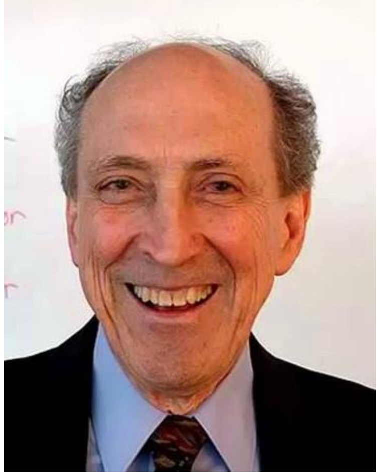
Multicore

Advanced Operating Systems

Overview

- Brief history on multicores
- Turning x86 cores on and off
- Consistency and scalability issues of multicore kernels
- Support for kernel parallelism

Dennard's Scaling (1972)



$$P = NCFV^2 + VI_{leakage}$$

P: power; N: transistors; C: capacitance; F: Frequency

- Power decides what you can do in your chip
- If you can reduce C, you can increase F (run things faster!)

Moore's Law (1965)

- Transistor size halves every 18 months (almost halving C)

$$P = NCFV^2 + VI_{leakage}$$

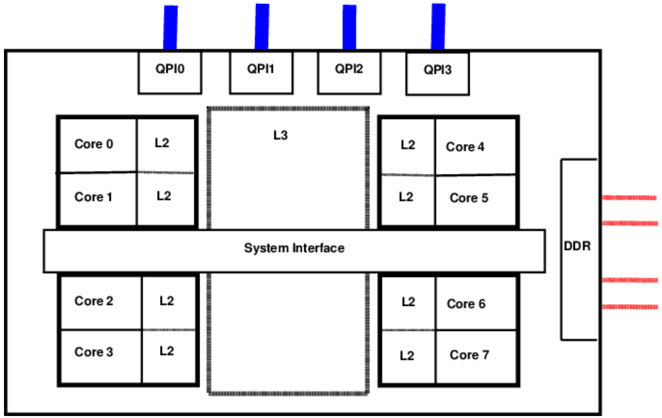
- What to do with the extra transistors?
  - Bigger caches, extra functions (e.g., SSE)
  - Instruction-level parallelism (pipelining and out-of-order execution)
  - ILP worked well with increased frequency
- This worked till 65nm transistors (ca. 2005)
  - $I_{leakage}$ problems < 65nm → heat dissipation

Post-Dennard (2005-Now)

- Do not increase frequency
  - Bounds power consumption
  - Solves the heat problem
- What to do with the extra transistors?
  - Instruction-level parallelism has diminishing returns
- Bigger caches
  - Last-level cache now in order of 10MB
- Additional hardware features
  - GPU/NIC/FPGA on chip
  - lbr, mpx, vmx, vmfunc, ept, sgx, pt, tsx, avx, cfi..
- Multicores!

Multicores

- Explicit parallel execution in a processor chip
  - Desktops, laptops, servers, mobile phones
- Cores share resources
  - Cache(s)
  - Memory
  - Memory controller
  - I/O devices
  - Coherence Traffic
  - System interconnect



Multicores and OS Kernels

- Kernel needs to take care of
  - Start/stop cores when necessary
    - x86 APIC
  - Deal with consistency/scalability issues
    - Kernel locking, partitioning, and replication
  - Schedule work on them
    - User/kernel threads

