Corey: An Operating System for Many Cores

Silas Boyd-Wickizer (MIT),
Haibo Chen, Rong Chen, Yandong Mao (Fudan University),
Frans Kaashoek, Robert Morris, Aleksey Pesterev (MIT),
Lex Stein, Ming Wu (Microsoft Research Asia),
Yuehua Dai (Xi'an Jiaotong University), Yang Zhang (MIT),
Zheng Zhang (Microsoft Research Asia)
What this talk is about

- **New OS interfaces** that help applications scale with the number of cores.
- Target applications: Web servers, MapReduce, mail servers, ...
Many applications spend time in the kernel

- Serving static web pages
  - Directory lookups and TCP processing
- Even applications implemented with multicore MapReduce spend time in the kernel
  - 30% of execution time spent growing address space on 16 cores
- Fraction of time in OS increases with the number of cores
  - OS becomes a bottleneck
The bottleneck is shared OS data structures

• Contention on shared data structures is costly:
  – serialization
  – moving data between caches

• Why does the OS need shared data structures?
  – OS semantics requires it
  – Simplifies resource management
Current practice for scaling the OS

- Redesign and reimplement kernel subsystems
  - Fine grained locking, RCU, etc.
- Lots of work: continuous redesign to increase concurrency
  - Linux changes: page cache, scheduler, RCU, memory management, ...
- Existing interfaces constrain designers
  - Even a small amount of shared kernel data limits performance with many cores
Our solution: change OS interface

• Applications don't always need to share all the data structures that existing interfaces share
• Allow applications to control how cores share kernel data structures
  – Avoid contention over kernel data structures
• We propose three interface changes
  – shares, address ranges, kernel cores
• Implemented in Corey OS
  – Partially implemented in Linux
New OS interfaces

- **Shares** control the kernel data used to resolve application references.
- **Address ranges** control page tables and the kernel data used to manage them.
- **Kernel cores** allow applications to dedicate cores to running particular kernel functions.
- Improve scalability of some applications by avoiding kernel bottlenecks
Idea #1: Shares
Object naming in an OS

- Kernel must map an application-visible reference into address of kernel object
  - Typically via per-process or global tables
  - Cores contend for shared data
Motivating example: file descriptors

- Shared kernel data structure: file descriptor table
- Measure the cost of using FD table
  - Threads dup-and-close a per-thread FD
  - 16 core AMD Opteron running a Linux 2.6.27
Ideal FD performance graph

- Expect throughput to scale linearly
Actual FD performance

- Notice two things:
Actual FD performance

- Notice two things:
  - Drop in throughput.
  - No improvement in throughput.
Actual FD performance

• Notice two things:
  – Drop in throughput.
  – No improvement in throughput.
Why throughput drops?

- Load fd_table data from L1 in 3 cycles.
Why throughput drops?

- Load fd_table data from L1 in 3 cycles.
- Now it takes 121 cycles!

```c
fd_alloc(void) {
    lock(fd_table);
    fd = get_free_fd();
    set_fd_used(fd);
    fix_smallest_fd();
    unlock(fd_table);
}
```
Why no improvement?

- Shared FD table is a bottleneck
  - A lock serializes updates to fd_table

```c
fd_alloc(void) {
  lock(fd_table);
  fd = get_free_fd();
  set_fd_used(fd);
  fix_smallest_fd();
  unlock(fd_table);
}
```
Can the performance be better?

- For some applications the OS shares kernel data structures unnecessarily
  - Should be able to improve performance

- Challenge: how should the OS figure out when to share and when not to?
  - More difficult is application has a mixture
Our solution: shares

- **Shares** allow applications to control how cores share the kernel data structures used to do lookups

- Applications specify when they need sharing, for example:
  - shared FDs allocated in shared table
  - private FDs allocated in private table

- Corey kernel uses shares for all lookup tables
Adding shares to Linux

- With minimal changes can add a share-like interface for FDs.
- FD system calls (sys_open, sys_dup, ...) take an optional shareid/fdtableid argument.
Linux FD share example
Linux FD share example

```c
fd2 = open("goo");
```
Linux FD share example

```c
fd2 = open("goo");
write(fd2, buf, 128);
```
Linux FD share example

```c
fd2 = open("goo");
fdtable1 = share_alloc();
write(fd2, buf, 128);
```
Linux FD share example

```
fd2 = open("goo");
fdtable1 = share_alloc();
fd0 = open("foo", share1);
write(fd2, buf, 128);
```
Linux FD share example

```c
fd2 = open("goo");
fdtable1 = share_alloc();
fd0 = open("foo", share1);
write(fd0, buf, 128, share1);
write(fd2, buf, 128);
```
Linux FD share example

- Cores manipulate FDs without contending for kernel data structures
Performance is now ideal

- Avoid contention on shared FD table:
  - No drop in throughput (no L1 misses)
  - Scales linearly (no serialization)
Benefit of shares

• Able to avoid unnecessary contention on kernel data structures
  – For example when application threads do not share FDs

• Applications can control how cores share internal kernel data structures

• Few kernel and application modifications to get scalability
Idea #2: Address ranges
Two options for multiprocessor shared-memory application

- Shared address space
  - Implemented with multiple threads

- Private address spaces
  - Implemented with multiple processes
  - Share memory with mmap(MAP_SHARED)
Cost of two options

Kernel data structure for managing address spaces.

shared address space

private address spaces

core 0  core 1

mm_struct

stack 0  array  stack 1

mm_struct

core 0  core 1

mm_struct

stack 0  array  stack 1
Cost of two options

- Contend on mm_struct: locks, counters, lists...

Contend on mm_struct, even for private mappings...
Cost of two options

...for example when both cores go to grow their stacks.

