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 whiteboard (⇒ attend the lectures)

Homework projects:

- a larger project, most probably with LEGO mindstorms
- (you may also be assigned a short (and easy) theoretical homework)

Evaluation:

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

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.

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)
- Medical
  - pacemaker, medical monitoring devices
  - robots used to move radioactive materials
- Transportation systems
  - computers in cars (ABS, MPFI, cruise control, airbag ...)
  - aircraft (FMS, fly-by-wire ...)
- Military applications
  - controllers in weapons, missiles, fighter aircraft, ...
  - radar and sonar tracking
- Multimedia – video telephony, multimedia center, videoconferencing
- ...

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

There are real time systems that are not embedded:

- stock market
- 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

- 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?

  ... 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

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

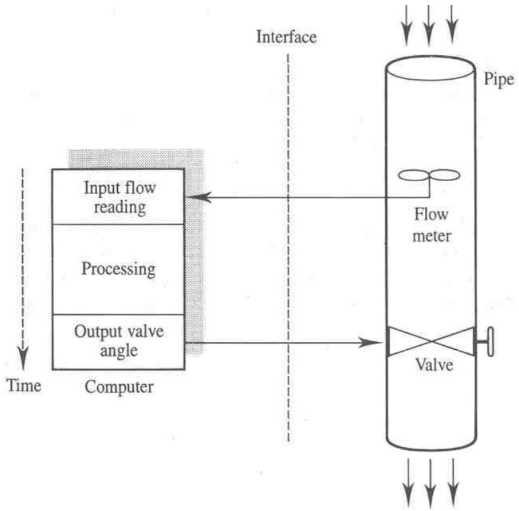
Many real-time systems combine “hard” and “soft” real-time tasks.

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

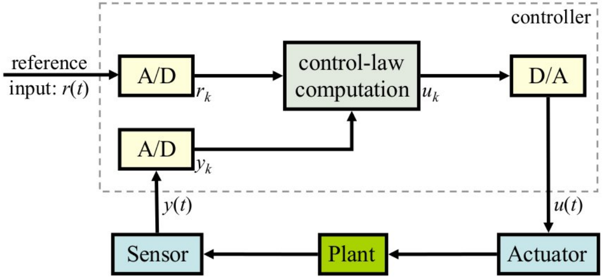
- Digital process control
  - anti-lock braking system
  - multi-point fuel injection
- Higher-level command and control
  - helicopter flight control
- Real-time databases
  - Stock trading systems

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

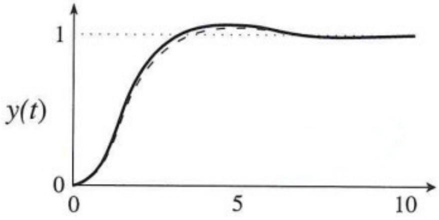
- Pseudo-code for the controller:

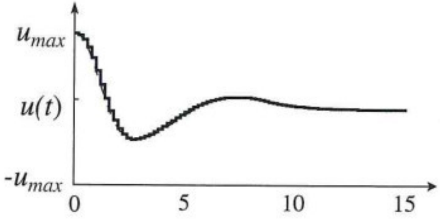
```
set timer to interrupt periodically with period $T$
foreach timer interrupt do
    analogue-to-digital conversion of $y(t)$ to get $y_k$
    compute control output $u_k$ based on $r_k$ and $y_k$
    digital-to-analogue conversion of $u_k$ to get $u(t)$
end
```

- 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

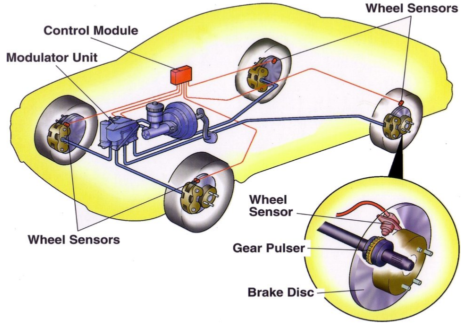
    ... but may result in unstable control