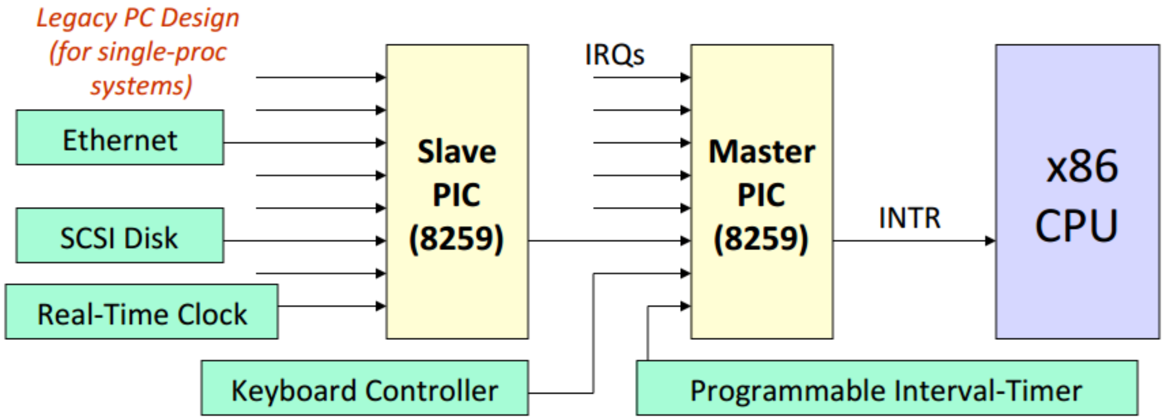
Multicores and OS Kernels

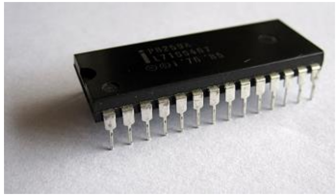
- Kernel needs to take care of
  - Start/stop cores when necessary
    - x86 APIC
  - Deal with consistency/scalability issues
    - Kernel locking, partitioning, and replication
  - Schedule work on them
    - User/kernel threads

Turning on x86 Cores

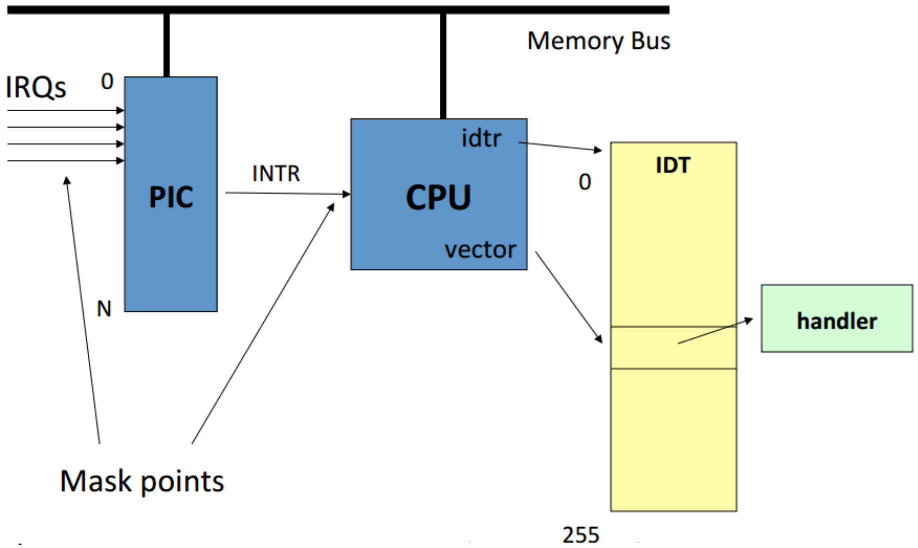
- BIOS hands the execution to one core
  - Intel calls this bootstrapping processor (BSP)
- BSP in turn can start other cores in the system
  - Intel calls these cores Application Processors or APs
  - BSP starts APs by sending them a special interrupt

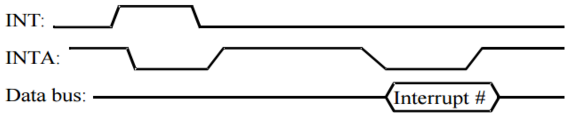
Programmable Interrupt Controller (PIC)





Programmable Interrupt Controller (PIC)





Advanced Programmable Interrupt Controller (APIC)

