OS Review

- Large and Complex
  - Use of standards for clarity/portability -- POSIX
  - Multiple processes/threads

- Concurrent control of system components
  - Programming via OS/Language (combination)

- Facilities for hardware control
  - Interrupt hooks/procedures - device access

- Extremely reliable and safe
  - Memory/Resource management

- Real-time facilities
  - Timing primitives
  - Overrun detection
  - Priority inheritance

- Efficiency of execution
  - Scheduling (auto)/admission/workload management
Ideal “open OS”

- seamlessly runs multiple applications
- independently designed/validated
- RT and non-RT

- Server Threads
- In built support for multiple schedulers

- provides admission/timing guarantees for RT applications
- Busy Interval Tracking
- Accurate Interval timers
- Resource reservation
- Usage monitoring/enforcement
- Extended Priority inheritance
  - Choice of PI Protocols
  - Priority tracking on interrupt handling
  - Message based priority inheritance
Two-level scheduler
Choosing a (non-ideal) OS

- Supports a standard
  - few system specific functions required

- Modularity
  - hooks for user level extensions
  - disable unneeded mechanisms/features

- Features
  - Pre-emption scheme(s): equal/non-equal priority
  - Priority assignment scheme(s): dynamic/fixed
  - # priority levels/tasks/timers/semaphores/mutexes/signals
  - Priority inheritance
  - Resource Usage tracking

- Performance
  - Timer resolution
  - Context Switching Time & Interrupt Latency
  - Time to block/deliver a signal
  - Time to grant/release semaphores/mutexes
POSIX (IEEE 1003.1b)

- Common OS interface for existing flavors of UNIX.
- Minimal level of functionality, which ALL compliant systems must provide.
- Original standard defines basic OS systems
  - device /file-system I/O
  - basic IPC
  - time services
  - process management
- Modified to deal with real-time issues (Pthreads)
  - priority inheritance (mutex)
  - priority-based scheduling policy
  - contention scope
  - increased timing specification

μITRON4.0

- TRON project: specification pre-dated any implementations
- Real-time Standard designed to deal with myriad platforms of varying size
- Provides various levels of compliance, which the designer can adhere to
  - minimum (R)
  - automotive
  - standard (S)
  - full (E)
- non-RT aspects unspecified
  - device/file I/O

Both API standards, with user-level hooks/extensions

http://www.opengroup.org/onlinepubs/009695399/idx/realtime.html
http://www.tron.org/index-e.html
**Kernel**

- Responsible for essential services
  - scheduling,
  - timing,
  - SIMPLE inter-process communication (mutex/signal)
  - other interrupts/exceptions
  - low-level hardware-specific tasks

- Two flavors
  - libraries to be included in application binary
  - base-OS: the core of an operating system

- Typically does not provide more advanced OS functions
  - file system
  - user interface
  - memory/resource management/protection
    - potentially reduced reliability of the system
  - more complex IPC (semaphore/message)

- For RT, must provides
  - load-independent timing for all functions

---

**Why use a kernel? Do we need a kernel?**
RT Application Compromises

- Implementation of periodic tasks using
  - sleep & wait-timer functions
- Hardware interrupt handling
  - split approach
- Thread scheduling:
  - queue structures,
  - resource sharing
  - messages for communication
  - usage monitor,
  - TCB’s
- Mapping design priorities to available priority levels
Implementing periodic tasks

- **Relative time:**
  - busy-wait
  - sleep-wait

- **Absolute time:**
  - timer

- **Time Properties:**
  - tick size (timer interrupt)
  - nominal resolution (granularity)

---

From: [DoAC02, Lecture 5: http://www.control.lth.se/~kurstr/L5 02 slides4.pdf]
Aperiodic/Sporadic tasks

- **Polling**
  - Idle Task
  - Periodic Server

- **Interrupts**
  - caused by
    - device notification
    - events/signals e.g. reset; alarm; overflow
  - ALWAYS higher priority than tasks
    - can cause unpredictable behaviour

- **Split handler**
  - interrupt routine kept minimal
  - sets flags/data for periodic routine to handle
  - minimises unpredictability