Example





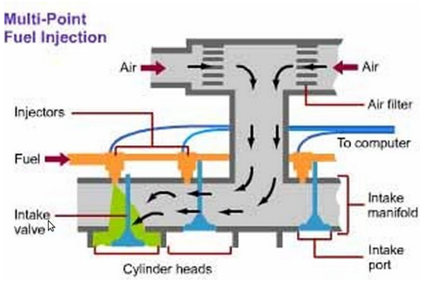
Anti-Lock Braking System



- 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

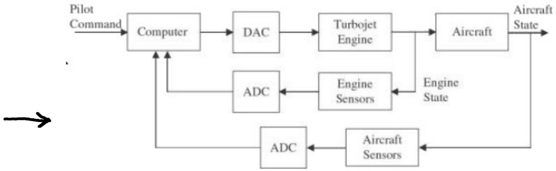
Multi-Point Fuel Injection (MPFI)



Multi-Point = fuel is injected to individual cylinders

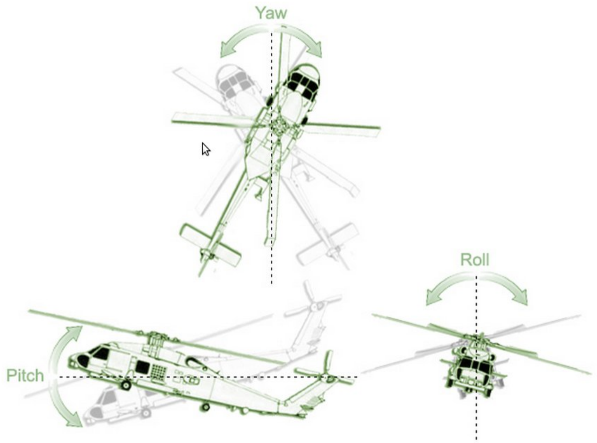
- The controller monitors throttle valve
  whenever the valve opens up more, fuel rate is increased
  (in anticipation of more air entering the engine)
- The fuel rate must increase as soon as the throttle valve opens
- The controller determines when and how much fuel is injected
  The algorithm is quite complicated and depends on data from various sensors

Multi-rate Digital Process Control



- A plant usually has more than one degree of freedom
  - more state variables
    (e.g. rotation speed and temperature of an engine)
  - multiple sensors and multiple actuators
- Sampling periods may be different for different variables
  sampling rate for RPM (rotations per minute) is higher than for temperature measurements
- Often sampling periods are chosen in harmonic way
  i.e. each longer period is an integer multiple of every shorter period
  - easier to implement
  - advantage from the standpoint of achievable processor utilization

Multi-Rate DPC – Helicopter Flight Control



There are also three velocity components

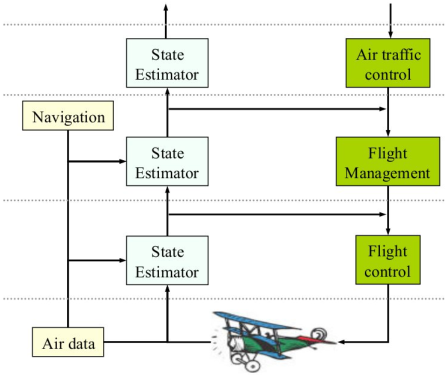
Two control loops: pilot's control (30Hz) and stabilization (90Hz)

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
- 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)

By The Way ...

The following BMW 745i

- approx. 2 000 000 lines of code,
- over 60 microprocessors,

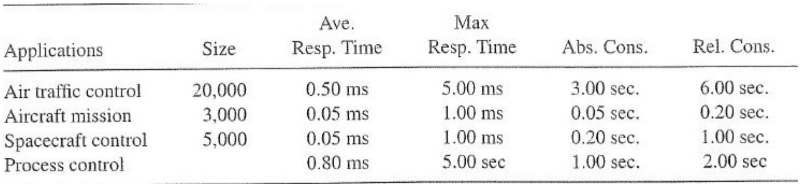
  53 x 8-bit, 11 x 32-bit, 7 x 16-bit
- multiple networks,
- controlled by Windows CE OS.

