Outline

- Definition and Motivation for Software Fault Tolerance
- Process pairs
- Robust data structures
What is Software Fault Tolerance?

- Three alternative definitions
  1. Management of faults originating from defects in design or implementation of software components
  2. Management of hardware failures in software
  3. Management of network failures
- We will follow the classical definition (1) due to Avizienis in 1977

Motivation for Software Fault Tolerance

- Usual method of software reliability is fault avoidance using good software engineering methodologies
- Large and complex systems \( \Rightarrow \) fault avoidance not successful
  - Rule of thumb fault density in software is 10-50 per 1,000 lines of code for good software and 1-5 after intensive testing using automated tools
- Redundancy in software needed to detect, isolate, and recover from software failures
- Hardware fault tolerance easier to assess
- Software is difficult to prove correct

<table>
<thead>
<tr>
<th>HARDWARE FAULTS</th>
<th>SOFTWARE FAULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Faults time-dependent</td>
<td>Faults time-invariant</td>
</tr>
<tr>
<td>2. Duplicate hardware detects</td>
<td>Duplicate software not effective</td>
</tr>
<tr>
<td>3. Random failure is main cause</td>
<td>Complexity is main cause</td>
</tr>
</tbody>
</table>
Consequences of Software Failure

- General Accounting Office reports $4.2 million lost annually due to software errors
- Launch failure of Mariner I (1962)
- Destruction of French satellite (1988)
- Problems with Space Shuttle and Apollo missions
- SS7 (signaling system) protocol implementation - untested patch (mistyped character) (1997)
- Therac 25 (overdose of medical radiation 1000’s of rads in excess of prescribed dosage)
- Toyota Prius recall (2004) due to bug in embedded code

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 exists 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 for software defects
Forms of Software Testing

- Exhaustive testing of reasonable sized applications is impossible.
- Approach is to define equivalence classes of inputs so that only one test case from each class suffices.
- Techniques proposed include:
  - Path testing
  - Branch testing
  - Interface testing
  - Special values testing
  - Functional testing
  - Anomaly analysis
- Studies have shown path testing and interface testing while difficult to design afford good coverage for large number of applications.

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 fault through multiple implementations of a given function.
Temporal Redundancy

- Reexecution of a program when error is encountered
- Error may be faulty data, faulty execution or incorrect output
- Reexecution will clear errors arising from temporary circumstances
- Examples: Noisy communication channel, Full buffers, Power supply transients, Resource exhaustion in multiprocess environment
- Provides fault containment
- Possible to apply to applications with loose time constraints

Multi-Version Software Fault Tolerance

- Use of multiple versions (or “variants”) of a piece of software
- Different versions may execute in parallel or in sequence
- Rationale is that multiple versions will fail differently, i.e., for different inputs
- Versions are developed from common specifications
- Three main approaches
  - Recovery Blocks
  - N-version Programming
  - N Self-Checking Programming
Recovery Blocks

- Checkpoint and restart approach
  - Try a version, if error detected through acceptance test, try a different version
  - Ordering of the different versions according to reliability
- Checkpoints needed to provide valid operational state for subsequent versions
- Acceptance test could be on output or embedded in code

Due to Brian Randell, first appeared in ToSE 1975

N-Version Programming

- All versions designed to satisfy same basic requirement
- Decision of output comparison based on voting
- Different teams build different versions to avoid correlated failures

Due to Al Avizienis, first appeared in CompSAC 1977
N Self-Checking Programming

- Multiple software versions with structural variations of RB and NVP
- Use of separate acceptance tests for each version

Due to J. C. Laprie, FTCS 87

Reliability Analysis of Multi-Version Approaches

- Three postulates of software development:
  P1: Complexity Breeds Bugs: Everything else being equal, the more complex the software project is, the harder it is to make it reliable.
  P2: All Bugs are Not Equal: You fix a bunch of obvious bugs quickly, but finding and fixing the last few bugs is much harder, if you can ever hunt them down.
  P3: All Budgets are Finite: There is only a finite amount of effort (budget) that we can spend on any project. That is, if we go for n version diversity, we must divide the available effort n-way.

- $R(t) = e^{-\lambda t}$
- Failure rate $\lambda \propto 1/$Effort (E)
- Failure rate $\lambda \propto$ Complexity (C)
Reliability of NVP vs. single version

Reliability of RB vs. Simplex
Process Pairs

- Used in HP Himalaya servers as part of their NonStop Advanced Architecture
- Bragging rights of the architecture
  - Run the majority of credit and debit card systems in N.America
  - More than US$3 billion of electronic funds transfers daily
  - Run many of the E911 systems in North America
- Primary and backup processes on two different processors
- Primary process executes actively
  - Backup process is kept current by periodically sending state of primary process
- Processors execute fail-stop failure
  - When processor failure detected, backup takes over

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 Mechanism in Tandem Guardian OS

