Designing for the Real World: Software Fault Detection, Recovery, and Analysis

Corey Minyard
MontaVista CGE Linux
Architect
Your Software Has Bugs

In sports, even the best offense cannot win games without a good defense.

The same is true for software. Even the best techniques for removing bugs (software review, testing, static analysis, etc.) will not eliminate all bugs. You need defensive techniques to handle when bugs occur.
Tools and Techniques Covered

- Event-driven programming
- Microstate Accounting (MV)
- FSAD (MV)
- RtAP (MV)
- IPMI logs
- Syslog tools
- Live application coredump (MV)
- RMON (MV)
- OOM Killer

/proc
Fatal signal handling
Page tables to make read-only memory.
## Things not covered

<table>
<thead>
<tr>
<th>Hardware stuff</th>
<th>Redundancy</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPMI</td>
<td>SAF AIS</td>
<td>SELinux</td>
</tr>
<tr>
<td>HPI</td>
<td>RAID</td>
<td>Tripwire</td>
</tr>
<tr>
<td>SMART</td>
<td>Kernel Debugging</td>
<td>PAM</td>
</tr>
<tr>
<td>EDAC</td>
<td>kgdb</td>
<td>SSL</td>
</tr>
<tr>
<td>LVM</td>
<td>kdump</td>
<td>IPSec</td>
</tr>
<tr>
<td></td>
<td>lkcd</td>
<td>samhain</td>
</tr>
<tr>
<td></td>
<td>crash</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flight recorder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>console logs and IPMI SoL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IPMI kernel logs</td>
<td></td>
</tr>
</tbody>
</table>
Non-Event-Driven Programming

Process

Thread

Data
Non-Event-Driven Programming

Thread

Data

Process
Non-Event-Driven Programming

Process

Thread
Data
Non-Event-Driven Programming

Process
Event-Driven Programming

- Process
- Thread
- Database
- Data
Event-Driven Programming
Event-Driven Programming

- Auditing
- Overload Control
- Number of operations not limited by max thread count
- Each operation has a well defined start/stop point
  - Data replication
  - Operation timing

In the author's experience people can conceptually understand non-event-driven programming more easily, but it's generally easier to correctly implement a system driven by events with an event-driven system.
Defensive Programming Techniques

Thread timing to detect CPU hogs/infinite loops
Catching signals and attempting recovery
Mutex wrapper for cleanup on an error
Write-protecting memory except when explicitly accessing it
Database recovery for cleanup on an error
Watching what's going on to look for faults
  RMON
  /proc
Programming the OOM killer
Thread Timing

As a thread takes a work item, it registers with a watch thread.

The watch thread uses microstate accounting to measure the CPU used by the thread.

If the thread uses too much CPU or remains blocked for too long, the watch thread sends it a signal.

The thread handles the signal to recover from the problem.

Microstate accounting can watch for a large number of things besides CPU, like blocking on I/O, paging, etc.

Will not work without microstate accounting.
Catching Signals

If a fatal signal comes in (like a SEGV, or a signal from the watch thread), it is possible for the thread to recover from the situation.

Two basic mechanisms:

  - libunwind
  - setjmp/longjmp

When a signal occurs, it is possible to “jump back” to a known starting point and recover the thread.

Logging tracebacks is very helpful for debugging.

But what about mutexes and data....
Mutex Wrapper

To handle recovery from fatal signals, mutexes should be wrapped in a way that allows them to be released by the thread cancellation. Mutexes are registered and when a fatal exception occurs, they are released automatically.

Race conditions make this code is tricky to write correctly, attention to detail and all the possibilities is important.

See pthread_cancel_push() and friends for details, though this may not be the best way to accomplish the task.

May also be useful for tracking dynamic memory allocations and other things of that nature.
Database for Global Data

For event-driven programming, a database of items is generally required to lookup the context of an operation. This is not necessarily a fancy SQL database, it could be in-memory, a hash table, etc.

Can be used for data replication.

Use of page table access control (mprotect()) can be used to write-protect data when not being written.

The application can check out a copy of the data, make its modifications to the data, and then check them in when complete. This means that the commit to the database is atomic; if the thread fails for some reason then the contents in the database are unaffected.
Monitoring the System

Looking in /proc

Many things to monitor: state of network connections, memory usage, irq rate, device state, microstate accounting info, etc.

RMON

Allows real-time monitoring of some things. A MontaVista User Guide exists. Can look at memory usage, network transfer rates, thread statistics. Use can create their own statistics. Statistical modifications can allow rate (bytes/sec, packets/sec, etc.) or leaky bucket analysis. Thresholds allow immediate alarms to be raised, in-kernel or userland monitoring.

Syslog

Offline storage
daemontools for monitoring
The OOM Killer

If the system runs out of memory, the kernel will run an algorithm to kill a process and recover memory.

The kernel may not make the same choice you would.

You can modify the algorithm with /proc/<pid>/oom_adj. This is highly recommended.

The OOM Killer section of the MontaVista Linux User's Guide for CGE 5.0 has details.
Debugging Applications in the Field

Application coredumps

You should generally change /proc/sys/kernel/core_pattern to place coredumps in a standard location and make sure the ulimit is set at init time.

Live application coredump

Allows you to take a coredump of an application while an application runs without stopping the application.

Allows overriding coredump parameters (maximum coredump size, etc.).

FSAD

Debug an application while it runs. It does not stop the application.

Debugger is linked into the application and uses tracepoints instead of breakpoints.

Current user interface is crude.
Fixing Problems in the Field

RTAP

Allows modification of an application while it runs. The application does not need to stop, go simplex, etc. while the patch applies.
Lots of safety nets to avoid problems when patching.
Patches can be removed if they cause problems or are no longer needed.

Patching is only a small piece of software management, the next presentation by Bob Monkman of Enea will demystify that process.
Questions?