• Contend on mm_struct: locks, counters, lists...
Cost of two options

- Contend on `mm_struct`: locks, counters, lists...
- No contention on `mm_struct`
More costs: soft page faults

- Linux lazily instantiates page tables

- Contend on mm_struct: locks, counters, lists...

- No contention on mm_struct
More costs: soft page faults

- Linux lazily instantiates page tables
- Content on mm_struct: locks, counters, lists...
- No contention on mm_struct
More costs: soft page faults

- Contend on mm_struct: locks, counters, lists...
- No contention on mm_struct

When an application allocates memory the OS doesn't actually fill in the PTEs.
More costs: soft page faults

- The first time a page is touched, the core will signal a memory fault...
- Contend on mm_struct: locks, counters, lists...
- No contention on mm_struct

Linux lazily instantiates page tables
More costs: soft page faults

- Contend on mm_struct: locks, counters, lists...
- One soft page fault per page
- No contention on mm_struct

...and the OS allocates a physical page and adds and adds a PTE to the pgtable.

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Ptable</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mm_struct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack 0</td>
<td>array</td>
<td>stack 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Ptable</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ptable</td>
<td></td>
</tr>
<tr>
<td>mm_struct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack 0</td>
<td>array</td>
<td>stack 1</td>
</tr>
</tbody>
</table>
More costs: soft page faults

- Linux lazily instantiates page tables:
  - Each mm_struct has a different pgtable, so each core soft page faults on each page.
  - Contend on mm_struct: locks, counters, lists...
- One page fault per page

- No contention on mm_struct
- Each core soft page faults on each page
The problem

- Neither option accurately represents how the application is using kernel data structures:
  - shared address spaces – the mm_struct is global
    - contention
    - unnecessary for private memory
  - private address spaces – the mm_struct is private
    - extra soft page faults, because no PTE sharing
Our solution: address ranges

- **Address ranges** provide benefits of both shared and private address spaces:
  - avoid contention for private memory
  - share PTEs for shared memory
Address ranges: avoid contention

Cores have a private "root" address range.

Diagram:
- Core 0
- ar_struct
- Stack 0
Address ranges: avoid contention

- core 0
- ar_struct
- stack 0

that maps private memory
Address ranges: avoid contention

Start another core with a private root address range.
Address ranges: avoid contention

A core allocates another address range
Address ranges: avoid contention

and the other core maps it into its root address range
Address ranges: avoid contention

cores map shared memory into the address range
Address ranges: avoid contention

No contention when manipulating root address ranges
Address ranges: avoid contention

Only contend both cores try to manipulate shared address ranges
Address ranges: share PTEs
Address ranges: share PTEs

When one core faults on a shared page...
Address ranges: share PTEs

...and fills in the PTE...

core 0 → pglevel1

pglevel1

pglevel1 → core 1

pglevel0

ar_struct

ar_struct

stack 0

ar_struct

array

stack 1
Address ranges: share PTEs

...the PTE is part of the other core's page table.
Address ranges good for complex memory sharing patterns

- Typical applications have more complex memory sharing patterns
  - Not just global or private
  - Example: MapReduce library designed for multicore
Inverted index with MapReduce
Inverted index with MapReduce

Map

<key, val> buckets

core 0

apple, 0 ... ...

... ... ...

core 1

banana, 50 ... ...

... ... ...

Apple ... ... banana ...

....

....

....
Inverted index with MapReduce

Map

<key, val> buckets

Reduce

core 0

apple, 0 ... ... ...

... ... ... ...

core 0

core 1

banana, 50 ... ... ...

... ... ... ...
Inverted index with MapReduce

Map
<key, val> buckets

Reduce
MapReduce sharing goals for kernel data structures

- No contention when growing the address space during Map
  - No contention in the mm_struct/ar_struct
- Share PTEs between Map and Reduce
Address ranges meet the goals
Avoids contention when growing the address space during Map

Each Map task's bucket array is mapped by a different ar_struct.
Avoids contention when growing the address space during Map

No contention while growing bucket array.
Avoids contention when growing the address space during Map

No contention while growing bucket array.
Avoids contention when growing the address space during Map

No contention while growing bucket array.
During Reduce PTEs are shared
During Reduce PTEs are shared

When Reduce on core 0 processes results from the Map tasks...
During Reduce PTEs are shared

the Map results associated PTEs will already be filled in.
During Reduce PTEs are shared
Address ranges in Corey

- Corey is a small experimental OS
- Low-level kernel interface for mapping memory and address ranges
- MapReduce uses Corey's user-level malloc
MapReduce application

- Word inverted index
- Measured the time to build the index of a 1Gbyte file
- For Linux, shared address space is faster than private address spaces
  - Fewer soft page faults
MapReduce reverse index results

- For Linux cores contend on mm_struct
  - Linux page fault handler is faster than Corey's
- With address ranges there is no contention
Benefit of address ranges

- Able to avoid contention, but able to share what is necessary
- Applications control how cores share internal OS data structures
- Few application modifications to improve scalability
Related work

- Research on NUMA operating systems.
  - K42 and Tornado: clustered objects
  - Disco and Cellular Disco on Flash: “distributed kernel”
- Research on multicore:
  - Linux performance studies
  - McRT
  - Barrelish
  - Thread clustering, constructive caching
- KeyKOS segments
Future work

- Finish Linux interface changes
  - Other interface changes
  - Bigger workloads
  - How much does OS interface need to change?

- Use caches better
  - Reduce cost of manipulating shared data
  - Large aggregate cache, small per-core caches
  - Kernel cores help
Summary

- New OS interfaces that help applications scale with the number of cores

- Allow applications to control how cores share kernel data structures
  - Avoid contention from unnecessary sharing
  - Share state when its beneficial