Real-Time & Embedded System

Real-Time System - Intro

- Definition
  - logical correctness
    - accuracy of result is with respect to specification
  - predictable timing
    - the time of result delivery is with respect to specification
- Typical characteristics:
  - Adherence to set time constraint
  - **Predictability**
    - repeatable result in terms of timing and value
  - Fault Tolerance
    - robustness in the presence of forseeable fault
  - Accuracy
  - Other common features:
    - concurrent, distributed and complex hardware
    - frequently evaluated as part of pyhsical system
- Real-Time Operating System
  - Predictability
  - Passivity
  - Small footprint
  - Instrumentation
    - Provides sensors, converters etc. to perceive environment
  - Usually integret into harware & environment (no operating system)

    $\Rightarrow$ Embedded System
- Real-Time Languages / Environments
  - Predictability
    - Always forseeable timing behaviour
  - Time
    - Specified granularity, operations based on time, scheduling
  - High Integrity
    - Fault detecting as early as possible
  - Concurrency & Distribution
    - Solid & high-level synchronization and communication primitives

    - Soecific & Scaling
        - Mapping physical interfaces into high-level concept and data types
- Comparison: Real-Time Languages vs. Real-Time OS vs. Libraries
    - Real-Time Languages:
        - Compiler level check
        - Runtime environment acting as Real-Time OS
    - Real-Time Operating System:
        - Loss: Complier level check

          (Can use equivalent tools => need additional language as it needs specification as well)
        - Still have: Scheduling, interrupt handling (migrating from runtime environment to OS)
    - Libraries:
        - Loss: Compiler level check; Block structure & Scoping
- Comparison: Real-Time & Embedded OS vs. General OS
    - Real-Time & Embedded:
        - memory: pre-allocated; predictable usage
        - scheduling: (usually) higher prority task absolutely run first & finish first
    - General:
        - memory: dynamic allocation
        - scheduling: focus more on fairness

Real-Time Languages

- Criteria
    - Predictability
    - Highly integrity: express and understand time

1. General Features of Languages / Systems

- Transformational ↔ Interactive ↔ Reactive
    - Transformational (functional)
        - Generating output based on input and then stop
        - Small number of internal states
    - Interactive
        - Server systems in long-term operation
        - Request occasional input
        - Accept request when resource available
    - Reactive (reflex)
        - React to external stimuli only and output stimuli
        - Similiar to, predictable functional system with always enough resources to ensure specified reaction time
- Dataflow Oriented ↔ Control Oriented

    - Dataflow
        - Continuous data-streams (Independent dataflow + functional processing)

          ⇒ high bandwidth
    - Control
        - Discrete signals controlling data-streams and processes

          ⇒ low bandwidth
- Causality and Synchronous
    - Synchronous
        - instantaneous (zero delay assumption)
        - **iff** total worst case computing time is smaller than the minimal time between two observable change in the environment

          ⇒ assum a logical & discret time

          ⇒ Easier verification, hardware implementation; Stronger analysis
    - Causality
        - Future does NOT influence the past
        - Current state completely depends on the past
    - Causal synchronous program
        - Reactive ⇒ well-defined output for each signal sequence
        - Deterministic ⇒ exactly one out put for each signal sequence
    - Cyclic Dependecy (in Time):
        - Result in potential causality problem in synchronous language
        - Cyclic programs can be reactive & deterministic (thus, causal synchronous)
- Physical coupling
    - Into embedded system ⇒ physicahl coupling as component of system

2. Real-Time Languages Example

- Ada
    - Features
        - Concurrency & Synchornization (shared memory & message passing)
        - Strong runtime environment

          ⇒ stand-alone runtime environment for embedded system
        - Maintainability, high-integrity and efficiency
        - Generic on the level of package
        - OO concepts

          ⇒ 'synchronizd interface' (abstract) allows only 1-step inheritance
    - High-integrity ⇒ Standardized language-annexes as part of language (Ex. Ravenscar)
        - Real-Time features
        - Distributed programming

    - Capability
        - General workhours
- Real-Time Java - Specific Java engine with class libraries
    - Disadvantage:
        - Underlying object orientation design

          ⇒ Predictability questionable

          ⇒ special concern for inheritance
        - Libraries usages destroies hard real-time integrity
        - Different Java engine implementation allowed
    - Capability
        - Very soft real-time application
- Esterel - Control-dominated reactive system
    - Features
        - Synchronous (zero delay assumption)
    - **"Zero delay" Assumption**
        - Logical term: All operations are instantaneous
        - Pyhsical term: No observable delay between stimuli and reaction
        - Comp sci term: All operations are finished before next input sampling
    - Capability
        - Alternative for high-integrity application
- VHDL - Hardware description language
    - Features
        - Can be data-flow or control-flow oriented
        - Programming in large ⇒ mapping hardware to high-level concept
        - High concurrency & synchronization primitives
        - Module can be clock differently, or not at all
        - Compile real-time dataflow and independent, asynchronous control path into hardware
- Timed CSP - process algebras
    - Features
        - Events are instantaneous
        - Unlimited concurrency
        - All communications are synchronous
        - Allows alebraic transformation to prove execution traces can/cannot happen
- PEARL (Process and Experiment Automation Real-Time Language)
    - Features
        - Established standard
        - Time as part of language (keywords)
    - Disadvantages
        - Synchronization on semaphore level

        - Designed for small scale
- POSIX
    - Disadvantages
        - Rely only on libraries calls
        - Signals instead of interrupts
- C
    - Features
        - Popular
        - Fast even without compiler optimization
        - Small footprint
    - Disadvantages
        - No abstraction for large system
        - Can NOT take advantage of hardware specific feature (as Assemblers do)
- Assemblers
    - Features:
        - Predictable result (as predictable as underlying hardware)
        - Closest to hardware
        - Small footprint
    - Disadvantages
        - No abstraction for large system
        - Hard to read (messy)

Physical Coupling

- Definition
    - Transform one dominant phenomenon into one eletrical signal
- Example: Measuring Temperature
    - Observable Effect caused by Temperature Changes:
        - Noise voltage change
        - Volum change (Ex.gas, liquid, metals)
        - Thermovoltage change (热电压)
        - Change in conductor (导体) / semiconductor
        - State change (gas, liquid, solid)
    - Analysis on Different Phenomenon (when used for measuring)
        - Thermocouple

          Pros: accept high temperature; small size; relatively cheap

          Cons: need stable amplifier (放大器), constraint on cables
        - Thermoresistor

          Pros: potentially higher accuracy; more linear; long-term stability; absolute temperature

          Cons: limited range; slow response time; heating itself up by electricity; big; expensive

        - Temperature sensitive Semi-conductor

          Pros: cheap; sensitive; long-term stability (some)

          Cons: Further limited range; strongly non-linear; larger; general inaccurate & instable
        - Noise Temperature Measurement (Noise Voltage)

          Pros: linear; highly accurate; long-term stability; wide temperature range;

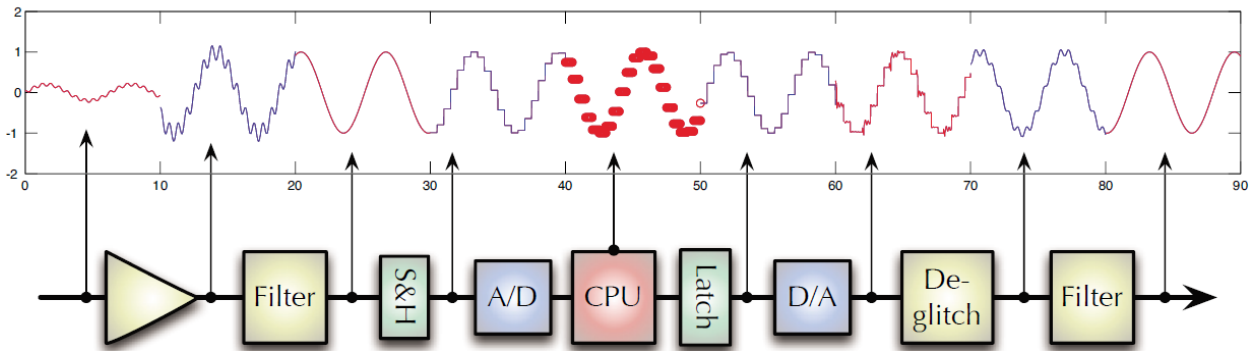
          Cons: expensive; large; complex amlifier requested
- Example: Measuring Range & Speed
    - Range:
        - By triangulation: not suited for randomly curved surface
        - By time of flight/phase/light: linear; resolution depends on detecting signals
    - Speed:
        - Doppler current profilers
- Effect on Realt-Time Embedded System:
    - Need to understand the loss / accuracy
        - especially when analyzing minimum / maximum error
    - Need to understand the meaning of value from hardwares

Interface

- Note: convert sampled value into valid data signal (bits)

1. Signal Processing Example - A/D-D/A

- Overview



- Processing Steps:
    - Original Signal: weak, noise-, interference-affected (carries additional higher freqency signals)

      ↓ Amplifier
    - Amplified Signal: strong, but still, noise-, interference-affected

      ↓ Filter
    - Filtered ("low passed") Signal: higher frequency eliminated - pre-cond for next stage

      ↓ S&H (Sample-and-Hold)
    - Signal after Sample-and-Hold: samples are taken over short time-span, and held until next sample

↓ A/D

- Bits (in CPU): discrete values inside CPU, quantized representation

  ↓ Latch

- Signal after Latch: output constant voltage over small time-span corresponding to value in CPU

  ↓ D/A

- Analogue Signal after D/A: glitches-affected (due to limited bandwidth)

  ↓ Deglitch

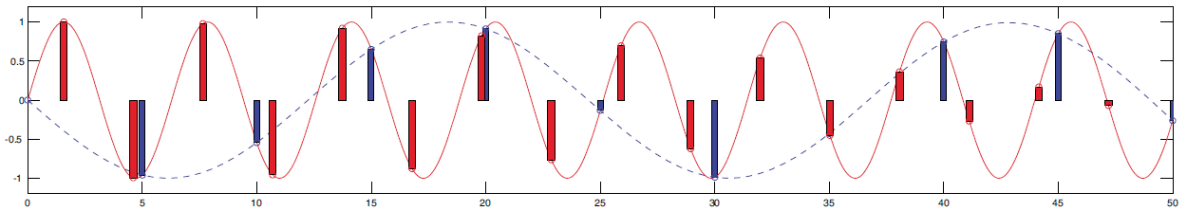
- Smoothed ("deglitched") Signal: predictabl, analogue transition steps (due to synchronized filter)

