Lecture 6:
OS Scalability
+
Multiprocessor Scheduling

(Thanks to Jonathan Appavoo, Todd Mowry, and Angela Demke Brown)
Operating System Scalability

- We have looked at various synchronization strategies
  - Scalability has been a key concern
- Most user applications actually aren’t very scalable
- Most exceptions use few OS services anyway
  - E.g. scientific computing
- Most multiprocessor systems support independent processes (multi-user workloads)
- Why does OS scalability matter?
Systems View of Scalability

User Commands

User Commands

Scale up

Add Scripts & Processors (SMP)

Proc 0
Proc 1
Proc 2
The Problem

- SDET benchmark

![Graph showing throughput vs processors for Linux 2.4 benchmark.](image-url)
Scaling Existing OSes

- Internal shared structures affect “independent” requests
- Limits scalability

External Service Requests Scale up

Service Interface

Software Structures

Add brick

Processors
Memory

Processors
Memory

Processors
Memory
Areas of Concern

• Statistical counters
  • Widely used to track variety of system properties
  • Frequently updated, rarely read

• Processor scheduling
  • We’ll look at this closely in the next 2 lectures

• Memory management
  • In tutorial this week (also, Assignment 2!)
Simple Shared Counter Example

Average time per request

- Single Shared Variable

Ideal – flat line
Solution: Per-CPU data

- OS assigns each CPU an integer \( id \) at boot time
  - Linux: access with \texttt{smp_processor_id()}\)

- Basic data structure is array with entry for each CPU
  - \texttt{counter[smp_processor_id()]} is data structure for current CPU
Simple Shared Counter Example

Average time per request

- Single Shared Variable
- Array of Per-CPU counters

Ideal – flat line
What went wrong?

• Per-CPU array can lead to *false sharing* problem
  • Each CPU has own variable
  • Several per-CPU variables are on same cache line
  • Modification of one causes invalidates in other CPUs' caches

• Solutions?
  • Use *padding* so each per-CPU variable lies on different cache line
Simple Shared Counter Example

- Single Shared Variable
- Array of Per-CPU counters
- Padded Array of per-CPU counters

Average time per request

0 3,750 7,500 11,250 15,000 18,750

1 3 5 7 9 11 13 15 17 19 21 23
Summary

• Taking a traditional OS and making it scale well on shared memory multiprocessors is hard
  • Fast uniprocessor solutions typically don’t scale
  • Designing for scalability can hurt uniprocessor performance
  • Maintaining scalability with every change is hard

=> Must design a system from the ground up, with scalability in mind
Insights and Approaches

- Scalability must be considered in system design
- Shared data is the enemy
  - Distribute data structures
  - Use per-cpu data whenever possible
    - With padding to cache lines!
- Minimize locking and expensive atomic ops
- Ideas from research have been adopted by mainstream
  - UofT/IBM Tornado/K42 projects showed techniques to improve scalability
    - Some applied to Linux scalability project
  - More recently, MIT Corey project and OSDI paper on improving Linux scalability further
Multiprocessor Scheduling

• Why use a multiprocessor?
  • To support multiprogramming
    • Large numbers of independent processes
    • Simplified administration
    • E.g. CDF wolves, compute servers
  • To support parallel programming
    • “job” consists of multiple cooperating/communicating threads and/or processes
    • Not independent!
  • First - the easy case: scheduling threads
Basic MP Scheduling

- Given a set of runnable threads, and a set of CPUs, assign threads to CPUs
- Same considerations as uniprocessor scheduling
  - Fairness, efficiency, throughput, response time…
- But also new considerations
  - Ready queue implementation
  - Load balancing
  - Processor affinity
Straightforward Implementation

- Scheduling events occur per CPU
  - Local timer interrupt
  - Currently-executing thread blocks or yields
  - Event is handled that unblocks thread
- Scheduler code executing on any CPU simply accesses shared queue
- What might be sub-optimal about this?
Alternative Ready Queue Design

- Advantages of Distributed Queues?
  - different policies per queue
  - easy to maintain affinity
  - minimal locking contention to access queue

- Cons?
  - Load balancing
Load Balancing

- Try to keep run queue sizes balanced across system
  - Main goal – CPU should not idle while other CPUs have waiting threads in their queues
  - Secondary – scheduling overhead may scale with size of run queue
    - Keep this overhead roughly the same for all CPUs
- **Push** model – kernel daemon checks queue lengths periodically, moves threads to balance
- **Pull** model – CPU notices its queue is empty (or shorter than a threshold) and steals threads from other queues
- Many systems use both
Work Stealing with Distributed Queues

