Paranoid Programming

Techniques for Constructing Robust Software

Rick Harper
Stratus Computer, Inc.

rick_harper@alum.mit.edu

“The action cannot be completed because Unknown is busy.”
$\mu$Word, when opening this document.
Attitude Adjustment

Stratus sells Continuous Availability (CA) computers

Customers expect CA computers to run 24 hours per day, 365 days per year

Software errors are a leading cause of system downtime

Software quality and robustness are especially important
Prime Directive

Code written for Continuously Available systems should

Work correctly regardless of input, system load, or state
   Not be the source of system failure through action or inaction
Contain and not propagate errors
   Properly diagnose and reject all bad input
   Recover from errors and bad state
   Make the consequence of the error proportional to its severity
Log significant events for later debugging
Evolve compatibly over time

It should be paranoid!
Paranoid Programming Course

This presentation outlines some techniques for developing paranoid code

Based on intensive one-day course taught at Stratus

Course Objectives

Understand the effects of other people’s hardware and software faults on computer system dependability

Acquire a tool kit of software construction techniques to help reduce the occurrence of failures due to other people’s hardware and software faults

Be able to implement these techniques effectively on current and planned projects
Course Outline

- Nature and Significance of the Problem
- Terminology and Buzzwords
- Software Techniques for Tolerating Errors
  - General Framework and Observations
  - Checkpoint and Rollback
  - State Rejuvenation
  - Recovery Block
  - Process Pairs
  - MultiVersion Programming
  - Process Groups
  - Robust Data Structures
  - Structure Marking
  - Control Flow Monitoring
  - Programming in a High-Availability Environment

- Techniques for Reducing Bugs
  - Maintaining a Ship and Debug Version
  - Assertions
  - Designing Error-Resistant Interfaces
  - Avoiding Memory Theft
  - Making the Compiler Work for You
  - Avoiding Risky Coding Style
  - Error Handling and Reporting Principles
  - Concurrent Programming
  - Testing
  - Inspections
NATURE AND SIGNIFICANCE OF THE PROBLEM
Causes of Outages

<table>
<thead>
<tr>
<th></th>
<th>Non-Fault Tolerant Systems</th>
<th>Fault Tolerant Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware</strong></td>
<td>50%</td>
<td>8%</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>25%</td>
<td>65%</td>
</tr>
<tr>
<td><strong>Communications / Environment</strong></td>
<td>15%</td>
<td>7%</td>
</tr>
<tr>
<td><strong>Operations / Procedures</strong></td>
<td>10%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Software-induced outages dominate hardware-induced outages in fault tolerant systems.

Procedural and operations-induced outages are significant.

Sources of Outage-Inducing Software Flaws: Tandem 1989

For Tandem, most outage-inducing software flaws are in communications and database software

*From “A Census of Tandem System Availability Between 1985 and 1990,” by Jim Gray*
## Tandem Guardian Software Halts by Cause

<table>
<thead>
<tr>
<th>Fault Category</th>
<th>%</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect computation</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Data fault</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Data definition fault</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td><strong>Missing operations:</strong></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Uninitialized pointers</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Uninitialized nonpointers</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Not updating data structures on occurrence of certain events</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Not telling other processes on the occurrence of certain events</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Side effect of code update</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>Unexpected situation:</strong></td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>Race/timing problem</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Errors with no defined error-handling procedures</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Incorrect parameters or invalid calls from user processes</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Not providing routines to handle legitimate but rare operational scenarios</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Microcode defect</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Others</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td><strong>Unable to classify</strong></td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

20% of halts caused by missing operations
29% were caused by unanticipated situations
Tandem Guardian Software Halts by Severity

Many software halts take down more than one processor

<table>
<thead>
<tr>
<th>Fault Severity</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Processor Halt</td>
<td>79%</td>
</tr>
<tr>
<td>Multiple Processor Halt</td>
<td>18%</td>
</tr>
<tr>
<td>Halt during Reboot</td>
<td>1%</td>
</tr>
<tr>
<td>Unable to Classify</td>
<td>2%</td>
</tr>
</tbody>
</table>

Most software halts are caused by known bugs

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Occurrence</td>
<td>24%</td>
</tr>
<tr>
<td>Recurrence</td>
<td>61%</td>
</tr>
<tr>
<td>Unidentified</td>
<td>15%</td>
</tr>
</tbody>
</table>
### Analysis of Tandem Error Logs

#### What process was running just prior to halt?

<table>
<thead>
<tr>
<th>Active Process</th>
<th>Cause Breakdown</th>
<th>%</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Handler</td>
<td>process control</td>
<td>5%</td>
<td>41%</td>
</tr>
<tr>
<td></td>
<td>memory management</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>message system</td>
<td>14%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>processor control</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hardware-related</td>
<td>16%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unknown</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>System monitor</td>
<td>memory management</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>Memory Manager</td>
<td>process control</td>
<td>31%</td>
<td>32%</td>
</tr>
<tr>
<td></td>
<td>memory management</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>All Other Privileged Processes</td>
<td>process control</td>
<td>1%</td>
<td>14%</td>
</tr>
<tr>
<td></td>
<td>memory management</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>communication product</td>
<td>8%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TMF</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tape process</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unknown</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td>hardware-related</td>
<td>7%</td>
<td>9%</td>
</tr>
<tr>
<td></td>
<td>message system</td>
<td>2%</td>
<td></td>
</tr>
</tbody>
</table>

Interrupt handling and memory management code seem to be particularly troublesome

**Touches hardware and is highly concurrent**
### Fault Propagation in the UNIX OS

Lee and Iyer injected 500 simulated hardware and software faults into SunOS 4.1.2 kernel

#### Results of hardware fault injection:

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Without Self-Reboot</th>
<th>With Self-Reboot</th>
<th>System Hang</th>
<th>Multiple User Application Failure</th>
<th>No error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory fault in text segment</td>
<td>0.02</td>
<td>0.22</td>
<td>0.02</td>
<td>0</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.68</td>
</tr>
<tr>
<td>Memory fault in data segment</td>
<td>0.02</td>
<td>0.14</td>
<td>0</td>
<td>0.08</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.58</td>
</tr>
<tr>
<td>Bus fault on address line</td>
<td>0</td>
<td>0.82</td>
<td>0</td>
<td>0.1</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.02</td>
</tr>
<tr>
<td>Bus fault on data line</td>
<td>0</td>
<td>0.76</td>
<td>0</td>
<td>0.1</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>CPU fault in registers</td>
<td>0</td>
<td>0.66</td>
<td>0</td>
<td>0</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Most injected hardware faults in SunOS 4.1.2 either caused reboot or were never detected

**BUT**

Memory faults in text segment caused a system hang...very bad
Fault Propagation in the UNIX OS, cont’d

Results of software fault injection:

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>System Failure</th>
<th>Multiple User Application Failure</th>
<th>No</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With Self-Reboot</td>
<td>System Hang</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uninitialized pointer</td>
<td>0.46</td>
<td>0</td>
<td>0</td>
<td>0.54</td>
</tr>
<tr>
<td>Misassigned pointer</td>
<td>0.4</td>
<td>0</td>
<td>0</td>
<td>0.6</td>
</tr>
<tr>
<td>Missing condition check</td>
<td>0.22</td>
<td>0</td>
<td>0.2</td>
<td>0.56</td>
</tr>
<tr>
<td>Incorrect condition check</td>
<td>0.26</td>
<td>0</td>
<td>0</td>
<td>0.62</td>
</tr>
<tr>
<td>Uninitialized / misassigned pointer data</td>
<td>0.26</td>
<td>0.02</td>
<td>0.06</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Most injected software faults in SunOS 4.1.2 either caused reboot or were never detected

BUT

Pointer faults caused a system hang
VOS 1992 Crash Data

63% of all VOS crashes occurred around hardware events
30% occurred around maintenance events
33% occurred around other events

A hardware event is
When the hardware is not in a normal running state (e.g., booting or power-fail)
Some unusual event is happening with a piece of hardware
Hardware maintenance is occurring
Tandem Integrity NonStop/UX Field Data