  ↓ Filter

- Filtered ("low passed") Signal: higher frequency signals eliminated (introduced by conversion step)

2. Signal Sampling

- Aliasing Problem
  - Low sampling frequency results in wrongly observed signal



  - Nyquist Criterion
    - Analog signal with bandwidth f_a must be sampled at frequency $f_s > 2f_a$
    - Perfect measurement taken at $f_s > 2f_a$ result in NO information loss due to sampling

      $\Rightarrow$ over sampling ($f_{\text{real}} >> 2f_a$) is required, due to quantized measurement

- Quantization
  - Assume N bits resolution
    - LSB (Least Significant Bit) $q = \frac{1}{2^N}$
  - Criteria
    - Root Mean Square error (RMS) - inevitable
    - Signal to Noise Ratio (SNR)
    - Effective Number Of Bits ($ENOB$)

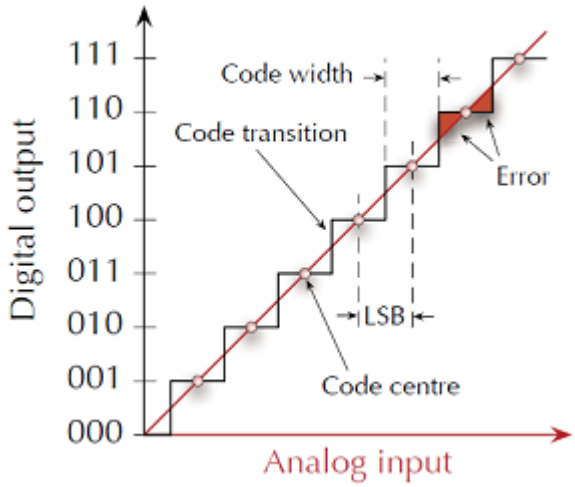
      $SNR_{\text{actual}} = SNR_{\text{ideal}} \Rightarrow ENOB = N$
    - Integral Non-Linearity (INL)

      maximal difference between the actual and ideal code center
    - Differential Non-Linearity (DNL)

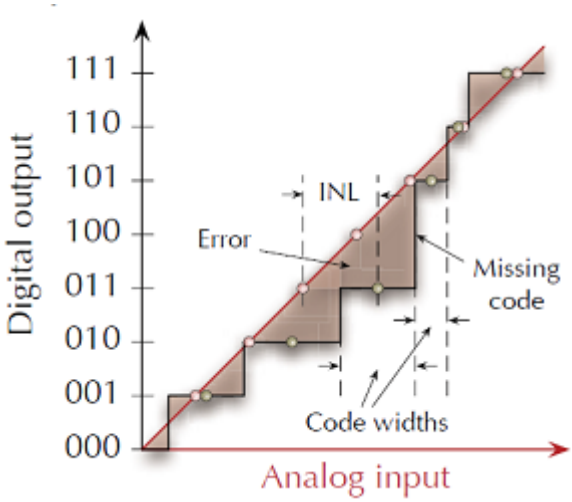
      difference between successive code widths
    - Missing codes

      result in $6.02dB$ decrease in SNR for each missing code

        - Response time/ Latency
        - Throughput /Maximal sampling rate
    - Idea Measurement & Error



    - Realistic Measurement & Error



3. Interface Example - A/D Converters

- Criteria
    - Throughput: maximal sampling frequency
    - Accuracy: ENOB, SNR
    - Latency
    - Power comsumption
    - Price: usually affected by complexity of the design

  ⇒ Trade-off to be expected

  Ex. throughput↑, then power consumption↑ and accuracy↓

- Converters
    - Integrating A/D - Single Slope
        - For each sampled voltage, accumulate a small U_{ref} to measure voltage value by comparing
        - Accuracy depends on voltage constant to accumulate; Throughput depends on input signal;
        - Pros: Simple

    - Cons: Slow
- Integrating A/D - Dual Slope
    - Accumulate in-signal A_{in} for constant time then subtract it by U_{ref} and measure the time it takes to substract $\Rightarrow$ measure the approximate derivative of A_{in}
    - Pros: can smooth the output; suppress (抑制) sepecific frequency
- Flash A/D
    - Concurrent comparators; Accuracy depends on accuracy of reference voltage
    - Pros: Fastest;
    - Cons: Complex for high resolution (size grows exponentially)
- Pipelined A/D
    - All pipelined stages run concurrently; Each stage convert m bits and pass substracted analogue signal & accumulated output to next stage $\Rightarrow$ In total p stages, then is a pm bits converter
    - Accuracy depends on components
    - Pros: Keep throughtput (concurrency); Increase resolution; low cost
    - Cons: Latency increases (trade-off)
- Successive Approximation Register (SAR) A/D
    - 1-bit ADC to convert one bit a time, starting from most significant bit (eg. $\frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{2^N}$); Accuracy depends on 1-bit AD&DA
    - Pros: minimal circuitry (with S&H); almost independent from resolution
    - Cons: slow
- Tracking Register (TR) A/D
    - 1-bit ADC continuous 1-bit conversion caompares current digital ouput and analogue input to count the register up/down accordingly; Accuracy depends on 1-bit AD&DA (Rarely used today)
    - Pros: minimal circuitry (no S&H); almost independent from resolution
    - Cons: Speed depends on amplitude change in input;
- $\Sigma - \Delta$ A/D
    - Σ: adder, to calculate the difference between input and current output

      Δ: integrator, decides to increase / decrease output value by output more 1/0 in the bit stream
    - Encoding: bitstream signal, where density of '1s' denotes input signal
    - Latency depends on bitsteam frequency; Accuracy depends on num of bits decimated; Inherently linear
    - Pros: can be high accuracy (16-24 bits); moderate throughput;

      effctively construct $\Sigma - \Delta$ D/D (D/A) with same encoding;

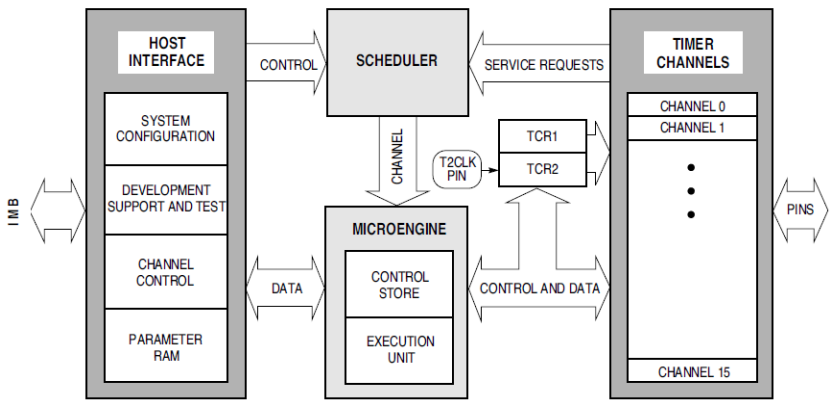
      avoid aliasing because integrator implicitly acts as a low pass filter
- Higher Order $\Sigma - \Delta$ A/D
    - Improve ENOB by adding further integrator stages
- Table for Comparison

	Integrating	SAR	Σ-Δ	Pipeline Flash	Flash
Throughput	$O(1/2^N)$	$O(1/N)$	throughput/ latency vs. accuracy – typ. high accuracy	$O(1)$	$O(1)$
Latency	$O(2^N)$	$O(N)$		$O(p)$	$O(1)$
Accuracy	can be high	medium-high		typ. low-medium	typ. low
Resolution	typ. 8-16 bit	typ. 8-24 bit	typ. 16-24 bit	typ. 8-16 bit	typ. 4-8 bit
Size / Components	$O(1)$	$O(1)$	$O(1)$	$O(p \cdot 2^m)$	$O(2^N)$
Notes	Can suppress some frequencies	Cost efficient and can be very accurate	Flexible architecture, typ. very accurate	Compromise between speed and cost	Fastest converter

3. Micor-controller

- Definition & Components
  - CPU: typ. 4-64 bit word size - also multi-cores
  - Memory: typ. hundre Bytes to MBytes; includes RAM, ROM, EPROM and/or flash
  - Clock generator
  - Timers, General interrupt logic
  - I/O:
    - Basic: GPIO pins, PWM generators, signal wdith detectors, counters, timers, ADC, DAC
    - Modules/Channels: Ethernet, U(S)ART, SPI, I^2C
- Special-purpose Microcontroller:
  - Time Processing Unit (TPU)
    - Access: through dual-ported RAM (atomicity from harware support)
    - **States** : non-interruptible µ-code-blocks.
    - **Functions** : constructed of one or multiple states.
    - **Channels** : 16 digital I/O lines with match and capture.
    - **Priorities** of channels.
    - **Timers** : $2 \cdot 16$ bit time-bases.

    ☞ Associate *functions, time-bases, channels* and *priorities* … and let it run!



