Using Hardware Transactions for OS Reliability

Spring 2014

Eran Harpaz
Hagar Porat

Supervisor: Noam Shalev

NSSL lab

CSR: Core Surprise Removal in Commodity Operating Systems

Noam Shalev
Technion
noams@campus.technion.ac.il

Eran Harpaz
Technion
seharpaz@campus.technion.ac.il

Hagar Porat
Technion
hagarp@campus.technion.ac.il

Idit Keidar
Technion
idish@ee.technion.ac.il

Yaron Weinsberg
IBM Labs
yaron@il.IBM.com

Abstract

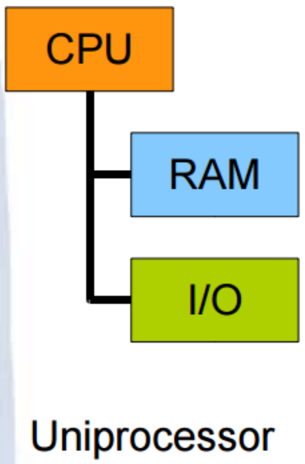
One of the adverse effects of shrinking transistor sizes is that processors have become increasingly prone to hardware faults. At the same time, the number of cores per die rises. Consequently, core failures can no longer be ruled out, and future operating systems for many-core machines will have to incorporate fault tolerance mechanisms.

In this paper we present CSR, a strategy for recovery from unexpected permanent processor faults in commodity operating systems. Our approach overcomes surprise removal of faulty cores, and also tolerates cascading core failures. When a core fails in user mode, CSR terminates the process executing on that core, and migrates the remaining processes in its run-queue to other cores. We further show how hardware transactional memory may be used to overcome failures in critical kernel code. Our solution is scalable, incurs low overhead, and is designed to integrate into modern operating systems on multi-core architectures. We have implemented it in the Linux kernel, using Haswell's Transactional Synchronization Extension, and tested it on a real system.

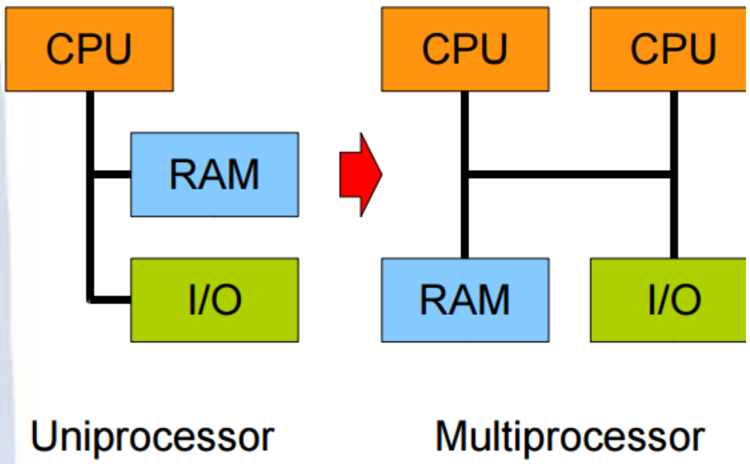
size and voltage supply. However, with hardware shrinking, the probability of physical flaws on the chip significantly rises [7, 15], and chances for processor faults become substantial [57]. With many tens or even hundreds of cores per chip, core failures can no longer be ruled out. Recent studies show that, even on consumer machines, hardware faults are not rare [42], and memory errors are currently dominated by hard errors, rather than soft-errors [54]. We therefore believe that, with technology advances, tolerating failures of individual cores shall become inevitable.

Current operating systems, including Linux and Windows, crash in the face of any permanent core fault and most chip-originated soft errors. As we explain later, the reason for the crash lies in the various kernel mechanisms that require the collaboration of the faulty core. Thus, the failure of a single core, out of hundreds in the foreseeable future, brings down the entire system. Cloud systems, for example, usually consolidate multiple virtual machines on a single server in order to improve its utilization [6, 31, 33, 52, 56]. In such settings, the failure of a single core crashes all the VMs running on the server.

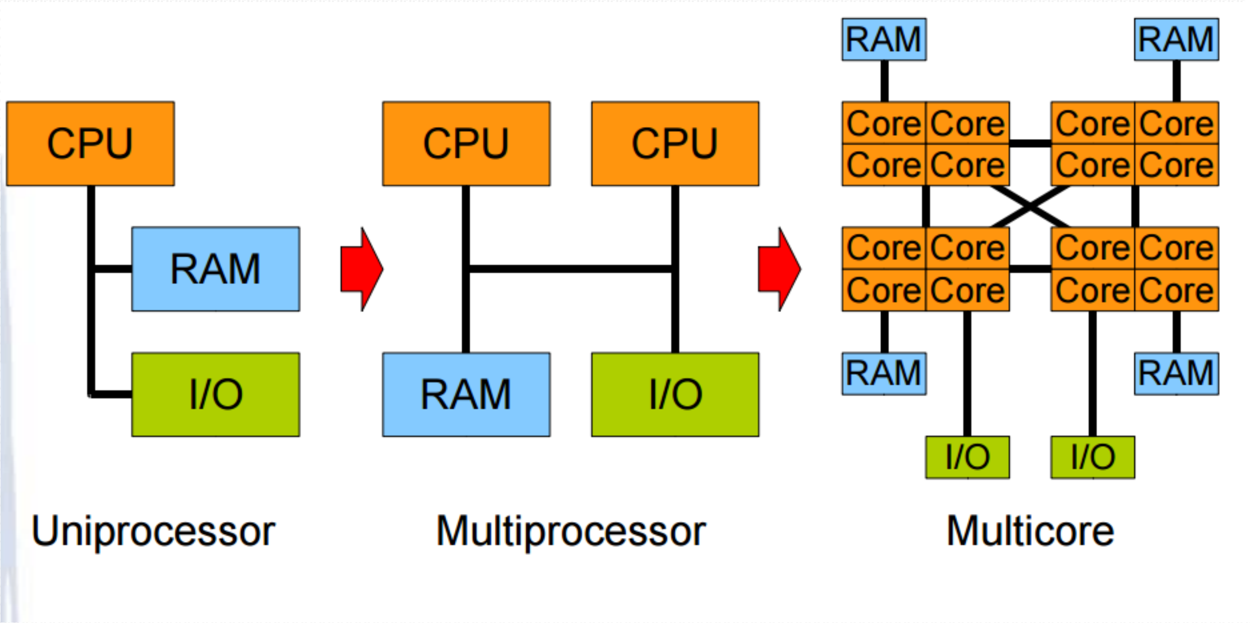
Background



Background

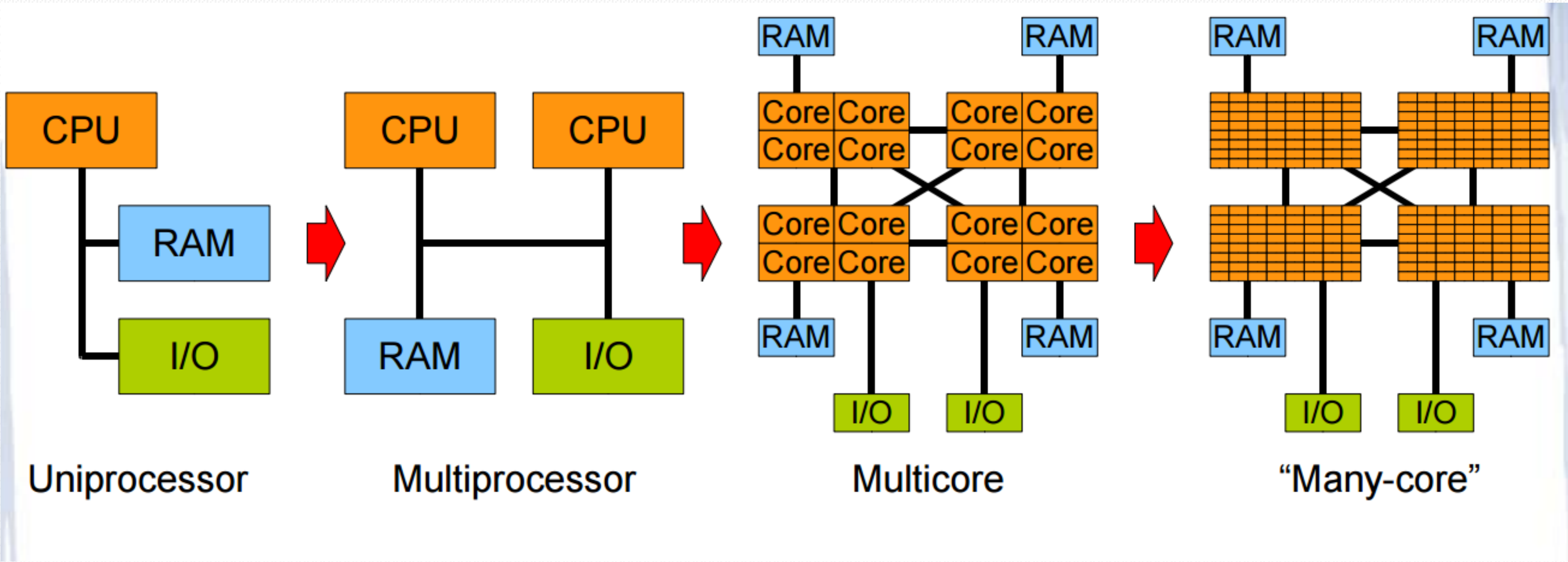


Background


CPU
RAM
I/O
Uniprocessor
CPU
CPU
RAM
I/O
Multiprocessor
RAM
RAM
Core Core Core Core
Core Core Core Core
Core Core Core Core
Core Core Core Core
RAM
RAM
I/O
I/O
Multicore