Modules containing panic-inducing faults

<table>
<thead>
<tr>
<th>Module</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Drivers (async, ethernet, etc.)</td>
<td>31</td>
</tr>
<tr>
<td>Memory Subsystem</td>
<td>16</td>
</tr>
<tr>
<td>Streams Mechanism</td>
<td>12</td>
</tr>
<tr>
<td>Process Management</td>
<td>6</td>
</tr>
<tr>
<td>Machine-Dependent VM Code</td>
<td>8</td>
</tr>
<tr>
<td>Shutdown / Boot Process</td>
<td>8</td>
</tr>
<tr>
<td>Filesystem</td>
<td>10</td>
</tr>
<tr>
<td>I/O Subsystem</td>
<td>3</td>
</tr>
<tr>
<td>Mirror Driver</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt Handling</td>
<td>1</td>
</tr>
<tr>
<td>Diagnostic / Integration</td>
<td>3</td>
</tr>
<tr>
<td>MIDAS (monitoring facility)</td>
<td>1</td>
</tr>
</tbody>
</table>
## Tandem Integrity NonStop/UX Field Data

### Programming errors causing panic-inducing faults

<table>
<thead>
<tr>
<th>Programming Error</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer made NULL and later used</td>
<td>17</td>
</tr>
<tr>
<td>Pointer assigned to wrong location</td>
<td>9</td>
</tr>
<tr>
<td>Stale pointer left from before</td>
<td>2</td>
</tr>
<tr>
<td>Missing check for an exception</td>
<td>26</td>
</tr>
<tr>
<td>Incorrect algorithm or code placement (includes major algorithm mistakes)</td>
<td>26</td>
</tr>
<tr>
<td>Unaligned data structures</td>
<td>4</td>
</tr>
<tr>
<td>Memory allocation / deallocation</td>
<td>11</td>
</tr>
<tr>
<td>Unnecessary code left in the OS</td>
<td>4</td>
</tr>
</tbody>
</table>
Summary of Empirical Data

Software-induced outages increasingly dominate hardware-induced outages

For Tandem, most outage-inducing software flaws are in comm and DB software

20% of Tandem Guardian halts caused by something somebody forgot to do

29% of Tandem Guardian halts caused by situations somebody didn’t anticipate

18% of Tandem Guardian faults take down more than 1 processor

61% of Tandem Guardian faults are caused by known bugs

Most Tandem Guardian halts occurred during interrupt handling (41%) and memory management (32%) code
Summary of Empirical Data

Most injected hardware and software faults in SunOS 4.1.2 either caused reboot or were never detected, but some memory and pointer faults caused system hangs.

63% of 1992 VOS crashes occurred around hardware events.
Ariane 501 Crash

4 June 1996 maiden flight of Ariane 5

40 secs into flight, Flight Control Computer (FCC) commands nozzle actuators to hard over position

Ariane 5 undergoes aero breakup and subsequent destruction

DM 1200 million down the drain
Ariane 501 Causal Factors

Inertial Reference System (IRS) informed FCC the missile is flying sideways

IRS emitted diagnostic information to FCC
This was interpreted as attitude data

Both Primary and Backup IRS failed
Unhandled exception due to overflow when converting 64-bit float to 16-bit integer in IRS horizontal velocity calibration code
Other conversions were protected by exception handlers
No justification for omitting protection for this variable
IRS cal code reused from Ariane 4
Continues to run after liftoff in case of short launch hold
Not needed in flight for Ariane 4
Not needed at all for Ariane 5
Ariane 5 horizontal velocity >> Ariane 4
Ariane 501 Causal Factors

Error not excited during test
   IRS cal code not tested under Ariane 5 trajectory
   Not flight critical

Spec says halt IRS when unhandled exception
   Assumed random hardware faults only
   Assumed software is perfect
Ariane 501 Lessons

IRS-FCC interface was insufficiently robust
Fault model incomplete: did not include SW errors
Error handling requirements not appropriate for common-mode SW errors
Critical assumptions were not documented, justified, or reviewed
New operational conditions violated design-time assumptions of re-used software
Gratuitous functionality does not go away just because it is no longer needed
Testing under realistic operational conditions was omitted
Software Error Genesis

Design Errors

60-65% of all SW faults introduced here
Incomplete, missing, inadequate, inconsistent, unclear requirements
Requirements not fitting physical models
Correction cost is 10X cost of correcting coding errors

Implementation Errors

35-40% of all SW faults introduced here
# errors proportional to
Size of code
Number of paths through code
TERMINOLOGY AND BUZZWORDS
Dependability

Property of a computing system which allows justifiable reliance to be placed upon delivered service

Means for achieving dependability
  - fault prevention: writing bug-free code
  - fault removal: testing and fixing bugs
  - fault forecasting: predicting and avoiding failures
  - fault tolerance: tolerating failures as they occur

Quantifications of dependability are numerous
  - Reliability, availability, N-fail/op, ...

A system can be dependable without being fault tolerant

A system can be fault tolerant without being dependable

Failure

Deviation of delivered service from specification

failure domain
  value - the value of the delivered service does not comply with the specification
  timing - the timing of the delivered service does not comply with the specification

failure perception
  consistent - all users have identical perceptions of the failure
  inconsistent - users have different perceptions of the failure

failure severity
  benign - failure consequences are of same order of magnitude as benefit of service delivery
  catastrophic - failure consequences are incommensurably greater than benefit of service delivery
Error

Corruption of system state liable to lead to failure
   Latent - not recognized by detection mechanism
   Detected - recognized by detection mechanism

Example
   Corrupted contents of RAM in text area
Fault

Adjudged or hypothesized cause of error(s)

Active - capable of producing an error
Dormant - incapable of producing an error

Physical Fault Models

Stuck-at
Inversion
Symmetric / asymmetric
Permanent / intermittent / transient

Software Fault Models

Bohrbugs: permanent; Heisenbugs: transient

These are all subsets of “Byzantine faults”:

Arbitrary (even malicious) fault manifestations
Fault Occurrence and Error Processing Behavior

There are several steps involved in handling faults correctly
Not all systems go through all steps
The name of the game is to prevent faults from causing failures

- fault arrival and error production
- error propagation
- error manifestation
- error compensation
- error detection
- fault diagnosis
- error processing - recovery
- fault passivation - reconfiguration
Error Compensation

Also known as fault masking

Possible when system state contains enough redundancy to enable delivery of error-free service from erroneous internal state

Needed when glitch-free service delivery is required

May be all that is needed for some mission regimes

Does not imply error detection, error recovery, or fault passivation

Examples

N-modular redundancy with voting for general-purpose computation

Error correcting codes for data transmission and storage

Averaging effect of many signal processing algorithms
Error Containment Region

Errors should not propagate past error containment region boundaries

Errors which do so can result in system failure

Hardware Examples
  Voting plane
  Decoder at memory output
  Decoder at bus interface
  Voting actuator

Software Examples
  Software voter
  Recovery Block acceptance test
  Data structure integrity check
  Control flow check
Error Manifestation Boundary

Faults are detected via error manifestation

Error manifestation boundaries should be defined

Error detection mechanisms reside at error manifestation boundaries

Errors should be quickly flushed to error manifestation boundaries to expedite detection, diagnosis, and recovery

The sooner the error is detected, the easier recovery will be

“Fix it so it breaks”
Error Detection

The use of error detection mechanisms at Error Manifestation Boundaries to determine the existence of errors

- proactive: go out hunting for errors
- reactive: wait for errors to happen

Facilitates subsequent error recovery, fault diagnosis, and fault passivation

Examples

- Syndrome attached to voter/comparator
- Parity on memory fetches
- Block codes on data transmissions
Error Processing - Recovery

The process of returning the system to an acceptable physical and computational state

Explicit recovery operations may or may not be required
  - Depends on how well error propagation can be controlled
  - Depends on whether state information is stored redundantly
  - Depends on temporal constraints

Recovery is essentially a semantic process
  - Must consider the physics of the system in order to subdue erratic behavior
  - Corrective action must not aggravate any transient already caused by the failure
Error Recovery

Computational state must be corrected

If state information is stored stably then a valid copy can be retrieved in a straightforward manner

If only a single copy of state information existed then the system state has to be reconstructed

Internal parameters with limited history can be reinitialized to a known state if the resulting transient is not too great (e.g., digital filter values)
Fault Diagnosis

The identification of the fault location or ambiguity group

Enables fault passivation

Fault diagnosis issues
  Coverage with respect to a class of faults
  Assuring that all participants arrive at consistent diagnoses
  Ambiguity group localization
  Verification and validation of coverage
  Intrusiveness
  Physical and temporal overhead
  Fault classification: permanent, transient, etc.
Fault Passivation (Reconfiguration)

Reconfiguration is the process of

- Isolating a failed element so it no longer has any influence on system behavior
- Reassigning the function of the failed element to a good element or group of elements

Isolation and reassignment may be

- Logical – there are multiple sources for a parameter and the bad one is simply ignored
- Electrical – removing power from the failed element or switching in a replacement element
- Physical – elements are separated by a reconfiguration actuator
Fault Passivation (Reconfiguration)

Reconfiguration can be automatic or initiated by a human operator.