4. Handling Device (More in Scheduling)

- Configure Device
  - Load & Store

        - Ada: can define record- and bit- ordering, encoding of value; single write access to each register
        - C: not portable as record- and bit- ordering are not defined
- Response from Device
    - Asynchronous:
        - Synchronized with real-time tasks without violating real-time constraints
        - Exclusive resources (ex. TPU, DMA...)
        - Provide periodically slot
    - With Constant Delay:
        - Pre-schedule device-process
    - Immediate:
        - Busy loop
- Language Requirement
    - Specify device interface (protocols and formats) in all detail
    - Handling asynchronous hardware messages from devices

      ⇒ strong abstraction & being time and physics specific

Time & Space

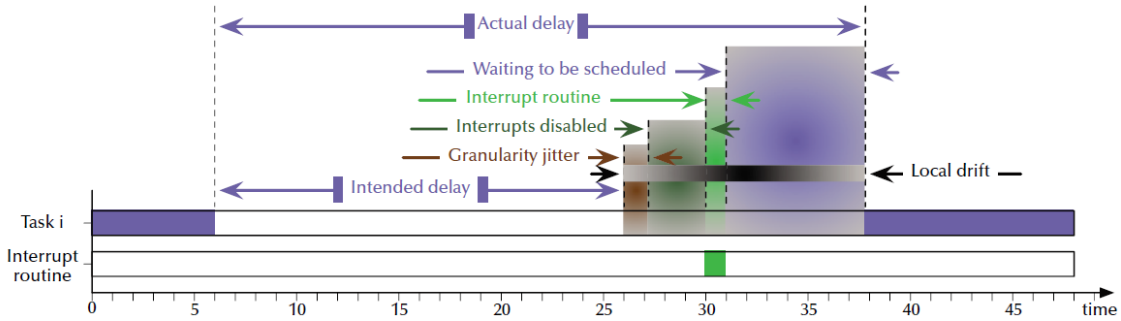
1. Time

- Dependency between Time and Space:
    - Einstein's general theory of relativity eliminates the independence of time (space) and events in time (space)
        - Direct consequence: clock in satellites need to be adjusted accordingly
- Real-Time in the System
    - “Real-Time”
        - the notion of an actual, accessible, generated or external, measured-by-event reference time
    - "Real-Time" Clock: usually understood as an external time source
    - Real-Time in the System
        - Not important: how time is defined / interpreted
        - Imortant: which accessible reference time is denoted "real-time"
    - Time Frame for Real-Time
        - Generate internally: interrupts from local timers; emply a RTC-module (timer based on the notion of seconds)
        - Import Externally: employ time-stamps / sequence numbers for received sensor-readings; receive UTC/TAI radio (in some countries)
- Accuracy of Real-Time:
    - Guarantee with a 'resolution' and 'accuracy' of time

	Syntactical resolution	Required range	Required resolution	Actual resolution detectable
Java	ms	undefined	undefined	no
RT Java	ms, ns	undefined	undefined	yes
Ada	ms, µs, ns	> 50 years	< 1 ms	yes
POSIX	ms, ns	undefined	< 20 ms	yes
C	int (as seconds)	undefined	undefined	no
Hardware timers	$1/f_t$ seconds typically 10 ns … 1 µs	$2^n/f_t$ seconds typically 100 ms … 100's years	configurable	yes

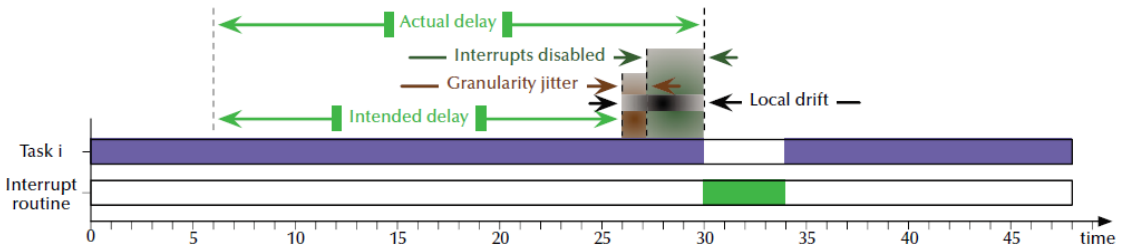
- Actual Error in Timing
  - Delay / Delay until statement (interrupt to initiate task switch & inform scheduler)

    Semantic: **Suspend** the current task **immediately** and **for (at least)** a **predefined time span** or **until (at least)** a **predefined absolute time**.



  - 'Time' interrupt (interrupt to perform context switch)

    Semantic: **Activate** a specified routine **after** a **predefined time span** or **at** a **predefined absolute time**.



    "Local delay drift" summarizes all additional (unintended) delays.

  ⇒ Only absolute delays & timers are allowed in strict real-time system

  (No relative delay because: delay time calculation is NOT atomic)

- Timing Constraint on Action - asynchronous transfer of conttrol (more in Asynchronous)
  - Common Concept in RT Sys: Timeliness is more important than Precision

    ⇒ get first approximation in fixed time well before deadline

    Inspect the deadline and proceed if time allowed

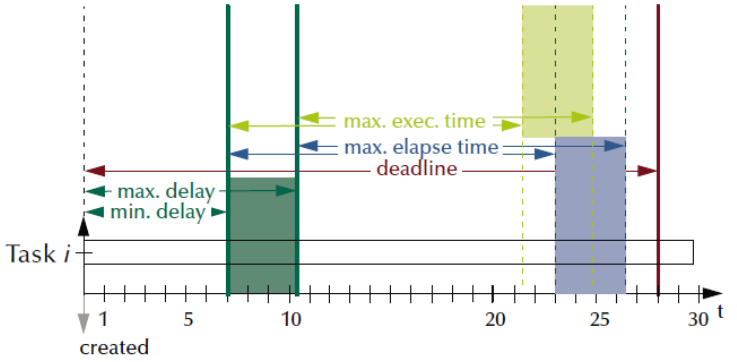
Improve the result with improvement recorded

Either achieve the best result before deadline, or use best-at-the-time result

- Different Timing Requirements
  - Timing Attributes for Tasks
    - when task created

Common attributes:

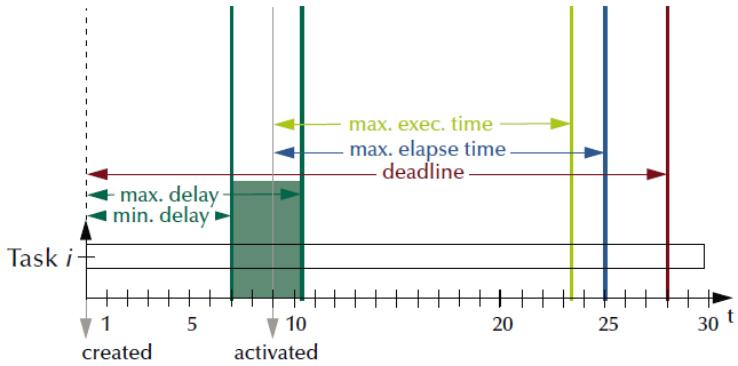
- Minimal & maximal **delay** after creation
- Maximal **elapsed time**
- Maximal **execution time**
- Absolute **deadline**



    - when task activated

Common attributes:

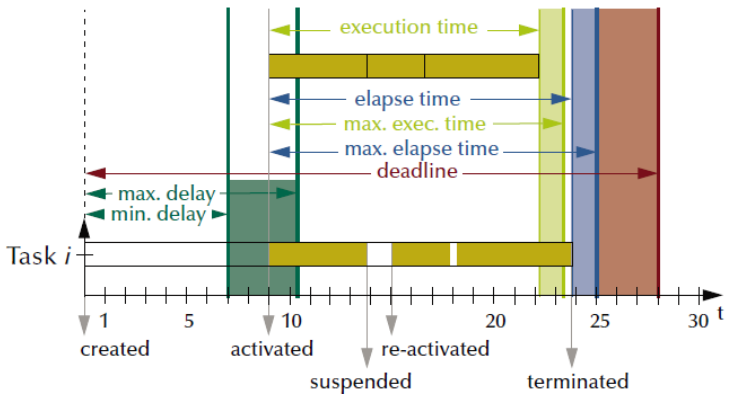
- Minimal & maximal **delay** after creation
- Maximal **elapsed time**
- Maximal **execution time**
- Absolute **deadline**



    - when task terminated

Common attributes:

- Minimal & maximal **delay** after creation
- Maximal **elapsed time**
- Maximal **execution time**
- Absolute **deadline**



  - Timing Attributes for Temporal Scope & Deadline

Temporal scopes can be:

Periodic	☞ controllers, routers, schedulers, streaming processes, …
Aperiodic	☞ periodic 'on average' tasks, i.e. regular but not rigidly timed, …
Sporadic / Transient	☞ user requests, alarms, I/O interaction, …

Deadlines can be:

Semantics defined by application	
"Hard"	☞ single failure leads to severe malfunction and/or disaster
"Firm"	☞ results are meaningless after the deadline
	☞ only multiple or permanent failures lead to malfunction
"Soft"	☞ results are still useful after the deadline

- Timing constraint in Languages
  - Time scope as part of the language
  - provided by standardized libraries (Ada)
  - Signal relation primitives as part of the language
  - Time primitives as part of the language
- Approach for Time-unbound Primitives
  - Exclude them
  - Expand them tp become individually safe (mandatory timeout)
  - Tag the code with additional constrains and Enable full pre-runtime analysis

- Staisfying Timing Requirement
  - Real-Time Logic approach
    - Procedure:
      1. Reduce system (ex. asynchronous, analogue, jitter, drift...) to fully synchronous & discrete
      2. Verify reduced system
      3. Compile reduced system into actual system
      4. Re-check actual system (complex systems approach)
    - Pros: correct solution according to specification
    - Cons: ignore real-world effects (ex. jitters) ⇒ usually assumption violated in real world
  - Complex Systems approach
    - Procedure:
      1. System identification & compile-time analysis (Scheduling)
      2. Run-time analysis and checks (Scheduling & Reliability)
      3. Supply fault-tolerant bahavior (Reliability)
    - Pros: real-world effects invovled; passes rigorous experiments
    - Cons: not complete or correct in any formal sense
- To operate under real-time constraints ⇒ Real-time system

2. Embodiment

- Embodiment - Definition & Explanation

    - Embodied Phenomena: occurs in real time and real space (not in our internal model) by there very nature

      ⇒ interact with the environment; essence of meaningful interation
    - Embodiment is the propery of any engagement with the real world which (may) makes this engagement meaningful
- Embedded System in Embodiment
    - Operational environment is supportive and employed by the system
    - The system is constructed as a part of the operational environment and according to the task
        - recognize:
            1. influence of pyhsical attributes on the result
            2. perception & action are highly coupled (observation affects result)
    - The task is meaningful considering the morphology and cognitive ability of the system as well as the response from environment
- To construct meaningful morphologies ⇒ Embedded system

Asynchronism

- Forms of Asymchronism
    - Interrupts
    - Exceptions
    - Asynchronous Transfer of Control