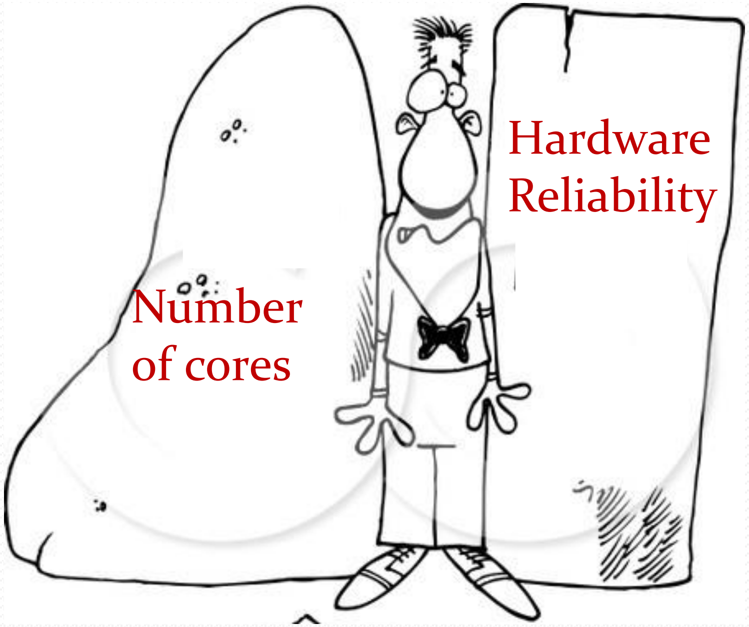
Background

- Many-core is here

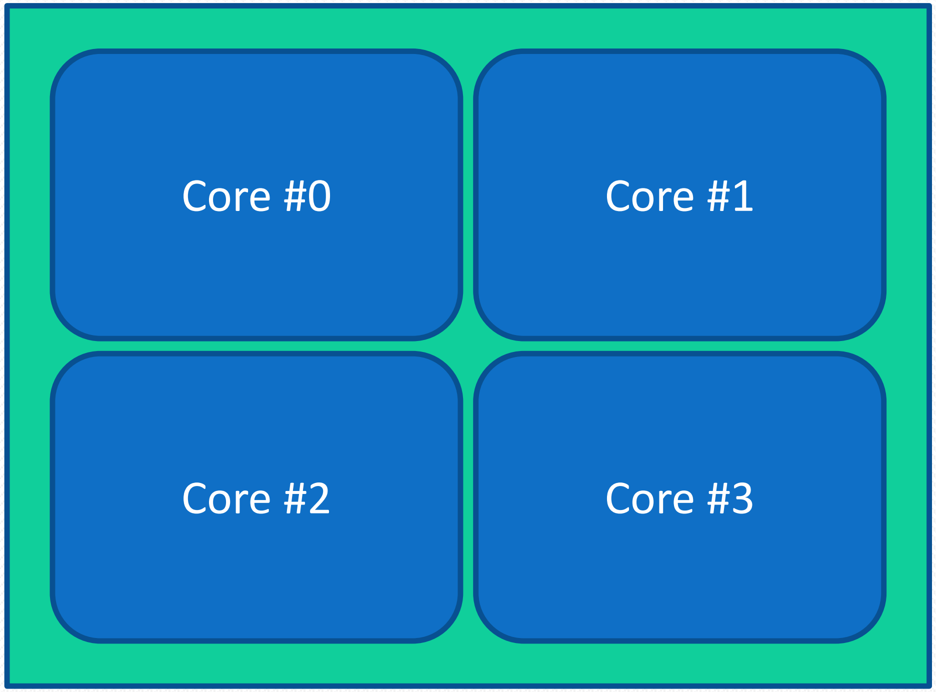


Background

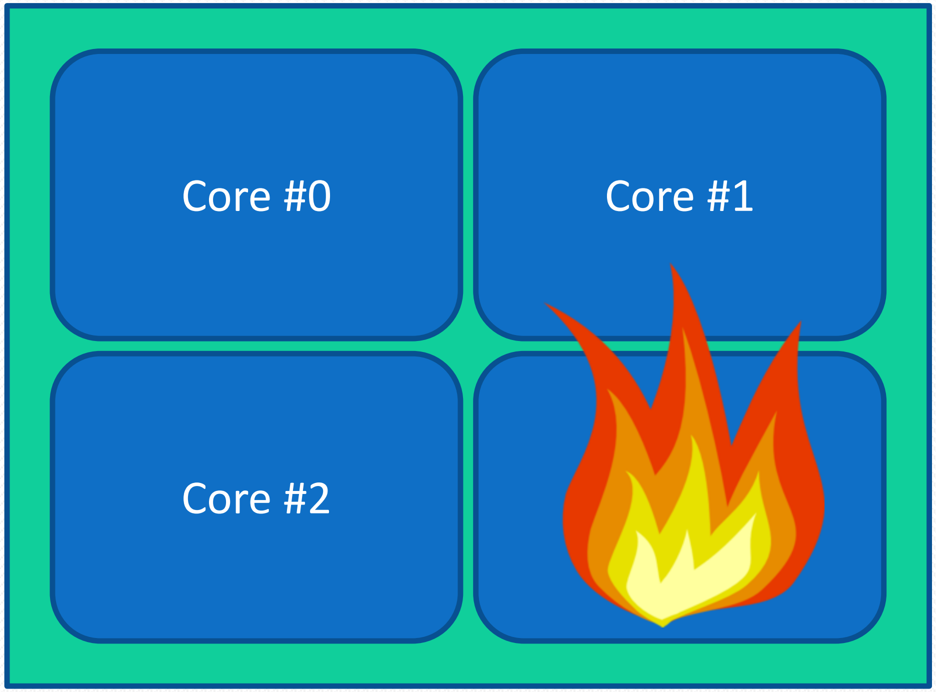
- Mid-quality hardware is favored
- Hardware reliability decreases
- Chances of core permanent hardware fault increase



