

Reliable Computing I

Lecture 12: Software Fault Tolerance

Instructor: Mehdi Tahoori

INSTITUTE OF COMPUTER ENGINEERING (ITEC) – CHAIR FOR DEPENDABLE NANO COMPUTING (CDNC)



KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.kit.edu

Why Software Fault Tolerance ?

- Can increase software reliability via fault avoidance using software engineering and testing methodologies
- Large and complex systems
 - fault avoidance not successful
- Redundancy in software may be needed to detect, isolate, and recover software failures
- Software is difficult to prove correct

Hardware vs. Software Faults



- | | |
|--|--|
| <ul style="list-style-type: none">■ Hardware faults<ul style="list-style-type: none">■ Faults time-dependent■ Duplicate hardware detects■ Mainly due to random cause | <ul style="list-style-type: none">■ Software faults<ul style="list-style-type: none">■ Faults time-invariant■ Duplicate software not effective■ Complexity is the main cause |
|--|--|

Sources of Unreliability: Software Failures



- High complexity of software is the major contributing factor of Software Reliability problems
- Software failures causes
 - Errors
 - Ambiguities
 - Oversights or misinterpretation of the specification
 - The software is supposed to satisfy
 - Carelessness or incompetence in writing code
 - Inadequate testing
 - Incorrect or unexpected usage of the software
 - Other unforeseen problems...

Experiences with Current Software



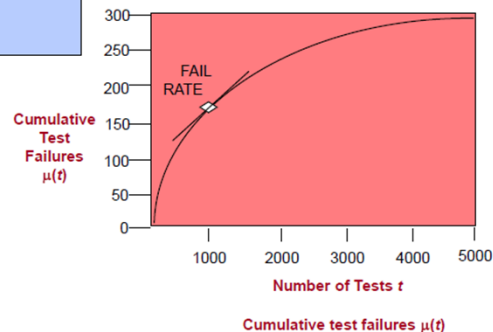
- Many computer crashes are due to software
- Even though one expects software to be correct, it never is
- Mature software exhibits fairly constant failure frequency
- Number of failures is correlated with
 - Execution time
 - Code density
 - Software timing, synchronization points

Experiences with Current Software



Key parameters and variables (with defect reintroduction)

Defect Detection Time Constant s	17.2 Weeks
Defect Repair Time Constant t	4.7 Weeks
Code Delivery	589810 Lines
Initial Error Density α	0.00387 Defects per Line
Defect Reintroduction Rate β	33 Percent
Deployment Time T	Week 100
Estimated Remaining Defects ERD_T	664 Defects
Estimated Current Defects ECD_T	445 Defects
Testing Process Quality TPQ_T	90 Percent
Testing Process Efficiency TPE_T	60 Percent



Difficulties



- Improvements in software development methodologies reduce the incidence of faults, yielding fault avoidance
- Need for test and verification
- Formal verification techniques, such as proof of correctness, can be applied to rather small programs
- Potential of faulty translation of user requirements
- Conventional testing is hit-or-miss.
 - “Program testing can show the presence of bugs but never show their absence,” - Dijkstra, 1972.
- There is a lack of good fault models

Approaches to Software Fault Tolerance



- **ROBUSTNESS**: The extent to which software continues to operate despite introduction of invalid inputs.
 - Example:
 - 1. Check input data
 - ask for new input
 - use default value and raise flag
 - 2. Self checking software
- **FAULT CONTAINMENT**: Faults in one module should not affect other modules.
 - Example:
 - Reasonable checks
 - Watchdog timers
 - Overflow/divide-by-zero detection
 - Assertion checking
- **FAULT TOLERANCE**: Provides uninterrupted operation in presence of program faults through multiple implementations of a given function

Approaches to Software FT



- N Version Programming
- Recovery Blocks
- Process Pairs
- Robust Data Structures
- ...