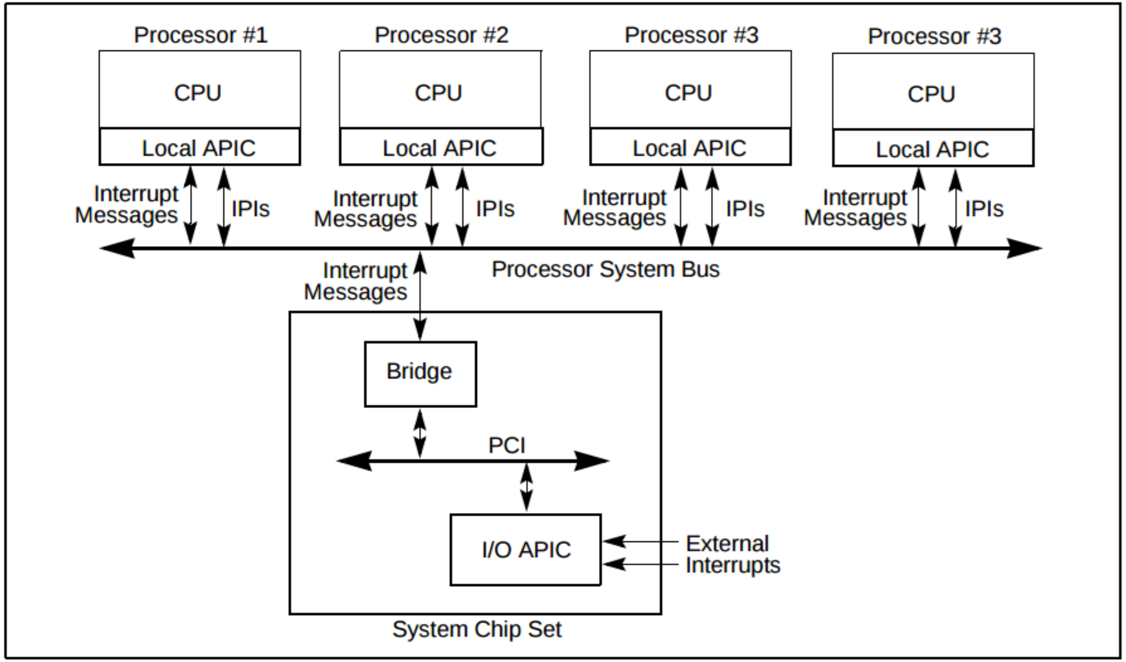


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

APIC Features

- Interrupt reporting
  - Replaces 8259 PIC
- Advanced features
  - Thermal management, performance monitoring
  - Internal error reporting
- Inter-processor interrupts (IPI)
  - Send interrupts to another core
  - Forward an interrupt
  - Start another core

Starting Application Processors

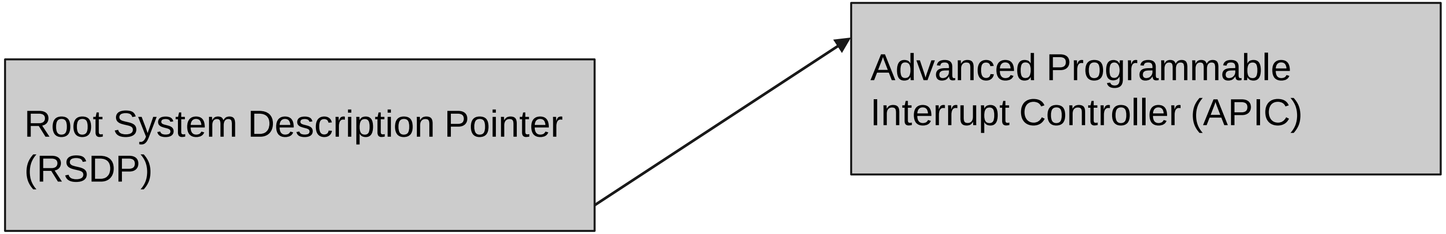
- Need to send an interrupt to remote cores
- Use the local APIC
  - How do you find the local/remote APIC?
  - How do you program the local APIC for this?

Starting Application Processors

- Need to send an interrupt to remote cores
- Use the local APIC
  - How do you find the local/remote APIC?
  - How do you program the local APIC for this?

Advanced Configuration and Power Interface (ACPI)

- BIOS sets it up in a predefined memory region
- Used to discover and configure various hardware components
    - APIC is a device
    - One LAPIC register entry per core (MADT)



Advanced Configuration and Power Interface (ACPI)

`sudo hd /proc/mem`

Root System Description Pointer (RSDP)

```
000fcfe0  52 53 44 20 50 54 52 20  da 44 45 4c 4c 20 20 02  |RSD PTR .DELL  .|
000fcff0  28 20 e3 79 24 00 00 00  c0 20 e3 79 00 00 00 00  |( .y$.... .y....|
000fd000  a0 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000fd010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Root System Description Table (RSDT/XSDT)

```
79e320c0  58 53 44 54 04 01 00 00  01 de 44 45 4c 4c 20 20  |XSDT......DELL  |
79e320d0  43 42 58 33 20 20 20 00  09 20 07 01 41 4d 49 20  |CBX3    .. ..AMI |
79e320e0  13 00 01 00 88 4f e7 79  00 00 00 00 a0 50 e7 79  |.....O.y.....P.y|
79e320f0  00 00 00 00 98 51 e7 79  00 00 00 00 e0 51 e7 79  |.....Q.y.......y|
```

Multiple APIC Description Table (MADT)