Since a substantial proportion of outages are due to maintenance and procedural errors, automatic means are preferred.

Reconfiguration/recovery must be completed quickly to prevent failure due to near-coincident faults.
A *fault containment region* is a bounded group of components or functionality.

An arbitrary fault inside a region cannot propagate across the boundary to cause another region to fail or to misbehave in any way.

Conversely, faults outside the region cannot physically affect proper operation inside the region.

However, errors (the effects of faults) may propagate to other regions.

Proper organization of fault containment regions is critical to achieving fault tolerance.

A physical fault containment region requires, (1) electrical isolation, (2) independent clocking, (3) independent power, (4) physical isolation.
Coverage

Numerical quantification of the effectiveness of a fault tolerance technique

Different coverage numbers will apply to different phases of fault and error handling

Example

   Effectiveness of a fault tolerance technique with respect to a class of faults
   Detection coverage of stopping faults
   Tolerance coverage of babbling faults
   Tolerance coverage of Byzantine faults
Coverage

Can be expressed probabilistically
  Probability of detecting stopping faults
  Probability of tolerating babbling faults
Can be determined empirically in some cases
Can not be determined empirically in most cases
SOFTWARE TECHNIQUES FOR TOLERATING FAILURES
General Observations and Implementation Principles

Define success criteria for the function you are developing
- Safety (what must never happen)
- Liveness (what must always happen)

Understand your environment, expected failure modes, and acceptable error handling
- What do you do when you can’t go on?
- Examples: single node (best-effort), cluster (fail-fast)

Select proactive or reactive technique
- Proactive techniques search for or attempt to predict errors
- Reactive techniques wait for errors to occur
General Observations and Implementation Principles

Define error containment boundaries
  Partition application so a portion can be down without entire application being down

Define error manifestation boundaries and detection mechanisms
  At least based on safety and liveness; preferably based on application specific checks
  Always check inputs and outputs
  Always check return values and error codes
  Balance overhead with detection coverage

Define error handling actions appropriate to safety, liveness, and environmental requirements
  Log significant events for later debugging
  Fail loudly, don’t fail silent...those who come after will thank you
  Don’t assume HW or SW are operating correctly
General Observations and Implementation Principles

Test all error detection and recovery code

- It is important but not core to central functionality; no revenue $ tied to it
- It gets implemented last
- It gets tested least
- It is hard to test
- It is invoked under periods of maximum system stress

In telecom applications (e.g., 5ESS)

- 50% LOC on core functionality
- 50% LOC on error handling
  
This is an appropriate mix for critical applications

Human error-making patterns are repetitive - categorize and log your errors and periodically review them
A menagerie of techniques
  Checkpoint and Rollback
  Recovery Blocks
  Process Pairs
  Transactions...

Overall Idea
  Save snapshot of correct state somewhere
  Do work, logging inputs and events
  Check for errors
  If error,
    Roll back or restore process(es) to state snapshot
    Optionally, inculcate nondeterminism
    Replay the computation
Else
Checkpoint and Rollback

A reactive technique

Applicability

- Where cost of failure is an annoyance
- Soft HW and (primarily) SW failures
- Works on nonredundant or redundant architecture
- Where you have time to retry a computation
- When you can identify checkpoints in your application
Checkpoint and Rollback Critical Assumptions

Errors can be detected

Checkpoints can be identified and efficiently copied to stable storage

Inputs and events can be logged

Computation can be replayed

Replay is deterministic with respect to applied inputs

Replayer can access checkpointed data

Rollback can be confined to a small number of processes, or interprocess interactions can be replayed or are idempotent
Checkpoint / Rollback Approach (1)

Develop error detection mechanisms

- Internal to application: code- or structure-based self checks
- External to application: probes, signals, null messages to app, heartbeats, ...

Determine data to be checkpointed

- Transparent to application
  - Compile-time
    - Run time: checkpoint all volatile state, checkpoint dirty data
  - Visible to application
    - Allocate data to be checkpointed to appropriate region: ISIS, libft, ...

- Must be stored in stable storage
- Must be accessible to process that is performing the retry
Checkpoint / Rollback Approach (2)

Determine events to be logged and replayed

Messages
Events
Transactions

Determine checkpoint times; options are:

Transparent to application
  Based on elapsed time
  Based on message arrival
  Based on amount of dirtied state

Visible to application
  Based on critical function invocation / exit

Figure out how you are going to replay the computation

Figure out what you are going to do if error is persistent
Example: Tandem HATS / AT&T “libft” Technology

watchd: distributed watchdog daemon

- Monitors registered application processes on primary and backup nodes for crash or hang; also monitors nodes
- Sends null message or signal to primary every T seconds, or awaits heartbeats
- If no response and primary node is unfailed, restarts process on primary
- If primary node is failed, restarts process on backup node
- Uses checkpoint data and message logs to replay computation

---

1 Also check out
http://www.cs.utk.edu/~plank/ckp.html and
http://warp.dcs.st-andrews.ac.uk/warp/systems/checkpoint/source.html
Example: AT&T “libft” Technology

libft: set of reusable UNIX library calls for checkpointing and message-logging

- Allows app programmer to designate variables to be checkpointed via “critical()” call
- Allows app programmer to trigger checkpoints via “checkpoint()” call
- Permits logging of received and transmitted messages on primary and backup nodes for replaying, via “ftread()” and “ftwrite()” calls
- Can reorder message arrivals in attempt to avoid Heisenbugs

nDFS: n-Dimensional File System

- Provides replicated stable storage to allow backups access to checkpointed data
Checkpoint and Rollback Cost Effectiveness

Development Cost

Based on AT&T libft experience, can insert watchd/libft/nDFS in existing telecom apps quickly (weeks)

*Using portable watchd/libft/nDFS library - didn’t have to write difficult checkpointing code from scratch*

Run time cost (no faults)

<14% for libft checkpointing approach

Effectiveness

On the order of 90% coverage of non-design errors

AT&T reports highly effective at tolerating certain known bugs that they can’t afford to fix
Checkpoint / Rollback Advantages

Works, mostly

AT&T has had success in tolerating faults it can’t afford to fix

Runtime overhead acceptably low: say 15%

Not all-or-nothing: can use judiciously in critical functions and integrate seamlessly
Checkpoint / Rollback Disadvantages

Defining checkpoint data and intervals is tricky

Checkpoint / rollback algorithms in concurrent systems are exceedingly complex and potentially slow

  Must establish recovery lines and avoid domino rollback

Irreducible overhead for checkpointing: processing, comm, storage

Efficient techniques are not transparent; transparent techniques are not efficient

Fault model is moderately weak

  Only as good as error detection means
  Doesn’t work for persistent software errors
  Error detection coverage is critical but usually neglected

Checkpoint placement and frequency are critical: price / performance tradeoff must be made

Could require double the run time to handle faults
State Rejuvenation

A proactive technique: use it to avoid failures

Applicability

Where long-running processes gradually degrade system state due to

Memory leaks, memory caching, weak memory reuse, memory fragmentation, unreclaimed resources, bitrot, ...

