Maksym Planeta    Björn Döbel

Operating Systems Meet Fault Tolerance
Microkernel-Based Operating Systems // Dresden, 22.01.2018
‘If there is more than one possible outcome of a job or task, and one of those outcome will result in disaster or an undesirable consequence, then somebody will do it that way.’

Edward Murphy jr.
Goal

- Fault tolerance
  - Problems
  - Solutions
- Operating systems techniques
Outline

- Murphy and the OS: Is it really that bad?
- Fault-Tolerant Operating Systems
  - Minix3
  - CuriOS
  - L4ReAnimator
- Dealing with Hardware Errors
  - Transparent replication as an OS service
Textbook terminology

Dependability threats:
- Failure
- Error
- Fault

Dependability means
- Prevention
- Removal
- Forecasting
- Tolerance
Why Things go Wrong

- Programming in C:
  
  *This pointer is certainly never going to be NULL!*
Why Things go Wrong

- Programming in C:
  *This pointer is certainly never going to be NULL!*

- Layering vs. responsibility:
  *Of course, someone in the higher layers will already have checked this return value.*
Why Things go Wrong

- Programming in C:
  *This pointer is certainly never going to be NULL!*

- Layering vs. responsibility:
  *Of course, someone in the higher layers will already have checked this return value.*

- Concurrency:
  *This struct is shared between an IRQ handler and a kernel thread. But they will never execute in parallel.*
Why Things go Wrong

- Programming in C:
  *This pointer is certainly never going to be NULL!*

- Layering vs. responsibility:
  *Of course, someone in the higher layers will already have checked this return value.*

- Concurrency:
  *This struct is shared between an IRQ handler and a kernel thread. But they will never execute in parallel.*

- Hardware interaction:
  *But the device spec said, this was not allowed to happen!*
Why Things go Wrong

- Programming in C:
  *This pointer is certainly never going to be NULL!*

- Layering vs. responsibility:
  *Of course, someone in the higher layers will already have checked this return value.*

- Concurrency:
  *This struct is shared between an IRQ handler and a kernel thread. But they will never execute in parallel.*

- Hardware interaction:
  *But the device spec said, this was not allowed to happen!*

- Hypocrisy:
  *I'm a cool OS hacker. I won't make mistakes, so I don't need to test my code!*
A Classic Study

- A. Chou et al.: *An empirical study of operating system errors*, SOSP 2001
- Automated software error detection (today: https://www.coverity.com)
- Target: Linux (1.0 - 2.4)
  - Where are the errors?
  - How are they distributed?
  - How long do they survive?
  - Do bugs cluster in certain locations?
Revalidation of Chou’s Results

- N. Palix et al.: *Faults in Linux: Ten years later*, ASPLOS 2011
- 10 years of work on tools to decrease error counts - has it worked?
- Repeated Chou’s analysis until Linux 2.6.34
Linux: Lines of Code

Figure: Linux directory sizes (in MLOC) [15]

- Drivers/Staging
- Arch
- Drivers w/o Staging
- File Systems (FS)
- Net
- Sound
- Other
Faults per Subdirectory (2001)

Figure: Number of errors per directory in Linux [5]
The clustering of bugs in Linux and other systems is developed by a wide range of programmers who tend to be more familiar with the device. We attempt to compensate for the unknown whether this set of bugs is representative of all errors.

Potential future work could use more sophisticated ranking algorithms (as with Intrinsa [11]). The result of this low-level focus is that good code can masquerade as good code if it does not happen. The clustering of bugs is almost three times greater than the rest of the kernel.

Figure: Rate of errors compared to other directories [5]
Faults in Linux: Ten Years Later

Checker Chou et al. Our results checked unchecked
Block 206 87 71
Null 124 267

In this section, we assess the extent to which the trends observed for Linux 2.4.1 and previous versions continue in Linux 2.6 kernels. Nevertheless, the distribution of these faults among the various directories is roughly comparable, and thus we find a similar result as that of Chou et al., regardless of any differences in the checkers.

