# Operating Systems

## Multiprocessor Scheduling

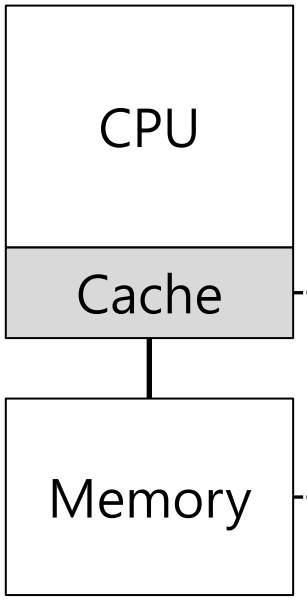
# Multiprocessor Scheduling

- The rise of the multicore processor is the source of multiprocessor-scheduling proliferation.
  - **Multicore**: Multiple CPU cores are packed onto a single chip.

- Adding more CPUs does not make that single application run faster.
  → You'll have to rewrite application to run in parallel, using **threads**.

**How to schedule jobs on Multiple CPUs?**

# Single CPU with cache




**Cache**

- Small, fast memories
- Hold copies of popular data that is found in the main memory.
- Utilize *temporal* and *spatial* locality

**Main Memory**

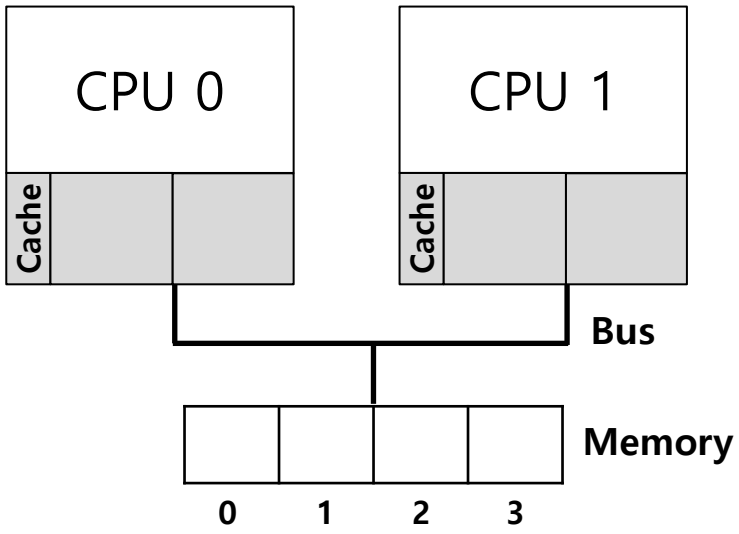
- Holds all of the data
- Access to main memory is slower than cache.