Where processes use canned (or old) code whose source can’t be modified

Critical Assumptions

No need to detect errors...you get them before they get you

Checkpoint data can be identified and copied to stable storage

Can generate checkpoint and restart scheme that lets you pick up where you let off

Rollback can be confined to a small number of processes
## State Rejuvenation Example

### Buggy Subroutine

```c
#define MEG (1024*1024)
unmodifiable_call(arguments)
{
    /* initialization */
    if(ptr1 = malloc(MEG)) == NULL) exit(1);
    if(ptr2 = malloc(MEG)) == NULL) exit(2);
    /* incredibly complex control flow */
    free(ptr1);
    /* more incredibly complex control flow */
    free(ptr2);
}
```

### Without State Rejuvenation

```c
main()
{
    while(1)
    {
        new_state = unmodifiable_call(some_arguments);
        write_output(new_state);
    }
}
```

### With State Rejuvenation

```c
main()
{
    int i = 0;
    while(1)
    {
        new_state = unmodifiable_call(some_arguments);
        write_output(new_state);
        if(i++ % REJUV_PERIOD == 0)
        {
            flush_outputs()
            checkpoint()
            rollback()
        }
    }
}
```
State Rejuvenation Cost Effectiveness

Development and maintenance cost

Somewhat less than checkpointing, since don’t have to design general purpose error detection and recovery techniques

Run time cost

Same as checkpointing; on the order of 15%
Runtime cost is predictable since you determine how often to rejuvenate

Effectiveness

Has been shown to be very effective when applicable: memory leaks, etc.
Can sometimes reduce execution time by aggregating fragmented state
Can run during slack times to minimize performance impact
Issues with State Rejuvenation

Limited applicability
   Essentially a boutique solution

Has most of checkpointing / rollback’s problems:
   Tricky to define checkpoint data
   Must empirically determine how often to rejuvenate
   Not transparent
   Serious difficulties in multiprocessing systems

Best used sparingly when you know that you have a longevity flaw
Recovery Block

A reactive technique

Applicability

Where cost of failure is severe
Where must deliver service at all costs
Soft HW and (primarily) SW failures
Typically, nonredundant architecture
Where you have time to retry a computation
Recovery Block Critical Assumptions

Faults are soft and primarily due to software errors
Software errors could be design or, more probably, coding errors
Faults do not cause app or system to crash
Faults can be corrected by retrying alternate version of code
Replica execution is deterministic
Recovery Block Flowchart

Enter RB with acceptable data

Error Containment Boundary

Recovery Point

Primary Alternative Module

Primary Alternative Module

Acceptance Test

Pass

Fail

Exit RB with acceptable data

Exit RB with failure indication

Really Fail
Example: Memory Allocation

Recovery Point
  State of pointer chains to be modified by allocation routine

Try block 1
  Allocate from heap 1

Try block 2
  Allocate from heap 2

Acceptance test
  Sum of allocated block sizes == requested size == size of free list decrement?
    Free / used pointer chains connected?

If fail, restore pointer chains from recovery point and retry
If error, perform error handling appropriate for the function and environment
Recovery Block Cost Effectiveness

Development and maintenance cost
  Approximately 60% increment for 2 try blocks

Run time cost (no faults)
  Typically in the 10-20% range
  Fujitsu reports 50% run time overhead for recovery block-protected UNIX system calls
  At least 2X in presence of faults

Effectiveness
  Approximately 90% coverage of non-design errors
Recovery Block Advantages

Works, mostly

Makes you figure out what the code is supposed to do by writing acceptance tests

Makes you think of at least two ways of solving the problem

Makes you figure out what you should do if your routine fails

Runtime overhead acceptably low: say 15%

Not all-or-nothing: can use judiciously in critical functions and integrate seamlessly
Recovery Block Disadvantages

Acceptance tests are critical

Single point of failure and a source of irreducible overhead

SW errors in boundary code occur significantly more often than in main routine

Price/performance tradeoff must be made: where and how often to place acceptance tests

Expensive

At least two copies of code must be constructed, tested, supported, etc.

Requires additional storage for input conditions to allow retries to commence

Could require double the run time to handle faults
Recovery Block Disadvantages

Fault model is weak

- Poorly tolerates design faults, hardware failures, OS crashes, app crashes

How to construct recovery point

- Sufficient data must be saved to enable retry

How to construct different try blocks

- How many copies to be developed
  - *Could use same code if trying to tolerate Heisenbugs and can inculcate nondeterminism*

What to do when all tests fail

- Remember: context dictates actions when you can’t go on

Multiprocessing systems suffer from domino rollback: avoidance is complex
Process Pairs

A reactive technique

Applicability

- Hard or soft HW and SW failures
- Loosely coupled redundant architecture; fail fast hardware optional
- Message-passing interprocess communication
- Works best for transaction-oriented applications

Critical assumptions

- Processes and hardware are fail-fast
- Errors can be corrected by re-executing same code in another environment
- No single points of failure in architecture
Overall Approach

Construct *fail-fast* processes

Either function correctly or detect a fault, signal failure, and stop

Both hardware and software may be designed to be fail fast

Fail fast processes may be constructed on non-fail-fast hardware

Enforce fault and error containment

No shared state; processes communicate via message passing

This prevents a process from corrupting state on its local processor

It also facilitates construction of process pairs

Two process pair types prevail

Checkpoint / restart / message

Persistent
Checkpoint / Restart / Message Process Pairs

Primary performs the work

Secondary listens for “I’m alive” messages

In checkpoint / restart scheme, primary logs state updates to stable storage accessible to secondary

In checkpoint / message scheme, state updates are piggybacked on (and may supplant) “I’m alive” messages

When secondary detects failure of primary, secondary refreshes state either from stable storage or from message log

Secondary then picks up processing where primary left off

“...it is the authors’ [Jim Gray and Andreas Reuter] experience that everyone who has written [a process pair] thinks that it is the most complex and subtle program they have ever written.”
Checkpoint / Restart / Message Example

Primary

- broadcast "I'm Primary"
- reply to last request
- any input?
  - yes: read it
  - no: become Primary
- compute new state
- send new state to backup
- reply

Secondary

- restart
- wait a second
  - am I default Primary?
    - yes: become Primary
    - no: wait a second
- new state in last second?
  - yes: any input?
  - no: newer state?
    - yes: set my state to new state
    - no: read it
- wait a second

requests from client

replies to client
Persistent Process Pairs

Suitable for transaction-based applications

Works in conjunction with transaction monitor

TM can undo partial transactions

Primary executes ACID transactions

BeginTransaction

code, code, code

EndTransaction or AbortTransaction

Amnesiac secondary (or TM or OS) listens for “I’m alive” messages

When primary fails-fast, TM undoes any transactions in progress and resubmits them (as well as subsequent client traffic) to the amnesiac secondary

Persistent process pairs provided as a primitive by NonStop operating system
Persistent Pair Example-with OS Support

Transaction Monitor

Primary

restart

primary? yes

BeginTX

read request

do work

reply to client

EndTX

no

Secondary

restart

primary? yes

BeginTX

read request

do work

reply to client

EndTX

no

requests from client

replies to client
Advantages

Extremely successful in Tandem OLTP applications

Persistent process pairs relatively easy to program

Fault model is moderately strong: tolerates hardware, OS, and app failures

High coverage (>90%) of hardware and software (including OS) faults

Does not consume too much performance at backup site: about 10% runtime overhead can be achieved
Disadvantages

Checkpoint / restart / message process pairs difficult to construct without significant toolkit or infrastructure investment

Must use checkpoint / restart / message process pairs in a non-transaction-based application

Must develop error detection checks and signaling techniques to make a process fail-fast

Works best on fail-fast hardware

Works best on message-passing interprocess communication

Works best on loosely coupled distributed hardware architecture
Multiversion Software

Applicability

Fast real-time critical applications where no dropout is acceptable: aerospace, nuclear, ground transportation
Where cost of failure is severe
Tolerates soft or hard faults, whether in HW or SW (primarily oriented towards tolerating SW coding faults)

Critical assumptions

Specification contains no flaws (omissions, inconsistencies, ambiguities)
Independent programming teams don’t make the same mistakes
Requires loosely synchronous redundant architecture
Replica execution is deterministic
Multiversion Software Development Steps

Develop specification
   Constraining enough to allow version comparison
   Flexible enough to allow diversity

Develop version voter
   Plurality voting
   Approximate voting

Generate diverse programs
   Give specification to three or more independent programming teams
       Random or enforced diversity

Run diverse versions on independent hardware

Perform periodic voting of version outputs
Multiversion Software Flowchart

[Flowchart image showing the flow of data and decision-making processes involving Input, Version 1, Vote, Version 2, Vote, Version 3, Vote, and FCR/FCR 2/FCR 3.]
MultiVersion Software Cost

Development and maintenance cost
   About 2.26 times the cost of single version for three versions