```
79e750a0  41 50 49 43 f4 00 00 00  04 ef 44 45 4c 4c 20 20  |APIC......DELL  |
79e750b0  43 42 58 33 20 20 20 00  09 20 07 01 41 4d 49 20  |CBX3    .. ..AMI |
79e750c0  13 00 01 00 00 00 e0 fe  01 00 00 00 00 08 01 00  |................|
79e750d0  01 00 00 00 04 06 01 05  00 01 00 08 02 02 01 00  |................|
79e750e0  00 00 04 06 02 05 00 01  00 08 03 04 01 00 00 00  |................|
```

LAPIC mapped at 0xfee00000

Proc 1 - APIC 0
Proc 2 - APIC 2
Proc 3 - APIC 4
…

Starting Application Processors

- Need to send an interrupt to remote cores
- Use the local APIC
    - How do you find the local/remote APIC?
    - How do you program the local APIC for this?

Enabling APIC

- Find out where memory mapped
  - Stored in MADT
  - Same between cores – no access to remote LAPIC
- Set up spurious interrupt vector
- Disable 8259 PIC (both master and slave)
  - Mask all interrupts
  - Remap IRQs to ≥ vector 32

Sending IPIs

- Need to know destination core's APIC ID
  - List of mappings in MADT
- Need to send IPI
  - Write to LAPIC Interrupt Command Register (ICR)

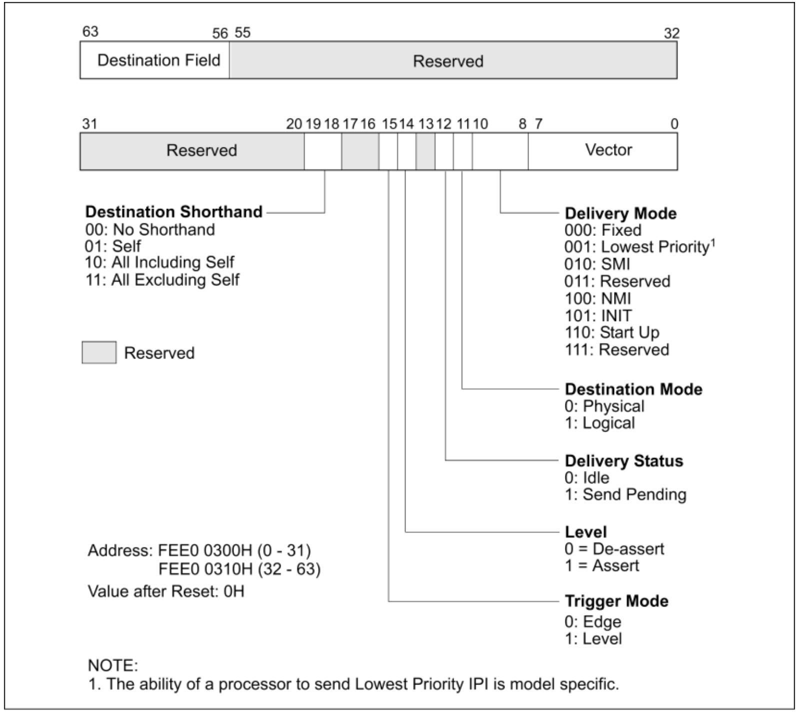


Figure 10-12. Interrupt Command Register (ICR)

Starting APs

- Send INIT IPI
  - Resets core specified with its APIC ID
- Send start-up IPI (SIPI)
  - Remote core starts executing given entry point
- Think about
  - (per-core) kernel stack
  - What happens at interrupts

Multicore OpenLSD

Lab 6: The more the merrier (Multicore)

Start APs

Dark Silicon and the End of Multicore Scaling (ISCA'11)

- Switching draws much power with many cores
- Cannot keep all transistors on at all times
- At 8nm (ca. 2019) 50% of the chip needs to be turned off
- Solution?
  - Lower frequency
  - Specialized cores
  - Turn cores on/off quickly

Multicore OpenLSD

Lab 6: The more the merrier (Multicore)

Bonus: Support core unplug

Multicores and OS Kernels

- Kernel needs to take care of
  - Start/stop cores when necessary
    - x86 APIC
  - Deal with consistency/scalability issues
    - Kernel locking, partitioning, and replication
  - Schedule work on them
    - User/kernel threads

Dealing with Concurrency

- Cores can execute kernel concurrently
  - Need to keep kernel state consistent
- Shared kernel subsystems
  - Frame allocator, I/O devices (e.g., console), scheduler, etc.
- Protecting the shared state
  - Lock the state (performance problems)
  - Partition the state (underutilization)
  - Replicate it (stale state)

Locks

- Spinlocks
  - Pros: easy to use, fast
  - Cons: waste cycles, scalability issues (cache coherence)
- Mutexes
  - Pros: easy to use, wasteless
    (core can do something else while waiting for the lock)
  - Cons: very high latency
- Read-copy-update (RCU)
  - Use replication to make locks scalable
  - Pros: fast, scalable in mostly-read scenarios
  - Cons: waste memory, difficult to use

Big Kernel Lock (BKL)

- Single (spin)lock protects entire kernel state
  - Grab this lock every time you enter kernel
- Pros: minimal complexity
- Cons: only one core can do work in kernel
  - E.g., fs cannot write to disk while webcam running
- Example: old versions of Linux

OpenLSD BKL

Lab 6: The more the merrier (Multicore)

Protect your kernel with a BKL

(When starting cores)

Fine-grained Locking

- Partition BKL
  - Start with separate locks for each subsystem
  - Continue with fine-graining the subsystem lock further down to data structures
  - Stop when the system does not wait on locks
  - Repeat when users complain
- Pros: concurrent execution in the kernel
- Cons: complexity
  - What may get concurrently accessed?
  - Fixed lock ordering everywhere to avoid deadlocks

Things Easily Get Out of Hand (linux/mm/rmap.c)