**By keeping data in cache, the system can make slow memory appear to be a fast one**

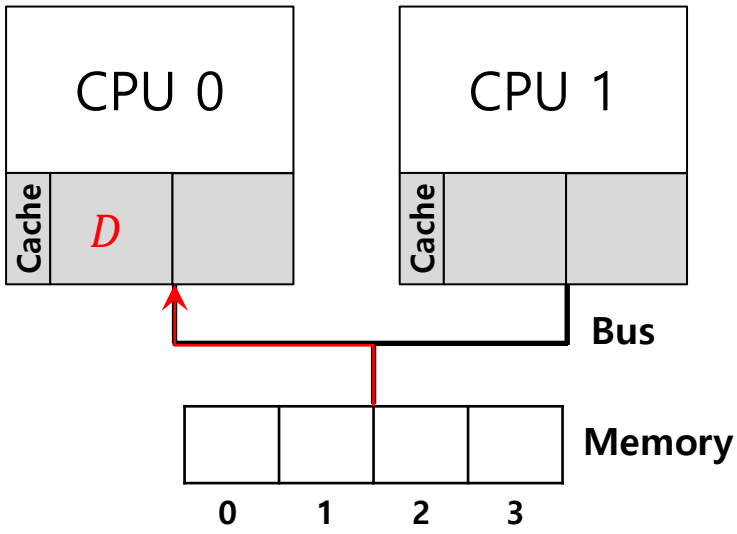
# Cache coherence

- Consistency of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory

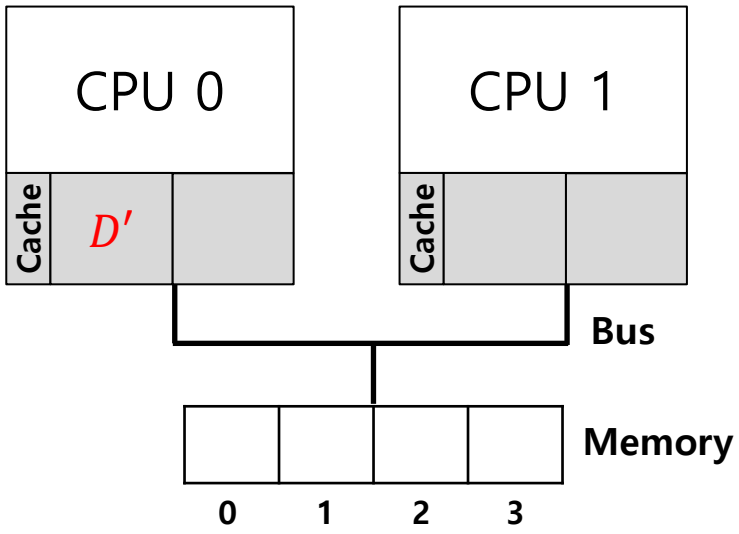



1. CPU0 reads a data at address 1.

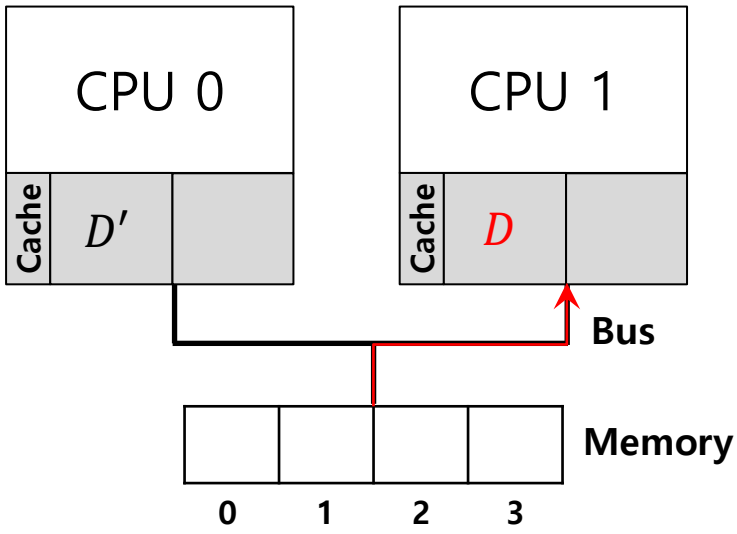


# Cache coherence (Cont.)

2. $D$ is updated and CPU1 is scheduled.




3. CPU1 re-reads the value at address A




**CPU1 gets the old value $D$ instead of the correct value $D'$.**

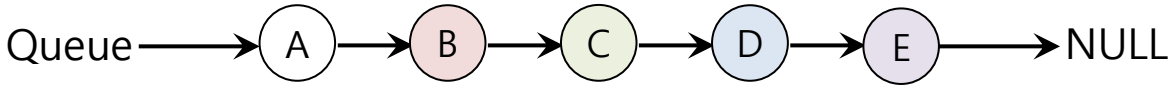
# Cache Affinity

- Keep a process on the same CPU if at all possible
  - A process builds up a fair bit of state in the cache of a CPU.
  - The next time the process run, it will run faster if some of its state is *already present* in the cache on that CPU.