Run time overhead
   On the order of 10 to 25%
   Same in presence of faults

Hardware overhead
   At least 3X
MultiVersion Software Effectiveness

Case 1: Aircraft autoland control law experiment

<table>
<thead>
<tr>
<th>Version</th>
<th>LOC</th>
<th># errors</th>
<th>error prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>ada</td>
<td>2256</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1531</td>
<td>568</td>
<td>0.00011</td>
</tr>
<tr>
<td>modula-2</td>
<td>1562</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pascal</td>
<td>2331</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prolog</td>
<td>2228</td>
<td>680</td>
<td>0.00013</td>
</tr>
<tr>
<td>t</td>
<td>1568</td>
<td>680</td>
<td>0.00013</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>3 versions: probability</th>
<th>5 versions: probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>No errors</td>
<td>.9998409</td>
<td>.9997807</td>
</tr>
<tr>
<td>Single errors in one version</td>
<td>.0001305</td>
<td>.0001915</td>
</tr>
<tr>
<td>Two distinct errors in multiple versions</td>
<td>.000002048</td>
<td>.000002275</td>
</tr>
<tr>
<td>Two coincident errors in multiple versions</td>
<td>.00002652</td>
<td>.00002210</td>
</tr>
<tr>
<td>Three Errors in multiple versions</td>
<td>0</td>
<td>.000003413</td>
</tr>
</tbody>
</table>

Case 2: Leveson, et al, found that code-based self-checks were far more effective than multiversion programming at finding programming errors
Multiversion Software Issues

Specification management

Make sure all teams are using same spec

Version construction

Ensure diversity

Version resolution

Voter construction is critical from detection and performance viewpoint

You get monstrosities like “approximate thresholding plurality voters”

When is bitwise agreement meaningful?

Placement and frequency of voting
Multiversion Software Issues

Version synchronization
   Requires fault tolerant synchronization mechanism

Version recovery
   Bring faulted version to same state as nonfaulty versions

Cross-channel exchange of input and output data
   Consumes bandwidth
Multiversion Software Advantages

Makes you figure out what the code is supposed to do by writing spec and voter

Helps validate and clarify the specification

Fault model is strong

Approved by Your Government for flight- and safety-critical systems

Constant execution time and no dropouts in presence of faults

Effective at coping with a few bad programs (if most versions are good)
Multiversion Software Disadvantages

Specifications DO contain flaws (omissions, inconsistencies, ambiguities)

Independent programmers DO make the same mistakes

Less effective when all programs are uniformly reliable: the hard stuff is hard for everybody

Versions may not agree in absence of faults due to roundoff error, non-commutativity, etc. ==> complex voters

Expensive: at least three copies of code must be developed and maintained

Requires 3X redundant hardware architecture plus interchannel communications network
Process Groups

Deterministic processes are replicated on multiple computing nodes of a distributed system

- The service provided by the replicated processes can be “continuously” available when some of the replicas fail
- Every replica has a copy of the common state
- State updates occur in the same order on all replicas
- Client communicates with a group address, not a single process address

Client and server process application code must execute group membership and communication calls

Client and server process application code must execute group membership and reliable multicast algorithms

- global, atomic, causal, fifo, hierarchical, and other broadcasts
- Virtual synchrony used to ensure consistency of group view
Process Groups Example: Database Server

Atomic update: Either all servers get the update, or none do.
#include "isis.h"

#define UPDATE 1
#define QUERY 2

main() {
    isis_init(0);
    isis_entry(UPDATE, update, "update");
    isis_entry(QUERY, query, "query");
    pg_join("salary_DB", PG_XFER, send_state, rcv_state, 0);
    isis_mainloop(0);
}

update(mp)
register message *mp; {
    char name[32]; int salary;
    msg_get(mp, "%s, %d", name, &salary);
    set_salary(name, salary);
}

query(mp)
register message *mp; {
    char name[32]; int salary;
    msg_get(mp, "%s", name);
    salary = get_salary(name);
    reply(mp, "%d", salary);
}

send_state() {
struct sdb_entry *sp;
for (sp = head(sdp); sp != tail(sdb); sp = sp->s_next)
    xfer_out("%s, %d", sp->s_name, sp->s_salary);
}

rcv_state()
register message *mp; {
    update (mp);
}


Process Groups Example: Client Code

```
#include "isis.h"
#define UPDATE 1
#define QUERY 2
address *server
main() {
    isis_init(0);
    server = pg_lookup("salary_DB");
    pc_client(server);
    ...
    ...do client work: calls to update and get_salary...
    ...
}

update(name, salary)
char *name; int salary; {
    abcast(server, UPDATE, "%s,%d", name, salary, 0);
}

get_salary(name)
char *name; {
    int salary;
    fbcast(server, QUERY, "%s", name, 1, "%d", &salary);
    return(salary);
}
```
Process Group Advantages and Disadvantages

Advantages

- Tolerates hardware, operating system, and some application failures
- Works over widely distributed heterogeneous systems
- Not all-or-nothing - can be deployed for critical services only

Disadvantages

- Not application transparent
- Does not support mutually preemptible threads very well
- Hierarchical groups not supported well
- Performance may be low
- Fault model is weak: usually fail-stop
- Error detection latency may be a few seconds
Robust Data Structures

A proactive technique

Use to seek out and destroy errors before they cause failures

A number of techniques is available

*We will use linked lists to demonstrate the concept*

Intended to achieve

Semantic integrity: the data’s meaning is uncorrupted

Structural integrity: the data’s organization is correct

*We will discuss structural integrity techniques today*
Robust Data Structures

Applicability

Critical data structures having a clearly defined and regular structure
Most highly developed for linked lists and trees

Critical Assumptions

Data structures can be fitted with redundancy
Execution time is available to audit and correct data structures
Storage is available to store essential redundant information
Robust Data Structures Approach

Robust data structures contain redundant data which allow erroneous changes to be detected and corrected by checks.

Checks may perform on-line error detection and correction.
- Overhead and effectiveness are functions of data structure access frequency.

Detection / correction programs (audits) may be used.
- Overhead may be tuned based on performance considerations.
- Checks may be run during slack time.
Robust Data Structures Approach

Robust data structures are classified as N-detectable and M-correctable

- **N-detectable**: all sets of N or fewer changes to structure can be detected
- **M-correctable**: all sets of M or fewer changes to structure can be corrected

Changes are defined to be arbitrary (malicious) modifications to the data structure
Example: Linked List

Nonrobust design

0-detectable, 0-correctable
Robust Linked List Design 1

Robustness additions

Store count of number of nodes
Add unique structure identifier field to each node
Link end of list to header

1-detectable, 0-correctable
Robust Linked List Design 2

Robustness addition

Add pointer from successor to predecessor node

2-detectable, 1-correctable
Robust Linked List Design 3

Robustness addition

Add pointer from successor to predecessor’s predecessor node

3-detectable, 1-correctable
Code to Correct Errors in Design 2 Linear Linked List (1)

```c
void *correctlist (void *H) {
    void *P, *prevP;
    int J = 0;
    prevP = H;
    P = H->next
    /* Scan main body of data structure */
    while (P != H) {
        J++;
        if ((P->prev == prevP) && (P->ID == H->ID)) {
            /* This node looks OK */
            prevP = P;
            P = P->next
        } else /* We have a problem */{
            if (backscan(H, P, prevP) == CORRECTED_ERROR)
                return (CORRECTED_ERROR);
            else
                return (UNCORRECTED_ERROR);
        } /* end if */
    } /* Done scanning main body of data structure */

    /* Now examine header of data structure */
    if ((H->prev != prevP) || !correct(ID(H))) {
        /* An error */
        if (backscan(H, P, prevP) == CORRECTED_ERROR)
            return (CORRECTED_ERROR);
        else
            return (UNCORRECTED_ERROR);
    } /* end if */

    /* Lastly, check count field in header */
    if (H->numnodes != J) {
        /* An error */
        H->numnodes = J;
        return (CORRECTED_ERROR);
    } /* No Errors */

    return (NO_ERRORS);
} /* end of correctlist */
```

