8. Multi-CPU Scheduling

multicore processor

multiple CPU cores are packed onto a single chip

a typical application only uses a single CPU

adding more CPUs does not make that single application run faster.
=> have to rewrite your application to run in parallel, using threads

Multi-threaded applications can spread work across multiple CPUs and thus run faster when given more CPU resources.

HOW TO SCHEDULE JOBS ON MULTIPLE CPUs?

Single-CPU hardware vs. Multi-CPU hardware

: the use of hardware caches (CPU has its own cache) & how data is shared across multiple processors

Cache coherence problem

if there are multiple CPUs, coherence problem could happen when rewrite a data and read it

solution is provided by the hardware : monitoring memory access
ex. bus snooping
: each cache pays attention to memory updates by observing the bus that connects them to main memory.
=> When a CPU then sees an update for a data item it holds in its cache, it will notice the change and either

- invalidate its copy (remove it from its own cache) or
- update it (put the new value into its cache too)

Write-back caches make this more complicated...

Synchronization

Given that the caches do all of this work to provide coherence, do programs (or the OS itself) have to worry about anything when they access shared data? = YES

When accessing (especially, updating) shared data items or structures across CPUs, mutual exclusion primitives (ex. locks) are needed to guarantee correctness

Without locks, adding or removing elements from the queue concurrently will not work as expected, even with the underlying coherence protocols.

One needs locks to atomically update the data structure to its new state as the number of CPUs grows, access to a synchronized shared data structure becomes quite slow

Cache Affinity

a process, when run on a particular CPU, builds up a fair bit of state in the caches (and TLBs) of the CPU.
The next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU.

=> multiprocessor scheduler should consider cache affinity when making its scheduling decisions, preferring to keep a process on the same CPU if at all possible

Single-Queue Multiprocessor Scheduling (SQMS)

simply putting all jobs that need to be scheduled into a single queue

advantage : simplicity

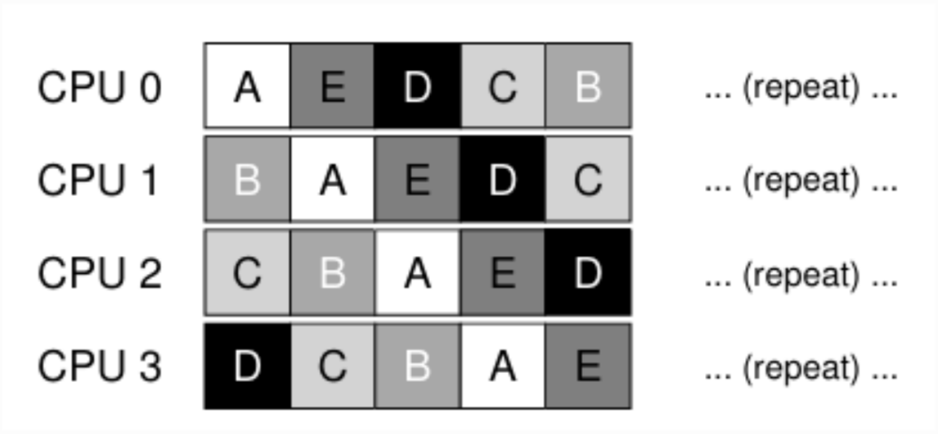
- does not require much work to take an existing policy that picks the best job to run next and adapt it to work on more than one CPU

disadvantage #1: lack of scalability Makes sure only one CPU modifies the queue at a time.

- to ensure the scheduler works correctly on multiple CPUs, need to insert locking into the code
- Locks can greatly reduce performance, particularly as the number of CPUs in the systems grows