Diagram after: http://www.it-infothek.de/images/fhtw_ra_bs_112.gif
Context switching mechanics

- CPU executes the “next” instruction defined by the Program Counter
- Branch(goto) and Call instructions change the value in the program counter
- Interrupt “stops” current program and executes a pre-defined routine before returning to the interrupted program
- We can use this interrupt routine to change the current task by:
  - saving the Program Counter & other registers
  - re-placing them with Program Counter & registers for another task
- To implement multi-tasking
  - use a “regular”/clock tick interrupt,
  - use a queue of the programs to be executed
Managing queues

- Static queue - move the index
- Dynamic queue - move the objects

Which is better?
- How do we handle static/cyclic schedules?
- How do we handle priority ordering/pre-emption?
- How do we handle blocked processes?
- How do we handle dynamically admitted processes?
Program states
Resource Sharing

- **Pre-emption locks**
  - disable interrupts
  - change priority level
  - system-wide blocking time

- **Mutex/Semaphore**
  - shared memory variables
  - protect access/release with pre-emption lock
  - synchronous (timeout facility)
  - asynchronous (availability test)
  - minimise time held for blocking
  - minimise use for efficiency

- **Reservation**
  - resource used by single process
  - less overhead than mutex/lock
  - less flexible resource allocation

- **Protection**
  - prevent process from accessing resource reserved by another process
  - ALL resource access must be checked - potentially large overhead

Which is best?
Communication

- signals - single bit
- repeat signals ignored
  - counter
- interrupts task if not masked

- message - packet of data
- another use for queues
- processed in FIFO order
- careful: overflow vs. blocking

Process A sends a message M to Process B.
B has too many messages in the queue and discards M.

Process A sends a message M to Process B.
B has too many messages in the queue, raises his priority to remove first message

Process A sends a high priority message M to Process B.
M becomes the first message in the queue, but Process B waits until Process C is suspended

http://linuxdevices.com/articles/AT4627965573.html
Usage Monitor

- **Process**
  - Execution Time
  - Response Time
  - Current Priority
  - Initial Priority
  - # active instances
    - Instance Deadline

- **Resource**
  - Requesting processes
  - Locking processes
  - Locking Times
  - Resource Priority

- **Priority Changes**
  - Inheritance
    - Message Queue
    - Resource Tracking
  - Dynamic Priority scheduling

- **System**
  - Utilization
  - Busy Interval tracking
  - Detection/correction
    - deadlock
    - overload
    - overrun

Suggest some methods
Thread Control Block (TCB)

- **Thread**
  - “basic unit of work handled by the scheduler”
  - used to implement jobs (task sub-units)

- **Thread Control Block**
  - data structure used to keep information needed to manage/schedule the thread
  - **Thread context**
    - registers and other volatile data;
  - **Task parameters**
    - type, phase, period, relative deadline, number of instances, event list
  - **Scheduling information**
    - current / assigned priority
  - **Timer information**
    - multiple timers
    - ordering for expiry

| Thread ID |
| Starting Address |
| Thread Context |
| Task Parameters |
| Scheduling Information |
| Synchronisation Information |
| Time Usage Information |
| Timer Information |
| Other Information |
Coping with insufficient priority levels

- Map assigned priorities \( (i \text{ of } n) \) to system priorities \( (\pi \text{ of } s) \)
  - Uniform mapping
  - Constant ratio mapping

- Multiple queues
  - in each task state
  - at different priority levels

How do we find the next “ready” task

- Between equal priority processes
  - FIFO
  - RR
  - EDF

- Similar scheme(s) can be used for mapping message priorities

If there are different numbers of message and task priorities, how do we implement priority inheritance from messages?

\[
\pi_i = \begin{cases} 
\left\lfloor \frac{i}{s} \right\rfloor & ; i < n \\
\frac{i}{s} & ; i = n
\end{cases}
\]

\[
r = n^{\frac{1}{s}}
\]
Adding terms to schedulability analysis

- **Overhead in scheduling/interrupt task & context switching**
  - jitter in release time due to tick size
  - increase execution time of pre-empting tasks
    - twice context switch time
  - a “new” highest priority task with
    - period = tick size
    - execution time for monitor, schedule, and update functions