Concepts of N-Version Programming

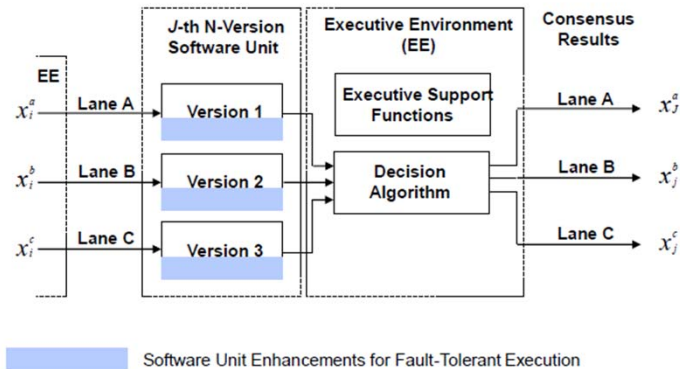


- $N \geq 2$ versions of functionally equivalent programs
- “Independent” generations of programs
 - carried out by N groups of individuals who do not talk to each other with respect to programming process
 - different algorithms, different programming languages, translation
- Initial specification formally done in some formal spec. language
 - states **unambiguously** the functional requirements
 - leaves widest possible choice of implementation
- By making the development process diverse it is hoped that the versions will contain diverse faults
- The inventors of NVP emphasized that:
 - “the definition of NVP has never postulated an assumption of independence and that NVP is a rigorous process of software development”

NVP Basic Model



The N-version software (NVS) model with $n=3$



Independence in N-Version Programming ?



- Do the N versions of a program fail independently (similar to hardware)? Are faults unrelated?
- Does $\text{Prob}(\text{failure of N-version system}) = \text{Prob}(\text{failure of one version})^N$??
 - If so, then the system reliability can be very high
- Why such an assumption may be false?
 - People make same mistakes, e.g. incorrect treatment of boundary conditions
 - Some parts of a problem more difficult than others
 - statistics show similarity in programmer's view of "difficult" regions

Limitation of N-Version Programming



- All N -versions originate from the same initial specifications whose correctness, completeness, and unambiguity should be assumed
 - Use formal correctness proofs on specs, rather than proofs on implementations
 - Exhaustive validation
- Based on an assumption that software faults are distinguishable:
 - faults that will cause disagreement between versions at specified voting points might be a result of independent programming efforts to remove identical software defects

Concepts of Recovery Blocks

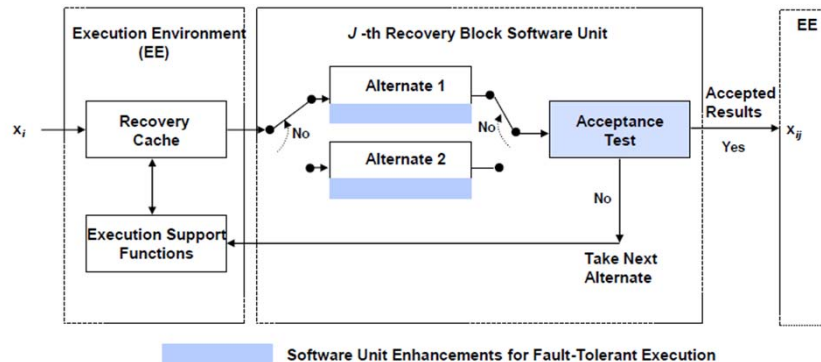


- Characteristics:
 - Incorporates general solution to the problem of switching to spare
 - Explicitly structures a software system so that extra software for spares and error detection does not reduce system reliability
 - First to consider a single sequential process; later extended to
 - Multiple processes within one system
 - Multiple processes in multiple systems → distributed recovery blocks
- Can view progress as sequences of basic operations, assignments to stored variable
- Structured program has BLOCKS of code to simplify understanding of the functional description
- Choose blocks as units for error detection and recovery.

Recovery Blocks Basic Model



The Recovery Block (RB) Model



Acceptance Tests



- Function: ensure the operation of recovery blocks is satisfactory
- Should access variables in the program, NOT local to the recovery block, since these cannot have effect after exit. Also, different alternates use different local variables.
- Need not check for absolute “correctness” - cost/complexity trade-off
- Run-time overheads should be LOW
- NO RESIDUAL EFFECTS should be present, since variables, if updated, might result in passing of successive alternates

Restoration of System State



- Restoring system state is automatic
- Taking a copy of entire system state on entry to each recovery block is too costly
- Use Recovery Caches or “Recursive” Caches
- When a process is to be backed up, it is to a state just before entry to primary alternate
- Only NONLOCAL variables that have been MODIFIED have to be reset

Recovery Blocks vs. NVP



■ Advantages of Recovery Block

- Most software systems evolve by replacement of some modules by new ones - can be used as alternates
- Nice hierarchical design - structured approach

■ Disadvantages of Recovery Block

- System state must be saved before entry to recovery block -- excessive storage
- Difficult to handle multiple processes -- might have domino effect
- Difficult to undo effects in real-time systems
- Effectiveness of acceptance test
- Higher coverage is more complex
- Lack of formal method to check

Recovery Blocks vs. NVP



■ Advantages of N-Version Programming

- Immediate masking of software faults -- no delay in operation
- Self-checking (acceptance tests) not required
- Conventional fault tolerant systems HW and SW have redundant hardware e.g. TMR (easier to include N-version software on redundant hardware)

■ Disadvantages of N-Version Programming

- How to get N-versions?
 - Impose design diversity, since randomness does not give uncorrelated software faults
- Extremely dependent on input specifications (formal correctness proofs...)