      *Note: need to have at least 2 concurrent entities to be asynchronized
- Interrupt vs. Asynchronous Transfer of Control
    - Interrupt:
        - communication with slow / asynchronous / sporadic devices
        - sampling / control loops
        - closely coupled reflective system
    - Asynchronous Transfer of Control
        - Error recovery: support atomic action & forward recovery
        - Mode changes: sudden change from 'normal' operation into 'emergency' measures
        - Partial computation: when timeliness more important than precision
        - Operator intervention (介入): user trigger mode changes
- Interrupt vs. Exception
    - Interrupt: linked with specific interrupt handler
    - Exception: can have different handler routine in different context

1. Interrupts

- Individal Device Level - generating
    - Select interrupt source
        - choose certain local event(s) to trigger an outgoing interrupt
    - Provide interrupt status register for reading

        - indicates the currently active interrupts (needed when sharing interrupt lines)
- System Interrupt Controller Level - forwarding
    - Processing Interrupts:
        - Collect all interrupts lines from devices
        - Identifies device status & Encodes inetrrupts into a common inetrrupt vector
        - Mask interrupts according to the priority level of currently active interrupt
        - Trigger an actual CPU interrupt
    - Form of Controllers in System:
        - Embedded into a copmlex micro-controller (an intrisinc part)
        - Dedicated inetrrupt controller (a set of similar/identical devices)
        - Universal interrupt controller (usually, cannot fetch interrupt status)
- Operating System Level - handling
    - Interrupt Service Routines
        - Pros:
            1. Full access to interrupt controller & device status
            2. Predictable delay before the routine activated ⇒ strict real-time constraints
        - Cons: cannot operate as a thread/task!
    - Communicating Interrupts to Processes
        - Synchronize process to events or data structures (ex. semaphore)
    - Transforming into Signals
        - Notify process via scheduler (work-around when missing interrupt propagation)

          ⇒ Cons: unpredictable (as via scheduler); carry little information

2. Exceptions

- Criteria for Exception Handling in RT environment
    - Predictability
        - no or minimal overhead when exception not occurs
        - predictable run-time overhead when exception occures
    - Appropriate recoveries (with sufficient information)
- Exception Mechanism (Ada as Example):
    - Exception Forms

<table>
<tr><td>raised:</td><td>from:</td><td>Run-time environment</td><td>Application</td></tr>
<tr><td>Synchronously</td><td colspan="3">Exceptions</td></tr>
<tr><td>Asynchronously</td><td colspan="2">Interrupt handler</td><td>Asynchronous transfer of control</td></tr>
</table>

    - Exception Granularity - block level

        - Handle one specific / any exception(s) at the end of a defined code block
    - Exception Attributes - information about souce
        - Run-time environment automatically
    - Exception Scope & Propagation
        - Exceptions declared for the whole module
        - Handler determined at run-time (by propagation) - by dynamic link

          Comparison: Real-Time Java:

          procedure declares every potentially raised exception + propagate exception outside static scope if not handled (becomes asynchronous)

          ⇒ messy
    - Uncaught Exception Propagation
        - task terminated ⇒ no propagation to parent-process (no surprise for parent)
        - Catch-All handler provided (though common reactioin is PANIC)
    - Exception Recovery (after exception handled)
        - Termination: return to one block outside (Ada, RT-Java)
        - Block Resumption: re-do the whole block

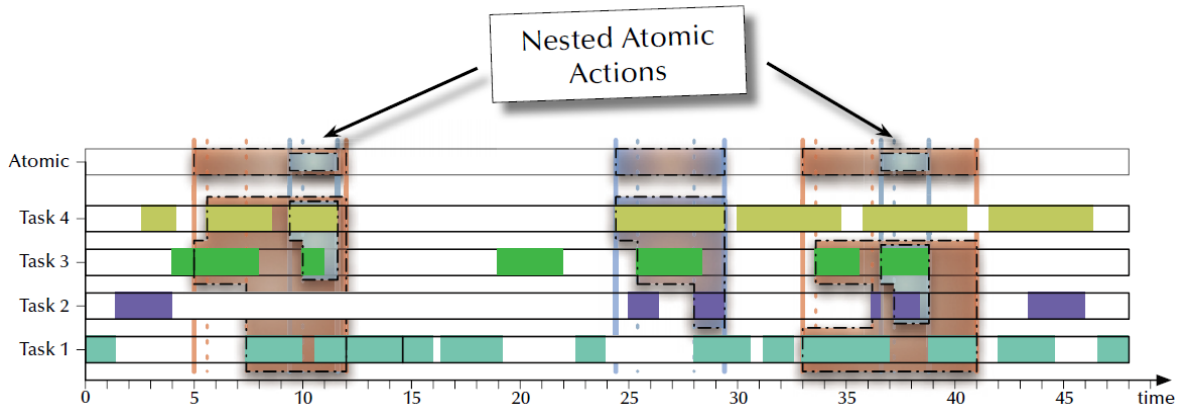
          Pros: contract enforced

          Cons: code needs to be aware of re-try; real-time constraint in danger
        - Hybrid: decide to return or re-try after exception handled
    - Special Cases of Exceptions:
        - in task declaration: propagated to parent task (stlll within the control of parent task)
        - in rendezvous: not propagated to either side if not handled

3. Atomic Action - Hiding Asynchronism

- Definition
    - Atomic action is either performed fully or not at all
    - Atomic action is declared as failed if any part of it fails
- Awareness of Atomic Action
    - be pared to be interrupted (due to failure of others)
    - be prepared to reset to their initial state at any time
- Nested Atomic Action
    - Can be nested iff all processes in nested atomic action are a ture subset of processes in the enclosing atomic action

      ⇒ client task (task calling for the atomic action) can invovle in atomic action (at least partially)



- Atomic Action Failure
  - Strategies:
    - Inner action fails but preserves full atomicity until it ends; then fail the outer action & clean up
    - Inner action fails then breaks the atomicity & propagate failure; then all clean up immediately
- Atomic Actoin in Real-Time Environment
  - Well-defined boundaries
    - Define entry & exit points for all processes in the atomic action
    - Processes can enter at different time, but need to be released at once
    - Separate the invovled processes from rest of the system
  - Indivisibility (Isolation)
    - Prohibit / restrict communications to outside processes and resources

      ⇒ Employing result from one atomic action into another implies strict serialization
  - Nesting
    - if and only if nested one forms a true enclosing relation
  - Concurrency
    - Independent atomic actions may be executed in any order and concurrently
  - Failure Recovery
    - failure (often) implies irreversible failures
    - backtracking (rollback) often pyhsically impossible

      ⇒ Forward error recovery more common
- Ada Code Example - Atomic Action
  - Only the triggering statement will trigger the“abort” behavior
    - There can be protected procedure/entry call inside the abort block

      ⇒ will abort the block right after the non-pre-emptive procedure
  - Need a task/protected object to be Monitor

    ⇒ maintain the state of the atomic action

  Select

  Triggeringstatement (Waiting on Monitor. failed)

...(Clean-up routine)

Then abort

...(Check in)

Select

... (Wait for maximum)

Then abort

End Select

Wait for othertask & Check out from Monitor

Exception

Monitor.report failure

End select

4. Asynchronous Transfer of Control

- Example in Ada
  - Routine for Asynchronous Transfer (select-abort)

```
select
   <entry-call | delay>
   [ … statements … ]
then abort
   … statements …
end select;
```

Execute the trigger (`entry-call` or `delay`), then:

1. If the **trigger can be completed**: ☞ The optional *statements following the trigger are executed* and the select statement is completed (the abortable part is never started).
2. If the **trigger is blocked** or requeued to a blocked entry: ☞ The statements in the *abortable part are executed*:

   2a. If the **abortable part completes before the trigger is completed**, ☞ An attempt is made to revoke the triggering statement. The select statement is completed after the cancelled or completed triggering statement.

   2b. If the **trigger is completed before the abortable part is completed**, ☞ The abortable part is aborted, and the optional *statements following the trigger are executed* and the select statement is completed.

  - Exception Handling in select-abort block
    - All part of select-abort can raise exception
    - In case of interruption of the abortable part, exceptions from abortable part are lost!

Synchronism

- Note on Mutual Exclusion
  - task safe ≠ interrupt handler safe
    - interrupt handler (a 'procedure') can NOT wait insed a queue (no PCB)

1. Shared Memory

- Passive: synchronize with passive objects

  *Note: Always rely on fundamentally hardware support

(Bakery algorithm needs atomic access to memory cell)

- Different Form
  - Semaphore
    - Wait & Signal

      ⇒ without OS: busy waiting; with OS support: suspend on semaphore
    - Pros: fast, minimal overhead (if only one waiting tasks allowed => transfer into 1 machine instruction)
    - Cons: messy, not bound to anything, no compiler check, no fairness
  - Conditional Critical Region
    - Pros:
      1. conditional synchronization is provided;
      2. associate code, data and sync primitives ⇒ compiler check
    - Cons:
      1. evaluate guards by waiting task ⇒ all tasks need to be activated to test guards
      2. access order undefined ⇒ potential livelocks
      3. conditional sync inside critical region needs to leave & re-enter the region
      4. still scattered around code;
  - Monitor
    - Pros: mutual exclusion solved elegently and safely
    - Cons: conditional sync on the level of semaphores still (messy inside)
    - Protected Object
      - Pros:
        1. all data & operations are encapsulated
        2. mutual exclusion solved elegently and safely
        3. fairness achieved, by queuing according to tasks' priority & "internal progress first" rule

           (1 outside queue + num of entries inside queues)
        4. conditional synchronization solved elegently

           (wait on corresponding internal queues + the leaving task evaluates entry guards)
        5. requeue to other entries (no internal conditional variable + no nested protected object call)

2. Message Passing

- Active: synchronize with an active entity (co-operate)
  - both entities need to be active
- Synchronous Message Passing
  - Direct message transfering
    - Sender / Receiver waits for the other task to reach rendezvous
    - Potentially infinite blocking ⇒ no control over remote task

(Ada: implements via scheduler ⇒ task suspended on predicate)