- **Loss in schedulability due to insufficient priority levels**
  - re-analyse using the available system priority
  - for tasks with equal priority
    - EDF/FIFO - maximum blocking time = sum of WCET for all other tasks
    - RR - maximum blocking time = WCET * # tasks

Is Utilization-based analysis still useful?
Integrating control, scheduling and Kernel/OS

- For real-time systems, we need to generate:
  - end-to-end task timing (derive job release/deadlines/periods)
  - execution guarantee requirements
  - stability guarantee requirements
  - storage/calculation requirements
  - schedules to maximize utilization/power usage of processor

- How do we ensure that Kernel/OS meets timing needs?
- How do we verify/guarantee control/application specifications?
- What scheduling algorithms/RT features can the Kernel/OS support?
- How are these features related to the control/application specifications?
- How do we ensure that the Kernel/OS can meet storage/calculation/utilization requirements?
High precision timing within Microsoft Windows: threads, scheduling and system interrupts

Goal: Produce uniform frame rate animation

54ms

55ms
BASEMENT: A Distributed Real-Time Architecture for Vehicle Applications

- **Red** -- hard processes
- assumed all Red processes are continuously operating at fixed frequencies
- cyclic, off-line, scheduling

- **Blue** -- soft processes
- intended for processes that are “less safety critical”
- event driven pre-emptive priority driven scheduling
- resources available to the Blue subsystem are those remaining after (the static) allocation of resources to the Red subsystem.
Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS

- **RTU**
  - task scheduling & semaphores
    - dynamic creation and deletion,
  - time management control
    - time ticks and delays
  - improved response time
  - reduced OS memory footprint

- **System-on-a-Chip Lock Cache (SoCLC)**
  - separate “lock cache” outside of memory
  - reduced delay in accessing a lock variable
UML diagrams

- **UML**
  - Unified Modelling Language
  - non-proprietary
  - extendable

- **Why model?**
  - visualise
  - specify
  - creation template
  - document design decisions

  Which function are we interested in?

- **(atomic) Action**
- **(sequence) Activity**

- **Structural (what)**
  - class
  - object
  - component
  - deployment

- **Behavioural (how)**
  - statecharts
  - activity diagrams
  - use cases (diagram)
  - interaction
    - collaboration diagrams
    - sequence diagrams
    - timing diagrams
    - interaction tables
    - overview diagrams
Statecharts

- State
- Transition (event)
- Start/End States
- Sub-states/Orthogonal States
- Actions/Signals
- Transition Guards/Actions
- Pseudo-states
  - Entry/Exit points
  - History
  - Join/Choice


Do these states reflect actions or activity?
Activity diagram

- Initial/Final
- Action
- Fork/Join/Merge
- Constraints
- Partitions (swimlanes)
- Nested activities
- Exceptions

Use case diagram

- Use case - verb
- Actor - noun
- Subject - system/context

Relation to prior diagrams?

Collaboration Diagram

- Scenario concept
- Objects/LifeLine
- Messages

**Sequence Diagram**

- Lifetime
- Multiple scenarios
- Time constraints
Timing Diagram
# Interaction Tables

- **Internal Messages**
- **External event list**
  - event
  - description
  - direction
  - pattern
  - response performance

## Table

<table>
<thead>
<tr>
<th>Lifeline Inst</th>
<th>Constraint</th>
<th>Message Sending Class</th>
<th>Message Sending Instance</th>
<th>Sequence ID</th>
<th>Message Name</th>
<th>Weak Order</th>
<th>Parameter Value</th>
<th>Message Receiving Class</th>
<th>Other End</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>Code</td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>e2i</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>Code</td>
<td>Code</td>
<td>e3o(ref)</td>
<td>e3o(ref)</td>
<td></td>
<td></td>
<td>C</td>
<td>e4i</td>
</tr>
<tr>
<td>C</td>
<td>alt[1] x=5</td>
<td>Code</td>
<td>Code</td>
<td>e5o(ref)</td>
<td>e5o(ref)</td>
<td></td>
<td></td>
<td>B</td>
<td>e6i</td>
</tr>
<tr>
<td>B</td>
<td>alt[2] x=0</td>
<td>Code</td>
<td>Code</td>
<td>e7o</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>e5o</td>
</tr>
</tbody>
</table>