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

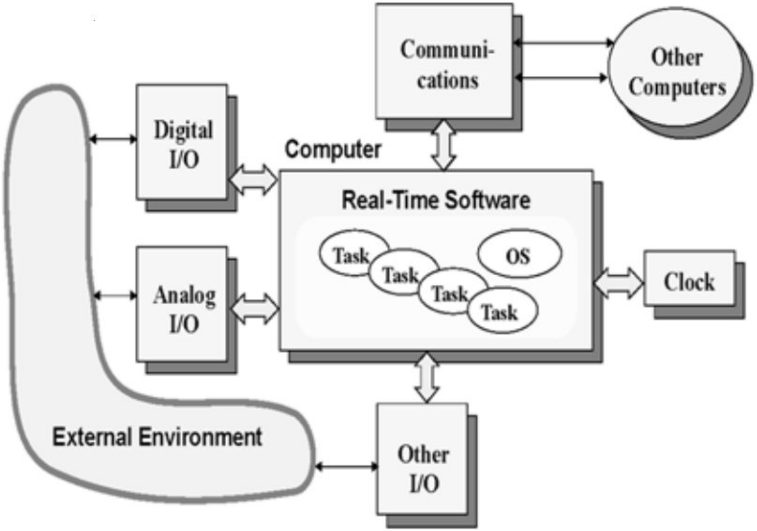
- A system for selling/buying stock at public prices
- Prices are volatile in their movement
- Stop orders:
  - set upper limit on prices for buying – buy for the best available price once the limit is reached

    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
- 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:

- 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

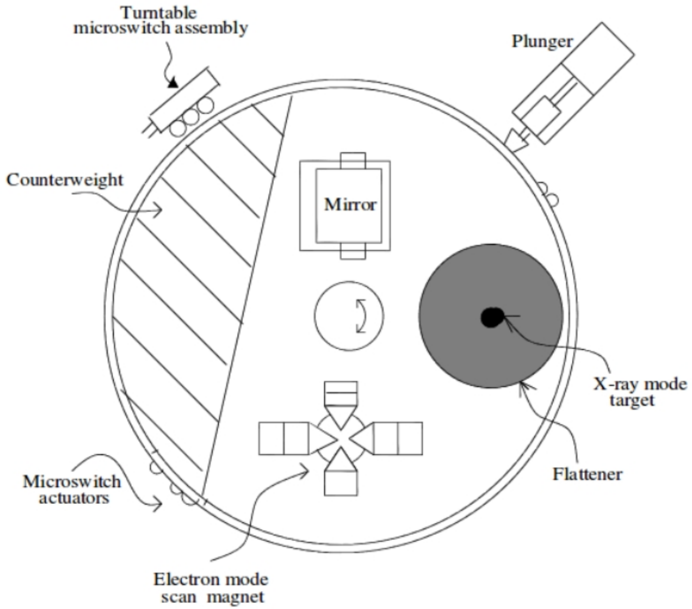
- between 1985 and 1987 (at least) six accidents involving enormous radiation overdoses to patients
- Half of these patients died due to the overdoses

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
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
- 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
- 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

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.

What happened?

Another incident proceeded as follows (again simplified):

- The operator used the light mode to check the patient's position when the treatment parameters had already been entered
- Then issued a "set" command, supposed to change the turntable position for the selected treatment
- Afterwards, the system displayed "beam ready" and the operator turned the beam on
- The patient received a "treatment" with the turntable in the light position

The cause:

- The set command rotates the turntable only when a shared variable CLASS3 is non-zero, while it is constantly being incremented during the checking phase
- However, the variable is only 8bit, so every 256th pass, the routine incorrectly thinks that everything is set
- The operator issued the "set" command precisely when CLASS3=0

Patriot missile mistiming

vs

Patriot missile mistiming

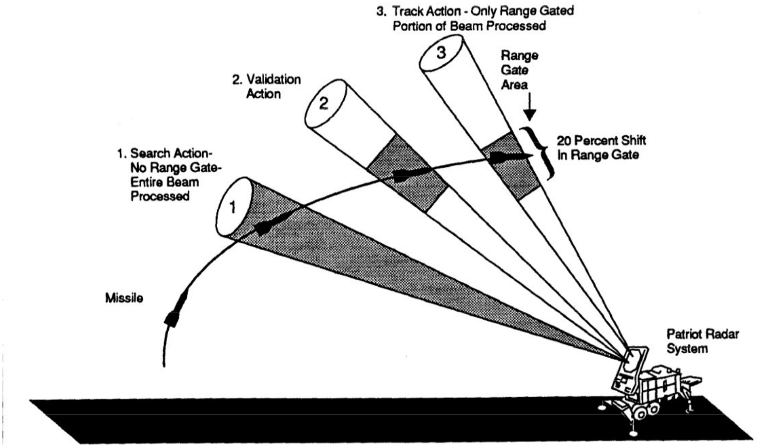
- Patriot – Air defense missile system
- Failed to intercept a scud missile on February 25, 1991 at Dhahran, Saudi Arabia