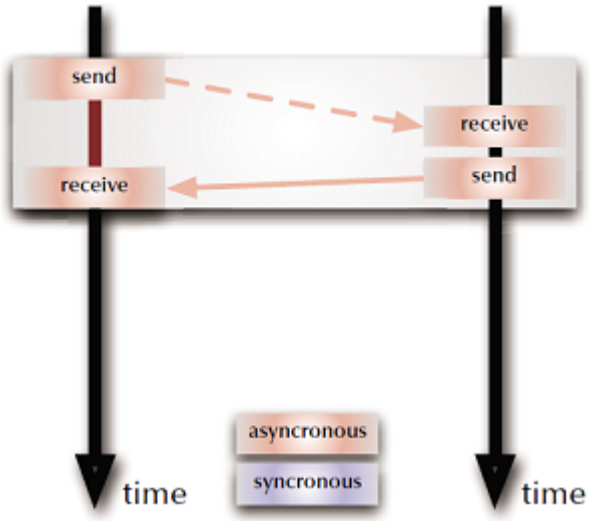
*Note: need hardware support

- Simulated by asynchronous message passing
  - sender voluntarily suspend until transaction completed
  - But:

    no immediate communication happening ⇒ never truely synchronized

    only sender (but nor receiver) knows the completion of transaction

    ⇒ cannot simulate the hardware support purely by software

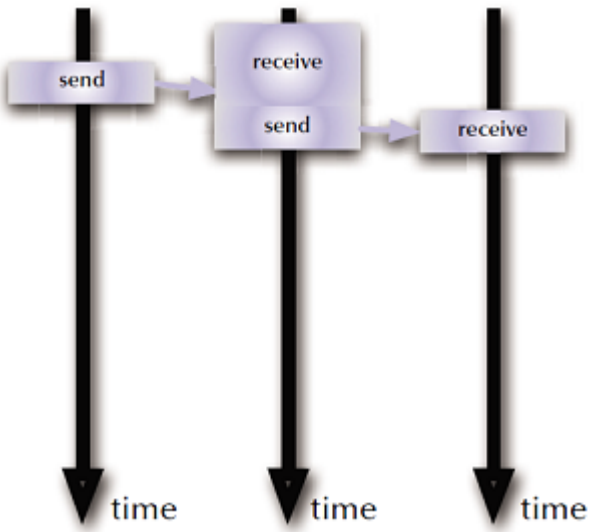


- Asynchronous Message Passing
  - Message not transferred directly ⇒ buffer needed to store data
    - Hardware support: dual-port memory

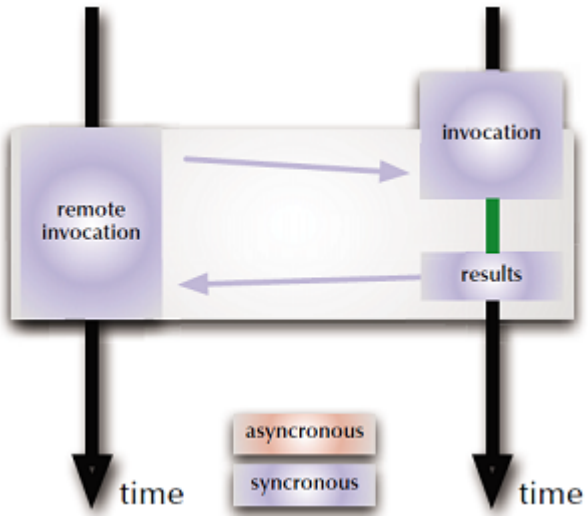
      Note: need overflow policy; yet, in real-time environment:

      buffer is pre-defined and system knows the necessary information needed

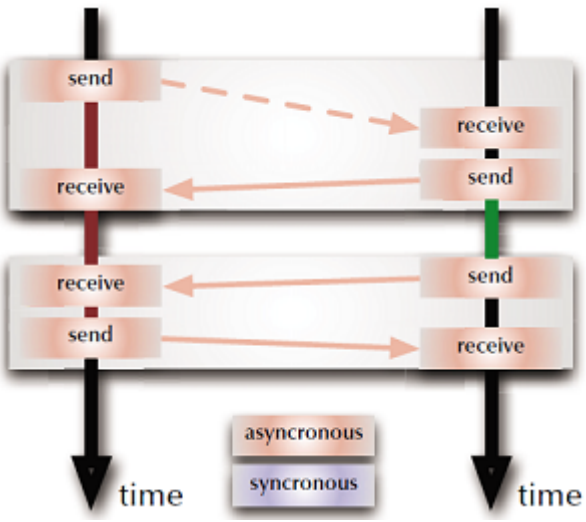
      ⇒ overwritten the full buffer straight away (more recent more useful)
  - Simulated by synchronous message passing
    - intermediate hot stand-by process as buffer (with its own sender & receiver)



- Remote Invocation
  - Procedure
    - Sender / Receiver wait the other task reach rendezvous
    - Sender passes parameters and keep blocked when receiver excutes local procedure
    - Receiver passes result back and release both processes



  - Simulated by Asyncchronous Message Passing
    - Simulate two synchronous message passing (never actually synchronized)



- Message Type
  - Current Problem
    - Communication system (often) outside typed language environment
    - Danger of machine dependent representations (especially in distributed environment)
  - Conversion Routines
    - manually (POSIX, C)
    - semi-automatic
    - automatic (compiler-generated) and type-persistent (Ada)

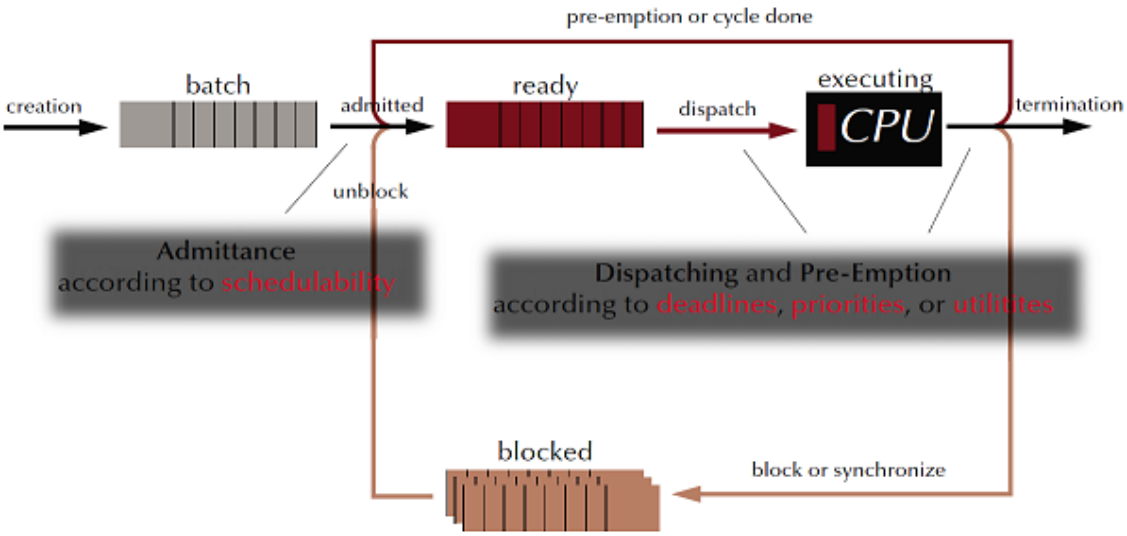
3. Synchronization in Large Scale Concurrency

- Avoid contention on sparse resource

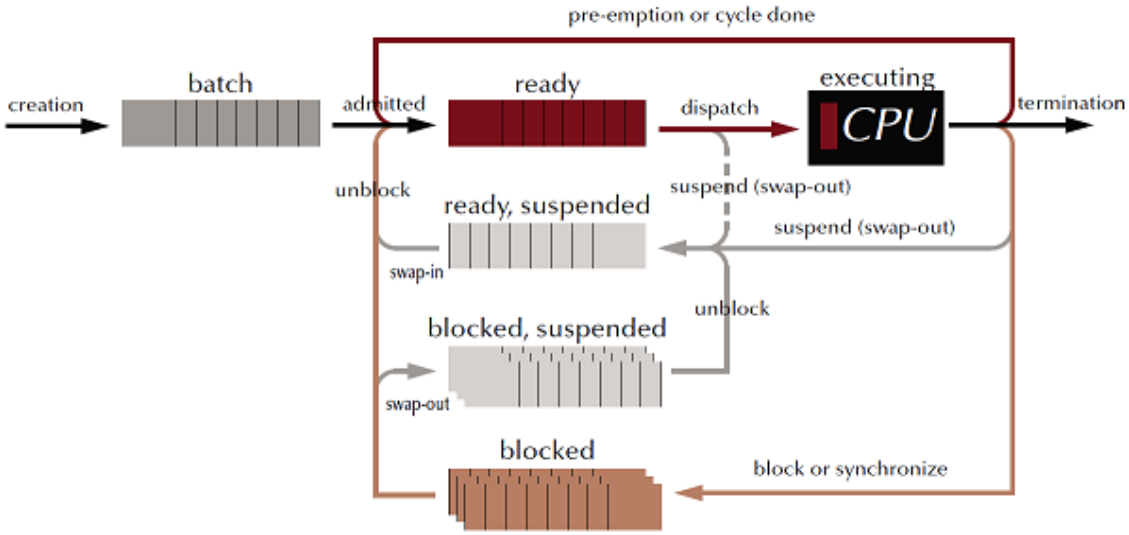
- Data assigned to processes
- Data integrity is achieved when concurrent entities need to be re-sync

Scheduling

- Purpose of Scheduling in Real-Time
  - Scheduling schemes to reduce non-determinism (in order to predict timing behaviour)
    - by both pre-run (at compile time) and post-run (at run-time) scheduler
  - Predicting worst-case behaviour when scheme applied
    - prediction used in:
      1. at compile-time: to comfirm the overall temporal requirements of application
      2. at run-time: to permit acceptance of additional usage/reservation requests
- Scheduling Forms in Real-Time
  - Rigid
    - All schedules are set off-line ⇒ full predictability (many high intergrity RT Sys)
  - Static
    - Schedule relations are statically ordered off-line ⇒ predictable response to disturbances (many RT sys)
  - Dynamic
    - Schedules depends on run-time situation ⇒ more flexible & efficient (most soft RT sys)
- Compared to Non Real-Time
  - In Real-Time:
    - always prefer tasks with higer priority (no general fairness, strict priority)
    - avoid unnecessary task swtiches



