Operating Systems Meet Fault Tolerance
Microkernel-Based Operating Systems

Maksym Planeta       Björn Döbel

30.01.2018
“If there’s more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way.” (Edward Murphy Jr.)
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!*

- 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!
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
Faults in Linux: Ten Years Later

To give an overview of the software we are studying, we first consider the evolution in code size of the Linux kernel between version 1.0, released in March 1994, and version 2.6.33, released in February 2010, as shown in Figure 1. We give the size of the development versions, including those for new drivers that are not yet mature enough to be included in the default Linux configuration, but also include only the ANSI C code. The code sizes are broken down by directory, highlighting the largest directories. For most directories, the code growth has been roughly linear since Linux 1.0. Some exceptions are drivers/staging, which has made up 57% of the source code since Linux 2.6.29, excluding drivers that were previously in the drivers directory.

An additional 180,000 lines of C code were added in Linux 2.6.28 as an incubator to run the checkers on the new code, and then repeat the correlation process. As our collected data contains information about Linux releases such as the release date and code size, and information about each group of reports to be considered by the user, we wrote more than 1,900 lines of PL/pgSQL code, amounting to, in total, 1.5 GB of data. To analyze the collected data, we imported their histories into the database, along with the associated notes. The database is provided to allow the user to classify each group of reports, whether considered to be unknown, and the user must indicate, via an interface based on the emacs "org" mode, whether they represent the same fault or unrelated ones. Once the correlation process is complete, a similar interface is provided to allow the user to classify each group of reports, and decreases. Many of the increases involve the introduction of new file systems, and are broken down by directory, highlighting the largest directories. Finally, staging grew substantially in 2.6.29. All in all, these extensions to a new version of Linux are released, it is only necessary to extend the results to a new version of Linux. When a new version of Linux is released, it is only necessary to run the checkers on the new code, and then repeat the correlation process. As our collected data contains information about Linux releases such as the release date and code size, and information about each group of reports to be considered by the user, we wrote more than 1,900 lines of PL/pgSQL code, amounting to, in total, 1.5 GB of data. To analyze the collected data, we imported their histories into the database, along with the associated notes. The database is provided to allow the user to classify each group of reports, whether considered to be unknown, and the user must indicate, via an interface based on the emacs "org" mode, whether they represent the same fault or unrelated ones. Once the correlation process is complete, a similar interface is provided to allow the user to classify each group of reports, whether considered to be unknown, and the user must indicate, via an interface based on the emacs "org" mode, whether they represent the same fault or unrelated ones. Once the correlation process is complete, a similar interface is provided to allow the user to classify each group of reports, whether considered to be unknown, and the user must indicate, via an interface based on the emacs "org" mode, whether they represent the same fault or unrelated ones. Once the correlation process is complete, a similar interface is provided to allow the user to classify each group of reports, whether considered to be unknown, and the user must indicate, via an interface based on the emacs "org" mode, whether they represent the same fault or unrelated ones.
Faults per Subdirectory (2001)

Figure: Number of errors per directory in Linux [4]
Fault Rate per Subdirectory (2001)

Figure: Rate of errors compared to other directories [4]
Faults in Linux: Ten Years Later

As shown in Figure 4(b), we obtain a similar result with distributions they considered. We can thus confidently degree of confidence observed by Chou et al. of confidence is comparable to the highest degree

Furthermore find that the drivers fault rate in the directory has a rate of over 8 for this case. With both

However, the distribution of these faults among the checker, for files of all ages and for larger func-

Nevertheless, the distribution of these faults among the drivers et al. similar to that found by Chou et al.

In any case, we find a definite lower fault rate. On the other hand, we find a definite

Finally, unlike Chou et al., we observe a high rate of

As shown in Figure 4(a), we also observe that the largest number of

In this section, we assess the extent to which the trends

In Linux 2.4.1 code.