Notice a problem though? => What about processor affinity?
Processor Affinity

- As threads run, state accumulates in CPU cache
- Repeated scheduling on same CPU can often reuse this state
- Scheduling on different CPU requires reloading new cache
  - And possibly invalidating old cache
- Try to keep thread on same CPU it used last
  - Automatic
  - Advisory hints from user
  - Mandatory user-selected CPU
- Called “affinity scheduling”
- Do they always find a warm cache though? What can happen?
Symbiotic Scheduling

- Threads load data into cache
- Expect multiple threads to thrash each others’ state as they run
- Can try to detect cache needs and schedule threads that can share nicely on same CPU
  - Examples? What kind of threads should be scheduled together?
  - E.g. several threads with small cache footprints may all be able to keep data in cache at same time
  - E.g. threads with no locality might as well execute on same CPU since almost always miss in cache anyway
Linux Scheduler Case Study

- per-CPU runqueues
  - Actually a red-black tree with the CFS scheduler
- CPUs are organized into *scheduling domains*
  - A set of CPUs whose load is kept balanced by kernel
  - Each domain contains a set of groups
- Load balancing is hierarchical
  - On each tick, recomputes local load statistics
  - Checks if time to invoke `load_balance()` for each domain from base to top-level
Linux Load Balancing

- Finds busiest group in current domain
- Finds busiest queue (CPU) in that group
- Invokes move_tasks to actually move threads
  - move_tasks attempts to preserve affinity when finding task to move
    - Can’t be currently executing
    - Target CPU must be allowable for task
    - Target CPU is idle OR process is not “cache hot” OR kernel has failed repeatedly to move processes
- Further reading (optional):
  - The Linux Scheduler – a decade of wasted cores (Eurosys 2016)
Parallel Job Scheduling

- “Job” is a collection of processes/threads that cooperate to solve some problem (or provide some service)
  - *Not* independent!
- How the components of the job are scheduled has a major effect on performance
  - Want scheduler to be *aware of dependences*
- We will look at two major strategies
  - **Space sharing** – each job has dedicated processors
  - **Time sharing** – multiple jobs share same processors
Why Job Scheduling Matters?

- Recall threads in a job are not independent
  - Synchronize over shared data
    - De-schedule lock holder, other threads in job may not get far
  - Cause/effect relationships (e.g. producer-consumer problem)
    - Consumer is waiting for data on queue, but producer is not running
- Synchronizing phases of execution (barriers)
  - Entire job proceeds at pace of slowest thread
Forms of scheduler-awareness

1. Know threads are related, schedule all at same time
   - Space sharing: all threads are from same job
   - Time sharing: group threads that should be scheduled together

2. Know when threads hold spinlocks
   - Don’t deschedule lock holder
   - Extends timeslice, but not indefinitely

3. Know about general dependences
   - E.g. infer producer/consumer relationships
Space Sharing Scheduling

- Divide processors into groups
  - Fixed, variable, or adaptive
- Assign job to dedicated set of processors
  - Ideally one CPU per thread in job
- Job waits until required number of CPUs are available (batch scheduling)

**Fixed:**
Always 2 groups of 4 CPUs.

**Variable:**
Currently 3 groups of 2, 4, and 2 CPUs; changes as jobs come and go.

**Adaptive:**
Job can ask for more CPUs as it runs.
Space Sharing

• Typically used on supercomputers
• Pros:
  • All runnable threads execute at the same time
  • Reduce context switch overhead (no involuntary preemption)
  • Strong affinity
• Cons?
  • Inflexible
  • CPUs in one partition may be idle while another partition has multiple jobs waiting to run
  • Difficult to deal with dynamically-changing job sizes
    • Adaptive scheme is complicated and uncommon
Choosing Jobs to Run

• At job creation, specify number of threads
• Scheduler finds set of CPUs
  • May negotiate with application

• How should scheduler choose jobs to assign to CPUs?
  What is optimal (in terms of average wait time)?
  • Uniprocessor scheduling  ➔ Shortest Job First (SJF) (shortest expected next CPU burst)
  • MP version  ➔ smallest expected number of CPU cycles
    (cycles == num_cpus * runtime)
Estimating Runtime

