CSE 120 Principles of Operating Systems

Spring 2018

Lecture 15: Multicore

Geoffrey M. Voelker

Multicore Operating Systems

- We have generally discussed operating systems concepts independent of the number of cores
- But many issues specific to a multicore environment
  - Cuts across many topics, makes more sense at end
- Today we'll discuss many of these issues and how operating systems tackle them
  - Architectural issues
  - Synchronization
  - Virtual memory
  - Scheduling
  - Scalability

Multicore Architecture

(Note: Wide variety of architectures in practice)

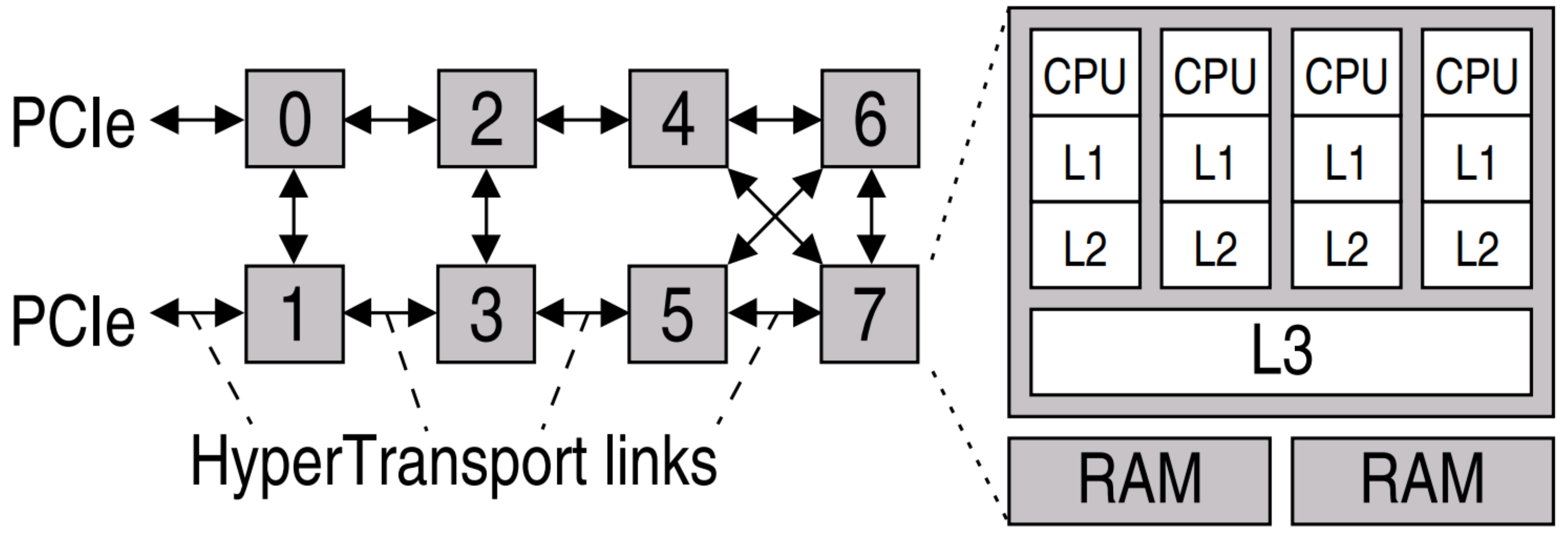

Figure 2: Node layout of an 8×4-core AMD system

(Image from A. Baumann et al., “The Multikernel: A new OS architecture for scalable multicore systems”, SOSP 2009)

Synchronization

- Disabling/enabling interrupts are per-core operations
  - Implemented with instructions, only apply to the CPU that executes those instructions
- Still needed
  - For synchronization interrupt handlers (as with single core)
  - Disable disk interrupts while handling disk interrupt
  - Disable timer interrupts while handling timer interrupt
- But for multicore synchronization
  - Need spinlocks (or equivalent)