  - In Non Real-Time
    - guarantee a general fairness (illusion of everything is in the air at the same time)
    - user-assigned priority usually ignored



- Assumptions of Real-Time Scheduling
  - Number of processes in the system is fixed
  - All processes are periodic and all periods (cycle times) are known
  - All processes are independent (usually not the case)
  - Task-switching overhead is negligible
  - Deadline indentical with process cycle times (periods)
  - Worst-case execution time for each process known
  - All process released at once
- Terms & Notations (with respect to task i)
  - T_i: Cycle time - period
  - R_i: Worst case response time - time from schedule request to process completion in the worst case
  - D_i: Deadline - relative deadline (related to the released time, released time $\neq$ request schedule)
  - C_i: Computation time
  - a : Release time - released at time a
  - B_i: Blocking time - time of t_i being blocked

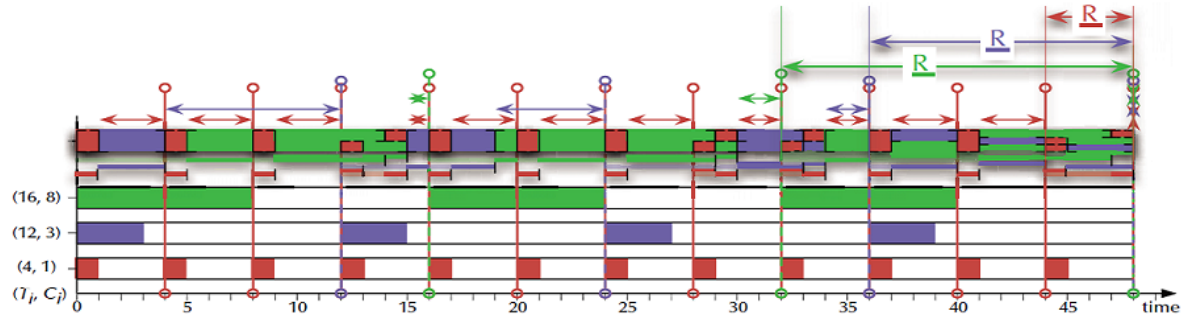
1. Earliest Deadline First (EDF)

- Scheme
  - Determine (one of) the proccess(es) with the earliest deadline
  - Execute the process until either, it finishes, or, another prosses's deadline is found earlier

    $\Rightarrow$ Pre-emptive scheme, Dynamic scheme - process selected at runtime given their deadline
- Maximal Utilization (assume for each task i, $D_i = T_i$)
  - $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$
    - sufficient and necessary test: pass $\Leftrightarrow$ schedulable
- Worst-case Response Time
  - Not necessarily when all tasked released at once (if assumption violated)
    - some long deadline tasks can be released eariler to have earlier deadline than the current task, and thus (potentially) interfering with the current task

⇒ all possible release combinations need to be considered

- if $U \leq 1 \Rightarrow$ bounded by cycle time T_i, as $D_i = T_i$



- Recurrent form (general) - for task j released at time a
  - $R_j^{t+1}(a) = \left\lfloor \frac{a}{T_j} + 1 \right\rfloor C_j + \sum_{k \neq j} \min \left\{ \left\lceil \frac{R_j^t(a)}{T_k} \right\rceil, \max \left\{ 0, \left\lfloor \frac{a + T_j - T_k}{T_j} \right\rfloor + 1 \right\} \right\} \cdot C_k$,

    $\text{where } R_j^0 = a + C_j, \text{ iterate until } R_j^{t+1}(a) = R_j^t(a) \text{ or } R_j^{t+1} > a + D_j, \ D_j \text{ relative to } a$

    $\left\lceil \frac{R_j^t(a)}{T_k} \right\rceil$ ⇒ if other task's cycle times fits in to my response time (assume $D = T$)

    $\max \left\{ 0, \left\lfloor \frac{a+T_j-T_k}{T_j} \right\rfloor + 1 \right\}$ ⇒ if other task is longer than me but can fit its deadline before mine

    use $\max$ to prevent negetive number from the righthand-side equation

    use $\min$ to select the actual interference time
- Actual worst-case response time
  - $R_j = \max\{R_j(a) - a\}_{a \in A}, \text{ where } A = scm(T_i)$ (scm = smallest common multiple - 最小公倍数)

- Pros & Cons
  - Pros:
    - can handle higher (full) utilization than FPS
    - $O(n)$ utilization test and passing test is equivelant to being schedulable
  - Cons:
    - degradation when resource is over-booked: any process can miss deadline

      ⇒ trigger a cascade of failed deadline

2. Fixed Priority Scheduling (FPS)

- Scheme
  - dispatch the runnable process with the highest process
    - Pre-emptive scheme, Static shceme - disptach order (preference) fixed and calculated off-line
- Priority Ordering
  - Definition of Optimal Priority Ordering
    - If a set of process is schedulable under an FPS-scheme, it is also scheduleble under FPS with the optimal ordering
  - Rate monotomic - Optimal

        - for tasks $i, j, T_i < T_j \Rightarrow P_i > P_j$ (more frequent, more important)
        - Maximal utilization (assume $D_i = T_i$)

          $$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \equiv U_{\max}$$

          sufficient test: pass ⇒ schedulabe, schedulable ⇏ pass
    - Deadline Monotonic Priority Ordering - Optimal
        - used when deadline NOT identical with cycle time $(D_i \neq T_i)$
- Worst-case Response Time
    - Recurrent form: $R_j^{t+1} = C_j + \sum_{k>j} \lceil \frac{R_j^t}{T_k} \rceil \cdot C_k$,

      $$\textbf{where } R_j^0 = C_j, \textbf{iterate until } R_j^{t+1} = R_j^t \textbf{ or } R_j^{t+1} > D_j$$
- Pros & Cons
    - Pros:
        - easier to implement, which implies less run-time overhead
        - graceful degradation when resource over-booked:

          processes with lower priority will always miss their deadlines first
        - $O(n)$ utilization test
    - Cons:
        - failing test does not mean not schedulable

3. Scheduling under More Realistic Assumptions

- Scheduling with Sporadic and Aperiodic Processes
    - Deferrable server task
        - server task is of highest priority & scheduled periodically
        - hard real-time task should be schedulable even if server task deploy its full length
        - server task deploys only if there are requests from sporadic / aperioadic task
    - Sporadic server task
        - based on deferrable server task
        - but, the server only replenishes (补充) / reschedules after a fixed time

          (Minimal time between activation need to be known / assumed)
    - FPS with dual priorities
        - start the sporadic / aperiodic tasks at highest priority
        - demote their priorities in time, so that hard real-time tasks is able to complete

          ⇒ dynamic scheme; push hard real-time tasks to their deadline
    - Introducing EDF server
        - equivalent to deferrable server: a periodic server task with immediate deadline

          ⇒ hard real-time pushed to limit

- Earliest deadline last (EDL)
  - earliest deadline last but still need to keep all deadlines for hard real-time task

    $\Rightarrow$ explicitly push all hard real-time tasks to their limits

- Scheduling when deadline NOT identical with cycle time $(D_i \neq T_i)$
  - $\textbf{when } D_i < T_i$ **:**
    - Deadline Monotonic Priority Ordering - Optimal:

      tasks $i, j, D_i < D_j \Rightarrow P_i > P_j$
    - Proof of DMPO optimality
      1. $\exists$ tasks $t_i, t_j \in$ $\textbf{tasks set } Q, P_i > P_j \textbf{ and } D_i > D_j \textbf{ in ordering } W$ (currently not DMPO)
      2. create W' $\textbf{by swapping } P_i . P_j \Rightarrow (P_i' < P_j') \wedge (D_i > D_j)$
      3. $W' \textbf{ schedules } Q \textbf{ because}$
         1. $\forall t_k \in Q \textbf{ with } P_k < P_j \textbf{ or } P_i < P_k$ are unaffected
         2. t_j schedulable in W': $P_j' > P_j \Rightarrow R_j' \leq R_j \leq D_j$
         3. t_i schedulable in W': $R_i' = R_j \leq D_j < D_i$, as $C_i + C_j < D_j$
  - $\textbf{when } D_i > T_i$ **:**
    - Assumptions: task t is released only after the former release of t is completed
    - if $R_i > T_i$ (only occasionally)

      following release of t_i is delayed by $R_i - T_i$, yet still $R_i < D_i$

      *Note: $R_i > T_i$ can NOT always hold $\Rightarrow$ otherwise t_i not schedulable as delay accumulates
    - Worst-case response time

      Need to consider each possible num of release, if the response time stablized (fixed-point):

      $$R_i(q) = B_i + qC_i + \sum_{k>i} \left\lceil \frac{R_i(q)}{T_k} \right\rceil, \textbf{ where } \forall q, R_i(q) - (q-1)T_i \leq D, \textbf{where}$$

      $B_i \textbf{ is blocking time (blocked by other tasks);}$
      $q \textbf{ is the num of release;}$
      $D \textbf{ relative to cycle (schedule) time}$

      $$\Rightarrow R_i = \max\{R_i(q) - (q-1)T_i \mid q \in \{1 \dots q_{\max}\}\}, \textbf{ where } q_{\max} = \max\{q | \frac{R_i(q)}{q} < T_i\}$$

- Scheduling in Dependent Tasks
  - Priority Inheritance
    - Scheme: when task of lower priority block higher priority task by occupying resource, the lowest task inherit priority from the waiting higher priority task when it starts waiting

      (inheritence starts when the higher priority task starts its waiting)
    - Problem: does Not make a difference for blocked task
  - Immediate Ceiling Priority Protocol
    - Scheme
      1. Each task t_i has a static priority P_i
      2. Each resource R_k has a static ceiling priority $C_k, \textbf{ where } C_k = \max\{P_i \mid t_i \textbf{ uses } R_k\}$

$\Rightarrow R_k$ has the maximal priority from tasks ever use it

3. Each task has a dynamic priority P_i^D,

$$\textbf{where } P_i^D = \max\{P_i, \max\{C_k^H\}\}, C_k^H \textbf{ is priority of } R_k \textbf{ currently held by } t_k$$

(priority inherentence starts from task touching (obtaining) the resource)

