CuriOS: Improving Reliability through Operating System Structure

Nils Asmussen

Paper Reading Group

08/29/2012

Outline

1. Introduction
2. Related OSs
3. Design of CuriOS
4. Evaluation
5. Conclusion

Outline

1. Introduction
2. Related OSs
3. Design of CuriOS
4. Evaluation
5. Conclusion

Motivation

- OS reliability is still a major issue
- Microkernels improve that by isolating components from each other
- But most of them don't support restartability or at least not in a satisfying way
- Problem 1: blindly restarting services does not help because of client-specific state
- Problem 2: Still too much rights (e.g. destroying state of client A while serving client B)

Alternatives

- Redundancy in HW and SW helps but is expensive
- Writing clients that are aware of faulting services is possible but difficult
- Checkpointing
  - Requires multiple checkpoints to avoid rolling back to a broken state
  - Leads to high memory and performance overhead

Outline

1. Introduction
2. Related OSs
3. Design of CuriOS
4. Evaluation
5. Conclusion

Brief Description

Minix3

- Reincarnation server is responsible for restarting crashed services and drivers
- Does only work well for stateless drivers/services
- Provides Datastore that can be used for checkpoints

L4/Iguana

- Collection of OS services running on top of L4Ka::Pistachio
- Offers resource management, protection and some device drivers
- No support for restartability

Brief Description

Chorus

- Services run in privileged mode and share address space of kernel
- “Hot restart” mechanism allows servers to maintain their state
- No technique to prevent corruption of state
- Chorus OS services don’t take advantage of “hot restart”

EROS

- Saves snapshots periodically to disk
- Performs some consistency checks and keeps multiple snapshots

Comparison

Kernel	**Restartability**
Minix3	Works only for stateless services
L4/Iguana	Might work for stateless services
Chorus	Does not work for stateful (?), stateless?
EROS	May work by restoring checkpoint

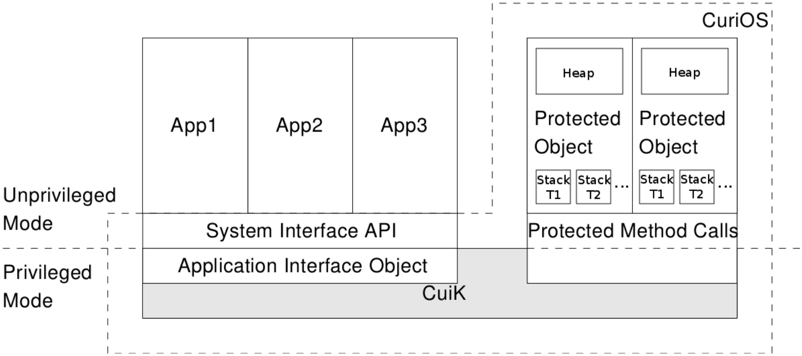
Observations

- Transparency of addressing
  → Clients should be able to use the same address
- Suspension of clients
  → No time outs or new requests during recovery
- Persistence of client-specific state
  → Results of previous requests should persist
- Isolation of client-specific state
  → An error should not corrupt state of unaffected clients

Outline

1. Introduction
2. Related OSs
3. Design of CuriOS
4. Evaluation
5. Conclusion

Overview


CuriOS
App1
App2
App3
Heap
Protected Object
Stack T1
Stack T2
...
Heap
Protected Object
Stack T1
Stack T2
...
Unprivileged Mode
System Interface API
Protected Method Calls
Privileged Mode
Application Interface Object
CuiK

Server State Management

Basics

- Servers that need client-specific state use the state management of CuiK
- A Server State Region (SSR) is an object that can be memory protected
- It is created if a client establishes a connection to a server
- A server can only access the SSR while it is processing a request from the corresponding client

Server State Management

Server types

1. Servers that do not require all client-states for operation
2. Servers that need all client-states

Server State Management

Server types

1. Servers that do not require all client-states for operation
2. Servers that need all client-states

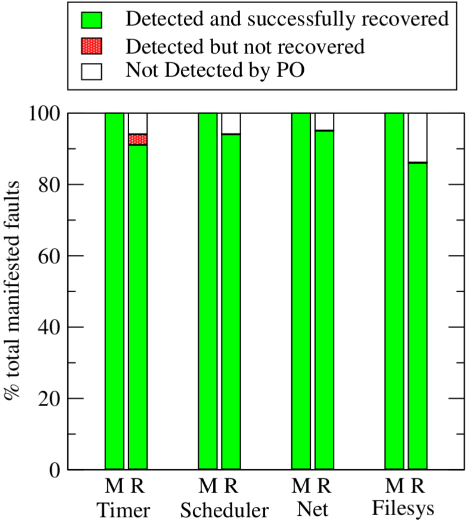
Consistency checks

1. Recovery routine uses magic numbers in objects that are checked
2. Server-specific checks can be implemented to ensure that pointers and numbers are within expected ranges

Outline

1. Introduction
2. Related OSs
3. Design of CuriOS
4. Evaluation
5. Conclusion

Error Recovery



Performance

Operation	**Instructions**	**Time**
Context Switch	?	$74\mu s$
Protected Call Without SSR	1594 ± 4	$195.7 \pm 0.5\mu s$
Protected Call With SSR	4893 ± 3	$378.9 \pm 0.9\mu s$
Error detection + Recovery	?	$X * 100\mu s$

Memory

SSRs

- Each SSR is memory protected and thus has to be on its own page (1KB on ARM)
- Assuming that typical client states are quite small, you waste nearly one page per client

Memory

SSRs

- Each SSR is memory protected and thus has to be on its own page (1KB on ARM)
- Assuming that typical client states are quite small, you waste nearly one page per client

POs

- Each PO has its own heap and a stack for every thread that uses the PO
- They say that the overhead per PO is in the order of tens of KBs
- Taking into account that they designed the file service to use one PO per open file, this is quite a lot

Outline

1. Introduction
2. Related OSs
3. Design of CuriOS
4. Evaluation
5. Conclusion

Conclusion

- Nice concepts for restartable services and protection against unaffected state corruption
- Unsatisfying evaluation
- A lot of open questions . . .

Discussion Questions

- How big is the private heap in POs and can it grow?
- How do they place programs in the single-address-space OS? PIC? statically specified?
- Shouldn't it be possible to build a similar system with multiple address spaces?
- Performance overhead? Comparison? Real workload?
- Is the memory overhead really acceptable?
- What about some kind of segmentation instead of paging?