Lecture 7: OS Scalability & Multiprocessor Scheduling

(Thanks to Jonathan Appavoo)
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

- cmd0
- cmd1

Scale up

Add Scripts & Processors (SMP)

Users Commands

- cmd0
- cmd1

- Proc 0
- Proc 1
- Proc 2

OS
The Problem

- SDET benchmark

![Graph showing throughput vs. number of processors for Linux 2.4](image)

Throughput

Processors
Scaling Existing OSes

External Service Requests → Scale up

Service Interface

Internal shared structures affect “independent” requests

➡ Limits scalability

Add brick
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
  • Lectures following scheduling (and Assignment 3)
Simple Shared Counter Example

Average time per request

Ideal – flat line
Solution: Per-CPU data

- OS assigns each CPU an integer *id* at boot time
  - Linux: access with `smp_processor_id()`

- Basic data structure is array with entry for each CPU
  - `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
  - But 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

18,750
16,000
13,500
12,000
10,500
9,000
7,500
6,000
4,500
3,000
1,500
0

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

→ Scalability must be considered in system design!
Insights and Approaches

- Shared data is the enemy
  - Distribute data structures
  - Use per-cpu data whenever possible
    - With padding to cache lines!
- Minimize locking and expensive atomic
- Ideas from research have been adopted mainstream
  - UofT/IBM Tornado/K42 projects showed techniques to improve scalability
    - Some applied to Linux scalability project
Multiprocessor Scheduling

- Why use a multiprocessor?
  - To support multiprogramming
    - Large numbers of independent processes
    - Simplified administration
    - E.g., wolf.teach.cs, compute servers
  - To support parallel programming
    - "job" consists of multiple cooperating and/or communicating threads or processes
    - Not independent!
- First, the easy case: scheduling independent 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
Centralized Queue

• 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

• Advantages? Disadvantages?
Alternative Ready Queue Design

Distributed Queues

- Ready Queue 0
  - CPU 0
- Ready Queue 1
  - CPU 1
- ... (repeated for CPUs 0 to N)
- Ready Queue N
  - CPU N

Centralized Queue

- Ready Queue
  - CPU 0
  - CPU 1
  - ... (repeated for CPUs 0 to N)

• Advantages of Distributed Queues?
• Cons?
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

Ready Queue 0 → CPU 0

Ready Queue 1 → CPU 1

Ready Queue N → CPU N

Notice a problem though?
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?
And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.

- Linus Torvalds, 2001

Linux scheduler illustrations from Jean-Pierre Lozi (https://www.i3s.unice.fr/~jplozi/wastedcores/files/extended_talk.pdf)
The Completely Fair Scheduler (CFS)

Conceptually, one runqueue where threads are globally sorted by runtime

Threads get their next task from the global runqueue

When a thread is done running for its timeslice, it is enqueued again

Some tasks have a lower niceness and thus have a longer timeslice (allowed to run longer, higher weight)

Core 0       Core 1       Core 2       Core 3
CFS on Multiprocessor

- Accumulated runtime is not a useful metric for load balancing
- Define CPU load of a thread: Load = Weight x %CPU
Linux Scheduler Basics

- 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
Hierarchical Load Balancing

L = 2000
Core 0

L = 3000
Core 1

L = 6000
Core 2

L = 1000
Core 3
Hierarchical Load Balancing

Balanced!

L = 2000

L = 3000

Core 0

L = 1000

L = 1000

Core 1

L = 3000

L = 1000

L = 1000

L = 6000

L = 1000

Core 2

Core 3

L = 1000

L = 1000

L = 1000

L = 1000

L = 1000

L = 1000

L = 1000

L = 1000

Balanced!
Hierarchical Load Balancing

L = 2000
L = 3000

Core 0
Core 1

L = 6000 ↔ L = 1000

Core 2
Core 3

Unbalanced
Hierarchical Load Balancing

L = 2000
Core 0
L = 3000
L = 3000
Core 1

L = 1000
L = 1000
L = 1000
L = 1000

L = 4000
Migrate
L = 1000
L = 1000
L = 1000
L = 1000

L = 3000
Core 2
Core 3
Balanced!
Hierarchical Load Balancing

Unbalanced

Average(L) = 2500 $\leftrightarrow$ Average(L) = 3500

- Core 0: L = 2000
- Core 1: L = 3000
- Core 2: L = 4000
- Core 3: L = 3000
Hierarchical Load Balancing

Average(L) = 3000

Balanced
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
- Actual load calculations in Linux are quite complex
  - Can fail to achieve balance and fairness in some scenarios
- Further reading:
  - The Linux Scheduler – a decade of wasted cores (Eurosys 2016)