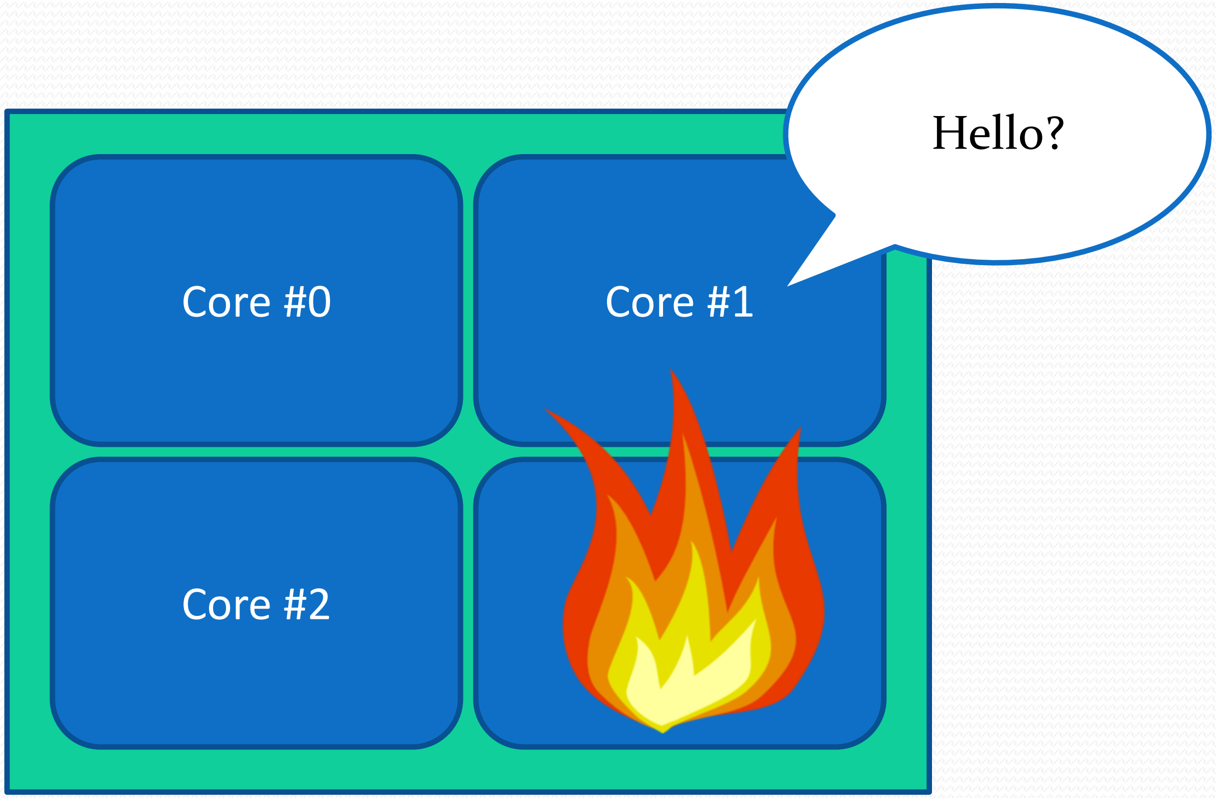
Failure Model



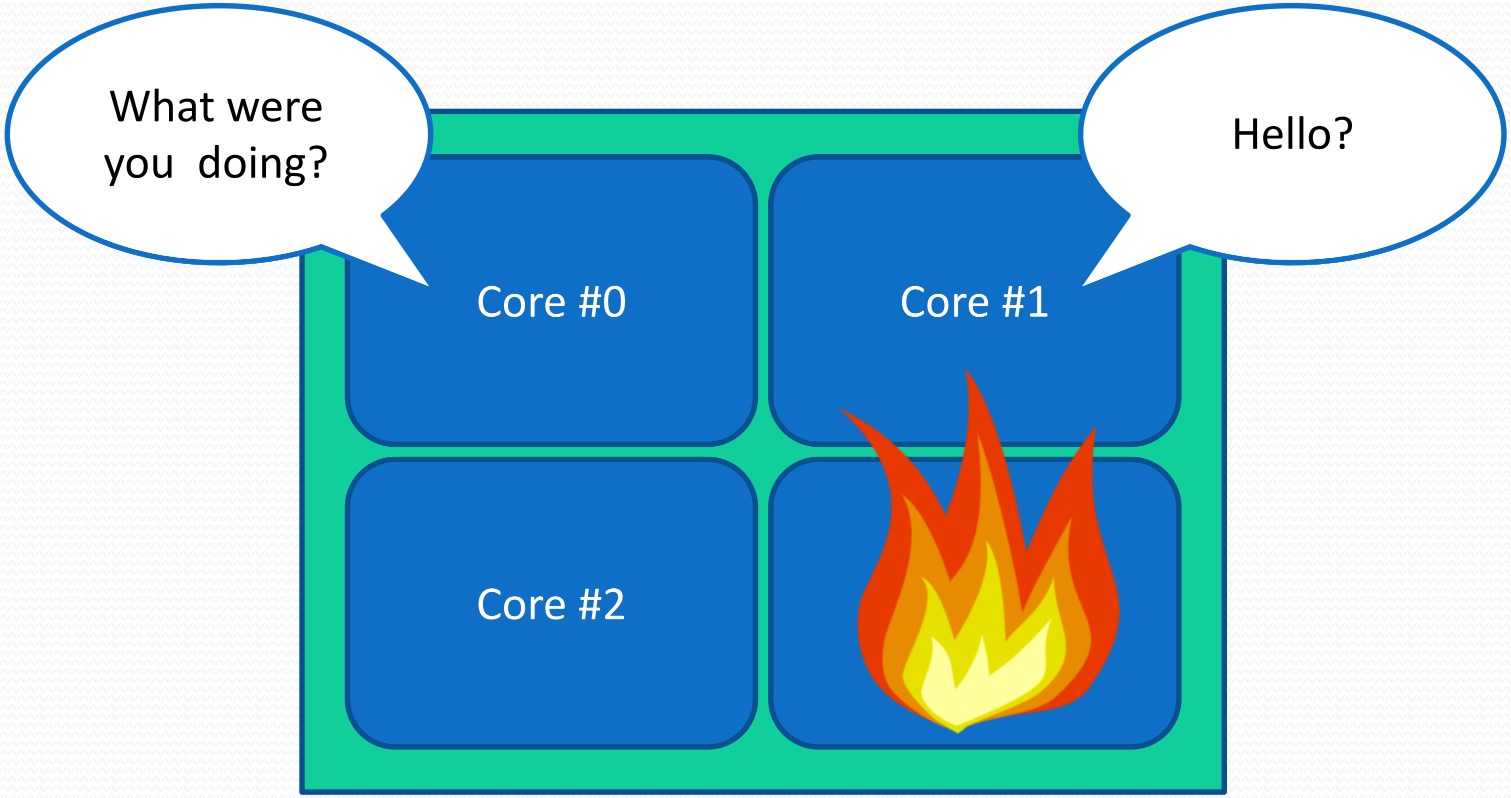
Failure Model



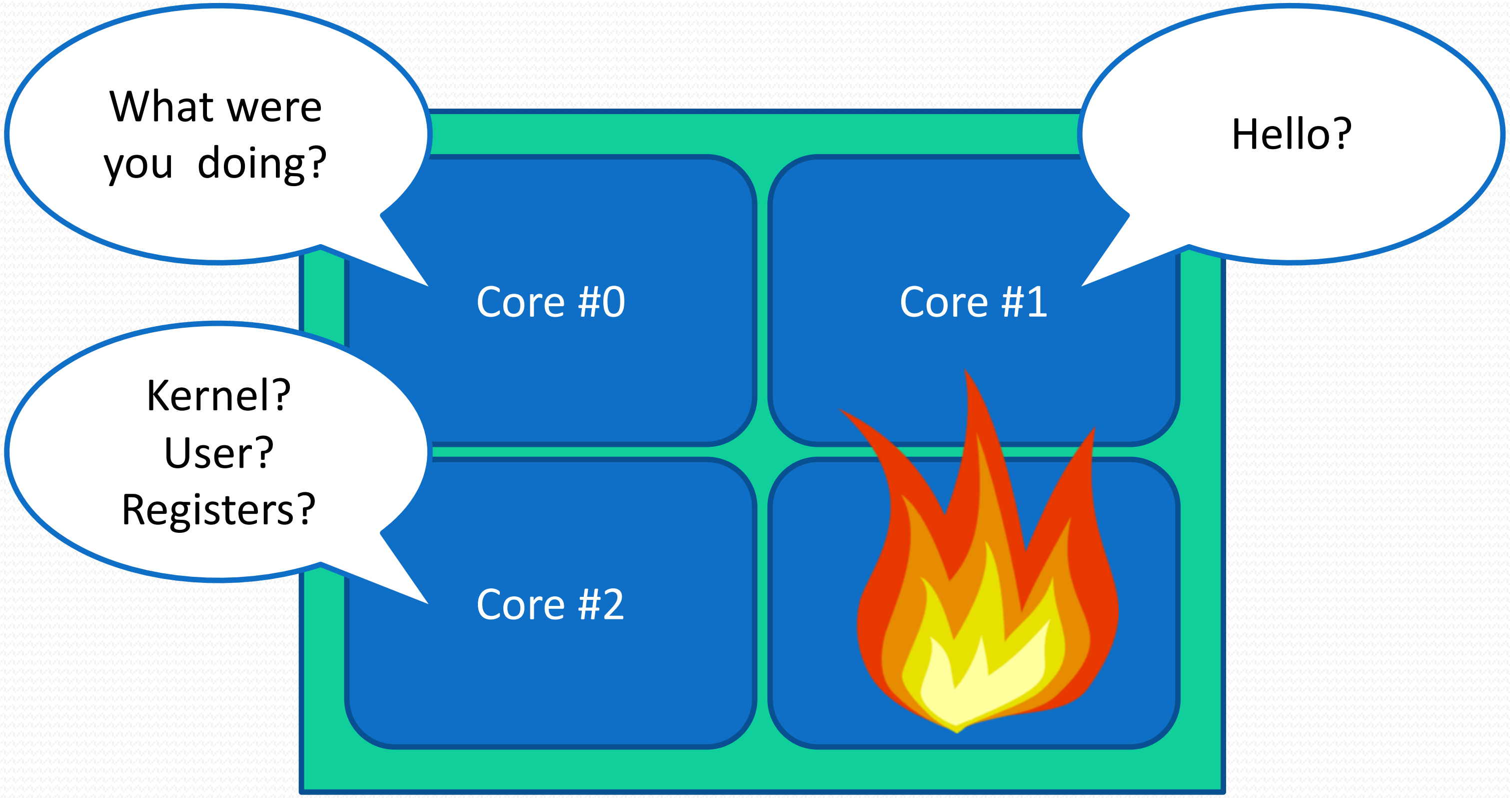
Failure Model



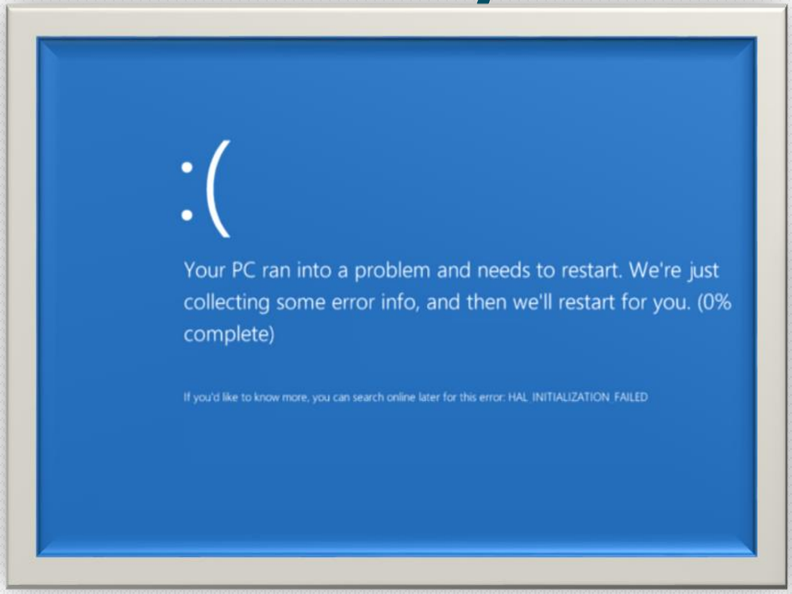
Failure Model



Failure Model

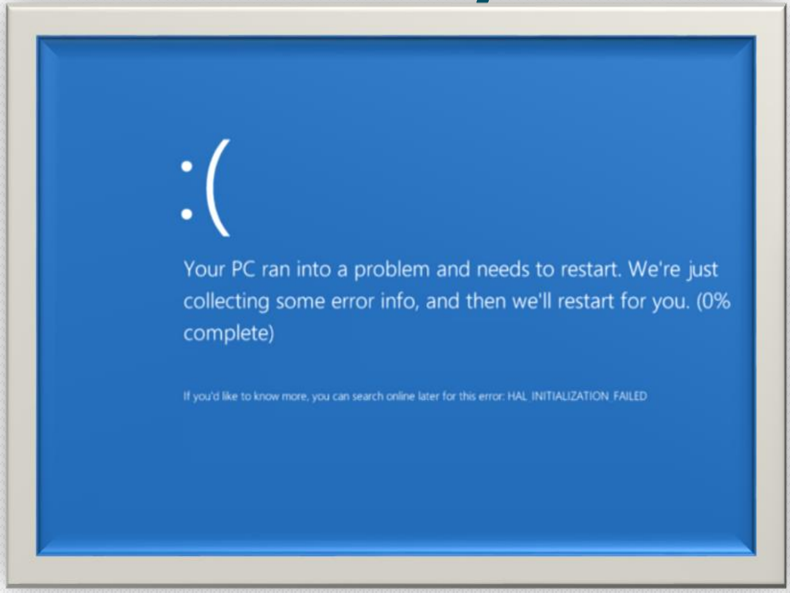


Nowadays


:(
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)
If you'd like to know more, you can search online later for this error: HAL_INITIALIZATION_FAILED


TECHNICAL DIFFICULTIES
PLEASE STAND BY

Nowadays



Do we really have to shut down the CPU completely?