```
* Lock ordering in mm:
*
* inode->i_mutex     (while writing or truncating, not reading or faulting)
*   mm->mmap_lock
*         page->flags PG_locked (lock_page)   * (see huegtlbfs below)
*         hugetlbfs_i_mmap_rwsem_key (in huge_pmd_share)
*         mapping->i_mmap_rwsem
*         hugetlb_fault_mutex (hugetlbfs specific page fault mutex)
*         anon_vma->rwsem
*         mm->page_table_lock or pte_lock
*         pgdat->lru_lock (in mark_page_accessed, isolate_lru_page)
*         swap_lock (in swap_duplicate, swap_info_get)
*                  mmlist_lock (in mmput, drain_mmlist and others)
*                  mapping->private_lock (in __set_page_dirty_buffers)
*                  mem_cgroup_{begin,end}_page_stat (memcg->move_lock)
*                  i_pages lock (widely used)
```

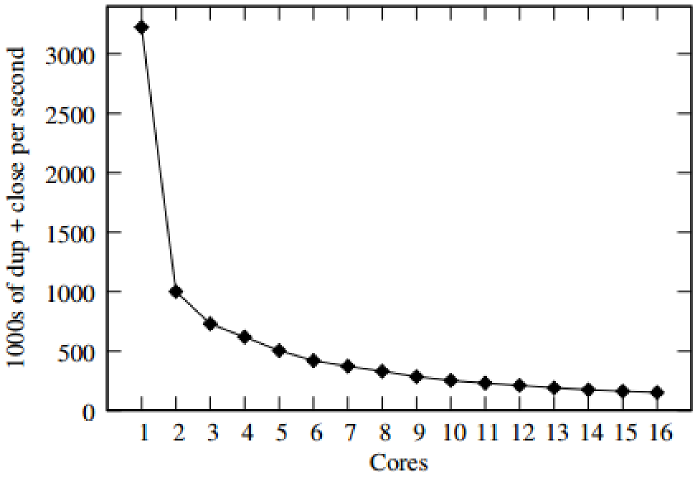
Killing the OpenLSD BKL

Lab 6: The more the merrier (Multicore)

Fine-grain your previous BKL

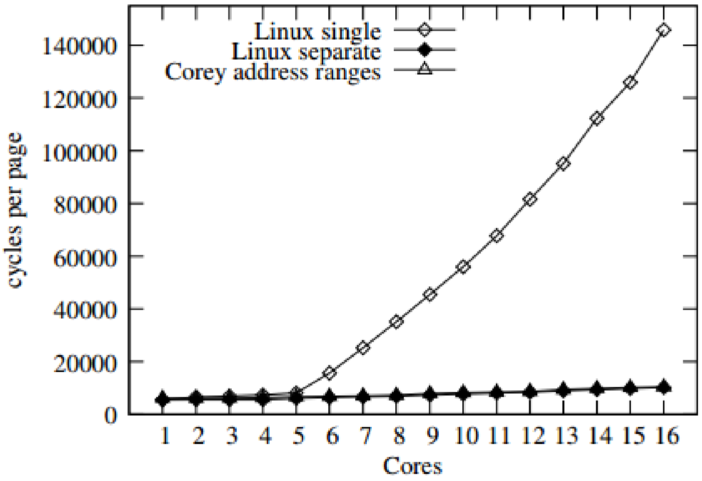
Overlocking