Overall, we find no particular difference between

In this section, we assess the extent to which the trends

In this section, we assess the extent to which the trends

In this section, we assess the extent to which the trends
Bug Lifetimes (2011)

(a) Per directory

(b) Per finding and fixing difficulty, and impact likelihood

Figure: Average fault lifespans [14]
Break

- Faults are an issue.
- Hardware-related stuff is worst.
- Now what can the OS do about it?
Minix3 – A Fault-tolerant OS

User processes
- User Processes
- Server Processes
- Device Processes

User processes:
- Shell
- Make
- User
- ... Other
- File
- PM
- Reinc
- ... Other
- Disk
- TTY
- Net
- Printer
- Other

Kernel
- Kernel
- Clock Task
- System Task
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 MINIX\textsuperscript{2}

```c
/* 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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pessimistic</td>
</tr>
<tr>
<td>PM</td>
<td>54.9</td>
</tr>
<tr>
<td>VFS</td>
<td>72.3</td>
</tr>
<tr>
<td>VM</td>
<td>64.6</td>
</tr>
<tr>
<td>DS</td>
<td>47.1</td>
</tr>
<tr>
<td>RS</td>
<td>49.4</td>
</tr>
<tr>
<td>Weighted average</td>
<td>57.7</td>
</tr>
</tbody>
</table>

**Figure:** Percentage of time inside recovery window

**Table IV.** Benchmark slowdowns

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DS</th>
<th>VFS</th>
<th>PM</th>
<th>VM</th>
<th>DS</th>
<th>VFS</th>
<th>PM</th>
<th>VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>dhry2reg</td>
<td>122.5</td>
<td>10.5</td>
<td>0.0</td>
<td>41.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execl</td>
<td>57.7</td>
<td>68.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux OSIRIS</td>
<td>48.5</td>
<td>11.9</td>
<td>0.0</td>
<td>39.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** Survivability under random fault injection

**Table III.** Survivability under random fault injection of faults

<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>
L4ReApplications: Restart on L4Re³

- 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*
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

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

![Diagram of CuriOS session protection]