Core Surprise Removal Mechanism (CSR)

- Recovery mechanism for Linux:
  - Faulty core detection
    - Watchdog
  - System is aware of faulty core

Core Surprise Removal Mechanism (CSR)

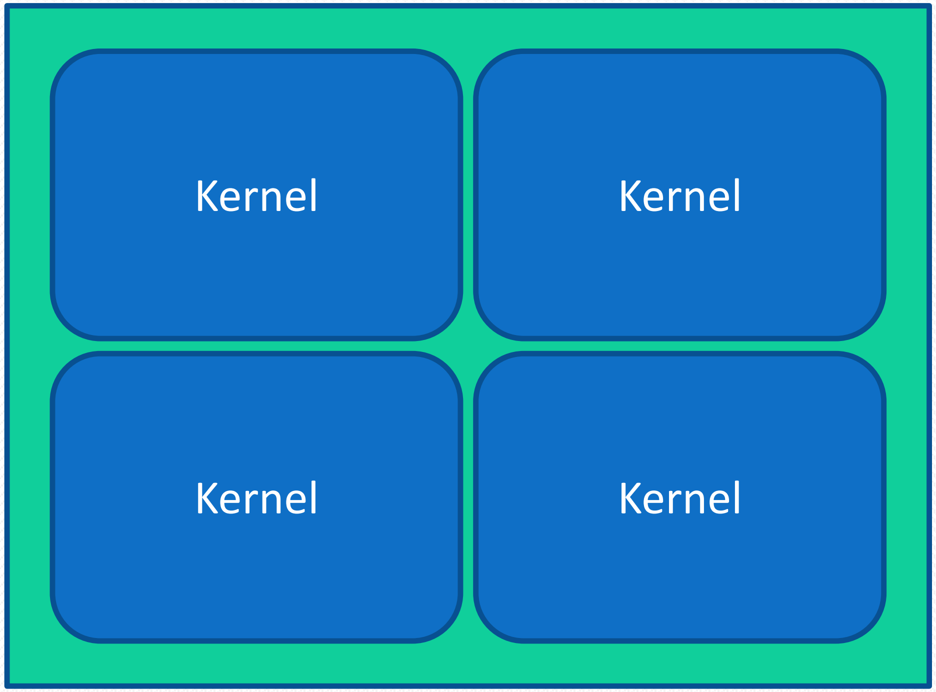
CSR – User Mode

CSR – User Mode

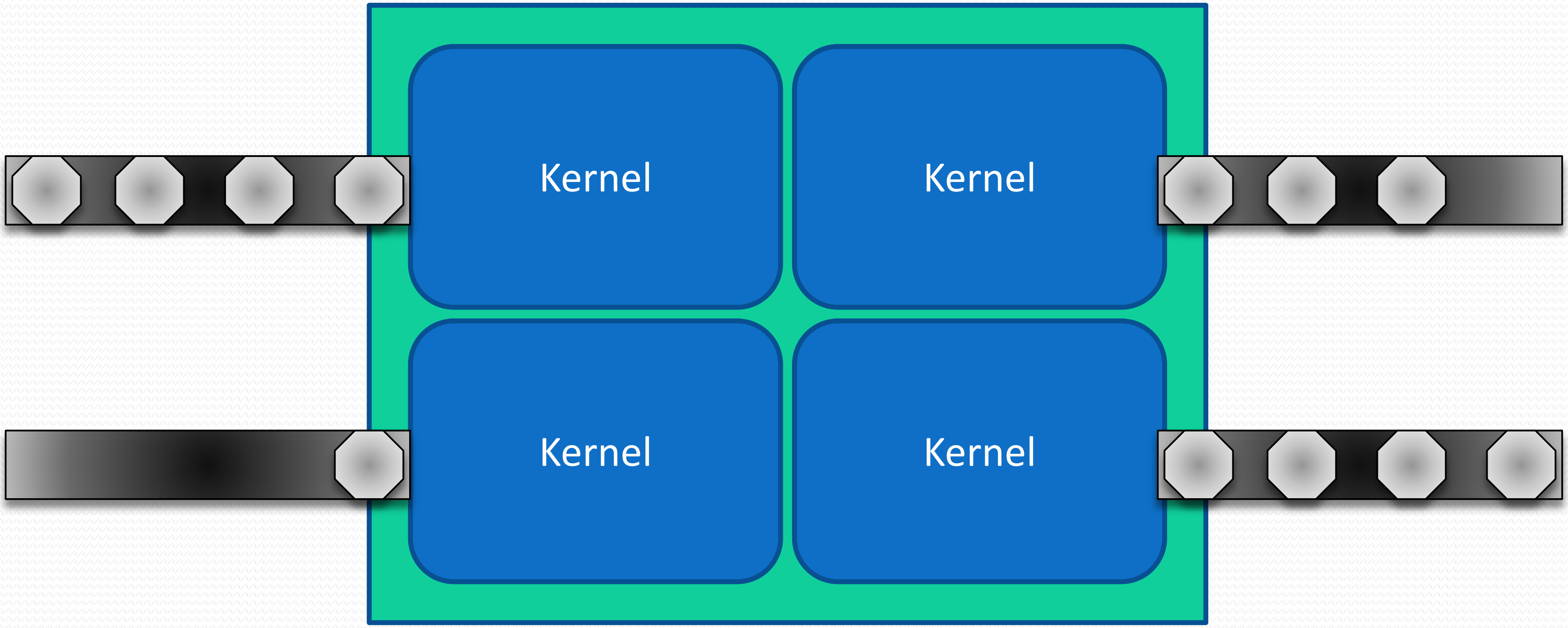
CSR – User Mode

“A fly in the ointment”