- Pros:

  Task is dispatched only if all employed resources are available

  (if my resource is held by other task, that task will have a priority at least as high as mine)

  $\Rightarrow$ Prevent deadlock! - single CPU, no conditional wait (no entry guards), no message passing

  (whereas banker algorithm: exhaustively search for a safe trace without deadlock)

  Reduce context swtich

- Maximal blocking time

  $B_i = \max\{C_j(r)\}_{r=1}^{R}, \textbf{ where}$

  $R = \textbf{num of critical resource},$
  $C_j(r) = \textbf{worst case computation time by task } t_j, \textbf{ whose } P_j < P_i, \textbf{ on critical resource } r$

  $\Rightarrow$ task can only be blocked by <u>ONE</u> lower priority task <u>ONCE</u>

  Overall maximal blocking time $B_{\max} = \max\{B_i\}$

- Cooperative $\Rightarrow$ Scheduling Scheduling when Tasks Not Pre-emptive
  - Scheme
    - Every task t_k divided into k NOT pre-emptive blocks of $C_{ik} < B_{\max}$
    - All critical sections completely enclosed into a single block C_{ik}
    - Every task offers a task switch at the end of each block
  - Response Time
    - $R_i = R_i^n + F_i$,

      $$\textbf{where } R_i^{k+1} = B_{\max} + C_i - F_i + \sum_{j>i} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j, \ F_i = \textbf{execution time for final block}$$

    - if $C = C_i = C_j = F_i = B_{\max} \ \Rightarrow R_i = R_i^n, \textbf{ where } R_i^{k+1} = C + \sum_{j>i} \left\lceil \frac{R_j^k}{T_j} \right\rceil C$
    - if further $\forall i, T = T_i \ \Rightarrow R_i = C + \sum_{j>i} C$
  - Pros & Cons
    - Pros:

      Task switch reduced

      Caches, pre-fetching and pipelines more efficient (as the exact timing of task switch known)

      Easier (a bit) to predict execution time

      Simpler schedules

      Inter-dependent tasks can be schedulable deadlock free by design

    - Cons:

      Special care for non-cooperative tasks (watch-dog timer to inspect as a fall-back)

$B_{\max}$ becomes the minimal delay for all tasks $\Rightarrow$ need to be acceptable for all tasks

- Fault Tolerant Scheduling
  - Extra Required Information:
    - worst case execution time C_i^f: Task t_k needs extra CPU-time C_i^f for recovery
    - minimum inter-arrival time between faults: T_f
  - Response Time (fixed-point)
    - $R_i = B_i + C_i + \sum_{j>i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max\left\{ \left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right\}_{k \geq i}$
    - when all recovery on highest priority: $R_i = B_i + C_i + \sum_{j>i} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max\left\{ \left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right\}_{k}$
- General Scheduling Methods (with no assumption)
  - $NP-\text{Hard}$ problem
    - Can be reduced to P with assumptions (ex. cycle times T is powers of 2)
- Language Support (Ada)
  - Prioritiy
    - Task and Interrupt priorities (static, dynamic, active)
    - Prioritized entry queues
    - Priority Ceiling Locking
  - Scheduler
    - FPS with FIFO within priorities (pre-emptive)
    - Round Robin
    - EDF
  - Task execution time measurement + Task attributes
  - NO sporadic servers

Resource Control

1. Resource / Service Request

- Criteria
  - For Resource/Service Request:
    - type (read / write)
    - time (arrival order, relative time...)
    - arritbutes, parameters, priority - of calling client
    - interal / synchrinization state of resourece - including timing behaviour
  - For Resource Sync Method:
    - expressive power; ease of use
  - Real-Time Perspective:
    - time constraint applied to resource request
    - special care required to extend deadlock analysis with real-time constraint
    - task failure more problematic with real-time constraint

- Method when Requested Resource Inavailable
  - Conditional wait:
    - accept request and suspend threads internally

      ⇒ client NOT able to revoke requests
  - Avoidance Synchronization:
    - suspend request at outer queue

      ⇒ client can easily revoke request
- Cons of Double Interaction (when synchronization method not expressive enough)
  - Request no longer atomic!
    - define atomic action for double interaction & sync methods need to be aware of definition
  - Wrong assumption about the other side (wrongly assume the client alive/dead)
    - verify client to be always alive in double interaction
    - eliminate double interaction by requeue...

      ⇒ 'requeue': request accept and no longer revocable

      'requeue with abort': request remain revocable (still at the outside queue)

2. Resource Reclaiming

- Motivation:
  - Increase reliability, which relies on spare resource
- Requirement on Reclaiming
  - Correctness: need to maintain feasibility (should still be schedulable)
  - Small Overhead: compared to the possible gains
  - Predictable: should be included in task's worst case response time ⇒ need bounded complexity
  - Effectiveness: improve the system reliability (can handle more failures)
- Expanded Scheduling for Resource Reclaiming
  - Feasible (prerun) schedule S: account for timing, resource, precedence constraint, R_i
  - Postrun schedule S': based on S and account for actual computation times C_i'
    - let st_i, ft_i are start and finish time in prerun S; st_i', ft_i' are start and finish time in postrun S'

      ⇒ Correct postrun schedule: $\forall$ **task** $t_i, (st_i' \leq st_i) \wedge (ft_i' < D_i)$

      Passing task: task t_i passes t_j **iff** $(ft_j < st_i) \wedge (st_i' < st_j')$ ⇒ strict order in S violated
- Types of Reclaiming Algorithm
  - Extreme cases
    - No reclaiming: dispatch as prerun schedule
    - Optimal reclaiming: globally re-schedule whenever reclaiming requested

      optimal scheduling of dynamic arrival, non-pre-emptive tasks in multi-processor: NP-Hard
  - Practical re-scheduling (for reclaiming)
    - Algorithm without passing tasks ⇒ bounded complexity (by constant)
    - Algorithm with passing ⇒ in general $O(logn)$, but bounded with restricted passing

- Practical Reclaiming Algorithm - dealing with Inter-dependent tasks
  - More Notation:
    - C_i': Actual computation time
    - $t_i \otimes t_j$: a set of resource conflict, i.e. t_i, t_j require a resource exclusively
    - $t_i < t_j$: a set of precedence constraints, i.e. t_i completes before t_j starts
  - Further Assumptions
    - m processors available
    - Tasks cannot migrate between processors once start (done on general OS for heat distribution)
    - At most one task per processor
    - One task-queue per processor; Task-queue in shared memory and maintains a consistent view
    - Tasks not pre-emptive (otherwise way easier)
  - Simplest Scheme
    - Reclaim simultaneous idling time
  - Detect concurrent tasks group for each task t_i - $O(m^2)$ with m processors
    - Based on S:

      $t_{<i} = \{t_k \mid ft_k < st_i\}$ - finish before t_i

      $t_{>}i = \{t_k \mid st_k > ft_i\}$ - start after t_i

      $t_{\sim i} = \{t_k \mid (t_k \notin t_{<i}) \wedge (t_k \notin t_{>i})\}$ - overlap with t_i

    $\Rightarrow$ allow task in $t_{\sim i}$ running concurrently to t_i, but not overlap with $t_{<i}$ or $t_{>i}$

    $\Rightarrow$ allow early start accordingly
  - Restriction vectors - static processor assignment
    - Restriction Condition ($\boxtimes$):

      $t_{i \boxtimes j} = (t_i < t_j) \vee (t_i \otimes t_j)$ - precedence or resource constraint
    - Restriction vectors for task i:

      $$RV_i[p] = \begin{cases} t_k \in t_{<i}(p) \mid \neg\exists t_j \in t_{<i}(p), st_j > st_k, & \text{if } i \text{ on processor } p \\ t_k \in t_{<i}(p) \mid t_{k \boxtimes i} \wedge (\neg\exists t_j \in t_{<i}, t_{j \boxtimes i} \wedge (st_j > st_k)), & \text{if } i \text{ not on processor } p \\ -, & \text{no such task} \end{cases}$$

      understatnding:

      each task stares at

      1. most recent task (in $t_{<i}$) queuing on my processor
      2. most recent task restricting me (in $t_{<i} \wedge \boxtimes$) queuing on other processor

      $\Rightarrow$ each task stares at at most m tasks (m = num of processors)
    - Completion Bit Matrix (CBM)

      $$CBM[i,p] = \begin{cases} 1 & \text{iff } t_i \text{ completed execution on processor } p \\ 0 & \text{otherwise} \end{cases}$$
    - Usage:
      1. Compute $RV_i(p)$ - at compile time
      2. For any task t_i next to be scheduled on processor p

Fetch the most recent CBM

If $\forall t_j \in RV_i(p) \wedge CBM(i,p) = 1 \Rightarrow \textbf{start } t_i$; Else idle until next CBM update

- Proof of Correctness

1. Given a feasible prerun schedule S, if $\exists t_i, st'_i > st_i$, then passing must have occurred

   Assume no passing occurred:

   $\Rightarrow \forall t \in t_{<i}, t$ have been dispatched before t_i

   $\forall t \in t_{>i}, t$ only dispatched after t completed

   $\forall t \in t_{\sim i}, t$ does not interfere with t_i as S feasible $\Rightarrow$ does NOT delay t_i

   $\Rightarrow st'_i \leq st_i, \textbf{ contradiction}$

2. Restriction Vector gives a correct postrun S'

   with above lemma: $S' \textbf{ incorrect } (\exists i, st'_i > st_i) \rightarrow \textbf{ passing occurred}$:

   $\Rightarrow$ with incorrect S', $\exists t_i, t_j, (ft_i < st_j) \wedge (st'_j < st'_i) \wedge (st'_i > st_i)$

   if t_i, t_j have resource or precedence conflicts

   $\Rightarrow$ included in RV_j ($t_i \in t_{<j}$)

   if t_i, t_j do not have such constraint

   t_j can not delay t_i by passing

   $\Rightarrow$ RV-algorithm allows for restricted forms of passing only

   and does not corrupt postrun S'