- Estimates typically come from users who submit the jobs
  - Low estimates make it “easier” to do scheduling
  - But cause trouble if not accurate!
  - Soln: kill jobs that exceed estimate
    - What user behaviour does that incentivize?
- How accurate are user estimates?
- Can automatic estimates based on history do better?
- How much does it matter?
Space Sharing - FCFS

- Scheduling convoy effect
  - Long average wait times due to large job
  - Exists with FCFS uniprocessor batch systems
  - Much worse in parallel systems
    - Fragmentation of CPU space

Scheduler queue (CPUs, time)

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time

CPUs
Solution: Backfilling

- Fill CPU “holes” from queue in FCFS order
- Not FCFS anymore. What can happen?
- Want to prevent “fill” from delaying threads that were in queue earlier
  - EASY (Extensible Argonne Scheduling System)
    - Make reservation for next job in queue

Scheduler queue (CPUs, time)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>Time</td>
<td>1,3</td>
<td>2,2</td>
<td>4,2</td>
<td>2,2</td>
<td>1,4</td>
</tr>
</tbody>
</table>
Variations on Backfilling

• EASY
  • 1. Used FCFS to order jobs in queue
  • 2. Made reservation for first blocked job in queue
  • 3. Backfilled jobs by looking at queue one at a time

• 1. Ordering alternative: include priority in queue
  • Administrative to distinguish between users
  • User to distinguish between own jobs
  • Scheduler to prevent starvation

• 2. Reservation alternatives
  • All queued jobs get a reservation (too much can go wrong)
  • Queued job gets a reservation if it has been waiting more than a threshold

• 3. Queue lookahead
  • Use dynamic programming to determine optimal packing
Parallel Time Sharing

- Each CPU may run threads from multiple jobs
  - But with awareness of jobs
- Co-scheduling (Ousterhout, 1982)
  - Identify “working set” of processes (analogous to working set of memory pages) that need to run together
- Gang scheduling
  - All-or-nothing $\rightarrow$ co-scheduled working set is all threads in the job
  - Get scheduling benefits of dedicated machine
  - Allows all jobs to get service
Gang Scheduling Example

- Multiprogramming level is typically controlled by either:
  - Monitoring memory demand, or
  - Fixed number of slots (rows)
    - e.g. IBM LoadLeveler Gang Scheduling allows up to 8 sets of jobs to be multiprogrammed on a set of CPUs
Gang Scheduling Issues

• All CPUs must context switch together
  • To avoid fragmentation, construct groups of jobs that fill a slot on each CPU
    • E.g., 8-CPU system, group one 4-thread job with two 2-thread jobs
  • Inflexible
    • If 4-thread job blocks, should we block entire group, or schedule group and leave 4 CPUs idle?

• Alternative 1: Paired gang scheduling
  • Identify groupings with complementary characteristics and pair them. When one blocks, the other runs

• Alternative 2: Only use gang scheduling for thread groups that benefit
  • Fill holes in schedule with any single runnable thread from those remaining
Example: Effect of Gang Scheduling

- LLNL gang scheduler on 12-CPU Digital Alpha 8400
  - Parallel gaussian elimination program
  - Run concurrently with 12 single-threaded interfering processes
  - Benefits due to synchronization effects and better cache use

Source:
Forms of scheduler-awareness

1. Know threads are related, schedule all at same time
   - Space sharing: all threads are from same job
   - Time sharing: group threads that should be scheduled together

2. Know when threads hold spinlocks
   - Don't deschedule lock holder
   - Extends timeslice, but not indefinitely

3. Know about general dependences
   - E.g. infer producer/consumer relationships
Knowing about Spinlocks

- Thread acquiring spinlock sets **kernel-visible flag**
- Clears flag on release
- Scheduler will *not immediately deschedule* a thread with the flag set
  - Gives thread a chance to complete critical section and release lock
  - Spinlock-protected critical sections are (supposed to be) short
  - Does not defer scheduling indefinitely
Forms of scheduler-awareness

1. Know threads are related, schedule all at same time
   - Space sharing: all threads are from same job
   - Time sharing: group threads that should be scheduled together

