Announcements

- Assignment 0 – expect to return next Monday
- Test #1 next week Wednesday (Oct. 24)
  - 2 hours, 4-6 p.m. (lecture + 1 additional hour)
  - BA 2139 – NOTE LOCATION!
  - Covers material to end of this week (Lecture 11)
- Google Tech Talk, tomorrow, 4:30-6pm, BA 1180
  - Chris Colohan (Google Software Engineer)
  - “Internet-scale Computing”
  - Open to all graduate students

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

• Option 2: Per-CPU Ready Queue
  • Scheduling code accesses 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

Limits of FCFS (Space Sharing)

- 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

Solution: Backfilling

- Fill holes from queue in FCFS order
- Not FCFS anymore
- Want to prevent "fill" from delaying threads that were in queue earlier
  - EASY (Extensible Argonne Scheduling System)
    - Make reservation for next job in queue

Variations on Backfilling

- EASY
  - Used FCFS to order jobs in queue
  - Made reservation for first blocked job in queue
  - Backfilled jobs by looking at queue one at a time
- Ordering alternative: include priority in queue
  - Administrative to distinguish between users
  - User to distinguish between own jobs
  - Scheduler to prevent starvation
- 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
- Queue lookahead
  - Use dynamic programming to determine optimal packing