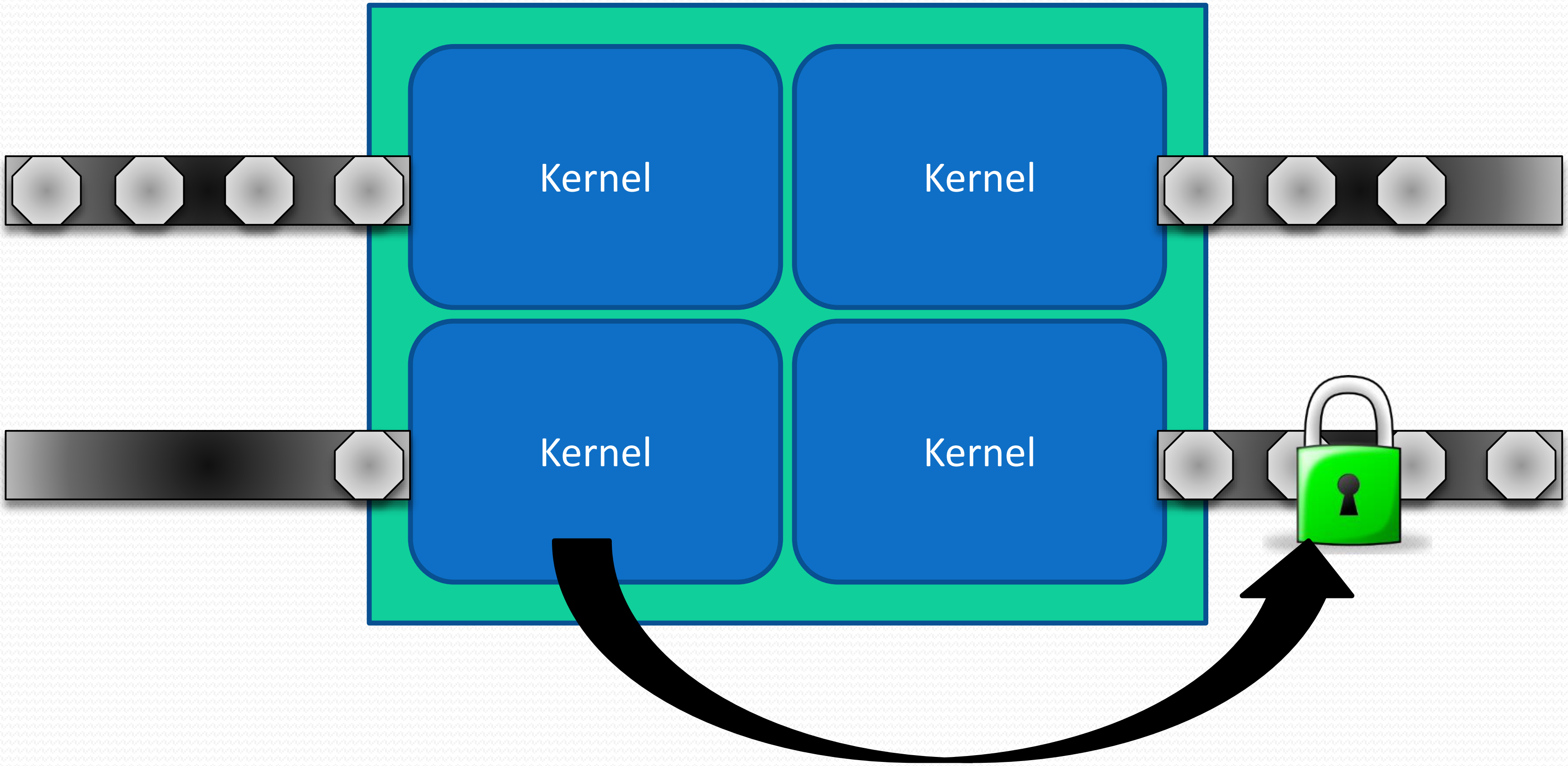
CSR – Kernel (OS) Mode



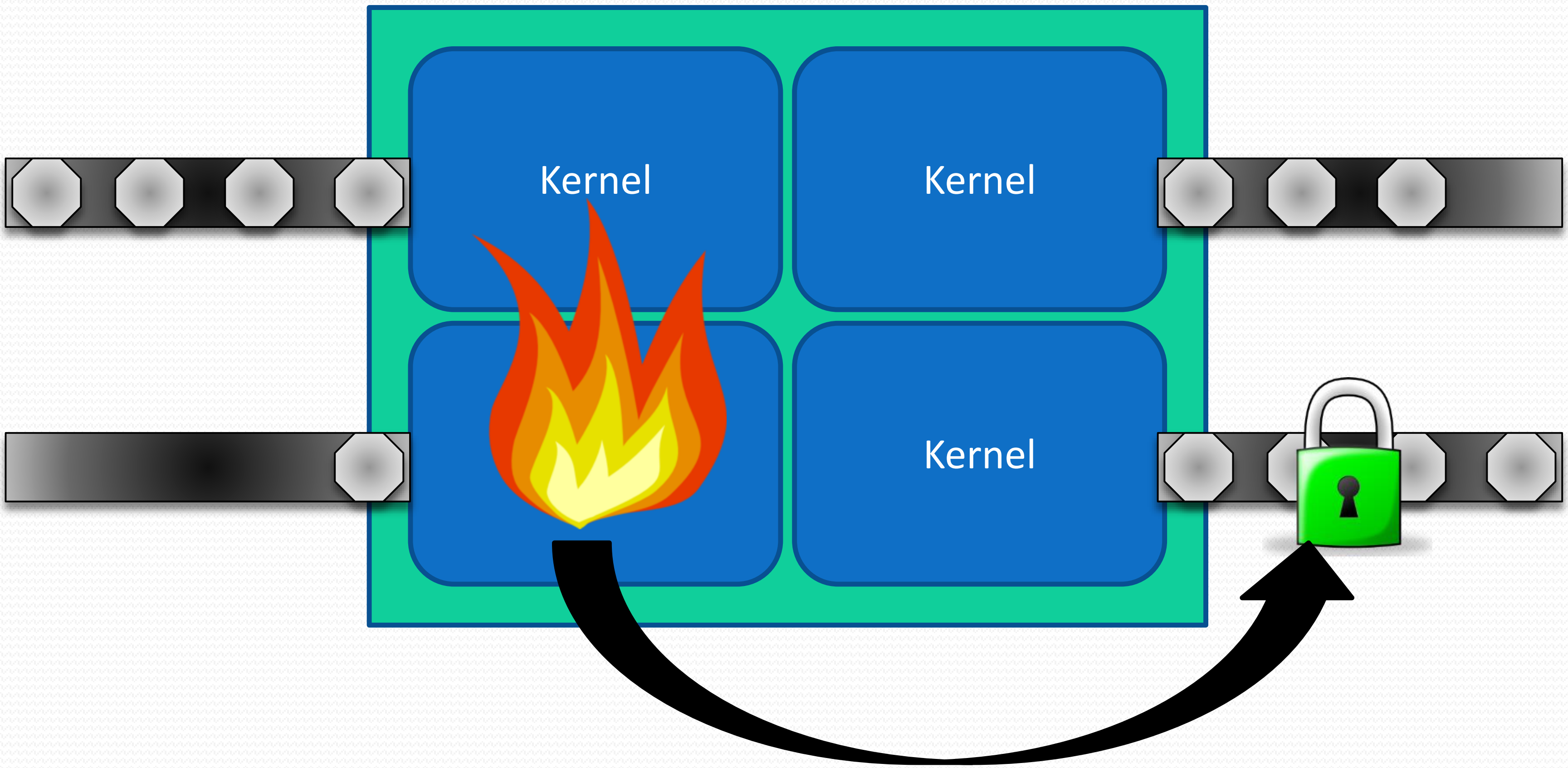
CSR – Kernel (OS) Mode



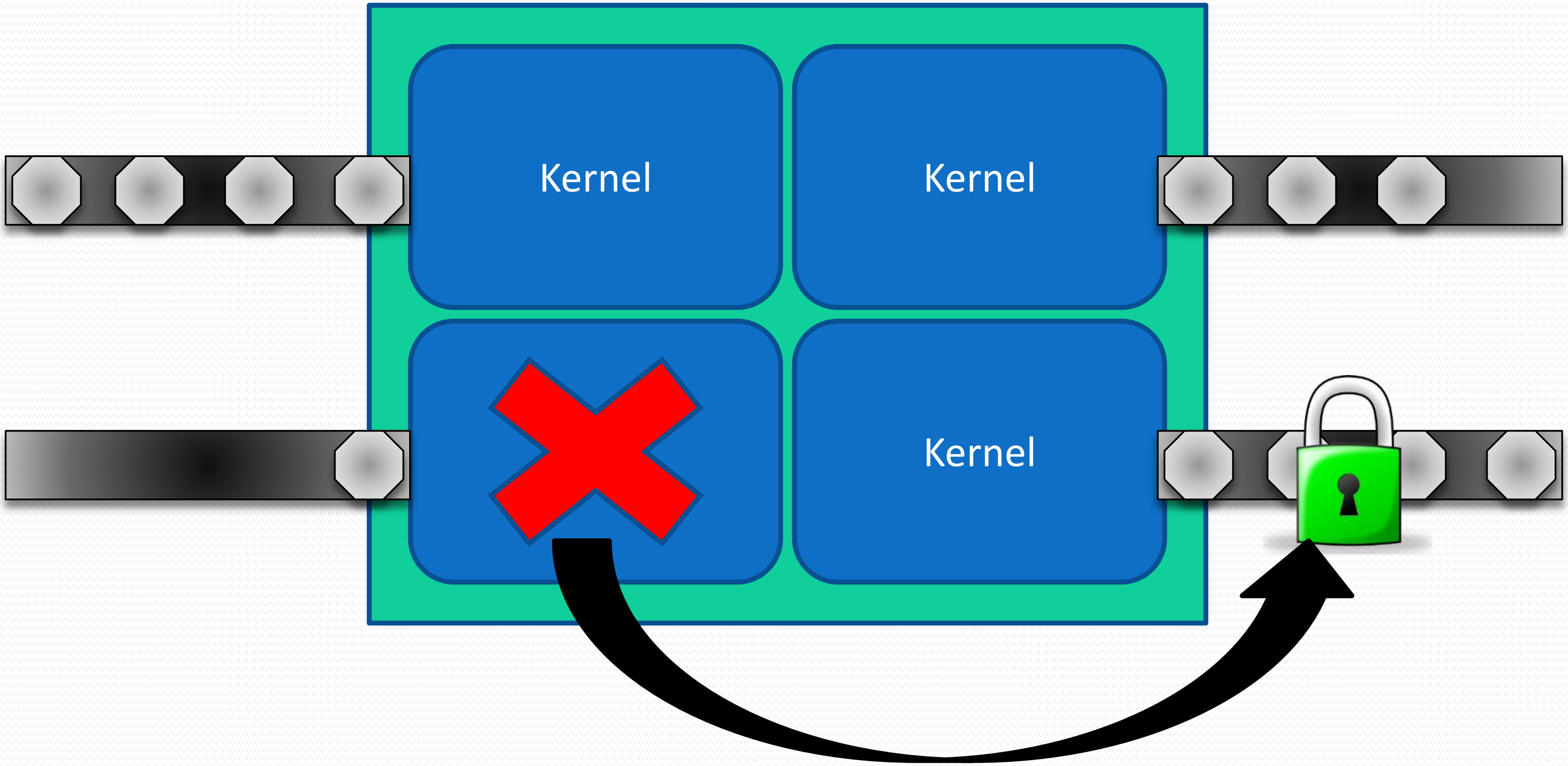
CSR – Kernel (OS) Mode



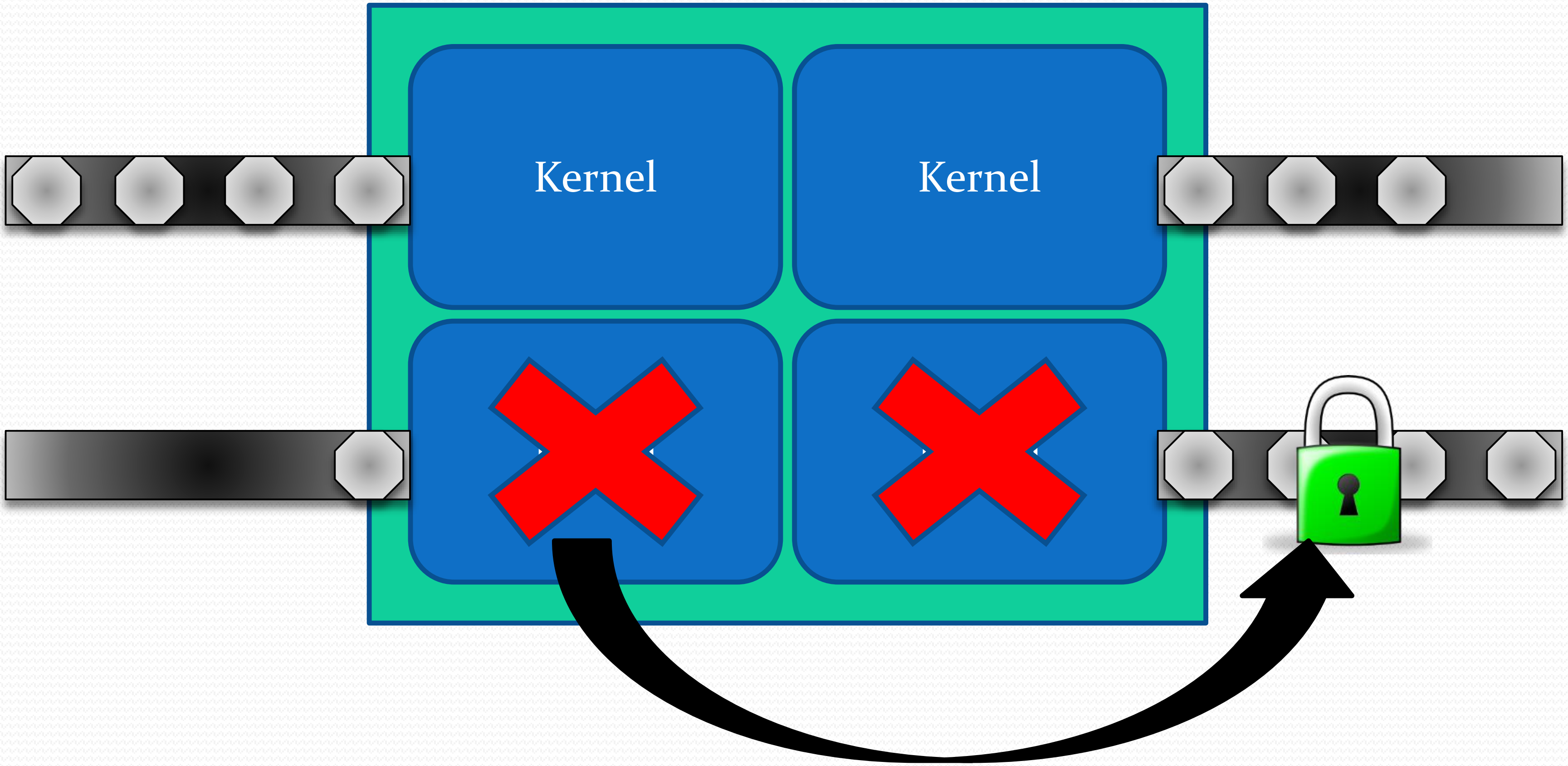
CSR – Kernel (OS) Mode



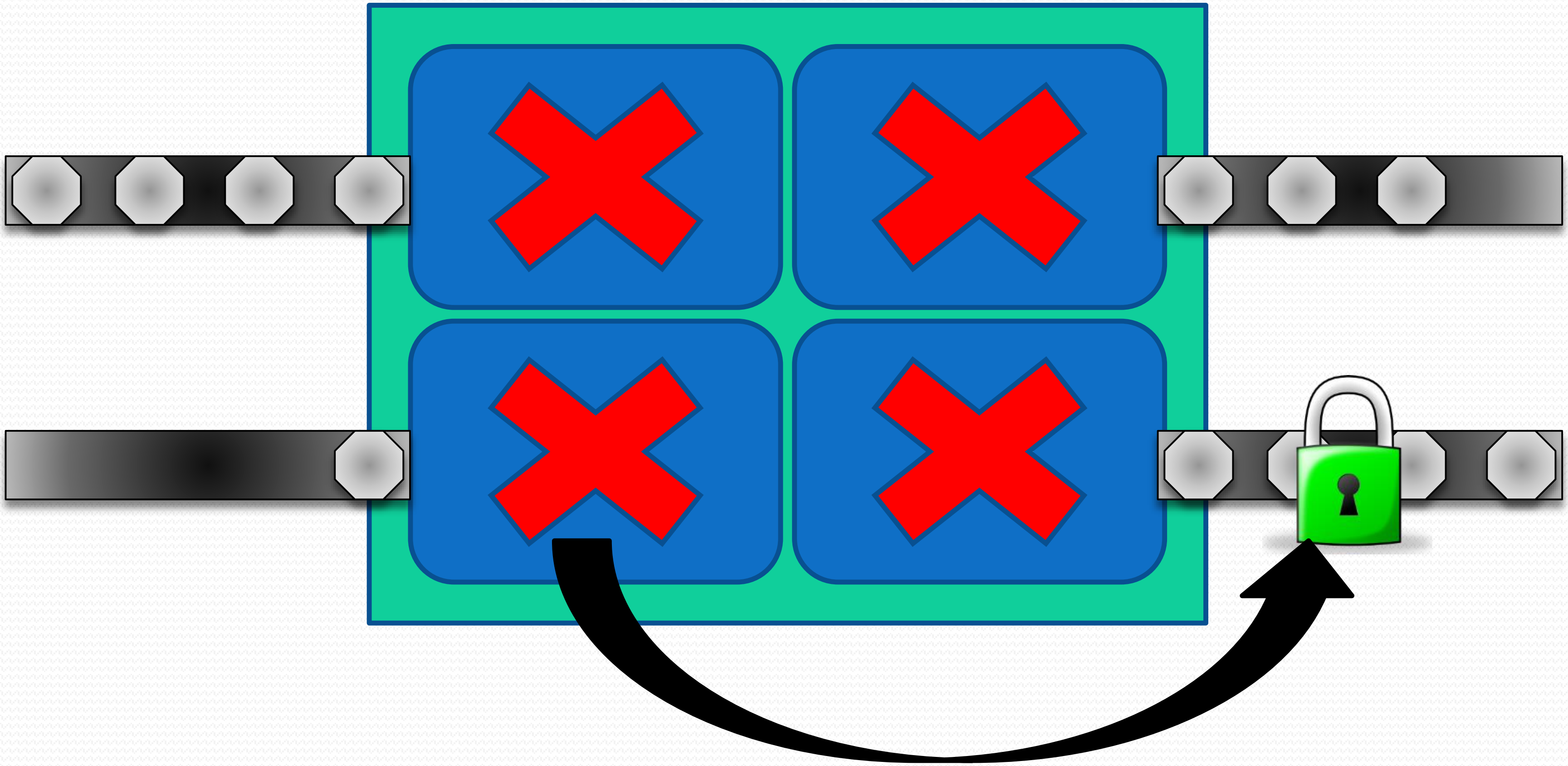
CSR – Kernel (OS) Mode



CSR – Kernel (OS) Mode



CSR – Kernel (OS) Mode



Goal

Tolerate core permanent hardware faults even during **kernel critical sections**

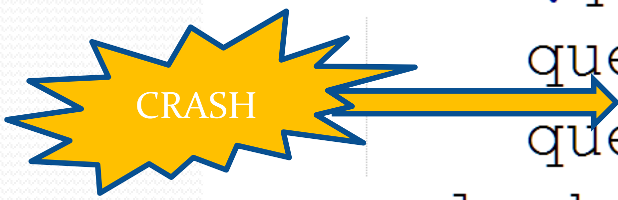
Take 1: Reclaim Locks

- Lock ownership
- System may end up in an intermediate-state
- We cannot tell what part of the critical section was executed

Take 1: Example

```
function queue_pop()
{
    lock(queue_lock)
        queue.elements_counter--
        queue.head = queue.head->next
    unlock(queue_lock)
}
```

Take 1: Example




```
function queue_pop()
{
    lock(queue_lock)
        queue.elements_counter--
        queue.head = queue.head->next
    unlock(queue_lock)
}
```

Take 1: Example

```
function queue_pop()
{
    lock(queue_lock)
        queue.elements_counter
        queue.head = queue.head->next
    unlock(queue_lock)
}
```

CRASH

$elements\_counter \neq Actual\ nu$

Our Solution:

Use Transactions

[Gray J.] The Transaction Concept: Virtues and Limitations. Tandem TR 81.3 , 1981

What is a transaction?

- Sequence of memory operations that either **commits** or **aborts**.
