Chapter 10 Multiprocessor Scheduling

10.1 Background: Multiprocess Architecture

Fundamental difference between

- Single CPU HW
- Multi-CPU HW

Difference centers around

- Use of HW caches and
- How they are shared across multiple CPUs

What is a cache?

- Help CPU run programs faster
- Small, fast memory that hold copies of “popular” data
  - Whose original is in main memory of the system
  - Main memory is slow, large memory, hold all the data
- Make use of temporal and spatial locality

Q: What happens when you have multiple processors,
with a single shared main memory?

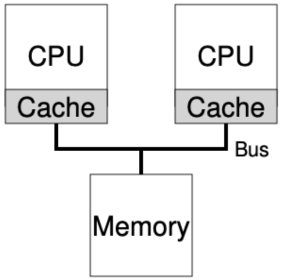


Figure 10.2: **Two CPUs With Caches Sharing Memory**

Example: Caching with multiple CPUs

- Program running on CPU1
  - Read data item (value D) at address A
  - Not in cache on CPU1
    - Fetch value D from main memory
  - Modify value at A, *only updating its cache to D’*
    - Writing data all the way to main memory is slow; usually done later
- OS stops running program: move program to CPU2
  - Read value at A
  - Not in cache on CPU2
    - Fetch from main memory
    - Gets value D instead of the correct value D’

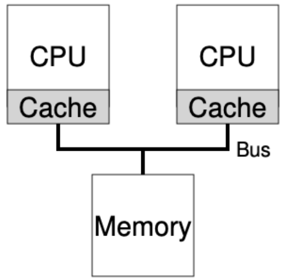


Figure 10.2: **Two CPUs With Caches Sharing Memory**

This problem is called: **cache coherence**

Solution is provided by the HW.

10.2 Don’t Forget Synchronize

Recall that accessing a shared data structure
from multiple threads (possibly on multiple CPUs)

- Need to add locks
- Problem:
  - Performance, as the number of CPUs grows
  - Access to synchronized shared data structures becomes quite slow!

10.3 One Final Issue: Cache Affinity

Issue:

- A process, when run on a particular CPU
- Builds up state in the caches (TLBs) of the CPU
- Hence, it is often advantageous to run the process on the same CPU
- Will run faster if some of its state is already present in the caches on that CPU

If process runs on a different CPU each time,
it will have to reload its state each time it runs.

This is not a safety issue:
It will still be correct due to cache coherence protocols built into the HW.

Multiprocessor scheduling should consider cache affinity,
when making scheduling decisions.

Try to keep a process on the same CPU if possible.

10.4 Single-Queue Multiprocessor Scheduling (SQMS)

Advantage: Simplicity

Put all jobs into a single queue
Policy: Pick X best jobs/processes to run (if there are X CPUs)

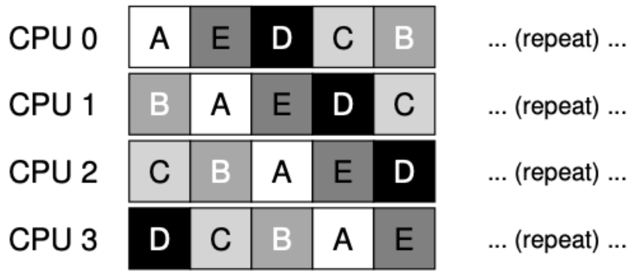
Disadvantage: Scalability

To ensure scheduler works correctly on multiple CPUs:

- Need to use locks to access the single queue
  - Problem #1: Reduces performance as the number of CPUs grow
    - Content for single lock
    - System spends time in lock overhead
    - Less time in doing real work
  - Problem #2: Cache affinity
    - Each CPU picks next job from globally-shared queue
    - Job can end up bouncing around from CPU to CPU
    - Doing exactly the opposite of what makes sense from a cache affinity perspective

Example: Five jobs (A, B, C, D, E). Four CPUs. Scheduling Queue looks this:

Jobs always switch CPU
No cache affinity advantage will be had.

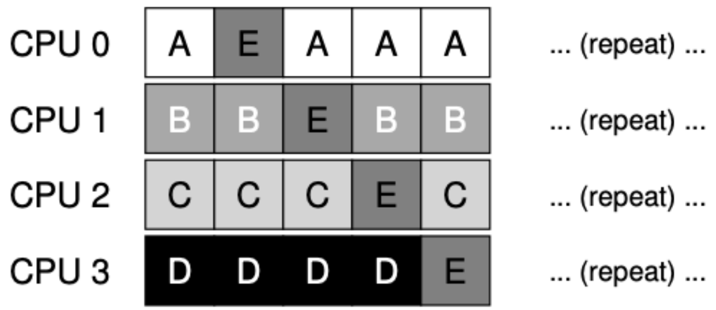


To handle this problem:

- Add affinity mechanism
  - Provide affinity for some jobs
  - Move around other jobs to balance the load

Example:

- Jobs A-D are not moved
- Job E migrate between CPUs
- Preserving affinity for most jobs



Can also migrate a different job next time — for better affinity fairness.

But this is complex.

10.5 Multi-Queue Multiprocessor Scheduling (MQMS)

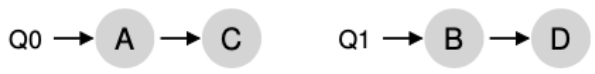
Multiple queues, e.g., one per CPU.

- Each queue follow a scheduling discipline, e.g., RR

When job arrive:

- Add job on exactly one queue
- Pick queue, e.g.,
  - Randomly
  - Queue with fewest jobs
  - Or some other heuristic
- Job scheduled independently
  - Avoiding problems of shared data structure needing synchronization

Example: Two CPUs, Two queues, Four jobs



With RR, we may get:

CPU 0	A	A	C	C	A	A	C	C	A	A	C	C	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...

Advantage:

- Scalable — more CPUs, more queues
  - Lock and cache contention should not become a problem
- Intrinsically provides cache affinity
  - Reusing cached content on each CPU

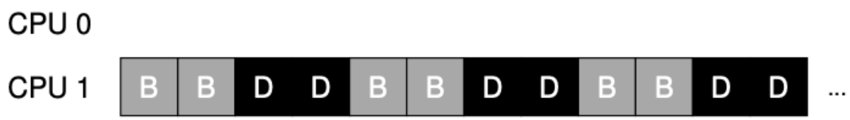
Problem:

- Load imbalance

Example: Four jobs, Two CPUs. Then job C finishes.

Now we get:

Next, consider that job A finishes. CPU 0 is left idle.



Q: How to handle load imbalance?

A: Obvious answer: move jobs around; technique: **migration**

Example: Move one of B or D to CPU 0 (for the last case)

Q0: B -> nil Q1: D -> nil

Example:

- First A is alone on CPU0
- B and D alternate on CPU1
- Then after some time slices:
  - B is moved to CPU0:
  - CPU0: A and B alternating
  - CPU1: D alone (for some time slices)

Continuous migration of one or more jobs. Keep switching jobs.

CPU 0	A	A	A	A	B	A	B	A	B	B	B	B	...
CPU 1	B	D	B	D	D	D	D	D	A	D	A	D	...

Q: How should the system decide to enact such migration?

Work Stealing

- A source queue with few jobs:
  - Peek at another target queue
  - If target queue is (notably) more full than source queue
    - Source queue will steal one or more jobs from the target queue

Black art policy:

- Find right threshold for how often to peek at another queue
- Tradeoff
  - Too frequent checks: high overhead — hurts scalability
  - Too infrequent: risk of sever load imbalances

Go runtime uses a work stealing algorithm to schedule goroutines.