```plaintext
Server State

Client A
  -> Client State A

Client B
  -> Client State B

call()
```
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

Diagram:

- Server State
  - Server
  - reply()

- Client State A
  - Client A
  - Client State A

- Client State B
  - Client B
CuriOS: Transparent Restart

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

![Diagram of CuriOS](image-url)
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

![Diagram of the seL4 design process](image)

**Figure:** The seL4 design process [12]
Refinement of verification

Figure: Refinement layers in the verification of seL4 [12]
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 [15]
  - ECC insufficient [11]

- Decreasing transistor sizes → higher rate of errors in CPU functional units [6]
Transparent Replication as OS Service [8, 7]

Application

L4 Runtime Environment

L4/Fiasco.OC microkernel
Transparent Replication as OS Service [8, 7]

Replicated Application

L4 Runtime Environment

Romain

L4/Fiasco.OC microkernel
Transparent Replication as OS Service [8, 7]

Unreplicated Application

Replicated Application

L4 Runtime Environment

Romain

L4/Fiasco.OC microkernel
Transparent Replication as OS Service [8, 7]

- Replicated Driver
- Unreplicated Application
- Replicated Application

- L4 Runtime Environment
- Romain

- L4/Fiasco.OC microkernel
Transparent Replication as OS Service [8, 7]
Romain: Structure
Romain: Structure
Romain: Structure

![Diagram of a system structure]

- Replica
- Replica
- Replica

- Resource Manager

- System Call Proxy

Master
Resource Management: Capabilities

Replica 1

1 2 3 4 5 6
Resource Management: Capabilities

Replica 1

1 2 3 4 5 6

Replica 2

1 2 3 4 5 6
Resource Management: Capabilities

Replica 1

1 2 3 4 5 6

Replica 2

1 2 3 4 5 6

Master

1 2 3 4 5 6
Partitioned Capability Tables

Replica 1
Marked used

Replica 2

Master
private
Replica Memory Management

Replica 1

Replica 2

Master
Replica Memory Management

Replica 1

Replica 2

Master
Replica Memory Management

Replica 1

Replica 2

Master
Replicating SPEC CPU 2006 [9]
Replicating SPEC CPU 2006 [9]

Sources of overhead:
- System call interception
  - Frequent memory allocation
- Cache effects
Error Coverage [9]
Error Coverage [9]

![Error Coverage Chart](chart.png)

- Bitcount
- Bitcount/TMR
- IPC
- IPC/TMR
- Dijkstra
- Dijkstra/TMR
- CRC32
- CRC32/TMR

Ratio of Total Faults in %

- No Effect
- Crash
- SDC
- Timeout
- Recovered (Compare)
- Recovered (Timeout)

39/53
How About Multithreading?

Replica 1

Replica 2
How About Multithreading?

Replica 1

Replica 2
How About Multithreading?

Replica 1

Replica 2
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 [13]
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 [13]
  - **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
Cooperation: Lock Acquisition

lock_rep(mtx)
Cooperation: Lock Acquisition

lock_rep(mtx)

- spinlock
  
  mtx.spinlock

Owner free?

Owner self?

spinunlock

Yield CPU

Store Owner ID

Store Owner

Epoch

spinunlock

Epoch matches?

return

Yes

No
Cooperation: Lock Acquisition

lock_rep(mtx)

spinlock
(mtx.spinlock)

Owner free?

Yield CPU

Store Owner ID

Epoch

spinunlock

Epoch matches?

return

Yes

No

No
Cooperation: Lock Acquisition

lock_rep(mtx)

- spinlock (mtx.spinlock)
- Owner free?
  - Yes
    - Store Owner ID
    - Store Owner Epoch
    - return spinunlock (mtx.spinlock)
  - No
    - Yield CPU
    - Store Owner ID
    - Store Owner Epoch
    - return spinunlock (mtx.spinlock)
Cooperation: Lock Acquisition

\[
\text{lock_rep}(\text{mtx})
\]

1. Spinlock: \(\text{mtx}.\text{spinlock}\)
2. Check if Owner is free?
   - No: Loop back to spinlock
   - Yes: Proceed to next step
3. Check if Owner is self?
   - No: Yield CPU and loop back to spinlock
   - Yes: Proceed to next step
4. Store Owner ID
5. Store Owner Epoch
6. Spinunlock: \(\text{mtx}.\text{spinlock}\)

If Epoch matches, return; otherwise, yield CPU and loop back to spinlock.
Cooperation: Lock Acquisition

```
lock_rep(mtx)

spinlock (mtx.spinlock)

Owner free?  Owner self?

Yes    Yes

Store Owner ID

Store Owner Epoch

spinunlock (mtx.spinlock)

return

No

Yes matches?

Yes

No
```

Cooperation: Lock Acquisition

```
lock_rep(mtx) ->
  spinlock (mtx.spinlock) ->
  Owner free? No ->
  Owner self? No ->
  spinunlock (mtx.spinlock) ->
  Yield CPU

Yes ->
  Store Owner ID

Yes ->
  Epoch matches?

Yes ->
  return
```
Cooperation: Lock Acquisition

lock_rep(mtx) →

spinlock (mtx.spinlock) →

Owner free? →

Yes → Store Owner ID →

Store Owner Epoch →

spinunlock (mtx.spinlock) →

Owner self? →

Yes → Epoch matches? →

Yes → return →

Yield CPU

No →

spinunlock (mtx.spinlock) →

No →

Epoch matches? →

No →

return
Overhead: SPLASH2, 2 workers [9]
Overhead: SPLASH2, 4 workers

Runtime normalized vs. native

Radiosity  Barnes  FMM  Raytrace  Water  Volrend  Ocean  FFT  LU  Radix  GEOM

Single Replica  Two Replicas  Three Replicas

3.93  2.94  2.02  2.02
Overhead: SPLASH2, 4 workers

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**: Daniel 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 II