[Line diagram and table generated from the PDF provided]
Requirements Analysis

- Requirements (Use cases)
  - Business
  - User (Functional)
    - primary/secondary
  - Constraints (Non-functional)

- Identification Techniques
  - textual description
  - interview (scenario)
  - comparable systems
  - existing systems
    - maintenance feedback
    - site visit
  - user working group
  - prototype

- Identify stakeholders (Actors)
  - underline the noun
    - interesting
    - uninteresting
    - attributes
    - synonyms
  - causal agents
  - real-world items
    - coherent services
    - physical devices
    - UI: visual/control elements

- Identify objects (actor/attribute)
  - transaction data (messages)
  - persistent information

- Draw UCDs
  - separate business / system contexts
  - check for completeness scenario
    walk-through
For each use case,
- select appropriate scenarios
- draw one UML diagram per scenario
- generate acceptance test for each scenario

“In software engineering, a design pattern is a general solution to a common problem in software design. A design pattern isn't a finished design that can be transformed directly into code; it is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.”


To Control (periodic)
- Detect actual operational state
- Evaluate desired state
- Compare actual and desired
- Control towards desired state (alternatives)
  - (Precondition: too high): Reduce
  - (Precondition: correct): Leave alone
  - (Precondition: too low): Increase

http://easyweb.easynet.co.uk/~iany/consultancy/goalpatt/goalpatt.htm

How do we choose?

Advantages?
Integrating UML, control, scheduling and OS

- For real-time systems, we need to generate:
  - end-to-end task timing (derive job release/deadlines/periods)
  - execution guarantee requirements
  - stability guarantee requirements
  - storage/calculation requirements
  - schedules to maximize utilization/power usage of processor

- How do we express scheduling & RT requirements in UML?
- How do we express OS features (e.g., interrupts) in UML?
- How do we verify/guarantee control/application specifications?
Efficient System Modelling of Complex Real-Time Industrial Networks using the ACCORD/UML methodology
Using UML-Based Rate Monotonic Analysis to Predict Schedulability
Experimenting with Real-time Specification Methods: The Model Multiplicity Problem

- “is inherent in any systems development method that uses a number of models to obtain a complete system specification”
- “likelihood of making errors increases with the number of models in the method.”
- “The students … significantly preferred OPM to OMT.”

<table>
<thead>
<tr>
<th>Issue</th>
<th>Proportion of Students that answered correctly in OMT</th>
<th>Proportion of Students that answered correctly in OPM</th>
<th>Test Statistic₁</th>
<th>p-Value (two-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifying timing exceptions</td>
<td>36/41</td>
<td>44/47</td>
<td>-0.95</td>
<td>0.34</td>
</tr>
<tr>
<td>Identifying cyclic processes</td>
<td>40/41</td>
<td>46/47</td>
<td>-0.01</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 2: Summary of results of the specification comprehension experiment for single-question issues

Table 4. Overall results of the Specification Quality part of the experiment.

<table>
<thead>
<tr>
<th>Overall number of error types per student</th>
<th>OMT Average</th>
<th>OMT STDV</th>
<th>OPM Average</th>
<th>OPM STDV</th>
<th>T-value two tailed</th>
<th>P-value</th>
<th>Difference in favor of</th>
</tr>
</thead>
<tbody>
<tr>
<td>all issues count the same</td>
<td>5.81</td>
<td>2.95</td>
<td>2.80</td>
<td>2.11</td>
<td>-5.36</td>
<td>0.01</td>
<td>OPM</td>
</tr>
<tr>
<td>issues in favor of OMT count 4 times as much</td>
<td>8.49</td>
<td>4.69</td>
<td>6.32</td>
<td>4.60</td>
<td>-2.16</td>
<td>0.05</td>
<td>OPM</td>
</tr>
</tbody>
</table>
Quantifying reliability