OS needs to ensure consistency according to the standard (e.g., POSIX)



Corey: Applications Should Control Sharing

Linux’s VMAs vs. Corey’s Address Ranges



(a) memclone

Tessellation: Space-Time Partitioning of the System

- Resource management does not scale to multicores for entire system
  - OS performs space-time partitioning
  - Applications perform resource management

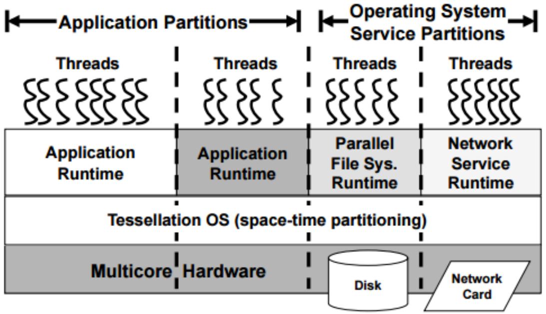
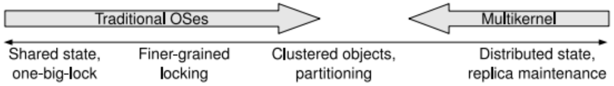


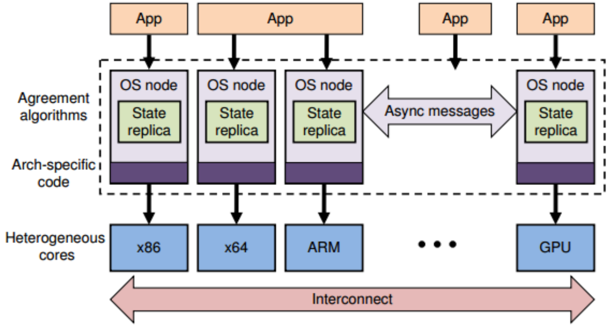
Figure 2: *Space-Time Partitioning* in Tessellation: a snapshot in time with four spatial partitions.

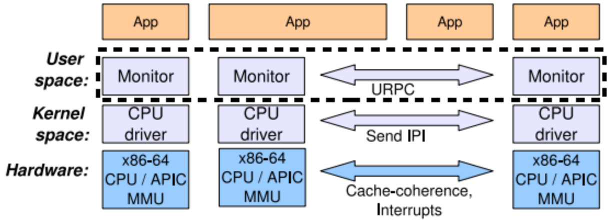
Multikernel: Replicate the State as a Principle

- Fine-graining locks does not scale from both performance and developer perspectives

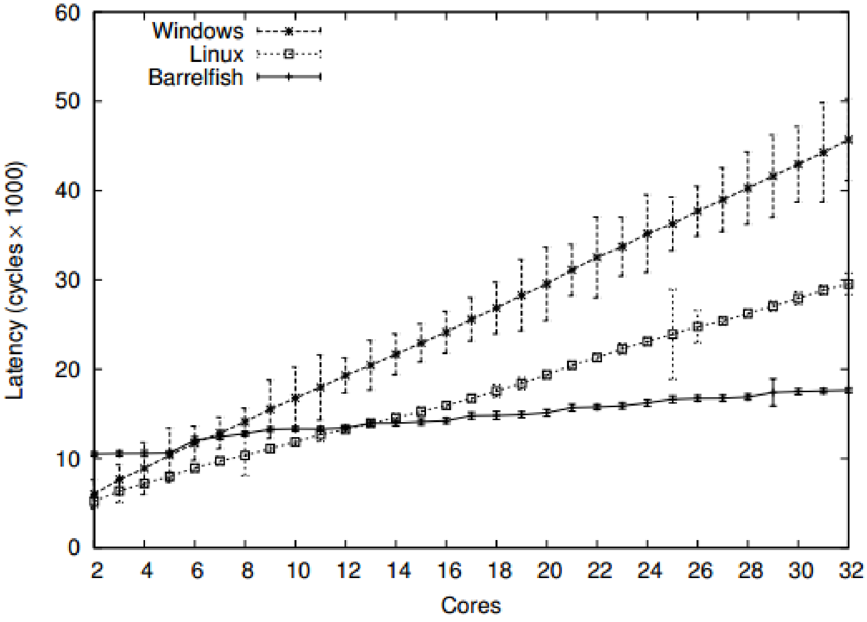


- Scalable kernel by design through replication





Example: unmap latency



Partial State Replication in Linux

- Create a copy of hot data structures per CPU
- Each CPU can access its own lock-free