- Upon **commit**, changes appear to have executed **atomically**.

What is a transaction?

- Sequence of memory operations that either **commits** or **aborts**.
- Upon **commit**, changes appear to have executed **atomically**.

```
function queue_pop()
{
    Begin_TX{
        queue.elements_counter--
        queue.head = queue.head->next
    }COMMIT
}
```

TRANSACTIONS [Rajwar et al. 2001]

- More concurrency than locks

R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. DC, USA, 2001. IEEE Computer Society.

HARDWARE TRANSACTIONS [Herlihy et al. 1993]

- Hardware transactional memory – HTM
- Much more time efficient than software

M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. SIGARCH Comput. Archit. News, 21(2):289–300, May 1993.

Hardware Transactions

- Much more time efficient than software
- More concurrency than locks

Hardware Transactions

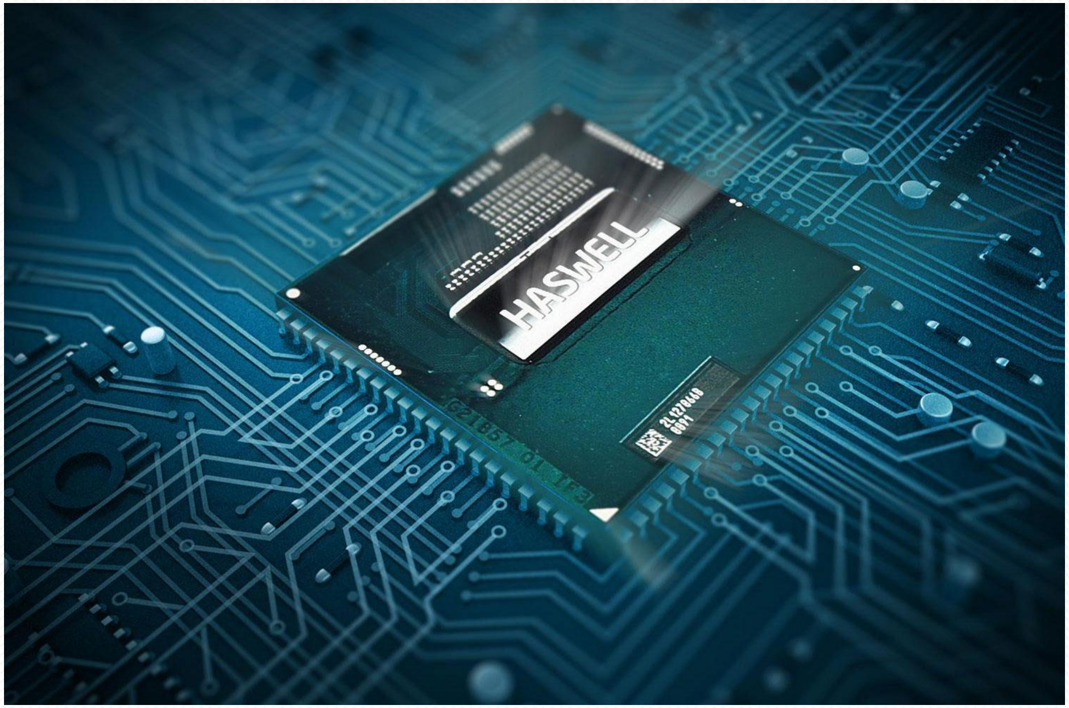
- Much more time efficient than software
- More concurrency than locks

Transactions may be used by the kernel

Intel TSX feature

Transactions may be used by the kernel

- XBEGIN
- XEND
- XTEST
- XABORT



[Intel] Intel R Architecture Instruction Set Extensions Programming Reference, chapter Transactional Synchronization

Replace Kernel Locks

- Update Linux code to use transactions instead of locks
- TxLinux [Rossbach et al. 2007]
  - Simulator
  - Seeking Performance

C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In SOSP, 2007.