---

Code to Correct Errors in Design 2 Linear Linked List (2)

```
backscan (void *H, void *P, void *prevP) {
    void *Q, *prevQ;
    int J = 0;
    prevQ = H;
    Q = H->prev
    while( J++ < H->numnodes) {
        if( (Q->next == prevQ) && (Q->ID == H->ID) ) {
            prevQ = Q; Q = Q->prev;
        } else {
            if( repair(P, prevP, Q, prevQ) == CORRECTED_ERROR)
                return (CORRECTED_ERROR);
            else
                return (UNCORRECTED_ERROR);
        } /* end of while */
    } /* end of backscan */
}

repair (void *P, void *prevP, void *Q, void *prevQ) {
    if( (P == Q) && (P->ID != H->ID) ) {
        P->ID = H->ID;
        return(CORRECTED_ERROR);
    } else if ( P == prevQ) {
        Q->prev = prevP;
        return(CORRECTED_ERROR);
    } else if( prevP == Q) {
        P->next = prevQ;
        return(CORRECTED_ERROR);
    } else
        return(UNCORRECTED_ERROR);
} /* end of repair */
```
Linear Linked List Correction Example
Cost Effectiveness of Linked List Designs

![Bar chart showing the cost effectiveness of different linked list designs. The x-axis represents Storage Cost, Detectability, Correctability, and Update Time. The y-axis represents the cost. The chart compares nonrobust, count, ID fields, loop, double-linked, and modified double-linked designs.]
Other Robust Data Structure Techniques

Other techniques are available in the literature

Structural techniques for popular data structures

Trees, stacks, fifos, heaps, queues, etc.

In general, a linked data structure is 2-detectable and 1-correctable iff the pointer network is bi-connected

Content-based techniques

Checksums, encodings
Robust Data Structure Cost Effectiveness

Development Cost

Not too bad, especially if can reuse code for common data structures (e.g., linked list manipulation / audit code)

Run Time Cost

Not excessive, especially if can run audits during slack times
Techniques can be selected / designed / tuned for low run time overhead

Effectiveness

Highly effective at ferreting out and correcting structural flaws

_Leveson et al found code-based self-checks far more effective than multiversion programming_

Less effective at tolerating semantic flaws
Issues with Robust Data Structures

Not a transparent technique

Relevant to a limited (but significant) class of errors
    Best at errors which corrupt the structure of the data

Data structure auditing / correction code is subtle, complex, difficult to program, and prone to programming errors

Not for the squeamish
Structure Marking

Add TYPE, SIZE, VERSION, and OWNER to data structure

TYPE: Unique number for each different structure
SIZE: in bytes
VERSION: Changed whenever structure declaration changes
OWNER: Unique ID of owner

Set them when the structure is instantiated

Check them before using structure instance

RunAssert( (s.TYPE == STYPE) &&
            (s.SIZE == SSIZE) &&
            (s.VERSION == SVERSION3) &&
            (s.OWNER == uid) );

Clear or 0xdeadbeef them when the structure is deallocated

s.TYPE = s.SIZE = s.VERSION = s.OWNER = 0xdeadbeef;
Structure Marking Tips

Don’t use common values (0, 1) for TYPE

OWNER must be independent of structure contents; can be UID, least significant bits of clock, etc.

Can find OWNER at sizeof(augmented structure) - sizeof(OWNER field)

Can use sizeof() to generate SIZE field

Don’t use common values for VERSION

VERSION field helps find integration errors

VERSION field can be used as parameter to procedure to indicate which version of structure to return; eases compatible evolution

Obvious applicability to object-oriented systems; set up the markings in constructors; check them in the methods

Can augment robust data structures with structural marking
Structure Marking Effectiveness

Protects against

Clobbered data: wild stores, bit rot, data overruns

Programming errors: logic errors, incorrect calls, using data after freeing or before allocating

System integration errors: notices cases where programs are unable to handle the data they are passed

Not so effective against

Design and compiler errors: wrong algorithm or compiler bug can produce perfect structure with incorrect contents

Experience

Highly successful in MULTICs

Performance overhead quite low
Control Flow Monitoring

A technique to ensure that control flow goes through intended paths

Example:

```c
/* beginning of monitored code */
... BFI 3...
if (swizzle) {
... BFI 5...
}
else {
... BFI 7...
}
... BFI 11...
/* end of monitored code */
```

(BFI = Branch-Free Interval)
Assign each BFI a prime ID called BID

```c
#define BID 3
... BFI 3...
if (swizzle) {
    #define BID 5
    ...BFI 5...
}
else {
    #define BID 7
    ...BFI 7...
}
#define BID 11
...BFI 11...
```

Assign each BFI a value for NEXT = product of all possible next BIDs

```c
#define BID 3
#define NEXT 5*7
... BFI 3...
if (swizzle) {
    #define BID 5
    #define NEXT 11
    ...BFI 5...
}
else {
    #define BID 7
    #define NEXT 11
    ...BFI 7...
}
#define BID 11
...BFI 11...
```
Add land mines: now there is no way to get through the BFIs incorrectly

```c
#define BID 3
#define NEXT 5*7
... BFI 3...

id = NEXT;
if (swizzle) {
    #define BID 5
    #define NEXT 11
    id = BID / (!(!(!id%BID) ; /* check for illegal entry */
    ...BFI 5...
    id = NEXT; /* only update id when done with BFI */
}
else {
    #define BID 7
    #define NEXT 11
    id = BID / (!(!id%BID) ; /* check for illegal entry */
    ...BFI 7...
    id = NEXT;
}
#define BID 11
id = BID / (!(!id%BID) ; /* check for illegal entry */
...BFI 11...
```

The final touch: now there is no way to get out of an incorrectly entered BFI

```c
#define BID 3
#define NEXT 5*7
... BFI 3...

id = NEXT;
if (swizzle) {
    #define BID 5
    #define NEXT 11
    id = BID / ( (!id%BID)) * (id%2) ; /* check */
    ...BFI 5...
    id = NEXT+ !!(id - BID);
}
else {
    #define BID 7
    #define NEXT 11
    id = BID / ( (!id%BID)) * (id%2) ; /* check */
    ...BFI 7...
    id = NEXT+ !!(id - BID);
}
#define BID 11
id = BID / ( (!id%BID)) * (id%2); /* check */
...BFI 11...
```
# Example

```c
#define NEXT 5*7
... BFI 3...
id = NEXT+ (!(id - BID));

if (swizzle) {
    /* if legal entry, id = 5*7 */
    /* if illegal entry, id != 5*7, probably */
    #define BID 5
    #define NEXT 11
    /* check for illegal entry */
    id = BID / ( !(id%BID) * (id%2) );

    /* if legal entry, would get id = 5 / ((!(35%5)) = 5 / (!(0)) = 5 / 1 = 5 */
    /* if illegal entry, would get id = 5 / ((!(id%5)) = 5 / (!(1)) = 5 / 0 = divide by zero error */

... BFI 5...
    /* if exit out of BFI5 before id = NEXT, will get caught at next illegal entry check with wrong id */
    id = NEXT+ (!(id - BID)); /* only update id when done with BFI */
} /* end of BFI 5 */
```

This term checks that BFI5 was legally entered from BFI 3.
This term checks that the previous BFI was not illegally entered after its entry check.
This term declares that the only valid next BFI is BFI 11.
This clobbers the id if BFI 5 was entered somewhere in its middle.
Control Flow Monitoring using ECCA

### ECCA: Assertion Version

```c
#define BID 3
... BFI 3...
#define NEXT 5*7
id = NEXT+ !(id - BID);
if (swizzle) {
#define BID 5
RunAssert ( !(id%BID)) * (id%2) ); id = BID;
...BFI 5...
#define NEXT 11
id = NEXT+ !(id - BID);
}
else {
#define BID 7
RunAssert ( !(id%BID)) * (id%2) ); id = BID;
...BFI 7...
#define NEXT 11
id = NEXT+ !(id - BID);
}
#define BID 11
RunAssert ( !(id%BID)) * (id%2) ); id = BID;
...BFI 11...
```
Control Flow Monitoring using ECCA

Advantages

Detects all single and most double control flow errors
Preprocessor can be easily implemented
Low overhead if BFI is large
Can easily add assertions
Can turn on for debugging and off for shipping

Disadvantages

Must modify source code
High overhead if BFI is small
Typical Configuration

Node 1
- Application 1
- HA SW, Scripts
- OS

Node 2
- Application 1
- HA SW, Scripts
- OS

Private Disks

Shared Disks

Heartbeat LAN

OS

Private Disks
Programming for a High-Availability Cluster Environment

Functional Architecture (Qualix lingo)