```
/* include/linux/percpu-defs.h */
#define DEFINE_PER_CPU(type, name) /* ... */

/* kernel/sched/sched.h */
DECLARE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);

#define cpu_rq(cpu)                   (&per_cpu(runqueues, (cpu)))
#define this_rq()                   this_cpu_ptr(&runqueues)
```

- Per-core caches to alleviate load on certain subsystems (e.g., frame allocator)
  - Refer to “Managing Physical Memory” lecture

OpenLSD Per-Cores

Lab 6: The more the merrier (Multicore)

Bonus: per-core frame allocation support and per-core run queues

Multicores and OS Kernels

- Kernel needs to take care of
  - Start/stop cores when necessary
    - x86 APIC
  - Deal with consistency/scalability issues
    - Kernel locking, partitioning, and replication
  - Schedule work on them
    - User/kernel threads

Multicore Parallelism

- Needs multiple threads of execution
- User-mode applications can spawn threads
  - Kernel schedules user-mode threads on top of idle cores
  - Refer to the “Multiprocessing” lecture
- Parallelism for kernel tasks
  - Post-interrupt work
  - Background maintenance
  - Filling up per-CPU frame caches
  - Writing dirty data to disk, swapping
  - RCU garbage collection

Parallelism for the OpenLSD Kernel

Lab 6: The more the merrier (Multicore)

Provide support for parallel work in the kernel

Some ideas:

1. Have an execution that always runs in the kernel context (scheduling?)
2. Have an execution that runs in the user context but does kernel work (communication?)
3. Take care of work on behalf of user processes (state management?)

Concurrency in Linux

- Interrupt handlers do as little as possible (top half)
- Deferred handling in interrupt execution context
  - Bottom Halves/SoftIRQs/Tasklets
  - No active process
  - No sleeping, just spinlocks/RCU
- Process execution context
  - Workqueues/kernel threads
  - Active process
  - Sleep allowed, mutex possible

Bottom Halves (BHs)

- Original approach: Bottom Halves
  - Enqueue interrupt handing
  - Execute before returning to user mode
  - Only one BH at a time
- Inefficient due to lack of parallelism
  - Now obsolete
  - Replaced by SoftIRQs and Tasklets

SoftIRQs vs. Tasklets

- SoftIRQ
  - Multiple (instances of) SoftIRQs run concurrently
  - Very efficient, hard to program
  - Limited number available (32)
- Tasklets
  - Multiple tasklets run concurrently, but just one instance of each
  - Less concurrency, easier to program
  - Implemented on top of SoftIRQs
  - Can dynamically register more

Linux SoftIRQs

- High-priority work (interrupts enabled)
- Run concurrently on multiple CPUs
  - Need spinlocks/RCU
- Per-CPU bitmask of SoftIRQs needing work
- Kernel checks bitmask right after interrupt and periodically with local_bh_enable()

Priorities

```
HI_SOFTIRQ
TIMER_SOFTIRQ
NET_TX_SOFTIRQ
NET_RX_SOFTIRQ
BLOCK_SOFTIRQ
IRQ_POLL_SOFTIRQ
TASKLET_SOFTIRQ
SCHED_SOFTIRQ
HRTIMER_SOFTIRQ
RCU_SOFTIRQ
```

SoftIRQ Kernel Threads

```
$ ps aux | grep ksoftirq
root   10  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/0]
root   18  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/1]
root   24  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/2]
root   30  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/3]
root   36  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/4]
root   42  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/5]
root   48  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/6]
root   54  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/7]
root   60  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/8]
root   66  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/9]
root   72  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/10]
root   78  0.0  0.0   0   0 ?   S    sep23   0:00 [ksoftirqd/11]
```

Linux Tasklets

- Dynamically created deferred work