1. The application executes as Primary
2. Primary starts a Backup on another processor
3. Duplicated file images are also created
4. Primary periodically sends checkpoint information to Backup
5. Backup reads checkpoint messages and updates its data, file status, and program counter
   - the checkpoint information is inserted in the corresponding memory locations of the Backup
7. Backup loads and executes if the system reports that Primary processor is down
   - the error detection is done by Primary OS or
   - Primary fails to respond to “I am alive” message
8. All file activities by Primary are performed on both the primary and backup file copies
9. Primary periodically asks the OS if a Backup exists
   - if there is no Backup, the Primary can request the creation of a copy of both the process and file structure

Evaluation of Process-Pairs

- Done for Tandem’s Guardian OS Studied Tandem Product Report (TPR) which are used to report product failures
- Problem classified as software fault only after analysts have pinpointed the cause
- Classes of software faults (not exhaustive)
  - Incorrect computation (3%)
  - Data fault (15%)
Results from Evaluation

- Out of total software failures, 138 out of 169 (82%) caused single processor halt (recoverable). This is a measure of the software fault tolerance of the system.
- Reasons for multiple processor fault
  - Same fault as in the primary: 17/28 (60%)
  - Second fault during job recovery: 4/28 (14.3%)
  - Second halt is not related to process pairs: 4/28 (14.3%)

Results from Evaluation

- Reasons for uncorrelated software fault
  - Backup reexecutes same task, but same fault not exercised: 29%.
    - Different memory state
    - Race or timing related problem
  - Example:
    - Privileged process on primary requests a buffer
    - Because of high user activity on primary, buffer exhaustion
    - Bug in buffer management routine and returns “success”
    - Primary privileged process uses uninitialized buffer pointer and causes processor halt
    - Backup process served the request after takeover
    - But buffer was available on the backup processor
Results from Evaluation

- Reasons for uncorrelated software fault
  - Backup does not reexecute failed request on takeover: 20%.
    - Processor monitoring task
    - Interactive task
  - Effect of error latency: 5%
    - Task that caused the error finished before detection
    - Example: I/O process for copying buffer from source to destination.
    - Copied an additional byte overwriting buffer tag.
    - No problem in data transfer.
    - The successful data transfer was checkpointed but not the corrupted buffer tag
    - Problem surfaces later when buffer manager verifies buffer.
    - No problem when reexecuting on backup.
Results from Evaluation

- Process pairs with checkpointing and restart recovers from 75% of reported software faults that result in processor failures.
- The complexity of process pairs introduces some faults
  - 16% of single processor halts were failures of backup processes.
- Counter-intuitive result since same software run on both processors.
- Loose coupling between processors, long error latency, operation using checkpoints and not lock-step.
- Are process triples better than process pairs?

Process Pairs
Advantages & Disadvantages

**Advantages**
- Extremely successful in Tandem OLTP applications
- Tolerates hardware, operating system, and application failures
- High coverage (> 90%) of hardware and software faults
- The backup does not significantly reduce the performance

**Disadvantages**
- Necessity of error detection checks and signaling techniques to make a process fail-fast
- Process pairs are difficult to construct for non-transaction-based applications
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

Focus on techniques for preserving the structural integrity

Robust Data Structures (cont.)

- 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
Robust Data Structures (cont.)

- Consider data structure which consists of a **header** and a set of **nodes**
  - the header contains
    - pointers to certain nodes of the instance or to parts of itself
    - counts
    - identifier fields
  - a node contains
    - data items
    - structural information: pointers and node type identifier fields

- Error detection and correction
  - in-line checks may be introduced into normal system code to perform error detection and possibly correction, during regular operation

---

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

```
header
  ↓
next

node
  ↓
data
  ↓
next

node
  ↓
data
  ↓
NULL
```

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

![Single-Linked List Diagram]

1-detectable and 0-correctable
- change to the count can be detected by comparing it against the number of nodes found 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)

Robust Data Structures
Double-Linked List Implementation

- Additions for improving robustness
  - a pointer added to each node, pointing to the predecessor of the node on the list

![Double-Linked List Diagram]

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
Error Correcting in Double-Linked List

- **Scan the list in the forward direction** until an identifier field error or forward/backward pointer mismatch is detected.
- When this happens, **scan the list in the reverse direction** until a similar error is detected.
- **Repair the data structure**

Header

- H
- ID
- count =3

Node B

- ID
- data
- C (F)

Node C

- ID
- data
- A

Correction

\[(\text{Local}_\text{Ptr}_B = \text{Back}_\text{Ptr}_C) \Rightarrow \text{Next}_\text{Ptr}_B := \text{Local}_\text{Ptr}_C\]

\[(\text{Next}_\text{Ptr}_B = \text{C})\]

Node F

Application of Robust Data Structures for Semantic Error Checking

- **Application to checking index corruption in B-trees**
- **See class presentation**
Robust Data Structures
Concluding Remarks

- 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

References

- Multi-version software
- Process pairs
- Robust data structures