- multicore processor is where multiple CPU cores are packed onto a single chip
- There are issues that come with it:
    - A primary one is that a typical application (i.e., some C program you wrote) only uses a single CPU; adding more CPUs does not make that single application run faster.
    - To remedy this problem, you'll have to rewrite your application to run in parallel, perhaps using threads
- Multithreaded applications can spread work across multiple CPUs and thus run faster when given more CPU resources.
- How should the OS schedule jobs on multiple CPUs? What new problems arise? Do the same old techniques work, or are new ideas required?

Multiprocessor Architecture

- The difference between multiprocessor and single CPU hardware is around the use of hardware caches and exactly how data is shared across multiple processors.
- In a system with a single CPU, there are a hierarchy of hardware caches that in general help the processor run programs faster.
    - **Caches are small, fast memories that (in general) hold copies of popular data that is found in the main memory of the system.**
    - **Main memory, in contrast, holds all of the data, but access to this larger memory is slower. By keeping frequently accessed data in a cache, the system can make the large, slow memory appear to be a fast one.**
    - For example:
        - consider a program that issues an explicit load instruction to fetch a value from memory, and a simple system with only a single CPU. the CPU has a small cache (say 64 KB) and a large main memory
        - The first time a program issues this load, the data resides in main memory, and thus takes a long time to fetch (perhaps in the tens of nanoseconds, or even hundreds).
        - The processor, anticipating that the data may be reused, puts a copy of the loaded data into the CPU cache.
        - If the program later fetches this same data item again, the CPU first checks for it in the cache; if it finds it there, the data is fetched much more quickly (say, just a few nanoseconds), and thus the program runs faster.
- Caches are thus based on the notion of locality, of which there are two kinds: temporal locality and spatial locality.
    - The idea behind **temporal locality** is that when a piece of data is accessed, it is likely to be accessed again in the near future; imagine variables or even instructions themselves being accessed over and over again in a loop.
    - The idea behind **spatial locality** is that if a program accesses a data item at address x, it is likely to access data items near x as well;
- what happens when you have multiple processors in a single system, with a single shared main memory?
    - Caching with multiple CPUs is much more complicated.

    - Cache coherence is a situation where multiple processor cores share the same memory hierarchy, but have their own L1 data and instruction caches. Incorrect execution could occur if two or more copies of a given cache block exist, in two processors' caches, and one of these blocks is modified.
- The basic solution is provided by the hardware: by monitoring memory accesses, hardware can ensure that basically the "right thing" happens and that the view of a single shared memory is preserved.
    - One way to do this on a bus-based system is to use an old technique known as 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 (i.e., remove it from its own cache) or update it (i.e., put the new value into its cache too).

Don't Forget 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?
- When accessing (and in particular, updating) shared data items or structures across CPUs, mutual exclusion primitives (such as locks) should likely be used to guarantee correctness (other approaches, such as building lock-free data structures, are complex and only used on occasion).
- For example, assume we have a shared queue being accessed on multiple CPUs concurrently.
    - 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.
- Locks are methods of synchronization used to prevent multiple threads from accessing a resource at the same time
- As the number of CPUs grows, access to a synchronized shared data structure becomes quite slow.

One Final Issue: Cache Affinity

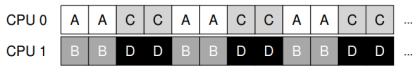
- One final issue arises in building a multiprocessor cache scheduler, known as cache affinity
- a process, when run on a particular CPU, builds up a fair bit of state in the caches (and TLBs) of the CPU.
    - The next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU. If, instead, one runs a process on a different CPU each time, the performance of the process will be worse, as it will have to reload the state each time it runs (note it will run correctly on a different CPU thanks to the cache coherence protocols of the hardware).
- Thus, a multiprocessor scheduler should consider cache affinity when making its scheduling decisions, perhaps preferring to keep a process on the same CPU if at all possible.