Node 1

Service Group 1
- Application
- Application
- Application

Service Group N
- Application
- Application
- Application

Service Manager

Resources (e.g., disk)

Operating System

Role Manager

Scripts for SG 1
- start
- restart
- stop
- test
- failed

Scripts for SG N
- start
- restart
- stop
- test
- failed

To/From Role Managers on other Nodes
Key Strategies

Automating application operation
Maximizing speed of application failover
Design for application migration
Insulate users from failover
Design applications to detect and recover from faults
Automating Application Operation

Application should be started or stopped without operator or user intervention

Define application startup and shutdown procedures

  No operator intervention
  Report startup/shutdown to HA monitor
  Don’t let shutdown accidentally cause failover
Maximize Speed of Application Failover

Replicate code and data on multiple nodes if possible

On shared disk, use raw volumes that do not require fsck

On shared disk, use journalled file system

Minimize data loss upon failure

  Minimize in-memory volatile data

  Use restartable transactions

  Use checkpoints

Run active-active
Design for Application Migration

Avoid system-specific information

Each application should have its own IP#
Each application should have its own “hostname”
Let DNS do the work
Avoid SPU ids or MAC addresses
Avoid uname(2)
Bind to a fixed port - don’t let the system choose one for you
Bind to a relocatable IP#
Call bind() before connect() to ensure the use of the relocatable IP#
Design for Application Migration

Give each application its own volume group

Volume groups are units of migration

If two applications use the same volume group, they must migrate together

Avoid file locking if possible

Local locks will be unknown to recovered application - could cause problems

Remote NFS locks will be unknown to recovered application and may never be released
Insulate Users from Failover

Try to require no user intervention to reconnect
  Design client software to try to reconnect automatically
  Use transaction processing monitor or message queueing software to distribute, retry, and enqueue transactions

Minimize re-entry of data
  Restartable transactions and checkpointing

Minimize impact of failure
  Design for reserve capacity to minimize performance degradation upon failure
Handling Application Failures

Modularize applications and make components tolerate each others’ failure

Attempt local restart of application components

Use techniques described in this course to detect and tolerate application failures
Developing “Bug-Free” Code

Observation

Cost to fix increases 10X for each stage (design, unit test, system test, released) that bug discovery and correction is delayed

Therefore we want to

Find bugs as early and as easily as possible
Find bugs automatically with minimal effort
Minimize the skill required to catch and fix bugs
Maintain a Ship and Debug Version

Maintain a ship and a debug version of the code

Debug version designed to catch bugs

Ship version designed to run quickly and reliably

Debug version

#ifdef DEBUG ... #endif

Debug version must behave exactly the same as the ship version

Don’t apply ship constraints to debug version

Trade size and speed for error detection

Ship version

Use defensive programming techniques previously described

Use run time assertions
Use Assertions

Three categories of assertions

  Compiler assertions: CompileAssert
  Debug time assertions: DebugAssert
  Run time assertions: RunAssert

Assertions are used to provide compile-time or run-time checking of design-time assumptions

General structure

  Assert (expression that should be true)

Failed assertions cause compilation or runtime error
Compile-Time Assertions

Objective

Verify design-time assumptions at compile-time

Especially useful when maintenance programmer unknowingly violates design assumptions

Usage

...char buffer[BUFSIZE];...

...CompilerAssert ( ISPOWER2 ( sizeof(buffer)));

A mechanization of CompilerAssert

#define CompilerAssert(exp) extern char _CompilerAssert [ (exp) ? 1 : -1 ]

#define ISPOWER2 (x) (!((x)&((x)-1)))
Debug Time Assertions

Debug time assertions are used to provide debug time checking of design-time assumptions

Use debug time assertions to catch bugs

Validate function arguments and outputs

Catch undefined behavior

Audit data structures, logs, etc.

Validate that the results of debug-only redundant algorithm are identical

Debug assertions MUST NOT

Disturb memory

Initialize data

Have ANY side effects
Debug Time Assertions

Usage: Should be true for execution to proceed

```c
void func (int nValue)
{
    DebugAssert(nValue>0)
    {
        ...guarded code...
    }
}
```

Definitions:

```c
#define DebugAssert(exp) if (!(exp)) { \
    ReportError(__FILE__, __LINE__); \
    return(FALSE); \
}\
else
ReportError( char *strFILE, unsigned uLINE)
{
    fflush(stdout); \
    fprintf(stderr, “Assertion failure: file %s, line %d\n”, strFILE, uLINE); \
    fflush(stderr); \
    exit(1);
}
```
Debug Time Assertion Tips

Debug time assertions are **not** used to catch errors that can occur in practice.

Example:

```c
char *strdup (char *str)
{
    char *strnew;

    /* CORRECT: tests for illegal condition that should never occur. */
    DebugAssert(str != NULL);

    strnew = (char *)malloc(strlen(str)+1);

    /* WRONG: tests for error condition what will occur in practice and should be handled. */
    DebugAssert(strnew != NULL);

    ...
```
Debug Time Assertion Tips

Debug-time assertions MUST NOT

  disturb memory
  initialize data
  have ANY side effects

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WRONG:</strong></td>
<td><strong>RIGHT:</strong></td>
</tr>
<tr>
<td>DebugAssert ((x/=2) &gt; 0);</td>
<td>x/=2;</td>
</tr>
<tr>
<td></td>
<td>DebugAssert ((x) &gt; 0);</td>
</tr>
</tbody>
</table>
Assertion Tips

Comment your assertions: what bug are they checking for, what should the programmer try instead

The programmer that fires your assertion may assume the assertion is erroneous, otherwise

Example of commented assertion

/* Do source and destination blocks overlap? Use memmove. */
DebugAssert( (pbTo >= pbFrom + size) || (pbFrom >= pbTo + size));

Example of assert.h from SunOS /usr/include

```
#ifndef NDEBUG
#define _assert(ex)   {if (!(ex)){(void)fprintf(stderr,"Assertion failed: file "%s", line
 %d\n", __FILE__, __LINE__);exit(1);}}
#define assert(ex)    _assert(ex)
#else
#define _assert(ex)
#define assert(ex)
#endif
```

```c
#define _assert(ex)   {if (!(ex)){(void)fprintf(stderr,"Assertion failed: file "%s", line
 %d\n", __FILE__, __LINE__);exit(1);}}
#define assert(ex)    _assert(ex)
```

```c
#define assert(ex)    _assert(ex)
#define assert(ex)
```
Design Error-Resistant Interfaces

In safety critical systems, most accidents occur due to interface errors

Assume that:

- Your functions will be called with erroneous arguments
- Your error codes will be ignored
- Functions you call will produce errors
Design Error-Resistant Interfaces

Use strong function prototypes

WRONG:

```c
void *memchr ( const void *pv, int ch, int size);
/* easy for caller to swap character and size args without compiler warning */
```

RIGHT:

```c
void *memchr ( const void *pv, unsigned char ch, size_t size);
/* no way, now. */
```
Design Error-Resistant Interfaces

Don’t bury error codes in return values: make it hard to ignore them

<table>
<thead>
<tr>
<th>WRONG:</th>
<th>RIGHT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>char c:</td>
<td>char c:</td>
</tr>
<tr>
<td>....</td>
<td>...</td>
</tr>
<tr>
<td>c=getchar();</td>
<td>BOOL fgetchar ( char <em>pch) /</em> function prototype */</td>
</tr>
<tr>
<td>if ( c == EOF)</td>
<td>...</td>
</tr>
<tr>
<td>{ end of file processing }</td>
<td>if(fgetchar (&amp;c))</td>
</tr>
<tr>
<td>else</td>
<td>{c has character}</td>
</tr>
<tr>
<td>{ character processing }</td>
<td>else</td>
</tr>
</tbody>
</table>
| {EOF and c is unchanged} |}
Design Error-Resistant Interfaces

Don’t write multipurpose functions

Complex code paths are hard to test

It is hard to validate all input argument combinations using assertions

Egregious example: realloc

Use simple functions

Simple function names

Simple code paths

Make each input and output represent exactly one type of data

Easy for caller to understand simple functions

Easier to validate arguments using rigid assertions
Design Error-Resistant Interfaces

