiding preemption of jobs is

   $$f \geq \max_i e_i$$

   (i.e. we want each job to have a chance to finish within a frame)

2. To minimize the number of entries in the cyclic schedule, the hyper-period should be an integer multiple of the frame size, i.e.

   $$\exists i : p_i \mod f = 0$$

Frame Based Scheduling – Frame Size

How to choose the frame length?
(Assume that periods are in $\mathbb{N}$ and choose frame sizes in $\mathbb{N}$.)

1. Necessary condition for avoiding preemption of jobs is

   $$f \geq \max_i e_i$$

   (i.e. we want each job to have a chance to finish within a frame)

2. To minimize the number of entries in the cyclic schedule, the hyper-period should be an integer multiple of the frame size, i.e.

   $$\exists i : p_i \mod f = 0$$

3. To allow scheduler to check that jobs complete by their deadline, at least one frame should lie between release time of a job and its deadline, which is equivalent to

   $$\forall i : 2 * f - gcd(p_i, f) \leq D_i$$

All three constraints should be satisfied.

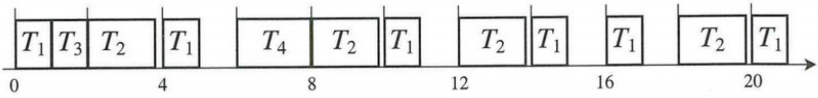
Frame Based Scheduling – Frame Size – Example

1. $f \geq \max_i e_i$
2. $\exists i : p_i \mod f = 0$
3. $\forall i : 2 * f - gcd(p_i, f) \leq D_i$

Example 12

$T_1 = (4, 1.0)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1.0)$, $T_4 = (20, 2.0)$
Then $f \in \mathbb{N}$ satisfies 1.–3. iff $f = 2$.

With $f = 2$ is schedulable:



Frame Based Scheduling – Job Slices

- Sometimes a system cannot meet all three frame size constraints simultaneously (and even if it meets the constraints, no non-preemptive schedule is feasible)

Frame Based Scheduling – Job Slices

- Sometimes a system cannot meet all three frame size constraints simultaneously (and even if it meets the constraints, no non-preemptive schedule is feasible)
- Can be solved by partitioning a job with large execution time into slices with shorter execution times

  This, in effect, allows preemption of the large job

Frame Based Scheduling – Job Slices

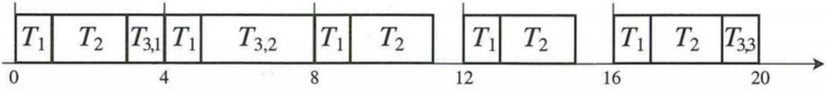
- Sometimes a system cannot meet all three frame size constraints simultaneously (and even if it meets the constraints, no non-preemptive schedule is feasible)
- Can be solved by partitioning a job with large execution time into slices with shorter execution times

  This, in effect, allows preemption of the large job

- Consider $T_1 = (4, 1)$, $T_2 = (5, 2, 7)$, $T_3 = (20, 5)$
- Cannot satisfy constraints: 1. $\Rightarrow f \geq 5$ but 3. $\Rightarrow f \leq 4$
- Solve by splitting T_3 into $T_{3,1} = (20, 1)$, $T_{3,2} = (20, 3)$, and $T_{3,3} = (20, 1)$

  (Other splits exist)

- Result can be scheduled with $f = 4$



Building a Structured Cyclic Schedule

To construct a schedule, we have to make three kinds of design decisions (that cannot be taken independently):

- Choose a frame size based on constraints
- Partition jobs into slices
- Place slices into frames

There are efficient algorithms for solving these problems based e.g. on a reduction to the network flow problem.

Scheduling Aperiodic Jobs

So far, aperiodic jobs scheduled in the background after all jobs with hard deadlines

This may unnecessarily delay aperiodic jobs

Scheduling Aperiodic Jobs

So far, aperiodic jobs scheduled in the background after all jobs with hard deadlines

This may unnecessarily delay aperiodic jobs

Note: There is no advantage in completing periodic jobs early

Ideally, finish periodic jobs by their respective deadlines.

Scheduling Aperiodic Jobs

So far, aperiodic jobs scheduled in the background after all jobs with hard deadlines

This may unnecessarily delay aperiodic jobs

Note: There is no advantage in completing periodic jobs early

Ideally, finish periodic jobs by their respective deadlines.