**A multiprocessor scheduler should consider cache affinity when making its scheduling decision.**

# Single queue Multiprocessor Scheduling (SQMS)

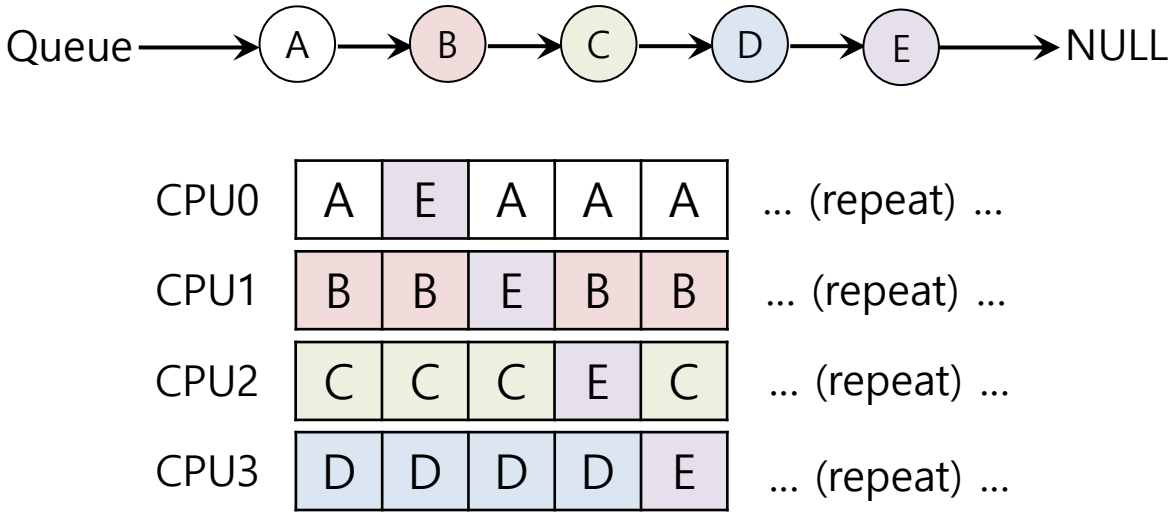
- Put all jobs that need to be scheduled into a single queue.
  - Each CPU simply picks the next job from the globally shared queue.
  - Cons:
    - Some form of **locking** have to be inserted → Lack of scalability
    - Cache affinity
    - Example:




    - Possible job scheduler across CPUs:

| | | | | | | |
|---|---|---|---|---|---|---|
| CPU0 | A | E | D | C | B | ... (repeat) ... |
| CPU1 | B | A | E | D | C | ... (repeat) ... |
| CPU2 | C | B | A | E | D | ... (repeat) ... |
| CPU3 | D | C | B | A | E | ... (repeat) ... |

