
Lecture 11: Multiprocessor Scheduling

CSC 469H1F

Fall 2006

Angela Demke Brown



Announcements

- Assignment 1 has been posted (Due Oct. 31)
 - You are encouraged to work with a partner
- Assignment 0 - expect to return next Tuesday
- Test #1 next week Thursday (Oct. 26)
 - 2 hours, lecture + tutorial time slot
 - Covers material to end of this week (Lecture 12)
- Distinguished Lecture today, 3pm, BA 1170
 - Vint Cerf (Chief Internet Evangelist, Google)
 - "Internet in the 21st Century"

Multiprocessor Scheduling

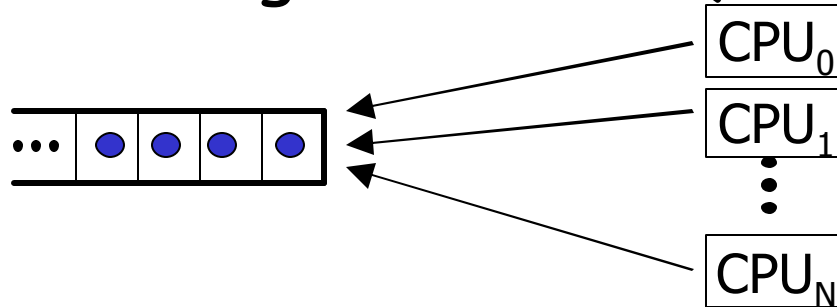
- Why use a multiprocessor?
 - To support multiprogramming
 - Large numbers of independent processes
 - Simplified administration
 - E.g. CDF wolves, compute servers
 - To support parallel programming
 - “job” consists of multiple cooperating/communicating threads and/or processes
 - Not independent!

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

Ready Queue Implementation

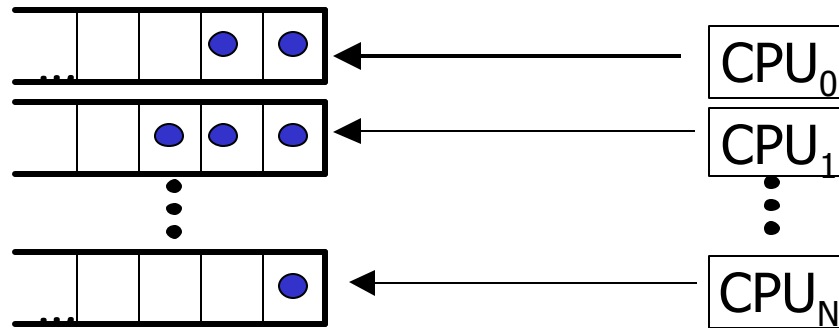
- Option 1: Single Shared Queue



- 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
 - Synchronization is needed

Ready Queue Implementation

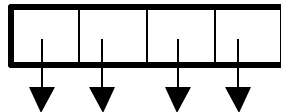
- Option 2: Per-CPU Ready Queue



- Scheduling code access queue for current CPU
- Issues
 - To which queue are new threads added?
 - What about unblocked threads?
 - Load balancing

Aside: Per-CPU data

- Assume shared-memory MP
- 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
 - `A[smp_processor_id()]` is data structure for current CPU
 - Often array contains just pointers



- 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
- Use padding so each per-CPU variable lies on different cache line

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

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

Symbiotic Scheduling

- Threads load data into cache
- Expect multiple threads to trash each others' state as they run
- Can try to detect cache needs and schedule threads that can share nicely on same CPU
 - 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

Parallel Job Scheduling

- “Job” is a collection of processes/threads that cooperate to solve some problem (or provide some service)
- How the components of the job are scheduled has a major effect on performance
- Two major strategies
 - Space sharing
 - Time sharing

Why Job Scheduling Matters

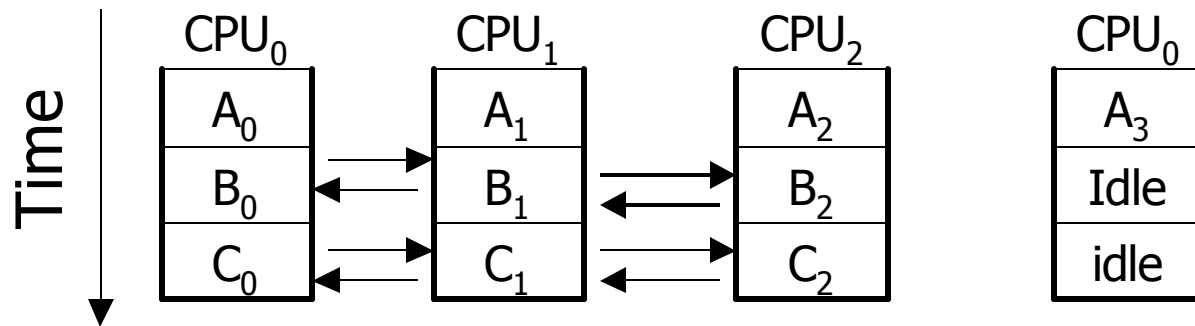
- Threads in a job are not independent
 - Synchronize over shared data
 - De-schedule lock holder, other threads in job may not get far
 - Cause/effect relationships (e.g. producer-consumer problem)
 - Consumer is waiting for data on queue, but producer is not running
 - Synchronizing phases of execution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