- Fault -> Error -> Failure
- Hardware
  - lifetime of component (before failure)
- Software
  - execution time (before failure)
  - reliability-growth (density of errors/faults)
- Hazard rate \( h(t) \)
  - instantaneous ratio of number of failures in time interval to number of survivors at beginning of interval
  - Typical bathtub shape
- Lifetime distribution \( Q(t) \)
  - probability distribution function (PDF) of component lifetime
  - i.e. likelihood of a given component failure over time
- Reliability \( R(t) \)
  - probability that system does not fail within required lifetime
- Mean time to failure \( MTTF \)

\[
Q(t) = 1 - e^{-\int_{0}^{t} h(\xi) \, d\xi}
\]

\[
R(t) = 1 - Q(t)
\]

\[
MTTF = \int_{0}^{\infty} R(t) \, dt
\]

How to achieve reliability given a “bathtub” curve. Any model assumptions?
Measuring & Estimating Reliability

- Component failure [hazard] rates $f(t)$ [$h(t)$]
  - Historical data (field/lab)
  - Accelerated (lab)
  - Distributions
    - Poisson (Constant - hazard) \( f(t) = \lambda e^{-\lambda t}; h(t) = \lambda \)
    - Weibull \( f(t) = \beta \psi(\psi t)^{\beta-1} e^{-(\psi t)^{\beta}}; h(t) = \beta \psi(\psi t)^{\beta-1} \)
      - \( \beta = 1 \) as for constant-hazard
      - \( 0 < \beta < 1 \) hazard rate decreases with time
      - \( \beta > 1 \) hazard rate increases with time

- Estimating structural reliability
  - series (chain)
  - parallel (1-out-of-n)
  - r-out-of-n
  - NMR

\[
\begin{align*}
R_s(t) &= \prod_{i=1}^{n} R_i(t) \\
R_p(t) &= 1 - \prod_{i=1}^{n} [1 - R_i(t)] \\
R_{r-out-of-n}(t) &= \sum_{k=r}^{n} \binom{n}{k} R(t)^k [1 - R(t)]^{n-k} \\
R_{n-out-of-NMR}(t) &= R_{voter}(t) \times \sum_{k=n+1}^{N} \binom{N}{k} R(t)^k [1 - R(t)]^{N-k}
\end{align*}
\]
Assumptions
- permanent failure
  - transient/intermittent faults
- no common mode failures
  - environment (correlated transients)
  - production flaws
- perfect voters
  - critical ratio
  - voter reliability drops (workload increases) with each input

Economics of reliability/repair
- cost of redundancy/standby’s
- cost of repair/replacement
- cost of outage (maintenance/failure)

RT performance
- Error propagation time
  - Historical data (field)
  - Simulation (lab)
- MTTR(Repair/Recover), MTBF (between)=MTTF+MTTR

Can we model response times from a hazard rate?
Software reliability models

- What is time?
  - clock time (cycles)
  - execution time
  - calendar time

- Software reliability can be characterised similar to hardware BUT
  - we observe execution failures/errors and attempt to fix faults
  - fix/change may introduce further flaws
  - we cannot identify a design fault unless specification changes

- Measures
  - predictive execution time/failure rate
    - based on historical data for similar designs and operations
    - presumes that different programmers/designs are same
  - reliability growth (error model)
    - measure the number of failures/errors per unit time as fixes are applied
    - stop when “desired” reliability achieved
    - presumes that faults/errors/failures are independent
  - we cannot identify a fault until it produces an error/failure

Is it realistic to use software reliability models?
Prevention, Masking, Recovery, Evasion

- **Error Reduction/Prevention**
  - design/create to minimise faults
  - prototype test for errors/failure
  - re-design to eliminate faults

- **Error Masking/Containment**
  - detect error
  - ignore fault
  - prevention of failure

- **Error Recovery**
  - detect error
  - id/correct fault
  - re-run -- prevention of failure

- **Error Evasion**
  - detect (potential) faults
  - action prior to error/failure
  - prevention of fault

Can you think of examples in each area/aspect?