Process Pairs



■ Applicability

- Permanent and transient hardware and software failures
- Loosely coupled redundant architectures
- Message passing process communication
- Well suited for maintaining data integrity in a transactional type of system
- Can be used to replicate a critical system function or user application

■ Assumptions

- Hardware and software modules design to fail-fast, i.e., to rapidly detect errors and subsequently terminate processing
- Errors can be corrected by re-executing the same software copy in changed environment

Process Pairs - Overview



- The user application is replicated on two processors as primary and backup processes, i.e., as process pairs
- Normally, only the primary process provides service
- The primary sends checkpoints to the backup
- The backup can take over the function when the primary fails
- The operating systems halts the processor when it detects non-recoverable errors
- The “*I am alive*” message protocol allows the other processors to detect the halt and to take over the primaries that were running on the halted processor

Robust Data Structures



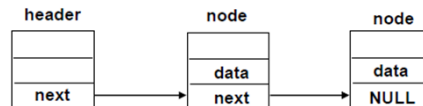
- The goal is to find storage structures that are robust in the face of errors and failures
- What do we want to preserve?
 - Semantic integrity - the data meaning is not corrupted
 - Structural integrity - the correct data representation is preserved
- A robust data structure contains **redundant data** which allow **erroneous changes** to be detected, and possibly corrected
 - a change is defined as an elementary (e.g., as single word) modification to the encoded (data structure representation on a storage medium) form of a data structure instance
 - structural redundancy
 - a stored count of the numbers of nodes in a structure instance
 - identifier fields
 - additional pointers

Link Lists



■ Non-robust data structure

- in each node store a pointer to the next node of the list
- place a null pointer in the last node



0-detectable and 0-correctable
changing one pointer to NULL can
reduce any list to empty list

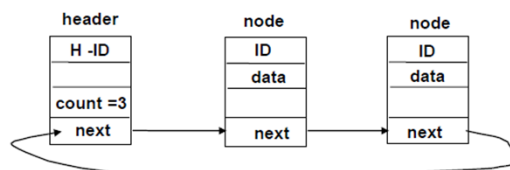
Linked Lists



■ Single-Linked List Implementation

■ Additions for improving robustness

- an identifier field to each node
- replace the NULL pointer in the last node by a pointer to the header of the list
- stores a count of the number of nodes



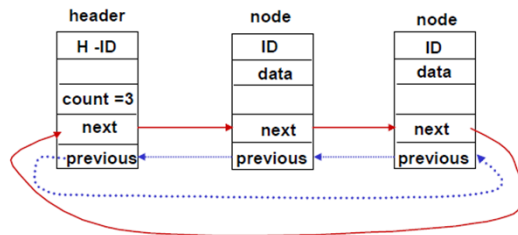
1-detectable and 0-correctable

- change to the count can be detected by comparing it against the number of nodes find by following pointers
- change to the pointer may be detected by a mismatch in count number or the new pointer points to a foreign node (which cannot have a valid identifier)

Linked Lists



- **Double-Linked List Implementation**
- Additions for improving robustness
 - a pointer added to each node, pointing to the predecessor of the node on the list



2-detectable and 1-correctable
the data structure has two independent, disjoint sets of pointers, each of which may be used to reconstruct the entire list

(c) 2013, Mehdi Tahoori

Reliable Computing I: Lecture 12

25

Robust Data Structures



- Commonly used techniques for supporting robust data structures
 - techniques which preserve structural integrity of data
 - binary trees, heaps, fifos, queues, stacks
 - linked data structures
 - content-based techniques
 - checksums, encoding
- Limitations
 - not transparent to the application
 - best in tolerating errors which corrupt the structure of the data (not the semantic)
 - increased complexity of the update routines may make them error prone
 - erroneous changes to the data structure may be propagated by correct update routines
 - faulty update routines may provoke correlated erroneous changes to several fields

(c) 2013, Mehdi Tahoori

Reliable Computing I: Lecture 12

26