As shown in Figure 4(b), we obtain a similar result with the degree of confidence observed by Chou et al. As shown in Figure 4(a), we also observe that the largest number of faults is in the drivers directory, with an average age of over 5 years, have a significantly lower fault rate. On the other hand, we find a definite difference among the checkers; the relative rate of faults for the various directories is roughly comparable, and thus we find fault rates of around 3% for the functions have a higher fault rate, of up to 3% for the NullRef checker. We also find fault rates of around 3% for the NullRef checker, for files of all ages and for larger functions.

Figure: Rate of errors compared to other directories [15]
Bug Lifetimes (2011) [15]

Figure: Per directory

Figure: Per finding and fixing difficulty, and impact likelihood
Software Engineering addressing faults

- QA
  Examples: Manual testing, automated testing, fuzzing
- Continuous Integration
- Static analysis
- Using compiled languages
- Guidelines, best practices, etc.
  Examples: MISRA C++, C++ Guideline Support Library
Example: MISRA C++ 2008

Rule 0-1-7

The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.
Example: MISRA C++ 2008

Rule 0-1-7
The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Rule 3-9-3
The underlying bit representations of floating-point values shall not be used.
Example: MISRA C++ 2008

Rule 0-1-7
The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.

Rule 3-9-3
The underlying bit representations of floating-point values shall not be used.

Rule 6-4-6
The final clause of a switch statement shall be the default-clause.
Rule 3-4-1

(Required) An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

<table>
<thead>
<tr>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining variables in the minimum block scope possible reduces the visibility of those variables and therefore reduces the possibility that these identifiers will be used accidentally. A corollary of this is that global objects (including singleton function objects) shall be used in more than one function.</td>
</tr>
</tbody>
</table>
Rule 3-4-1: Example

```c
void f(int32_t k)
{
    int32_t j = k * k; // Non-compliant
    {
        int32_t i = j; // Compliant
        std::cout << i << j << std::endl;
    }
}
```

In the above example, the definition of `j` could be moved into the same block as `i`, reducing the possibility that `j` will be incorrectly used later in `f`. 
• Faults are an issue.

• Hardware-related stuff is the worst.

• Now what can the OS do about it?
Minix3 – A Fault-tolerant OS

User processes:
- User Processes
- Server Processes
- Device Processes

Kernel:
- Kernel
- Clock Task
- System Task

- Shell
- Make
- User
- ... Other
- File
- PM
- Reinc
- ... Other
- Disk
- TTY
- Net
- Printer
- Other
Minix3: Fault Tolerance

• Address Space Isolation
  - Applications only access private memory
  - Faults do not spread to other components

• User-level OS services
  - Principle of Least Privilege
  - Fine-grain control over resource access
    - e.g., DMA only for specific drivers

• Small components
  - Easy to replace (micro-reboot)

---

Minix3: Fault Detection

- Fault model: transient errors caused by software bugs
- Fix: Component restart
- *Reincarnation server* monitors components
  - Program termination (crash)
  - CPU exception (div by 0)
  - Heartbeat messages
- Users may also indicate that something is wrong
Repair

- Restarting a component is insufficient:
  - Applications may depend on restarted component
  - After restart, component state is lost

- Minix3: explicit mechanisms
  - Reincarnation server signals applications about restart
  - Applications store state at data store server
  - In any case: program interaction needed
    - Restarted app: store/recover state
    - User apps: recover server connection
OSIRIS: Transparent recovery in MINIX²

/ * initialization */
while (true) {
    receive(&endpoint, &request);
    switch (request.type) {
        case REQ_TYPE_x:
            reply = req_handler_x(request);
            break;
        case REQ_TYPE_y:
            reply = req_handler_y(request);
            break;
        /* ... */
    }
    if (reply) send(endpoint, reply);
}

- Target typical server architecture
- Local checkpoints
- Recovery windows
- Compiler assisted state recording

OSIRIS: Results

<table>
<thead>
<tr>
<th>Server</th>
<th>Recovery coverage (%)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pessimistic</td>
<td>Enhanced</td>
</tr>
<tr>
<td>PM</td>
<td>54.9</td>
<td>61.7</td>
</tr>
<tr>
<td>VFS</td>
<td>72.3</td>
<td>72.3</td>
</tr>
<tr>
<td>VM</td>
<td>64.6</td>
<td>64.6</td>
</tr>
<tr>
<td>DS</td>
<td>47.1</td>
<td>92.8</td>
</tr>
<tr>
<td>RS</td>
<td>49.4</td>
<td>50.5</td>
</tr>
<tr>
<td>Weighted average</td>
<td>57.7</td>
<td>68.4</td>
</tr>
</tbody>
</table>