- Hardware
- Software
- Data (Information)
- Scheduling (Time)
- Environment (Layout)
Prevention: Design

- Use of standards
  - expression of design
  - pre-existing designs (templates)
  - manage design process
  - qualification procedures
- Standards exist in each area/aspect

“Integrating UML Real-Time and IEC 61131-3 with Function Block Adapters”
  - UML Real-Time: variant of UML used to control IPC (robot)
  - IEC61131-3: PLC (transport) language standard
  - “The main purpose of the FBA-Language is to give developers of both UML-RT and IEC 61131-3 a common language for the specification of adapters between components of their models.”
Test design
- full coverage; maximise errors
- black box (behavioural)
  - derive from specifications/use cases
- white box (structural)
  - derive from program/algorithm structure (state-charts)
- levels
  - unit, component, integration, regression, system, acceptance

“Integration Testing of Fixed Priority Scheduled Real-Time Systems”
- White box test based on task requirements.
- Job B (3 sub-jobs) accesses a shared resource, and when entering the critical section boost its priority.
- Execution Order Graph, shows all possible paths to completion.
- Ensure tests cover EOG.
Masking: Hardware Modular Redundancy

- Standby unit
  - switching

- Static Pair
  - no voter/correction
  - detects problem iff not common mode

- NMR (with spares)
  - voter
  - faulty modules removed
  - spares used to replace units detected as faulty and/or N decreased by 2.

- Non-identical modules
  - removes some common mode issues

- Synchronising inputs/outputs
  - value: clusters (voter)
  - timing: timeouts (timestamp)

- Voter Types
  - formalised majority
    - choose cluster > N/2; any member
  - generalised k-plurality
    - choose cluster > k; any member
  - generalised median
    - choose median output

- Multiple voters
  - Full/partial input set?
  - Replicating memory
    - vote before read/write

---

We have 4 identical processors: 2 static pairs, 3MR with voter. Which is more effective? Why?

We have 2 pairs of non-identical processors:
2 static pairs (same), 2 static pairs (different), 3MR with voter (which?)
Masking: Software Multiple versions

- **Copies**
  - use same executable
  - possibly store in different areas

- **Tools: Compiler/IDE**
  - use same source code
  - for different processors
  - using different compilers

- **Implementation Team**
  - same specification/tests
  - different source code/compiler

- **N-Version**
  - run different versions in parallel
  - vote for acceptable output

- **Retry (Recovery block)**
  - run different versions in sequence
  - stop at first acceptable output
  - roll-back to start between versions

“SIFT: Design & Analysis of a fault tolerant computer for aircraft control”
  - message passing; no shared memory
  - complete interconnection
  - OS responsible for reconfiguration/voting
  - 5MR - 78% overhead

How to define an acceptance test?
Teams/compilers truly independent?
Which method is most effective?
Masking: Data, Scheduling & Environment

- **Duplication**
  - D: data sent twice
  - S: copies scheduled at different priorities/times
  - E: separation, shielding

- **Data**
  - Parity
    - single bit (1 error)
    - multiple bits (1 error per portion)
    - P-word
    - interlaced
  - Checksum
    - “sum” of words in block
  - Codes
    - known set of codes maps 1-1 to valid data
    - cyclic: all codes are 1-bit shift away from another valid code

- **Scheduling/OS**
  - Watchdog
    - non-completion/ restart
  - Stable algorithms
    - HP processes unaffected

- **Environment**
  - Temp/pressure/radiation
    - sensor compensation

http://www9.dw-world.de/rtc/infotheque/digital_signal/fig3522.gif
Recovery: Hardware

- Re-configurable
  - spares
  - modular (maintenance)
  - reliable switching

- High Availability
  - %: how many nines?
  - redundancy/masking
  - degradation of service
    - “3-3-2-0”, “3-2-1-0”, “3-2-0”

- PC vs PLC
  - custom hardware vs. COTS
  - proprietary vs. open standards
  - service, spares, support

- “Throw-away” mentality
  - run diagnostics
  - return to service if passed
  - request replacement if failed

- “Windows Hardware Error Architecture”
  - 7-10% of all reported crashes are due to hardware errors
  - No common mechanism for discovery of hardware error sources
  - How will we repair? FPGA’s?

http://download.microsoft.com/download/.../TWAR05008_WinHEC05.ppt
Recovery Block
- software redundancy
- “roll-back” to start (check-point) when output is not acceptable
- Irreversible actions?