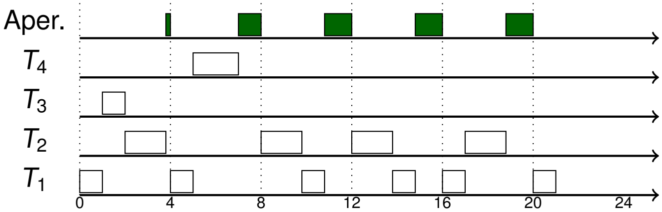
Slack Stealing:

- Slack time in a frame = the time left in the frame after all (remaining) slices execute
- Schedule aperiodic jobs ahead of periodic in the slack time of periodic jobs
  - The cyclic executive keeps track of the slack time left in each frame as the aperiodic jobs execute, preempts them with periodic jobs when there is no more slack
  - As long as there is slack remaining in a frame and the aperiodic jobs queue is non-empty, the executive executes aperiodic jobs, otherwise executes periodic
- Reduces resp. time for aper. jobs, but requires accurate timers

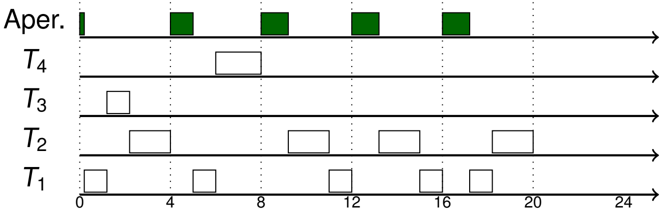
Example

Assume that the aperiodic queue is never empty.

Aperiodic at the ends of frames:



Slack stealing:



Frame Based Scheduling – Sporadic Jobs

Let us allow sporadic jobs

i.e. hard real-time jobs whose release and exec. times are not known a priori

Frame Based Scheduling – Sporadic Jobs

Let us allow sporadic jobs

i.e. hard real-time jobs whose release and exec. times are not known a priori

The scheduler determines whether to accept a sporadic job when it arrives (and its parameters become known)

- Perform *acceptance test* to check whether the new sporadic job can be feasibly scheduled with all the jobs (periodic and sporadic) in the system at that time

  Acceptance check done at the beginning of the next frame; has to keep execution times of the parts of sporadic jobs that have already executed
- If there is sufficient slack time in the frames before the new job's deadline, the new sporadic job is accepted; otherwise, rejected
- Among themselves, sporadic jobs scheduled according to EDF

  This is optimal for sporadic jobs

Frame Based Scheduling – Sporadic Jobs

Let us allow sporadic jobs

i.e. hard real-time jobs whose release and exec. times are not known a priori

The scheduler determines whether to accept a sporadic job when it arrives (and its parameters become known)

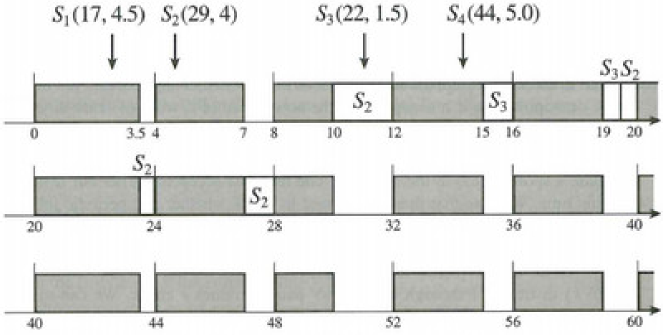
- Perform *acceptance test* to check whether the new sporadic job can be feasibly scheduled with all the jobs (periodic and sporadic) in the system at that time

  Acceptance check done at the beginning of the next frame; has to keep execution times of the parts of sporadic jobs that have already executed
- If there is sufficient slack time in the frames before the new job's deadline, the new sporadic job is accepted; otherwise, rejected
- Among themselves, sporadic jobs scheduled according to EDF

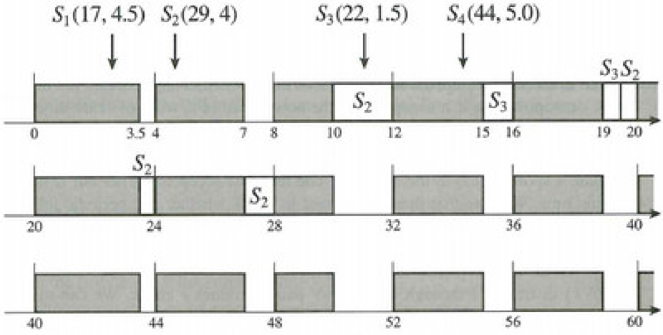
  This is optimal for sporadic jobs

Note: rejection is often better than missing deadline

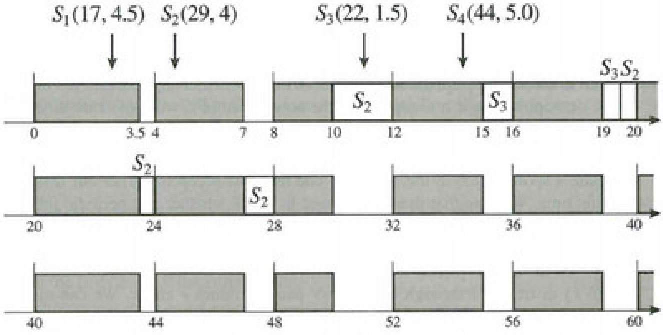
e.g. a robotic arm taking defective parts off a conveyor belt: if the arm cannot meet deadline, the belt may be slowed down or stopped