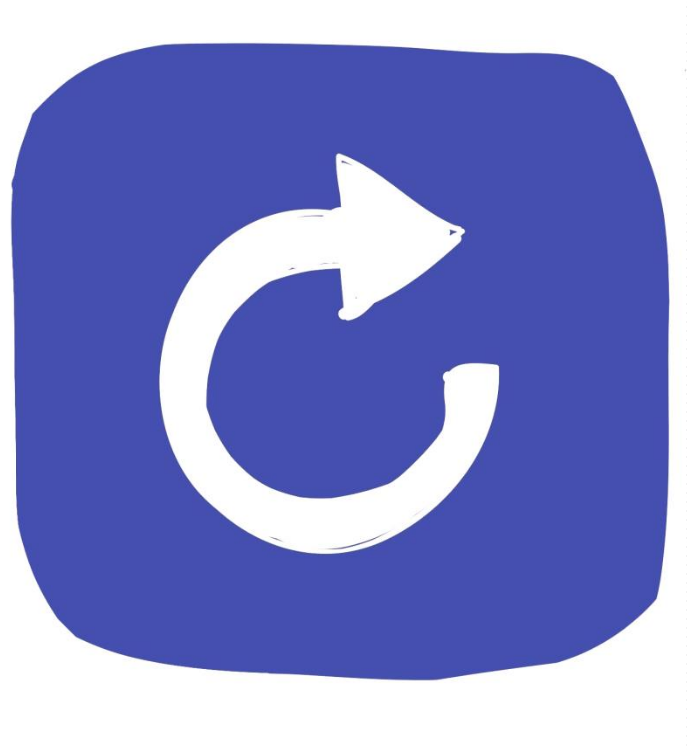
Step 1: Replace Kernel Locks

```
lock(lock_X)
    ...critical section...
        X = X + 2 ;
unlock(lock_X)
```

```
Begin_tx {
   ...critical section...
       X = X + 2 ;
}COMMIT
```

Step 2: Fallback (Abort Handler)

- **Fallback** must be provided to transactions
- Try again, and again, and again…



Step 3: Limited Retries

- We retry, but not forever
- After many retries, resort to locks

Step 3: Limited Retries

- We retry, but not forever
- After many retries, resort to locks
- System boots but runs too slow

Step 3: Limited Retries

- We retry, but not forever
- After many retries, resort to locks
- System boots but runs too slow

- **Only 10%** execute transactionally (**Commit Rate**)

Step 4: Fix Problematic Sections

- I/O operation
- Large sections

Step 4: Fix Problematic Sections

- I/O operation
- Large sections

- **Only 60% commit rate**

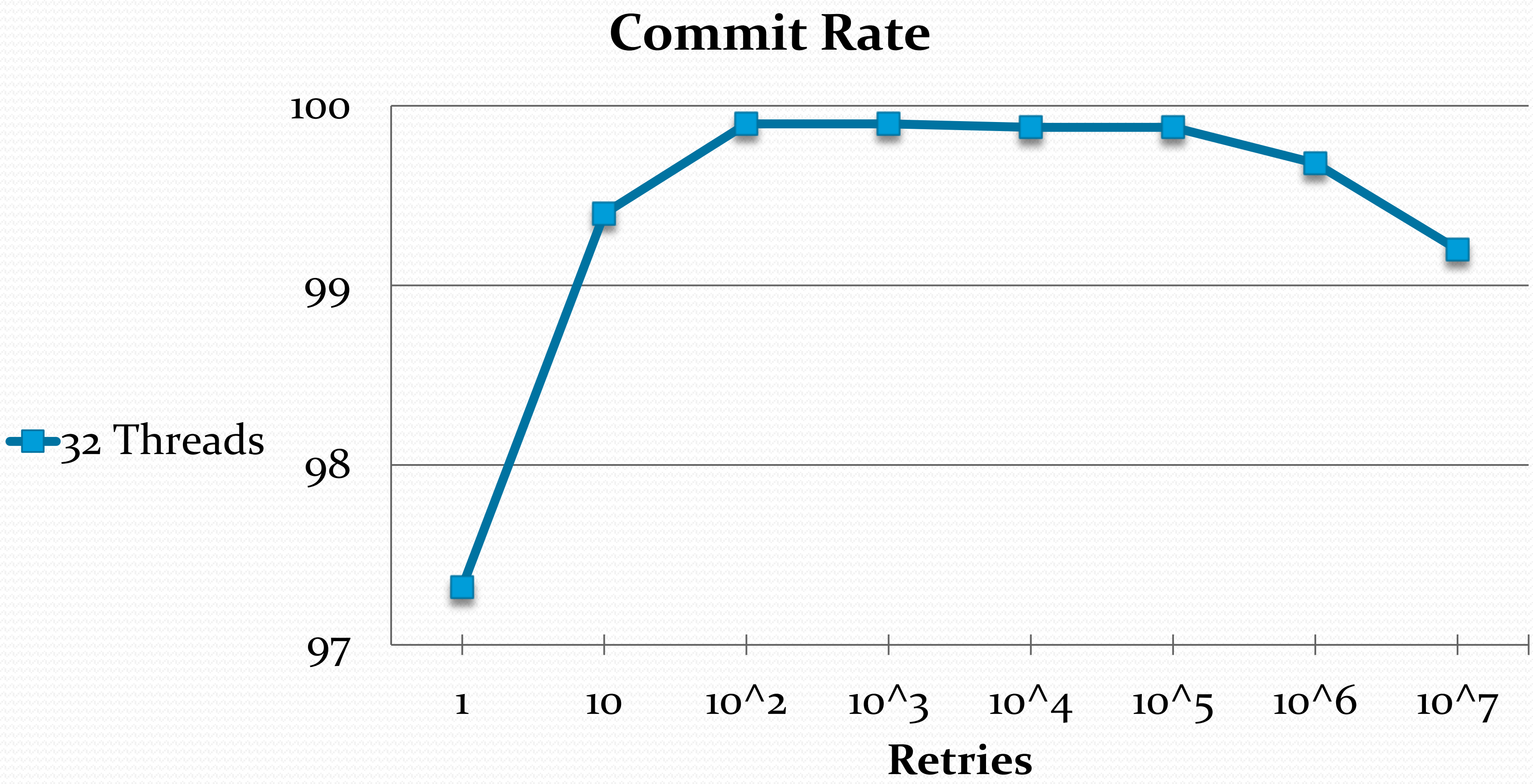
Step 5: Variant Retries

- 10 attempts for problematic section
- **99% commit rate**

Step 5: Variant Retries

- 10 attempts for problematic section
- **99% commit rate**

Step 6: Optimal Retries



Step 6: Optimal Retries

- 100 retries
- **99.9%**

Step 7: Transactions & Locks

- We updated ~50 critical sections
- There are still lock-based sections
- What if a transactional section conflicts with a locked one?

Transactions & Locks - Scenario 1

Time

Thread 1

```
Begin_tx{
   ...critical section #1...
      Write(X) = 0
}  COMMIT
```

Thread 2

```
lock(lock_X)
   ...critical section #2...
      temp = Read(X)
      Write(X) = temp+2
unlock(lock_X)
```

Transactions & Locks - Scenario 1

Time

Thread 1

```
Thread 1



Begin_tx{
      if (lock_X is locked){
         ABORT
      } else {
  ...critical section #1...
          Write(X) = 0
}  COMMIT
```

Thread 2

```
Thread 2
lock(lock_X)
  ...critical section #2...
       temp = Read(X)






       Write(X) = temp+2
unlock(lock_X)
```

Transactions & Locks - Scenario 2

Time ↓

Thread 1

```
Begin_tx{
      if (lock_X is locked){
         ABORT
      } else {
  ...critical section #1...
          Read(X)



          Write(X)
}  COMMIT
```

Thread 2

```


lock(lock_X)
  ...critical section #2...


     Write(X)
unlock(lock_X)
```

Transactions & Locks - Scenario 2

Time

Thread 1

```
Begin_tx{
      if (lock_X is locked){
         ABORT
      } else {
  ...critical section #1...
          Read(X)



          Write(X)
}  COMMIT
```

Thread 2

Abort Tx

```
lock(lock_X)
  ...critical section #2...


     Write(X)
unlock(lock_X)
```

Code Example

- Added ~500 lines of code

Code Example

- Added ~800 LOC

```
void scheduler_tick(void)
{
    spin_lock(&rq->lock);

        // do_something...

    spin_unlock(&rq->lock);
}
```

Code Example

- Added ~800 lines of code

Invoked every 4ms

```
void scheduler_tick(void)
{
    spin_lock(&rq->lock);

        // do_something...

    spin_unlock(&rq->lock);
}
```

Code Example

```
void scheduler_tick(void)
{

TX_LABEL:

    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction













}
```

Code Example

```
void scheduler_tick(void)
{

TX_LABEL:

    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction

        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }













}
```

Code Example

```
void scheduler_tick(void)
{

TX_LABEL:

    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction

        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        // do_something...









}
```

Code Example

```
void scheduler_tick(void)
{

TX_LABEL:

    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction

        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        // do_something...

        _xend(); // commit

}
```

Code Example

```
void scheduler_tick(void)
{

TX_LABEL:

    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction

        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        // do_something...

        _xend(); // commit


    } else { // fallback on abort

        if( ++retries < MAX_RETRIES ){
            goto TX_LABEL; // retry
        }








}
```

Code Example

```
void scheduler_tick(void)
{

TX_LABEL:

    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction

        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        // do_something...

        _xend(); // commit


    } else { // fallback on abort

        if( ++retries < MAX_RETRIES ){
            goto TX_LABEL; // retry
        }

        spin_lock(&rq->lock);

        // do_something...

        spin_unlock(&rq->lock);
    }

}
```

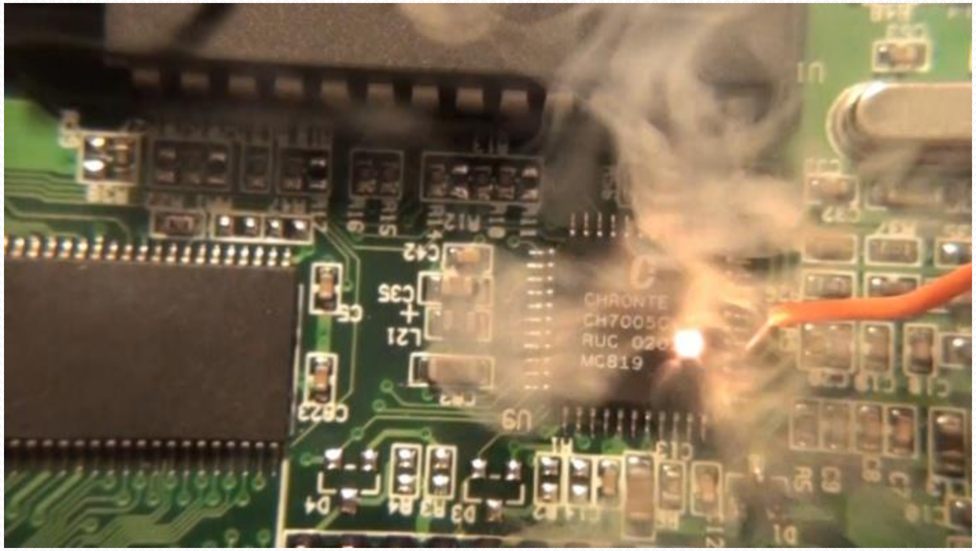
Original code

CSR & HTM

- We created a "bulletproof" Linux
  - CSR - Core surprise removal mechanism
  - HTM – Hardware Transactional Memory