Check-point
- “copy” of process state

Logging (audit trail)
- record of non-deterministic events subsequent to start (checkpoint)

Optimal checkpoint placement for recovery blocks

<table>
<thead>
<tr>
<th>PWD Assumed?</th>
<th>Uncoordinated Checkpointing</th>
<th>Coordinated Checkpointing</th>
<th>Pessimistic Logging</th>
<th>Optimistic Logging</th>
<th>Causal Logging</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Overhead</td>
<td>Low</td>
<td>Higher</td>
<td>Highest</td>
<td>Higher</td>
<td>Higher</td>
</tr>
<tr>
<td>Output Commit</td>
<td>Not possible</td>
<td>Very slow</td>
<td>Fastest</td>
<td>Slow</td>
<td>Fast</td>
</tr>
<tr>
<td>Checkpoint/process</td>
<td>Several</td>
<td>1</td>
<td>1</td>
<td>Several</td>
<td>1</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>Complex</td>
<td>Simple</td>
<td>Simple</td>
<td>Complex</td>
<td>Complex</td>
</tr>
</tbody>
</table>

Recovery
- Complex

Domino Effect
- Possible

Orphans
- Possible

Rollback Extent
- Unbounded
- Last checkpoint

“Survey of Rollback-Recovery Protocols in Message-Passing Systems”
- checkpointing: uncoordinated, coordinated, communication-induced
- logging: pessimistic, optimistic, causal
Recovery: Data, Scheduling & Environment

- **Data**
  - Resend
  - Estimate from history/other data

- **Scheduling**
  - add slack to allow for re-run - FT-RMA

- **Environment**
  - control fans, shields etc
  - OS kill unnecessary tasks

---

**“Minimum Achievable Utilisation for Fault Tolerant Processing of Periodic Tasks”**

- RMA: $U < 0.69$ for large $n$
- Consider:
  - system susceptible to a single fault
  - recovery action is the re-execution of all uncompleted tasks.
  - RMA priorities maintained even during recovery.
- $U < 0.5$ is the minimum achievable utilization that permits recovery from faults before the expiration of the deadlines of the tasks.
- Better than the trivial bound of $0.69/2$ if computation times were doubled to provide for re-execution.
- Can be used for re-execution on same/spare platform.
Evasion

- Data
  - diversity
    - deliberately perturb inputs/reconcile outputs in redundant systems
  - scrubbing
    - periodically correct memory

- Hardware
  - regular switchover
    - ensures spares are functional
  - diagnostics
    - self-tests on system

- Software
  - reversal checks
  - early termination
    - recursive: less accurate

- Scheduling
  - admission policy

- Environment
  - monitoring: power, temperature, pressure, radiation

“System-Level Power-Aware Design Techniques in Real-Time Systems”
- low power instructions
- reduce bus transitions
- voltage/frequency scaling
- optimize battery discharge
- reduce overhead
  - cache use
  - checkpointing
Malicious faults

Assumptions made
- reduction/prevention
  - consistent use of standards; good test data
- masking/containment
  - redundant systems receive same input
- recovery
  - perfect switching/storage
- evasion
  - correct evasion tactics

Byzantine fault
- “one in which a component of some system not only behaves erroneously, but also fails to behave consistently when interacting with multiple other components.”
- “an arbitrary fault that occurs during the execution of an algorithm”
  - crash
  - failure to correctly execute algorithm step
  - arbitrary execution of a step other than the one indicated by algorithm

How can we handle these faults?