- Restriction Vector - dynamic processor assignment (processor can exchange task-queue)
  - Restriction Vectors for task i:

    $$RV_i[p] = \begin{cases} t_k \in t_{<i}(p) \mid t_k \boxtimes i \wedge \neg \exists t_j \in t_{<i}(p), t_j \boxtimes i \wedge (st_k < st_j) & \textbf{if } t_k \textbf{ exists} \\ - & \textbf{no such task} \end{cases}$$

    understanding:

    each task stares at only the most recent task restricting me (no matter which processor)

  - Proof of Correctness

    Assume: task t_i is next to be scheduled yet blocked on idling processor P_x;

    task t_j is currently running on processor P_y

    The swapping of dispatching queue $DQ_x \leftrightarrows DQ_y$ of P_x, P_y does NOT intfere the correctness of S' if the swapping is permitted only if $st_i \geq ft_j$

    $\Rightarrow$ the current blocked task t_i will not be further delayed

    $\Rightarrow$ the next unrestricted t_k used to be blocked on P_y, can not start early on P_x

    $\Rightarrow$ no task further delayed by $DQ_x \leftrightarrows DQ_y$

- Evaluation of Resource Reclaiming
  - Criteria
    - Task graph density $P_p \in [0,1], \textbf{ with 0 being fully independent , 1 fully dependent}$
    - Aw-ratio $\frac{C'_i}{C_i}$ (actual to worst case ratio) $\Rightarrow$ in reality usually $\mathbf{1}$ (not trying to be faster)
    - Mig_attempts: number of checks on dispatch queues by RV algorithm for migrating

    - Resource Reclaiming Cost
        - $C_{\text{basic}} = O(m)$ - simultaneous idling, with m processors
        - $C_{\text{early\_start}} = O(m^2)$ - detect concurrent tasks group
        - $C_{\text{RV}} = C_{\text{early\_start}} + C_{\text{RV}}$, where C_{RV} is the cost for calculation of RVs in compile time
          - Restriction vector without DQ swapping
        - $C_{\text{RV\_migration}} = C_{\text{early\_start}} + C_{RV} + f(\text{Mig\_attempts}, C_{\text{early\_start}})$
          - Restriction vector with DQ swappings
    - Practical Measurement
        - Continuous gain from: basic → early-start → RV-reclaiming → RV-reclaiming-with-migration
        - RV-reclaiming-with-migration: overhead can be noticeable (extended communication / sync)
        - Need high-degree of tasks dependencies to justify application of RV-reclaiming-with-migration
        - Necessary support from Real-Time System:
            1. allow earlier task start time
            2. allow task migration
            3. can express task dependencies in the introduced form (to apply introduced algorithm)
- Recourse Control Issues in Pratical Real-Time System
    - Polices
        - Priority assignment problem: achieve timing constraint and reliability when assigning priority
        - Overload problem: predict and protect system from overload condition
        - Flexibility problem: locally adjust system to current timing constraint
    - Run-Time Environment
        - Enforcement problem: hadling tasks & resources exceeding their anticipated worst case limit
        - Measurement problem: record relevant information in a sufficient resolution and frequency
        - Coordination problem: synchronizing system components which are under different policies

Reliability

- Motivation
    - build Predictable & Dependable system in Real-Time Domain
- Terms
    - Reliability: measure of success, with which a system confirms to its specification
    - Failure: deviation of system from its specification
    - Error: system state which leads to failure
    - Fault: the reason for error

1. Dealing with Faults

- Types of Faults
    - Design
        - Inconsistent or inadequate specification (frequent source for disastrous faults)
        - Program errors (frequent source for disastrous faults)

        - Component & communication system failure (usually predictable)
    - Logic
        - Non-termination (watch-dog timer required)
        - Range violation & inconsistent state (run-time environment level exception handler)
        - value violation & wrong results (user-level exception handler)
    - Time
        - Transient faults (single 'glitches', hard to handle)
        - Intermittent faults (of certain regularity, need careful analysis)
        - Permanent faults (stay failure, easy)
- Fault Avoidance & Removal
    - hardware-level
        - reliable hardware under the working pyhsical environment
        - proper protection over one-point failure place - connectors, pins...
        - testing, yet exceptional conditions / remote environment hard to creat / simulate
    - software-level
        - verify consistency of specification (formal method if applicable)
        - automated deduction ar compile time
        - coding standard & code certification
        - run-time environment & languages with resonable support for requirement
        - testing, yet test space too large in Real-Time system
- Fault Tolerance - no method guarantees total absence of faults
    - Types:
        - Full fault tolerance:

          system still operates under forseeable error condition, without significant loss of functionalities or performance (but might reduce operational time)

          yet not maintainable for infinite operation time
        - Graceful degradation (fail soft):

          system still operates under forseeable error condition, with a partial loss of functionality or performance

          might have multiple levels of reduced functionality
        - Fail safe:

          the system fails yet maintains its integrity

2. Redundancy - main way for fault tolerance

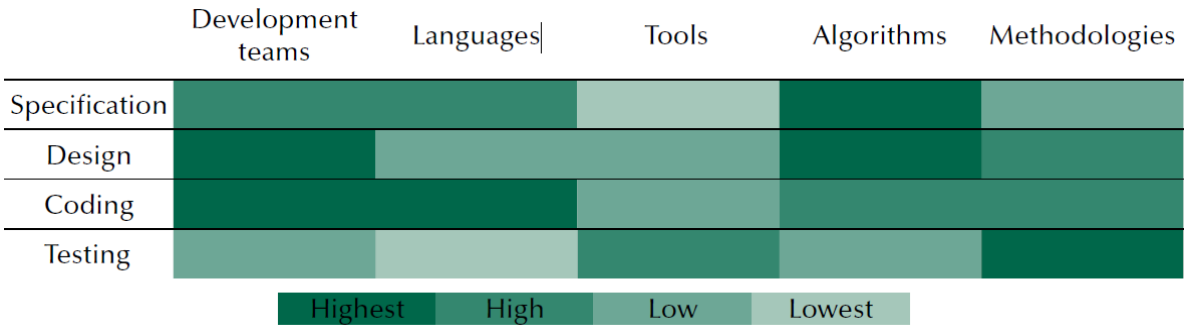
- N-Modular Redundancy (NMR) - Hardware Redundancy
    - Extra hardware Resource:
        - to detect and localize the faults
        - to handle exceptional situation and recover from error
        - yet,
            1. will adds to overall system complexity

2. usually cannot protect over one-point failure (connecotrs, bins...)

- Assumption:
  - functionally identical components are connected via voting / masking / comparing system
  - functionally identical components can be swapped dynamically (in a detected error-condition)
  - The fault is based on a physical phenomenon and applies only locally & the structure of functionally identical system is sufficiently different
- Redundant Sub-system
  - same specification
  - different computer systems; operation systems
  - different real-time languages & development environment (N-Version programming)

  ⇒ outputs from different part is slightly different

- N-Version Programming
  - Factors Impacting Software Diversity



  - Example: "The six-language project"
    - Testing result of different version

Failure category	Average … Single version	3-version	5-version
	… failure probabilities measured in … 5,127,400 cases	102,531,685 cases	30,757,655 cases
No errors	0.99993733	0.9998409	0.9997807
Single error	6.27×10^{-5}	13.05×10^{-5}	19.15×10^{-5}
Two distinct errors		0.20×10^{-5}	0.23×10^{-5}
Two coincident errors		2.65×10^{-5}	2.21×10^{-5}
Three errors			0.34×10^{-5}

    ⇒ 3-version & 5-version have lower failure rate than the "golden master"

    coincident errors invovling more than 2 versions were never observed

    ⇒ Control problem specially suitable for N-version Programming

    (in general, diverting results do NOT necessarily imply faults)

  - Issues
    - Arithmetic: integer usually identical; real value need to be consider tolerance

⇒ need to evaluate similarities between different systems

- Multiple solutions: the solution space allows multiple correct but structurally different solutions

  ⇒ N-version programming NOT helps

  (algorithm for voting / comparing results can be more copmlex than the actual one)
- Specification: hard to prevent people misunderstand the specification in the same way

  ⇒ specification is diversified and written in different ways but for logical the same thing
- Diversity assumption: co-incident error condition may happen (in desining / developing)
- Project costs: need to consider if a single version developed with same effort may show a similar level of reliability

- Dynamic Redundancy
  - Error Detection
    - from Environment: from CPU, controllers, run-time environment
    - from Application process: replication (N-version programming), coding (hamming), contracts, continuity (assume limited difference between consecutive controller value)
  - Damage Confinement and Assessment
    - Confinement: avoid transfer of fault-effects between system parts

      (modular decompostion, atomic action, 'firewalls')
    - Assessment: identify fault and its potential location and level

      (location of detected error, possible system paths leading to the error)

      *Note: fine-granular (confinement) system limits the length of paths in assessment
  - Error Recovery
    - Backward: checkpoint, log ⇒ applicable even if fault itself can not be identified

      (yet, need to ensure system-wide consistent checkpoints;

      system can NOT contains non-reversible components - e.x. **TIME**)
    - Forward: choice of most time critical parts of real-time and embedded system

      (highly application dependent; may involve complex priority, mode changes)
  - Fault Treatment - adjust, avoid, correct, or exchange
    - localize hardware fault to be easier & more precise than a software fault
    - On-line fault treatment can be hard and often limited to exchanges of complete modules
    - Finer granularity than static redundant systems
    - Excahnge of fault components is usually expensive and complex operation
    - Limited number of substitutable sub-system ⇒ often assume transient (短暫的) fault

      ⇒ log the event and continue operations, rather than throwing it away

3. Safety and Dependability

- Terminology
  - Safety

        - Definition:

          Freedom from those conditions that can cause death, injury, occupational illness, damage (or loss of) equipment (or property), or environmental harm. (Leveson, '86)

        - Varying standard:

          Standard (requirement) of safety changes according to socail pressure and acceptability.

    - Dependability Features
        - Availablity - ready to use
        - Reliability - absence of failure
        - Safety - absence of fatal failures
        - Confidentiality - absence of unaithorized disclosures
        - Integrity - no data corruptions
        - Maintainability - accessibility to changes and improvements

- Design Tool Chains towards 'Safety & Dependability'
    - Restriction
        - limit tools & environments to 'safer' operations: e.x. Ada Ravenscar, Ada Zero Footprint
    - Formalization
        - Proof correctness: Temporal logic, real-time logic
        - Provable predicates for languages