2. Know when threads hold spinlocks
   - Don't deschedule lock holder
   - Extends timeslice, but not indefinitely

3. Know about general dependences
   - E.g. infer producer/consumer relationships
Knowing General Dependences

- Implicit Co-scheduling (Arpaci-Dusseau et al.)
- Designed for workstation cluster environment
  - Explicit messages for all communication/synchronization
  - MUCH more expensive if remote process is not running when local process needs to synchronize
- Communicating processes decide when it is beneficial to run
  - Infer remote state by observing local events
    - Message round-trip time
    - Message arrival
- Local scheduler uses communication info in calculating priority
OS Noise

- Or: how to schedule OS activities
- Massively parallel systems are typically split into I/O nodes, management nodes, and compute nodes
  - Compute nodes are where the real work gets done
  - Run customized, lightweight kernel on compute nodes
  - Run full-blown OS on I/O nodes and mgmt nodes
  - Why?
- Asynchronous OS activities perturb nice scheduling properties of running jobs together
  - Up to a factor of 2 performance loss in real large-scale jobs
  - Need to either eliminate OS interference, or find ways to coordinate it as well
Illustration of OS Noise Issue

• The following 4 slides are extracted from a tutorial given at EuroPar 2004 ([www.di.unipi.it/europar04/Tutorial2/tutorial_europar04.ppt](http://www.di.unipi.it/europar04/Tutorial2/tutorial_europar04.ppt))
  
  • They illustrate the issue with OS Noise on the ASCI Q supercomputer, published in Supercomputing 2003 (“The case of the missing supercomputer performance”)

  “Achieving Usability and Efficiency in Large-Scale Parallel Computing”

  • Kei Davis and Fabrizio Petrini {kei,fabrizio}@lanl.gov
  • Performance and Architectures Lab (PAL), CCS-3
    • (Computer and Computational Sciences Division, Los Alamos National Labs)
Performance of SAGE on 1024 nodes

- Performance consistent across QA and QB (the two segments of ASCI Q, with 1024 nodes/4096 processors each) \( \leftarrow \) 4 processors / node
- Measured time 2x greater than model (4096 Processor Elements)

There is a difference why?

OS activity is the culprit!

Lower is better!
The effect of the noise

- An application is usually a sequence of a computation followed by a synchronization (collective):

- But if an event happens on a single node, it can affect all the other nodes
Effect of System Size

- The probability of a random event occurring increases with the node count
Tolerating Noise: Buffered Coscheduling

- We can tolerate the noise by coscheduling the activities of the system software on each node.
Example Cluster Scheduler: SLURM

- Simple Linux Utility for Resource Management
  - Performs resource management within single cluster => 3 roles:
    - Allocates access to computer nodes and their interconnect
    - Launches parallel jobs and manages them (I/O, signals, time limits, etc.)
    - Manages contention in the queue
- Developed by Lawrence Livermore National Lab (LLNL)
  - With help from HP, Bull (European high performance computing company), Linux NetworX, and others
  - Open Source
  - Extensible, provides flexible plugin mechanism
  - Active development still on-going
- Widely used on high performance compute clusters
SLURM Features

- Plugins support multiple scheduling policies
  - FIFO
  - Backfilling
  - Gang Scheduling
    - Requires multi-core awareness at slurmctld
  - Priority-based preemption
  - Topology-aware scheduling
    - Reduce contention on interconnect
- Includes many management & accounting features
SLURM Architecture

- Graphic from
  https://computing.llnl.gov/linux/slurm/slurm.sc08.bof.pdf
Further readings

• Scheduling problem encountered in many contexts (e.g., datacenters)
• Omega (Eurosys 2013)
  • https://research.google.com/pubs/pub41684.html
• Hawk (USENIX 2015)
  • https://www.usenix.org/conference/atc15/technical-session/presentation/delgado
• Mesos (NSDI 2011)
  • https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf
• Sparrow (SOSP 2013)
  • https://dl.acm.org/citation.cfm?id=2522716
• Thoth (VLDB 2013)
  • https://dl.acm.org/citation.cfm?id=2733062
• And many more ...
Announcements

• Midterm
  • Next week on Monday, in the lecture room
  • Bring T-card!
  • Covers up to this week (excluding)
  • Write in pen, cannot regrade tests written in pencil!

• Location and logistics:
  • Starts at 9AM, as discussed in the first week of classes
  • Please be on time, we start at 9:10 sharp => 110 minutes
  • Check website and Piazza announcements!

• A2 to be released! More details in tutorial!