- $S_1(17, 4.5)$ released at 3 with abs. deadline 17 and execution time 4.5; acceptance test at 4; must be scheduled in frames $2, 3, 4$; total slack in these frames is 4, i.e. rejected

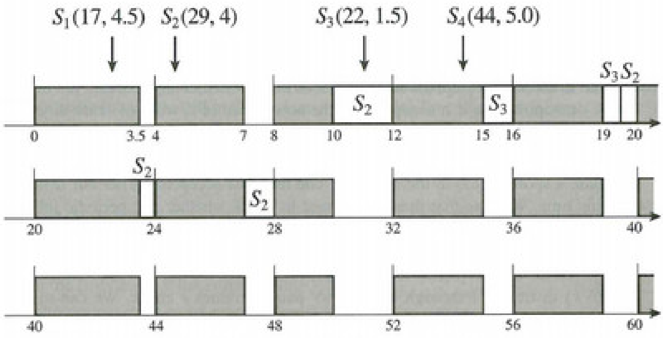


- $S_1(17, 4.5)$ released at 3 with abs. deadline 17 and execution time 4.5; acceptance test at 4; must be scheduled in frames $2, 3, 4$; total slack in these frames is 4, i.e. rejected
- $S_2(29, 4)$ released at 5 with abs. deadline 29 and exec. time 4; acc. test at 8; total slack in frames 3-7 is 5.5, i.e. accepted



- $S_1(17, 4.5)$ released at 3 with abs. deadline 17 and execution time 4.5; acceptance test at 4; must be scheduled in frames 2, 3, 4; total slack in these frames is 4, i.e. rejected
- $S_2(29, 4)$ released at 5 with abs. deadline 29 and exec. time 4; acc. test at 8; total slack in frames 3-7 is 5.5, i.e. accepted
- $S_3(22, 1.5)$ released at 11 with abs. deadline 22 and exec. time 1.5; acc. test at 12;

  2 units of slack in frames 4, 5 as S_3 will be executed *ahead of the remaining parts of* S_2 by EDF – check whether there will be enough slack for the remaining parts of S_2, accepted



- $S_1(17, 4.5)$ released at 3 with abs. deadline 17 and execution time 4.5; acceptance test at 4; must be scheduled in frames $2, 3, 4$; total slack in these frames is 4, i.e. rejected
- $S_2(29, 4)$ released at 5 with abs. deadline 29 and exec. time 4; acc. test at 8; total slack in frames 3-7 is 5.5, i.e. accepted
- $S_3(22, 1.5)$ released at 11 with abs. deadline 22 and exec. time 1.5; acc. test at 12;

  2 units of slack in frames $4, 5$ as S_3 will be executed *ahead of the remaining parts of* S_2 by EDF – check whether there will be enough slack for the remaining parts of S_2, accepted
- $S_4(44, 5.0)$ is rejected (only 4.5 slack left)

Handling Overruns

Overruns may happen due to failures

e.g. unexpectedly large data over which the system operates, hardware failures, etc.

Handling Overruns

Overruns may happen due to failures

e.g. unexpectedly large data over which the system operates, hardware failures, etc.

Ways to handle overruns:

- Abort the overrun job at the beginning of the next frame; log the failure; recover later

  e.g. control law computation of a robust digital controller
- Preempt the overrun job and finish it as an aperiodic job

  use this when aborting job would cause "costly" inconsistencies
- Let the overrun job finish – start of the next frame and the execution jobs scheduled for this frame are delayed

  This may cause other jobs to be delayed

  depends on application

Clock-drive Scheduling: Conclusions

Advantages:

- Conceptual simplicity
  - Complex dependencies, communication delays, and resource contention among jobs can be considered when constructing the static schedule
  - Entire schedule in a static table
  - No concurrency control or synchronization needed
- Easy to validate, test and certify

Clock-drive Scheduling: Conclusions

Advantages:

- Conceptual simplicity
  - Complex dependencies, communication delays, and resource contention among jobs can be considered when constructing the static schedule
  - Entire schedule in a static table
  - No concurrency control or synchronization needed
- Easy to validate, test and certify

Disadvantages:

- Inflexible
  - If any parameter changes, the schedule must be usually recomputed

    Best suited for systems which are rarely modified (e.g. controllers)
  - Parameters of the jobs must be fixed