Space Sharing

- Divide processors into groups
 - Fixed, variable, or adaptive
- Assign job to dedicated set of processors
 - Ideally one CPU per thread in job
- Pros:
 - Reduce context switch overhead (no involuntary preemption)
 - Strong affinity
 - All runnable threads execute at same time
- 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

Time sharing

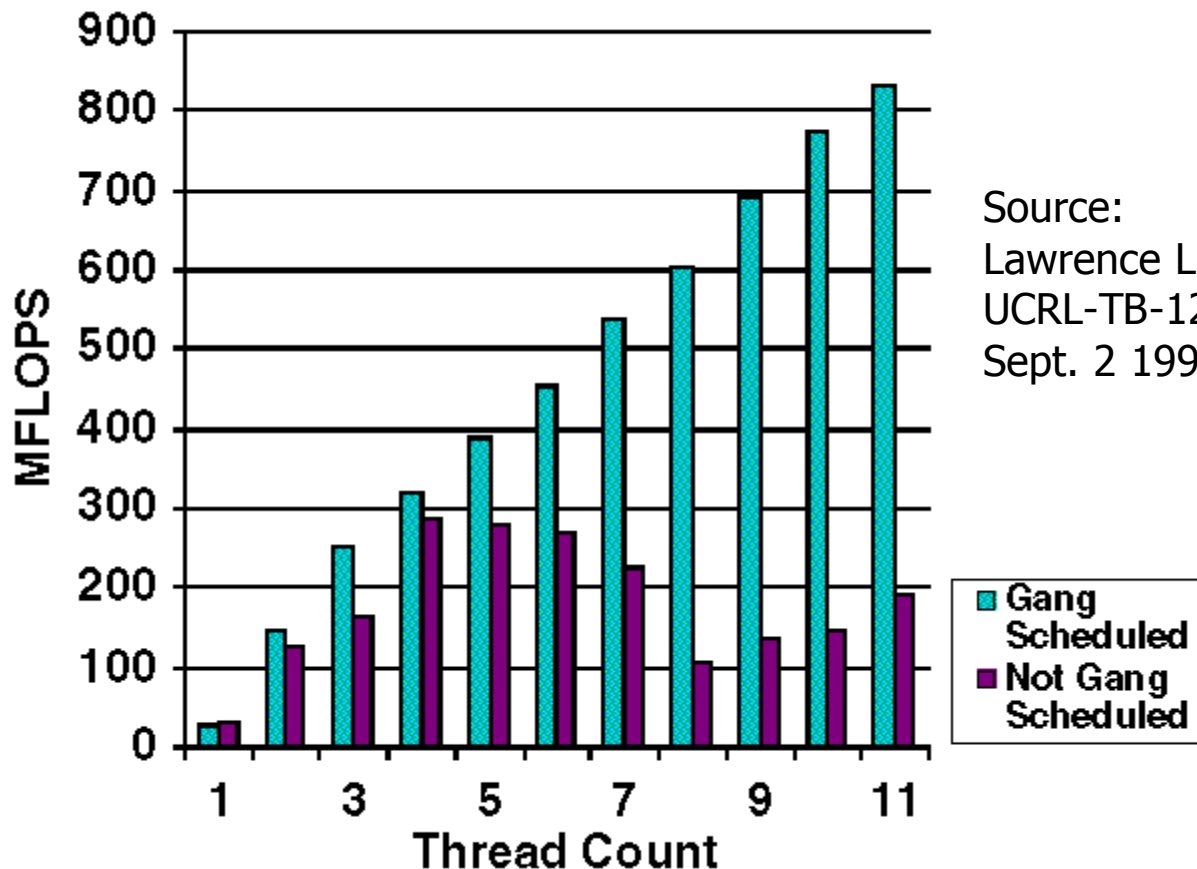
- Each CPU may run threads from multiple jobs
 - But with awareness of jobs
- Gang or Co-scheduling
 - All CPUs perform context switch together



- Bin packing problem to fill available CPU slots with runnable jobs

Example: Effect of Gang Scheduling

- LLNL gang scheduler on 12-CPU Digital Alpha 8400
 - Parallel gaussian elimination program
 - http://www.llnl.gov/asci/pse_trilab/sc98.summary.html



Source:
Lawrence Livermore Natl Lab
UCRL-TB-122379-Rev2
Sept. 2 1998