Reliably integrating UML, control, scheduling, OS

- For real-time systems, we need to generate:
  - end-to-end task timing (derive job release/deadlines/periods)
  - execution guarantee requirements
  - stability guarantee requirements
  - storage/calculation requirements
  - schedules to maximize utilization/power usage of processor


- How do we measure/estimate/express reliability?
- How do we express reliable/FT features in UML?
- How do we get reliable task execution using scheduling?
- What features are required in the OS for reliable/FT operation?
- How do we reliably meet control requirements?
- How can we use historical data to take preventative action?
A unified method for evaluating real-time computer controllers and its application

- **Performance Measures**
  - probability of dynamic failure
    - pass/fail criterion
    - increases with deadline
    - sub-tasks: conditional hard deadlines
  - mean modified cost
    - ranking criterion
    - determine if increase as deadline increases

- **State space**
  - area surrounding the desired trajectory
  - for a range of deadlines:
    - subdivide into regions which will/will not result in dynamic failure, if deadline missed
    - for region with no failure, determine cost for valid deadlines

- **Scheduling**
  - high priority to those with high incremental cost

- **Specification/Evaluation**
  - rank alternates by cost; specify acceptable cost

- **# Checkpoints**
  - tradeoff: storage time vs. replay time

- **Redundancy**
  - tradeoff: reduced failure vs. overhead for SIFT
Fault-Tolerant Air Data/Inertial Reference Unit

- **ADIRS**
  - connected to flight computers x 3
    - three buses - one each flight computer for r/w, the other two for r/o
    - bus I/O modules (dual redundant)
  - **ADIRU**
    - fault tolerant: fail-op/fail-op/fail-safe
    - laser gyros x 6
    - accelerometers x 6
    - processors x 4
    - clock x 4
    - power supply x 3
  - **SAARU (Secondary Air-data Attitude Reference Unit)**
    - 100% monitored
    - IFO gyros x 4
    - accelerometers x 4
    - processors x 2
  - **ADM (Air-Data Modules)**
    - static pressure x 3
    - total pressure x 3

- **Software**
  - OS manages
    - CPU time,
    - memory,
    - I/O functions
  - 3 tasks
    - fixed time slots (100Hz frame)
    - read-only access to other task memory

- **Voting**
  - bit-by-bit on output of I/O modules (frame sync)
  - clocks sync every 100Hz
  - power distribution buses w/ isolation

- **Redundant components** separated physically & electrically
Real time recovery of FT processing elements

- How to recover SIFT after transient fault causes failure?
  - 5 - 100 times more frequent than permanent failure
- Re-sync as additional task
- “tag” used memory segments
- tested on undersea vehicle
  - ten 24Hz control loops
  - normal align suspends for 48 cycles
  - algorithm re-aligned in 36 seconds, suspended control for 2 cycles

MD11-AFS

- Integrates functions of different degrees of criticality
- 3 dissimilar processors
- 3-version software
- Dual (mirrored) configuration
Software-implemented fault-tolerance and separate recovery strategies enhance maintainability (substation automation)

Fig. 2. Changing recovery strategies from three-and-a-spare to graceful-degradation.

Software-implemented fault-tolerance and separate recovery strategies enhance maintainability (substation automation).

- **Automatic power resumption** when a HV/MV transformer goes down, e.g. triggered by internal protection (temperature too high, oil alarm, …). It disconnects the MV lines connected to the busbar of the transformer, computes the load carried by the transformer just before the event happened, and if possible, causes the remaining transformer to take the entire load.

- **Parallel transformers** a series of automatic actions, assisting remote operators. E.g., an operator can request to switch on a transformer and function2 translates this request into a specific sequence of commands. Such a re-insertion scenario may be applied some time after transformer exclusion.

![Figure 1: Electric circuit (grey lines) and control architecture (black lines) of Primary Substation](image-url)
Case study guidelines

- Use Case diagram
  - Event table
  - Behavioural UML diagrams
  - Timing diagrams
- Task selection/specification
  - Resource sharing
  - Communication
- Choose Scheduler/OS
  - Compromises
  - Special features
- Implement
  - Pseudo code
  - UML diagrams (static)
- Analysis/Criticism

Cases:
- Flight controller
- Intruder alarm
- Submarine sonar
- Postal address
- 1000 tonne press
- Radar signal processing

- Does anyone really use this stuff?
  - Control
    - NIF
  - Telecommunications
    - Nortel CS2100
  - Power Generation/Distribution
    - EX2100 Generator Excitation Control
  - Electronics
    - XILINX FPGA's Functional TMR

Exam review