Using Test-And-Set

- Here is our lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock→held));
}
void release (lock) {
    lock→held = 0;
}
```

- When will the while return? What is the value of held?
- What about multiprocessors?

Atomic Instructions

- Hardware implements atomicity across all cores
  - Atomic instructions are special memory operations
  - Use cache coherency machinery to implement atomicity
  - Essentially take a lock on a cache line

Back to Spinlocks

- Spinlock implementations highly tuned
  - Common case by far is that lock acquire succeeds
  - Want this common case to be fast (a few instructions)
- Many variants of spinlocks
  - Blocking spinlocks: Spin for a while, then block
  - Read/write spinlocks: Multiple readers || one writer
  - Seqlocks: R/W locks optimized for many readers
- One drawback of locks is that threads have to wait
  - Can we synchronize without waiting?!?
  - Yes! Get ready…

Wait Free / Non-Blocking Synchronization

- Data structure accessed via shared pointer
- Threads reading do not acquire a lock
  - They just start reading, no lock overhead at all
- Threads writing, though, do extra work
  - Create a private copy of the shared data structure
  - Apply updates to private copy
  - Check pointer to shared data structure
  - Not changed since started?
    - Atomically change the pointer to private copy
    - Becomes new shared version
  - Changed since started?
    - Abort and restart from the beginning

Why Does This Work?

- Only threads reading
  - Easy, nothing to synchronize
- Readers, one writer
  - While writer works on private copy, readers see old version
  - When writer finishes, atomically updates pointer
  - Readers will either see old version or new, but not in between
  - Readers never have to wait in either case
- Multiple writers
  - Creating copy is just reading, so no need to synchronize
  - All updates are to a private copy, no need to synchronize
  - First writer atomically updates pointer
  - All other writers have to abort and restart

When Does This Work Best?

- Read dominated workload
  - Optimizes away lock overhead for readers
    - Same performance as single-threaded code without locks
  - Writers have to create copies → overhead
  - More simultaneous writers → more wasted work
    - Only one writer succeeds, all others abort
- Small shared data structures
  - Larger the data structure → more effort making a copy
  - Longer the copy time, higher probability of another writer
  - Spinning could be much shorter
- In sum: Have to be selective if using this approach

Read-Copy-Update (RCU)

- Linux implements a specific form of non-blocking synchronization called RCU
- Same basic idea, but writer update slightly different
  - Writer waits until all readers have finished using old version
  - Relies upon scheduling, simple write to update pointer
- Implementations tricky for complex data structures
  - First used for simple data structures
  - With experience over many years, now used extensively throughout Linux kernel

Virtual Memory

- Every core has its own page table pointer
  - All address translations on that core use that page table
- Each core can be using a different page table
  - Executing kernel threads in different processes
- Multiple cores using the same user-level page table
  - Executing different kernel threads in the same process
- Multiple cores using the kernel page table
  - Executing different kernel threads in the OS at the same time
  - Why we need spinlocks, RCU locks, etc.

TLB Coherency

- Cache coherency H/W does not apply to TLB entries
  - Burden on OS to keep TLBs consistent
- When the OS updates a PTE
  - e.g., evict a page → need to invalidate the PTE for that page
- Invalidating PTEs expensive on multiple cores
  - Invalidate not only in the core executing the code, but all cores that are using the same page table
    - Also known as “TLB Shootdown”
  - Use inter-processor interrupt (IPI) to have other cores invalidate the PTE in their TLB
    - Overhead scales with number of cores
  - Need to track cores using the page table
    - Only trigger IPIs on those cores

Scheduling

- Multicore scheduling adds new dimensions to the scheduling problem
  - Already lots of heuristics for single CPU schedulers
  - Multicore makes the problem much harder
- Granularity?
  - Schedule processes or threads?
- Where?
  - Which cores should run which processes/threads?
- When?
  - When do jobs with multiple processes, or processes with multiple kernel threads run on multiple cores?

Time Scheduling

- Job queues
  - Single queue for entire system
  - Multiple queues, one per core (more typical in modern OSes)
- Queues use some scheduling algorithm
  - MLFQ, proportional, etc.
  - No explicit coordination: Queues scheduled independently
    - Often default case
- Coordinated scheduling
  - Coscheduling, gang scheduling: Processes/threads scheduled on multiple cores at the same time
    - Dependent execution, can only make process if all scheduled
    - Early parallel machines, modern use in, e.g., GPUs

Space Scheduling

- Partition and dedicate cores among jobs
  - Jobs assigned cores for their lifetime
  - Processes and threads for job scheduled just on those cores
- Used in modern “batch” systems
  - Supercomputers, data-parallel processing (Hadoop, Spark)
  - Queue of jobs
  - High-level scheduler maximizes job throughput in system
- Challenges
  - How many cores to allocate for a job?
  - How to bin-pack jobs on machines?
    - Think clusters of multicore machines
  - Often implemented as a higher-level scheduler (for cluster)

Application Hints

- Applications may want to run processes/threads only on specific cores
  - Cache locality, NUMA locality, I/O device locality, etc.
    - OS scheduler does try to achieve this naturally
    - e.g., Linux scheduling domains
  - Known as processor affinity or CPU pinning
- OS will only schedule on that core (or set of cores)
  - sched_setaffinity (syscall), taskset (command line program)
  - pthread_setaffinity_np (thread granularity)
- Can also dedicate cores to specific processes
  - Affinity of process A to core 0, other processes to other cores
  - Not “fair”, but useful in server environments

Scalability

- Many multicore issues are correctness issues
  - Synchronization, TLB coherency, etc.
  - Want them to be fast, but need them for correctness
- Other multicore issues are performance issues
  - Straightforward implementations are correct
  - But do not scale
- “Scalability” for multicore OS implementations
  - Performance of OS operations scales with the number of cores
  - More cores → better OS performance
- Lots of implementation complexity added to improve OS scalability

Per-Core Data Structures

- Global shared OS data structures need to be protected by a lock
- More cores → more contention for lock → serialization
  - Think about your list of free physical pages in Nachos
  - Every core managing processes/VM needs to access this list
- Instead, create per-core data structures
  - Each core has a private data structure → no global lock needed, can just use a per-core lock
    - Per-core list of physical pages
  - Complexity in balancing resources across cores
- Very common implementation technique
  - Page lists, ready lists, allocation pools, etc.

Cache Contention

- Cache coherency implemented by hardware
  - Simplifies implementing parallel software
  - But can also introduce performance bottlenecks
- Cache line contention → serialization bottleneck
  - Writing to a cache line requires invalidating in other CPUs
  - Not much of a problem with 4 cores…
  - Can be a headache with 32 cores

Cache Contention

- Atomic instructions
  - Spinlocks use atomic instructions (XCHG, XADD)
  - Writing on the spinlock invalidates all other caches, expensive
  - RCU avoids atomic instructions, just uses memory operations
- Shared memory
  - Many processors updating the same data (e.g., counters)
    - Contention on cache line serializes execution
    - Can even happen with RCU
  - Have to partition data structure (yes, even counters)

Cache Contention

- False sharing
  - Two different variables on same cache line
  - One written often, the other read often (but independently)
  - Causes cache line contention
    - Writing one variable invalidates the other variable in other cores
    - When other cores read variable, need to get cache line from writer
    - Lots of time spent moving the cache line from one core to another
  - Once discovered, easy to fix: move variables to different cache lines (e.g., move fields around in struct)

Next time…

- Read Appendix B