Figure: Percentage of time inside recovery window

Table: Recovery coverage (%)

<table>
<thead>
<tr>
<th>Recovery mode</th>
<th>Pass</th>
<th>Fail</th>
<th>Shutdown</th>
<th>Crash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateless</td>
<td>19.6%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>80.4%</td>
</tr>
<tr>
<td>Naive</td>
<td>20.6%</td>
<td>2.4%</td>
<td>0.0%</td>
<td>77.0%</td>
</tr>
<tr>
<td>Pessimistic</td>
<td>18.5%</td>
<td>0.0%</td>
<td>81.3%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Enhanced</td>
<td>25.6%</td>
<td>6.5%</td>
<td>66.1%</td>
<td>1.9%</td>
</tr>
</tbody>
</table>

Figure: Survivability under random fault injection
L4ReAnimator: Restart on L4Re

- L4Re Applications
  - Loader component: ned
  - Detects application termination: parent signal
  - Restart: re-execute Lua init script (or parts of it)
  - Problem after restart: capabilities
    - No single component knows everyone owning a capability to an object
    - Minix3 signals won’t work

---

**L4ReAnimator: Lazy recovery**

- Only the application itself can detect that a capability vanished
- Kernel raises *Capability fault*
- Application needs to re-obtain the capability: execute *capability fault handler*
- Capfault handler: application-specific
  - Create new communication channel
  - Restore session state
- Programming model:
  - Capfault handler provided by server implementor
  - Handling transparent for application developer
  - *Semi-transparency*
Distributed snapshots

- Localized checkpoints
- Problem: Unlimited rollbacks
- Solution: Create global snapshot
- No synchronized clock
- No shared memory
- Only point-to-point messages

---

Break

- Minix3 fault tolerance
  - Architectural Isolation
  - Explicit monitoring and notifications
- L4ReAnimator
  - semi-transparent restart in a capability-based system
- Next: CuriOS
  - smart session state handling
CuriOS: Servers and Sessions

- State recovery is tricky
  - Minix3: Data Store for application data
  - But: applications interact
    - Servers store *session-specific* state
    - Server restart requires potential rollback for every participant

---

CuriOS: Server State Regions

- CuiK kernel manages dedicated session memory: *Server State Regions*
- SSRs are managed by the kernel and attached to a client-server connection

![Diagram](image_url)
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B's session state
CuriOS: Protecting Sessions

- SSR gets mapped only when a client actually invokes the server
- Solves another problem: failure while handling A's request will never corrupt B’s session state
CuriOS: Transparent Restart

- CuriOS is a *Single-Address-Space OS*:
  - Every application runs on the same page table (with modified access rights)
Transparent Restart

- Single Address Space
  - Each object has unique address
  - Identical in all programs
  - Server := C++ object

- Restart
  - Replace old C++ object with new one
  - Reuse previous memory location
  - References in other applications remain valid
  - OS blocks access during restart
seL4: Formal verification of an OS kernel

- seL4: https://sel4.systems/
- Formally verify that system adheres to specification
- Microkernel design allows to separate components easier
- Hence verification process is easier

---

Verification of a microkernel

Figure: The seL4 design process [13]
Refinement of verification

Figure: Refinement layers in the verification of seL4 [13]
Break

- seL4
  - Assumes correctness of compiler, assembly code, and hardware
  - DMA over IOMMU
  - Architectures: arm, x86
  - Virtualization
  - Future: Verification on multicores

- All these frameworks only deal with software errors.
- What about hardware faults?
Transient Hardware Faults

- Radiation-induced soft errors
  - Mainly an issue in avionics+space?