Make code intelligible at the point of call

Document calling example and emphasize potential hazards

Encourage programmer to cut and paste your recommended usage

Example

/* realloc (void *pv, size_t size) */
* typical use:
* void *pvnew; // used to protect pv if realloc fails
* pvNew = realloc(pv, sizeNew);
* if (pvNew != NULL) {
  // success...update pv
  * pv = pvNew;
  *
  } else
  * \failure – don’t destroy pv with the NULL pvNew
  *
  void *realloc(void *pv, size_t size)
...
Design Error-Resistant Interfaces

Don’t pass data in global or static memory

   Callers up or down the calling chain may be using or may
   clobber the data

Don’t use caller’s input buffers as a workspace

   You don’t really know how big they are or whether you can
   modify them

Use assertions to validate function arguments

Avoid Boolean arguments

   Easy to forget what “TRUE” means
   Easy for a fault to toggle TRUE and FALSE (NORAD fault)
Avoid Memory Theft

Don’t reference memory you don’t own or have freed
   Especially memory-mapped I/O
Don’t reference memory that you think you have locked but don’t
   This gave the SVR4 MP porters fits

Techniques
   0xdeadbeef and 0xfeedbabe memory
   Use robust data structures and structure marking
   Perform BOTH allocation and deallocation on same side of interface
Make the Compiler Work for You

Enable all optional compiler warnings, including “require prototypes for all functions”

Enable subscript range checking where possible
  Leave on in ship code if possible

Use lint

Tolerate no compiler warnings

Turn off all compiler optimizations in debug version to facilitate single-stepping through code
  There are probably more bugs in your code than in the compiler
  However, gcc optimization does provide some additional error checking for “uninitialized variable” and “return without value” errors
### Avoid Risky Coding Style

Use well-defined data types: rely only on what the ANSI standard specifically guarantees to be portable

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>0 .. 127</td>
</tr>
<tr>
<td>signed char</td>
<td>-127 .. 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>0 .. 255</td>
</tr>
<tr>
<td>short</td>
<td>-32767 .. 32767</td>
</tr>
<tr>
<td>signed short</td>
<td>-32767 .. 32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>0 .. 65535</td>
</tr>
<tr>
<td>int</td>
<td>-32767 .. 32767</td>
</tr>
<tr>
<td>signed int</td>
<td>-32767 .. 32767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0 .. 65535</td>
</tr>
<tr>
<td>long</td>
<td>-2147483647 .. 2147483647</td>
</tr>
<tr>
<td>signed long</td>
<td>-2147483647 .. 2147483647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>0 .. 2147483647</td>
</tr>
<tr>
<td>int i : n</td>
<td>0 .. (2^{(n-1)} - 1)</td>
</tr>
<tr>
<td>signed int i : n</td>
<td>-(2^{(n-1)} -1) .. (2^{(n-1)} -1)</td>
</tr>
<tr>
<td>unsigned int i : n</td>
<td>0 .. (2^{(n-1)} -1)</td>
</tr>
</tbody>
</table>

Unknown size, but at least \(n\) bits
Avoid Risky Coding Style

Look for underflows and overflows of variables

Avoid risky idioms

Don’t mix operator types

  If you must, use ( )’s to enforce precedence and type

  If you have to look up precedence in the manual, use ( )’s

Write boring code that is legible by the average programmer

Tight C does not guarantee efficient machine code; it does guarantee subsequent confusion

  We read code more often than we write it
Try Hungarian Notation

Makes it possible to identify types as you read the code without seeing the variable declaration

- `a` array
- `f` boolean flag
- `b` byte
- `ch` char
- `dw` dword
- `h` handle
- `l` long
- `lp` long pointer
- `n` int
- `p` pointer
- `w` word

Variable name = prefix + Descriptive name

Examples: `pchTo`, `phObjHandle`, `pbNew`, `phObjHandle->length`
Software Development Hygiene

Single-step through every code path of all new or modified code

Focus on data flow and state transformations

Don’t clean up old code unless absolutely necessary

Don’t implement nonstrategic or unnecessary features

Don’t implement “free” features

Don’t implement unnecessary flexibility
Error Handling and Reporting Principles

Always check for error codes returned by procedures and functions

Use common, gathered cleanup paths

Ensure that locks are released and memory is deallocated before calling error handling routine that may exit

Keep recovery code simple (remember the VOS outage data?)
Concurrent Programming

Concurrent programming is difficult to get right and difficult to debug

Don’t use concurrency unless you have to

Identify the benefits of concurrency before you use it

Avoid gratuitous nondeterminacy; it’s going to be hard enough to debug already

Don’t confuse semaphores with condition variables
Concurrent Programming

Learn concurrent programming from a good book

Concurrent Programming, Andrews

Concurrent Systems, Bacon

Multithreaded Programming with Windows NT, Pham and Garg

Use or build a library of standard concurrent programming primitives

Semaphores, monitors / condition variables

Ad hoc devices are almost certainly buggy, offer incomplete semantics, or are very hard to use
Testing

People are optimists and test to show that code does work

Most programmers quit testing when 60% of the code has been tested

Write and test code in small chunks as they are completed

Try to test under conditions that approximate reality

Fix bugs as you find them, not later

Use code coverage tool to grade testing effectiveness

Test all error handling and recovery (remember the VOS outage data?)

  It doesn’t get used very often

  It doesn’t get used unless there is already a problem

  It is hard to test
Eliminate Random Behavior, at Least in Debug Version

Force bugs to be reproducible

0xfeedbabe newly allocated memory

0xdeadbeef newly deallocated memory

Make sure routines can be made to produce same output for same input so regression testing will work

Be able to freeze dates, timestamps, random numbers, etc.

Augment data structures with auditable structures and logs

Be careful to ensure that logs and auditing do not cause behavior of debug version to differ from ship version

Unless you want to use auditable data structures in the ship version
Inspections

Inspection definition:

Group evaluation of a work product for the purpose of finding defects

Inspections are:

Formal: Well-defined roles, responsibilities, and procedures

   Documented in Stratus SED-2014, *Work Product Inspection Procedure*

Flexible: Applicable to all types of work products

   Specs, designs, code, test plans, ...

Economical: Allow defects to be uncovered and removed early in the process when they are easier and cheaper to fix

Efficient: Structured nature of inspections ensures that time is spent productively
Inspections

Inspections are NOT:

Brainstorming sessions to find solutions to problems
Performance review of author of the inspected work product
Inspection Process (1)

Planning

Identify author, moderator, recorder, inspectors and allocate their time

Setup

Distribute materials and book the room

Preparation

Inspectors review work product and record all defects

1:1 to 2:1 ratio preparation time to meeting time is appropriate

The Meeting

Walk through work product and record / classify defects and issues

4 possible outcomes: approved, not approved, conditional approval, and inspection incomplete
Inspection Process (2)

Reporting
  Written and distributed by moderator summarizing the inspection

Rework
  Author owns all defects and is responsible for addressing each

Verification
  If conditionally approved, verifier is appointed to confirm that defects have been addressed

Analysis
  Analyze results to see if process can be improved
  Provide statistics to show effectiveness of the process, provide planning data, demonstrate quality achievements, demonstrate productivity gains, etc.
  Can lead to checklist updates, process changes, documentation changes, training plans, etc.
Inspection Productivity

IBM, 1975
Inspecting test plans, test designs, and test cases reduced unit test time by up to 85%

Imperial Chemical Industries, 1982
Program maintenance effort was 0.6 minutes/line/year for inspected code, 7 minutes/line/year for uninspected code

ICL, 1986
1.58 person-hours cost to find defect in inspection, 8.47 person-hours cost to find defect in test

Stratus Continuum Languages Group, 1995
Inspection time / total project time = 10%
64% of all defects were found in inspection
Total inspection cost = cost of fixing 61 field bugs
Many More Techniques are Available

Process Improvements
Formal Specification and Verification
Structured exception handling
...

These techniques can be added to future courses as needed

Feel free to call on me at any time during your career at Stratus for consultation on using paranoid programming in your jobs
Recommended Reading

Your course papers

*Software Fault Tolerance*, by Michael Lyu, Ed.

*Safeware*, by Nancy Leveson

*Safer C*, by Les Hatton

*Proceedings of the IEEE Fault Tolerant Computing Symposia*

*C Traps and Pitfalls*, by Andrew Koenig

*Writing Solid Code*, by Steve Maguire

www.rstcorp.com
List of Course Papers