    As opposed to most priority-driven schedulers

Real-Time Scheduling

Scheduling of Reactive Systems

Priority-Driven Scheduling

Current Assumptions

- Single processor
- Fixed number, *n*, of *independent periodic* tasks
  i.e. there is no dependency relation among jobs
  - Jobs can be preempted at any time and never suspend themselves
  - No aperiodic and sporadic jobs
  - No resource contentions

Current Assumptions

- Single processor
- Fixed number, n, of *independent periodic* tasks

  i.e. there is no dependency relation among jobs

  - Jobs can be preempted at any time and never suspend themselves
  - No aperiodic and sporadic jobs
  - No resource contentions

Moreover, unless otherwise stated, we assume that

- **Scheduling decisions take place precisely at**
  - release of a job
  - completion of a job

  (and nowhere else)
- Context switch overhead is negligibly small

  i.e. assumed to be zero
- There is an unlimited number of priority levels

Fixed-Priority vs Dynamic-Priority Algorithms

A priority-driven scheduler is on-line
i.e. it does not precompute a schedule of the tasks

- It assigns priorities to jobs after they are released and places the jobs in a ready job queue in the priority order

  with the highest priority jobs at the head of the queue

- At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue

  i.e. one of the jobs with the highest priority

Fixed-Priority vs Dynamic-Priority Algorithms

A priority-driven scheduler is on-line
i.e. it does not precompute a schedule of the tasks

- It assigns priorities to jobs after they are released and places the jobs in a ready job queue in the priority order

  with the highest priority jobs at the head of the queue

- At each scheduling decision time, the scheduler updates the ready job queue and then schedules and executes the job at the head of the queue

  i.e. one of the jobs with the highest priority

Fixed-priority = *all jobs in a task* are assigned the same priority

Dynamic-priority = jobs in a task may be assigned different priorities

Note: In our case, a priority assigned to a job does not change. There are *job-level dynamic priority* algorithms that vary priorities of individual jobs – we won't consider such algorithms.

Fixed-priority Algorithms – Rate Monotonic

Best known fixed-priority algorithm is *rate monotonic (RM)* scheduling that assigns priorities to tasks based on their periods

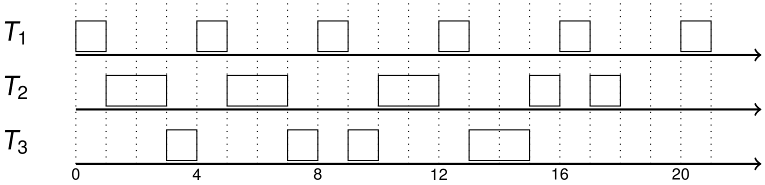
- The shorter the period, the higher the priority
- The *rate* is the inverse of the period, so jobs with higher rate have higher priority

RM is very widely studied and used

Example 13

$T_1 = (4, 1)$, $T_2 = (5, 2)$, $T_3 = (20, 5)$
with rates 1/4, 1/5, 1/20, respectively

The priorities: $T_1 \succ T_2 \succ T_3$



Fixed-priority Algorithms – Deadline Monotonic

The *deadline monotonic (DM)* algorithm assigns priorities to tasks based on their *relative deadlines*

- the shorter the deadline, the higher the priority

Fixed-priority Algorithms – Deadline Monotonic

The *deadline monotonic (DM)* algorithm assigns priorities to tasks based on their *relative deadlines*

- the shorter the deadline, the higher the priority

Observation: When relative deadline of every task matches its period, then RM and DM give the same results

Proposition 1

When the relative deadlines are arbitrary DM can sometimes produce a feasible schedule in cases where RM cannot.

Proof.

Consider e.g. $T_1 = (3, 1, 1)$ and $T_2 = (2, 1)$. □

Dynamic-priority Algorithms

Best known is *earliest deadline first (EDF)* that assigns priorities based on *current* (absolute) deadlines

- At the time of a scheduling decision, the job queue is ordered by earliest deadline

Dynamic-priority Algorithms

Best known is *earliest deadline first (EDF)* that assigns priorities based on *current* (absolute) deadlines

- At the time of a scheduling decision, the job queue is ordered by earliest deadline

Another one is the *least slack time (LST)*

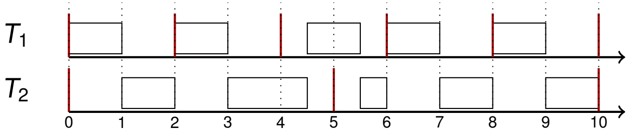
- The job queue is ordered by least slack time

Recall that the *slack time* of a job J_i at time t is equal to $d_i - t - x$ where x is the remaining computation time of J_i at time t

We focus on EDF here.

EDF – Example

$T_1 = (2, 1)$ and $T_2 = (5, 2.5)$



Note that the processor is 100% "utilized", not surprising :-)