```
/* include/linux/interrupt.h */
struct tasklet_struct
{
  struct tasklet_struct *next;
  unsigned long state;
  atomic_t count;
  void (*func)(unsigned long);
  unsigned long data;
};

static inline void tasklet_schedule(struct tasklet_struct *t)
{
 if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
   __tasklet_schedule(t);
}
```

Linux Workqueues

- SoftIRQ and Tasklets run in interrupt execution context
- Workqueues run in process execution context
  - Backed by kernel threads
  - Graceful resource sharing and can sleep

```
/* include/linux/workqueue.h */
typedef void (*work_func_t)(struct work_struct *work);
struct work_struct {
  atomic_long_t data;
  struct list_head entry;
  work_func_t func;
#ifdef CONFIG_LOCKDEP
  struct lockdep_map lockdep_map;
#endif
};
static inline bool queue_work(struct workqueue_struct *wq,
                              struct work_struct *work)
{
  return queue_work_on(WORK_CPU_UNBOUND, wq, work);
}
```

Linux kworkers

```
$ ps aux | grep kworker
root         6  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/0:0H]
root        20  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/1:0H-kb]
root        26  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/2:0H-kb]
root        32  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/3:0H-kb]
root        38  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/4:0H-kb]
root        44  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/5:0H-kb]
root        50  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/6:0H-kb]
root        56  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/7:0H-kb]
root        62  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/8:0H-kb]
root        68  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/9:0H-kb]
root        74  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/10:0H-k]
root        80  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/11:0H-k]
root        85  0.0  0.0   0   0 ?   I    sep23   0:03 [kworker/3:1-mm_]
root       141  0.0  0.0   0   0 ?   I    sep23   0:08 [kworker/1:1-eve]
root       151  0.0  0.0   0   0 ?   I    sep23   0:03 [kworker/8:1-eve]
root       152  0.0  0.0   0   0 ?   I    sep23   0:49 [kworker/9:1-eve]
root       170  0.0  0.0   0   0 ?   I    sep23   0:10 [kworker/11:2-ev]
root       171  0.0  0.0   0   0 ?   I    sep23   0:04 [kworker/6:2-mm_]
root       184  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/u25:0]
root       281  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/4:1H-kb]
root       301  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/8:1H-kb]
root       336  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/5:1H-kb]
root       351  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/6:1H-kb]
root       366  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/1:1H-kb]
root       367  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/10:1H-k]
root       438  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/9:1H-kb]
root       442  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/11:1H-k]
root       443  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/3:1H-kb]
root       506  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/2:1H-kb]
root       531  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/0:1H-kb]
root       668  0.0  0.0   0   0 ?   I<   sep23   0:00 [kworker/7:1H-kb]
root      1542  0.0  0.0   0   0 ?   I    sep23   0:08 [kworker/4:0-eve]
```

Linux Kernel Threads

- Unit of parallelism in the Linux kernel
- Use the waitqueue mechanism to get notified of jobs

```
/* kernel/fork.c */
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
  struct kernel_clone_args args = {
    .flags       = ((lower_32_bits(flags) | CLONE_VM |
                   CLONE_UNTRACED) & ~CSIGNAL),
    .exit_signal = (lower_32_bits(flags) & CSIGNAL),
    .stack       = (unsigned long)fn,
    .stack_size  = (unsigned long)arg,
  };

  return _do_fork(&args);
}
```

References

1. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1
2. Linux software interrupts and realtime, https://lwn.net/Articles/520076/
3. Kernel APIs, Part 2: Deferrable functions, kernel tasklets, and work queues, https://www.ibm.com/developerworks/library/l-tasklets/
4. Concurrency Managed Workqueue (cmwq), https://www.kernel.org/doc/Documentation/workqueue.txt
5. Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, Doug Burger. Dark Silicon and the End of Multicore Scaling, in *ISCA* 2011.
6. Sankaralingam Panneerselvam and Michael M. Swift. Chameleon: Operating System Support for Dynamic Processors. In *ASPLOS* 2012.
7. Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *OSDI* 2014.
8. What is RCU, Fundamentally? http://lwn.net/Articles/262464/
9. Rick Lindsley and Dave Hansen. BKL: One Lock to Bind Them All. Ottawa Linux Symposium 2002.
10. Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP* 2009.
11. Juan A. Colmenares, Sarah Bird, Henry Cook, Paul Pearce, David Zhu, John Shalf, Steven Hofmeyr, Krste Asanovic, and John Kubiatowicz. Resource Management in the Tessellation Manycore OS. In *HotPar* 2010.
12. Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *OSDI* 2008.