  (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



3. Track Action - Only Range Gated Portion of Beam Processed
3
Range Gate Area
2. Validation Action
2
20 Percent Shift In Range Gate
1. Search Action- No Range Gate- Entire Beam Processed
1
Missile
Patriot Radar System

Patriot Missile Mistiming

Prediction of the new area:

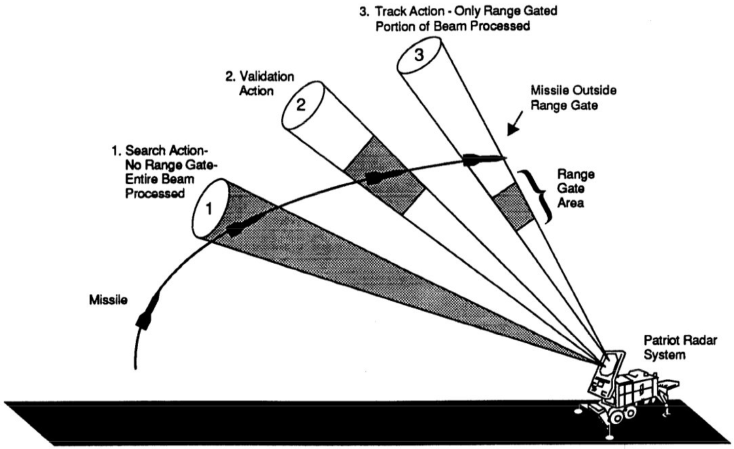
- a function of *velocity* and *time* of the last radar detection
- velocity represented as a real number
- **the current time kept by incrementing whole number counter counting tenths of seconds**
- computation in 24bit fixed floating point numbers

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



3. Track Action - Only Range Gated Portion of Beam Processed
3
2. Validation Action
2
Missile Outside Range Gate
1. Search Action- No Range Gate- Entire Beam Processed
1
Range Gate Area
Missile
Patriot Radar System

(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
  - Real-time programming languages

Outline – Scheduling

The Scheduling problem:

Input:

- available processors, resources
- set of tasks/jobs

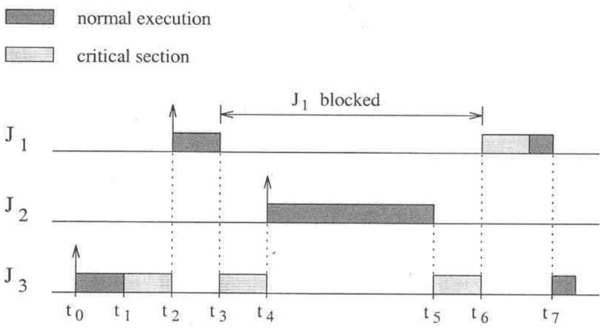
  with their requirements, deadlines, etc.

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

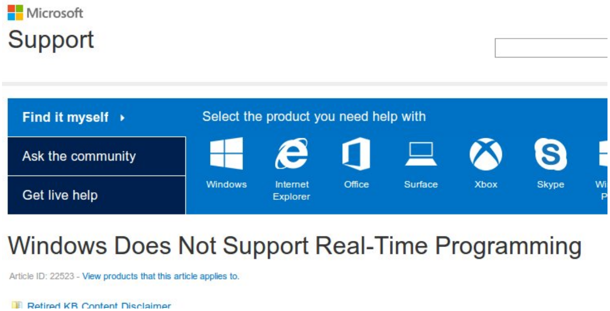
  offline and online algorithms, RM, EDF, etc.

Outline – Scheduling – Puzzle

- toast: each side for 2 minutes on a pan
- pan: at most two toasts at a time

What is the minimum time to make three toasts?

Outline – Programming



Windows Does Not Support Real-Time Programming

Article ID: 22523 - View products that this article applies to.