Single-Queue Scheduling

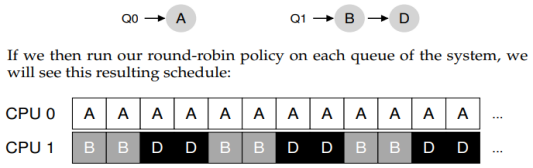
- To build a scheduler for a multiprocessor system,the most basic approach is to put all jobs that need to be scheduled into a single queue; we call this single queue multiprocessor scheduling or SQMS for short. This approach is simple.
- SQMS has obvious shortcomings. The first problem is a lack of scalability. To ensure the scheduler works correctly on multiple CPUs, the developers will have inserted some form of locking into the code. Locks ensure that when SQMS code accesses the single queue (say, to find the next job to run), the proper outcome arises.
- Locks, unfortunately, can greatly reduce performance, particularly as the number of CPUs in the systems grows
- As contention for such a single lock increases, the system spends more and more time in lock overhead and less time doing the work the system should be doing
- The second main problem with SQMS is cache affinity. For example, let us assume we have five jobs to run (A, B, C, D, E) and four processors.
    - Because each CPU simply picks the next job to run from the globally shared queue, each job ends up bouncing around from CPU to CPU, thus doing exactly the opposite of what would make sense from the standpoint of cache affinity.
    - To handle this problem, most SQMS schedulers include some kind of affinity mechanism to try to make it more likely that the process will continue to run on the same CPU if possible. Specifically, one might provide affinity for some jobs, but move others around to balance load.
- Thus, we see that SQMS has pros and cons; it is easy to implement but it does not scale well due to synchronization overheads, and it does not preserve cache affinity.

Multi-Queue Scheduling

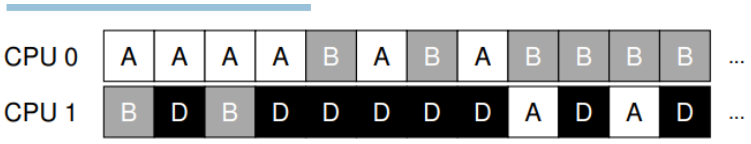
- Because of the problems caused in single-queue schedulers, some systems opt for multiple queues, e.g., one per CPU. We call this approach multi-queue multiprocessor scheduling (or MQMS).
- Each queue will likely follow a particular scheduling discipline, such as round robin, though any algorithm can be used.
- When a job enters the system, it is placed on exactly one scheduling queue, according to some heuristic (e.g., random, or picking one with fewer jobs than others). Then it is scheduled essentially independently, thus avoiding the problems of information sharing and synchronization found in the single-queue approach.
- MQMS has a distinct advantage of SQMS in that it should be inherently more scalable.
    - As the number of CPUs grows, so too does the number of queues, and thus lock and cache contention should not become a central problem.
    - In addition, MQMS intrinsically provides cache affinity; jobs stay on the same CPU and thus reap the advantage of reusing cached contents therein.
- However, we have a new problem, which is fundamental in the multi-queue based approach: load imbalance
    - For example, in CPU 1 we have jobs A and C and in CPU 2 we have jobs B and D. With Round Robin scheduling, the system might produce something like this:



- Let's assume we have the same set up as above (four jobs, two CPUs), but then one of the jobs (say C) finishes. We now have the following scheduling queues:



  - As we can see from this diagram, A gets twice as much CPU as B and D, which is not the desired outcome.
  - If jobs A and C both finish, CPU 0 will be sitting idle while jobs B and D are running.
- How should a multi-queue multiprocessor scheduler handle load imbalance, so as to better achieve its desired scheduling goals?
  - The answer to this query is to move jobs around, a technique which we refer to as migration. By migrating a job from one CPU to another, true load balance can be achieved.
  - From the previous example, if both jobs A and C finish and CPU 0 is sitting idle, we can just move job B or D to CPU 0.
  - Let's say however that only job C finishes so job A is still running on CPU 0 while B and D are running on CPU 1.
    - In this case, a single migration does not solve the problem.
    - One possible solution is to keep switching jobs, as we see in the following timeline. In the figure, first A is alone on CPU 0, and B and D alternate on CPU 1. After a few time slices, B is moved to compete with A on CPU 0, while D enjoys a few time slices alone on CPU 1. And thus load is balanced



- How should the system decide to enact such a migration?
  - One basic approach is to use a technique known as work stealing
  - With a work-stealing approach, a (source) queue that is low on jobs will occasionally peek at another (target) queue, to see how full it is. 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.
  - If you look around at other queues too often, you will suffer from high overhead and have trouble scaling
  - If, on the other hand, you don't look at other queues very often, you are in danger of suffering from severe load imbalances
  - Finding the right threshold remains, as is common in system policy design, a black art

Summary

- The single-queue approach (SQMS) is rather straightforward to build and balances load well but inherently has difficulty with scaling to many processors and cache affinity.
- The multiple-queue approach (MQMS) scales better and handles cache affinity well, but has trouble with load imbalance and is more complicated.