Simulate Core Hardware Fault

- Evaluate our enhanced OS
- Fail a core during critical section



```
void scheduler_tick(void)
{

TX_LABEL:
    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction
        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }







        // do_something...

        _xend(); // commit


    } else { // fallback on abort


        if ( ++retries < MAX_RETRIES )
            goto TX_LABEL; // retry

        spin_lock(&rq->lock);
        // do_something...
        spin_unlock(&rq->lock);
    }

}
```

Failed core is:

1. Unresponsive
2. Not changing anything

```c
void scheduler_tick(void)
{

TX_LABEL:
    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction
        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        if (processor_id() == cpu_to_hang) { //Flag is set by a system call
            block_interrupts();

        }
        // do_something...

        _xend(); // commit


    } else { // fallback on abort


        if ( ++retries < MAX_RETRIES )
            goto TX_LABEL; // retry

        spin_lock(&rq->lock);
        // do_something...
        spin_unlock(&rq->lock);
    }

}
```

Failed core is:

1. Unresponsive
2. Not changing anything

```c
void scheduler_tick(void)
{

TX_LABEL:
    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction
        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        if (processor_id() == cpu_to_hang) { //Flag is set by a system call
            block_interrupts();
            while (true);
        }

        // do_something...

        _xend(); // commit


    } else { // fallback on abort


        if ( ++retries < MAX_RETRIES )
            goto TX_LABEL; // retry

        spin_lock(&rq->lock);
        // do_something...
        spin_unlock(&rq->lock);
    }

}
```

Failed core is:

1. Unresponsive
2. Not changing anything

```
void scheduler_tick(void)
{

TX_LABEL:
    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction
        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        if (processor_id() == cpu_to_hang) { //Flag is set by a system call
            block_interrupts();
            while (true);
        }

        // do_something...

        _xend(); // commit


    } else { // fallback on abort


        if ( ++retries < MAX_RETRIES )
            goto TX_LABEL; // retry

        spin_lock(&rq->lock);
        // do_something...
        spin_unlock(&rq->lock);
    }

}
```

Failed core is:

1. Unresponsive
2. Not changing anything

```
void scheduler_tick(void)
{

TX_LABEL:
    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction
        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        if (processor_id() == cpu_to_hang) { //Flag is set by a system call
            block_interrupts();
            while (true);
        }

        // do_something...

        _xend(); // commit


    } else { // fallback on abort
        if (processor_id() == cpu_to_hang)
            goto TX_LABEL; // retry
        if ( ++retries < MAX_RETRIES )
            goto TX_LABEL; // retry

        spin_lock(&rq->lock);
        // do_something...
        spin_unlock(&rq->lock);
    }

}
```

Failed core is:

1. Unresponsive
2. Not changing anything

```
void scheduler_tick(void)
{

TX_LABEL:
    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction
        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        if (processor_id() == cpu_to_hang) { //Flag is set by a system call
            block_interrupts();
            while (true);
        }
        // do_something...

        _xend(); // commit


    } else { // fallback on abort
        if (processor_id() == cpu_to_hang)
            goto TX_LABEL; // retry
        if ( ++retries < MAX_RETRIES )
            goto TX_LABEL; // retry

        spin_lock(&rq->lock);
        // do_something...
        spin_unlock(&rq->lock);
    }

}
```

~~block_interrupts();~~ Cancelled upon transaction time out

Failed core is:

1. Unresponsive
2. Not changing anything

```c
void scheduler_tick(void)
{
    if ( processor_id() == cpu_to_hang) //Flag is set by a system call
        block_interrupts();

TX_LABEL:
    if ( _xbegin() == _XBEGIN_STARTED) { // start the transaction
        if ( raw_spin_is_locked(&rq->lock) ) {
            // necessary for mutual exclusion
            _xabort(1);
        }

        if (processor_id() == cpu_to_hang) //Flag is set by a system call
            while (true);

        // do_something...

        _xend(); // commit


    } else { // fallback on abort
        if (processor_id() == cpu_to_hang)
            goto TX_LABEL; // retry
        if ( ++retries < MAX_RETRIES )
            goto TX_LABEL; // retry

        spin_lock(&rq->lock);
        // do_something...
        spin_unlock(&rq->lock);
    }

}
```

Failed core is:

1. Unresponsive
2. Not changing anything

Results

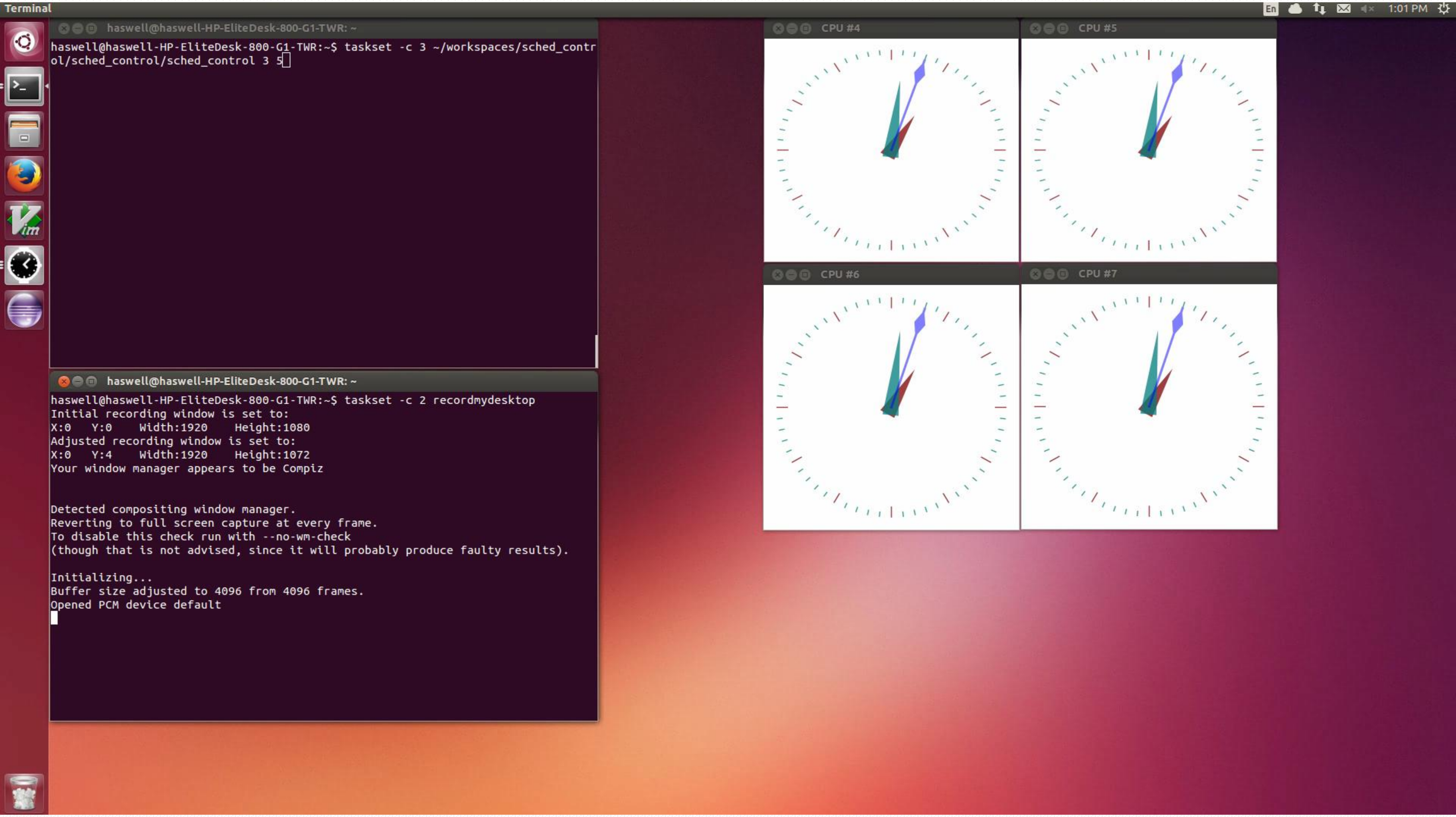
- We got a fault-tolerant OS
  - System survives single failure as well as cascading failures
- Performance gain
- Power consumption reduced

Results

- We got a fault-tolerant OS
  - System survives single failure as well as cascading failures
- Performance gain
- Power consumption reduced

Workload	Commit Rate	Performance Gain	Energy Saving
Idle	100%	-	4%
16-threads	99.9%	0%	1%
32-threads	99.9%	3%	3%
64-threads	99.8%	4%	2%

Demo


Terminal
1:01 PM
haswell@haswell-HP-EliteDesk-800-G1-TWR: ~
haswell@haswell-HP-EliteDesk-800-G1-TWR:~$ taskset -c 3 ~/workspaces/sched_control/sched_control/sched_control 3 5
haswell@haswell-HP-EliteDesk-800-G1-TWR: ~
haswell@haswell-HP-EliteDesk-800-G1-TWR:~$ taskset -c 2 recordmydesktop
Initial recording window is set to:
X:0 Y:0 Width:1920 Height:1080
Adjusted recording window is set to:
X:0 Y:4 Width:1920 Height:1072
Your window manager appears to be Compiz
Detected compositing window manager.
Reverting to full screen capture at every frame.
To disable this check run with --no-wm-check
(though that is not advised, since it will probably produce faulty results).
Initializing...
Buffer size adjusted to 4096 from 4096 frames.
Opened PCM device default
CPU #4
CPU #5
CPU #6
CPU #7

References

[Gray J.] The Transaction Concept: Virtues and Limitations. Tandem TR 81.3 , 1981

R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. DC, USA, 2001. IEEE Computer Society.

M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. SIGARCH Comput. Archit. News, 21(2):289–300, May 1993.

[Intel] Intel R Architecture Instruction Set Extensions Programming Reference, chapter Transactional Synchronization

C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In SOSP, 2007.

Thank you!