Retired KB Content Disclaimer

Basic information about RTOS and RT programming languages

- RTOS – overview (freeRTOS, RTLinux, VxWorks)
  - real-time in non-real-time operating systems
  - **implementation of theoretical concepts in freeRTOS**
- RT programming languages – short overview (Ada, C, Java)
  - **implementation of theor. concepts in real-time Java**

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
- 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

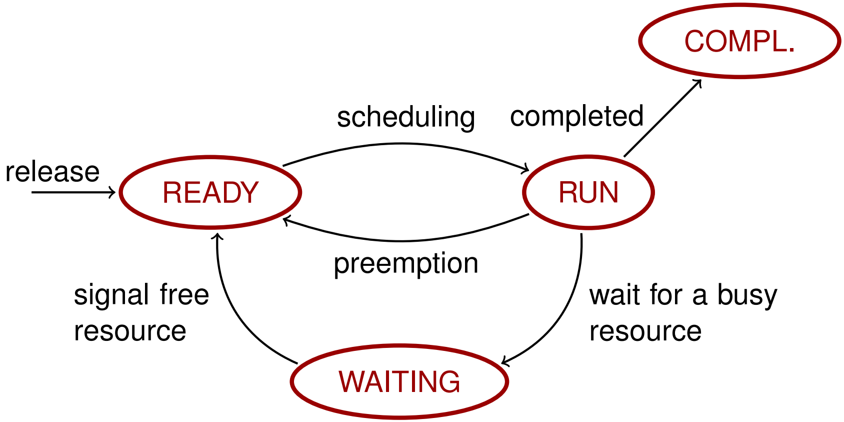
  compute a control law, transform sensor data, etc.
- A *task* is a set of related jobs which jointly provide some system function

  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.

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

- 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 $e_i = e_i^+$

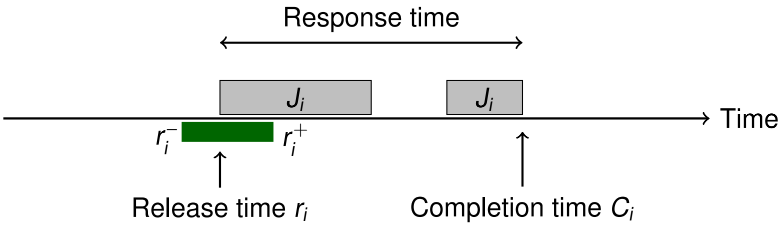
Job Parameters – Release and Response Time

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

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

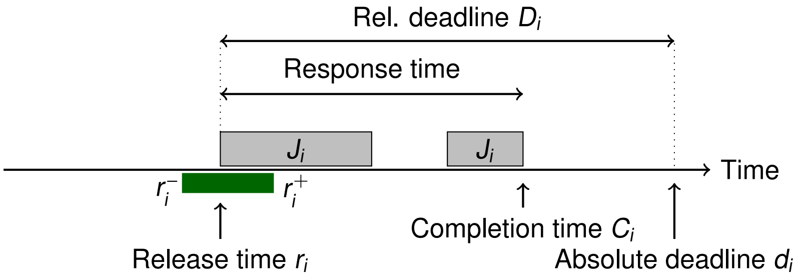


Job Parameters – Deadlines

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

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.

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.

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

- 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

- 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

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

The general case considered in literature:

m processors $P_1, \dots, P_m$, 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

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 $0 \leq t_1 \leq t \leq t_2$ 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*

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

Sometimes efficiency of scheduling algorithms is measured using a *cost function*:

- the maximum/average response time
- the maximum/average lateness – the difference between the completion time and the absolute deadline, i.e. $C_i - d_i$
- miss rate – the percentage of jobs that are executed but completed too late
- ...

Definition 7
A 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.

Real-Time Scheduling

Individual Jobs

Scheduling of Individual Jobs

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

Each J_i has a release time r_i, an execution time e_i and a relative 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? Minimize maximal lateness.

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) and minimizes the maximal lateness (always).

Proof.

Any feasible schedule σ can be transformed in finitely many steps to EDD schedule whose lateness is $\leq$ the lateness of σ (whiteboard). □

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
- determine lateness of each job in your schedule (is the maximal lateness minimized?)

  Recall that lateness of J_i is equal to $C_i - d_i$

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, Dependent (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) and minimizes the maximal lateness.

Proof.

Any feasible schedule σ can be transformed in finitely many steps to EDF schedule which is feasible and whose lateness is $\leq$ the lateness of σ (whiteboard). $\square$

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)

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.

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)

	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:

$$\begin{tikzcd}
 & & J_1 \arrow[dl] \arrow[dr] & \\
 & J_2 \arrow[dl] \arrow[dr] & & J_3 \arrow[d] \\
J_4 & & J_5 & J_6
\end{tikzcd}$$

Find a feasible schedule.

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*, which every job acquires when its execution starts and releases *R* when execution is complete – causes no preemption

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