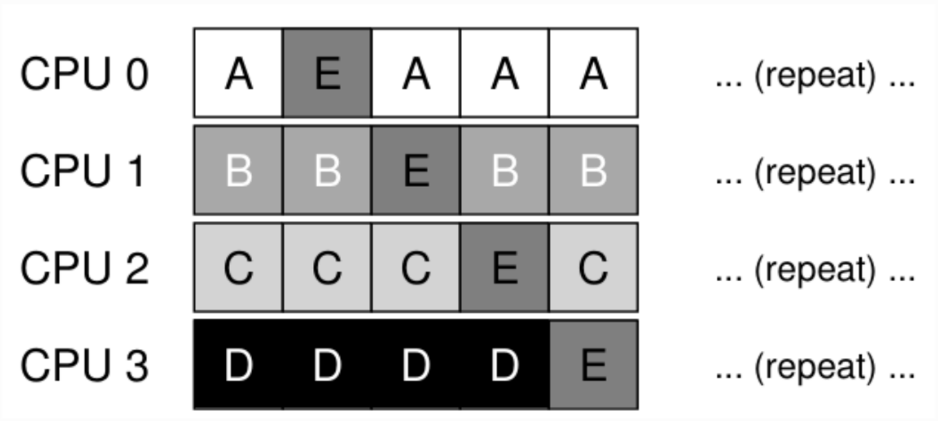
disadvantage #2: cache affinity

each job could ends up bouncing around from CPU to CPU, thus doing exactly the opposite of what would make sense from the standpoint of cache affinity.



=> Solution : include affinity mechanism to run on the same CPU if possible / can be complex

Ex) provide affinity for some jobs, but move others around to balance load



SQMS

- strengths
    - straightforward to implement given an exisiting single-CPU scheduler (has only a single queue)
- weaknesses
    - doesn't scale well (due to synchronization overheads)
    - doesn't readily preserve cache affinity

Multi-Queue Multiprocessor Scheduling (MQMS)
: multiple queues -> one per CPU

Procedure
1. a job enters the system
2. it is placed on exactly one scheduling queue, according to some heuristic
3. scheduled essentially independently, avoiding the problems of information sharing and synchronization found in the single-queue approach

inherently more scalable
as the number of CPUs grows, so too does the number of queues
=> lock and cache contention should not become a central problem

MQMS intrinsically provides cache affinity
= jobs stay on the same CPU

fundamental problem in the multi-queue based approach : **load imbalance**
=> HOW TO Deal With Load Imbalance? : Move jobs around = migration

by Migrating a job from on CPU to another, true load balance can be achieved

How should the system decide to enact such a migration?
: Work stealing

Procedure
1. a source queue that is low on jobs will occasionally peek at another target queue, to see how full it is.
2. If the target queue is notably more full than the source queue, the source will "Steal" one or more jobs from the target to help balance load

Tension

- If you look around at other queues too often -> high overhead & have trouble scaling (= the entire purpose of implementing the multiple queue scheduling)
  overhead **increases with the number of queues**
- If you don't look at other queues very often -> in danger of suffering from severe load imbalances.

=> Finding the right threshold remains, as is common in system policy design, a black art.

Linux Multiprocessor Schedulers

no common solution has emerged to building a multiprocessor scheduler

O(1) scheduler / Completely Fair Scheduler (CFS) / BF Scheduler (BFS)
Brain Fuck

O(1) & CFS : use multiple queues / BFS : uses a single queue
both approaches can be successful

O(1)

- priority-based scheduler (simliar to the MLFQ)
- changing a process's priority over time
- scheduling those with highest priority in order to meet various scheduling objectives
- interactivity is a particular focus
  is about "how the process behaves over time"
  $\updownarrow$

CFS

- Deterministic proportional-share (like Stride scheduling)

BFS

- only single-queue approach focus on simplicity
- proportional-share / based on a more complicated scheme known as Earliest Eligible Virtual Deadline First (EEVDF)
  to achieve good fairness and responsiveness.

Summary

SQMS

- Pros
  - straight forward to build
  - balances load well
- Cons
  - inherently has difficulty with
    - scaling to many processors
    - cache affinity

MQMS

- Pros
  - scales better
  - handles cache affinity well
- Cons
  - has trouble with load imbalance
  - more complicated

working set size : how much cache space it needs to run efficiently

How effectively a job runs on a particular CPU depends on whether the cache of that CPU currently holds the working set of the given job.
If it doesn't, the job runs slowly, which means that only 1 tick of its runtime is subtracted from its remaining time left per each tick of the clock.
This is the mode where the cache is 'cold' for that job (i.e., it does not yet contain the job's working set).
However, if the job has run on the CPU previously for 'long enough', that CPU cache is now 'warm', and the job will execute more quickly.