- DRAM errors in large data centers
  - Google study: >2% failing DRAM DIMMs per year [16]
  - ECC insufficient [12]

- Decreasing transistor sizes → higher rate of errors in CPU functional units [7]
Transparent Replication as OS Service [9, 8]
Transparent Replication as OS Service [9, 8]
Transparent Replication as OS Service [9, 8]
Transparent Replication as OS Service [9, 8]
Transparent Replication as OS Service [9, 8]
Romain: Structure
Romain: Structure

Replica

Replica

Replica

Master
Romain: Structure
Romain: Structure
Replica Memory Management

Replica 1

Replica 2

Master
Replica Memory Management

Replica 1

Replica 2

Master

rw  ro  ro

rw  ro  ro
Replica Memory Management

Replica 1

Replica 2

Master
Replicating SPEC CPU 2006 [10]
Replicating SPEC CPU 2006 [10]

Sources of overhead:
- System call interception
  - Frequent memory allocation
- Cache effects

Normalized Runtime

<table>
<thead>
<tr>
<th>Application</th>
<th>Single Replica</th>
<th>Two Replicas</th>
<th>Three Replicas</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>hmer</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>sjeng</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>libquant</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>h264ref</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>tonto</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>lmb</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>omnet++</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>astar</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>GEOM</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Error Coverage [10]
Error Coverage [10]
How About Multithreading?

Replica 1

A1
A2
A3
A4

Replica 2

A1
A2
A3
A4
How About Multithreading?
How About Multithreading?
Problem: Nondeterminism

Replica 1

Replica 2
Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- **Compiler solutions [2]:**
  No support for binary-only software
Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions [2]:
  No support for binary-only software
- **Workspace-Consistent Memory** [1]:
  Requires per- replica and per-thread memory copies
Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions [2]:
  No support for binary-only software
- Workspace-Consistent Memory [1]:
  Requires per-replica and per-thread memory copies
- Lock-Based Determinism
  - Reuse ideas from Kendo [14]
Solution: Deterministic Multithreading

- Related work: debugging multithreaded programs
- Compiler solutions [2]:
  No support for binary-only software
- Workspace-Consistent Memory [1]:
  Requires per-replica and per-thread memory copies
- Lock-Based Determinism
  - Reuse ideas from Kendo [14]
  - Only for lock-based software!
Enforced Determinism

- Adapt libpthread
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`
  - `__pthread_lock`
  - `__pthread_unlock`

- Lock operations reflected to Romain master

- Master enforces lock ordering
Cooperative Determinism

- Replication-aware libpthread
- Replicas agree on acquisition order w/o master invocation
- Trade-off: libpthread becomes single point of failure
- Alternative: place INT3 into four functions
Overhead: SPLASH2, 2 workers [10]
Overhead: SPLASH2, 4 workers

Run time normalized vs. native

- Radiosity: 3.93
- Barnes: 2.94
- FMM: 2.02
- Raytrace: 2.02
- Water: 2.02
- Volrend: 2.02
- Ocean: 3.93
- FFT: 2.94
- LU: 2.02
- Radix: 2.02
- GEOM: 2.02

Categories: Single Replica, Two Replicas, Three Replicas
Sources of overhead:
- System call interception
- Cache effects
- Lock density
Hardening the RCB

- **We need:** Dedicated mechanisms to protect the RCB (HW or SW)
- **We have:** Full control over software
- Use FT-encoding compiler?
  - Has not been done for kernel code yet
- RAD-hardened hardware?
  - Too expensive

Why not split cores into resilient and non-resilient ones?
Summary

- OS-level techniques to tolerate SW and HW faults
- Address-space isolation
- Microreboots
- Various ways of handling session state
- Replication against hardware errors
Further Reading

- **Minix3**: Jorrit Herder, Ben Gras,, Philip Homburg, Andrew S. Tanenbaum: *Fault Isolation for Device Drivers*, DSN 2009

- **CuriOS**: Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle and Roy H. Campbell *CuriOS: Improving Reliability through Operating System Structure*, OSDI 2008

- **Qmail**: D. Bernstein: *Some thoughts on security after ten years of qmail 1.0*

- **seL4**: Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick and others *Formal verification of an OS kernel*, SOSP 2009

- **Romain**:
  - Björn Döbel, Hermann Härtig: *Can We Put Concurrency Back Into Redundant Multithreading?*, EMSOFT 2014
Bibliography I


Bibliography II


Bibliography III


Bibliography IV