# Scheduling Example with Cache affinity



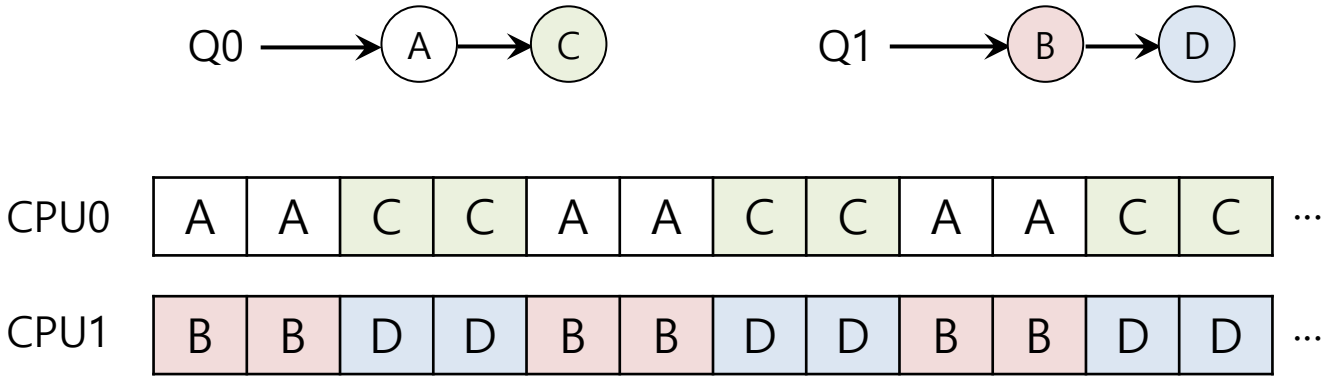

- Preserving affinity for most
  - Jobs A through D are not moved across processors.
  - Only job E Migrating from CPU to CPU.
- Implementing such a scheme can be **complex**.

# Multi-queue Multiprocessor Scheduling (MQMS)

- MQMS consists of multiple scheduling queues.
  - Each queue will follow a particular scheduling discipline.
  - When a job enters the system, it is placed on **exactly one** scheduling queue.
  - OS has to decide into which queue to place each job
  - Avoid the problems of information sharing and synchronization.

# MQMS Example

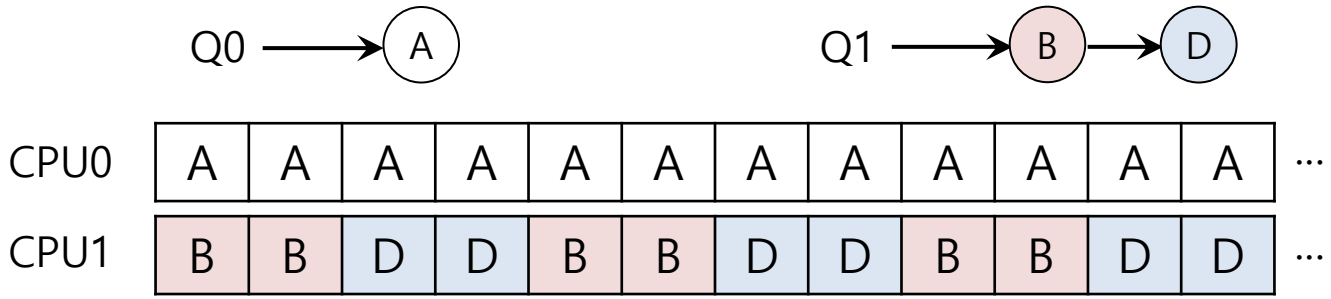
- With **round robin**, the system might produce a schedule that looks like this:




**MQMS provides more scalability and cache affinity.**

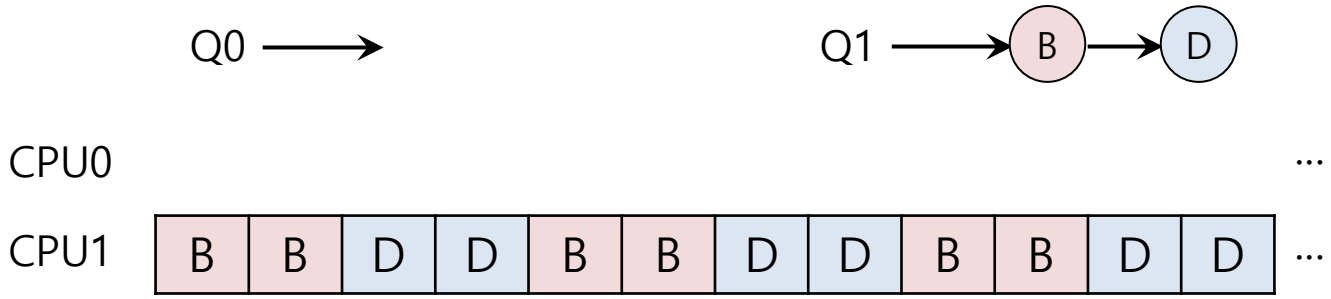
# Load Imbalance issue of MQMS

- After job C in Q0 finishes:




**A gets twice as much CPU as B and D.**

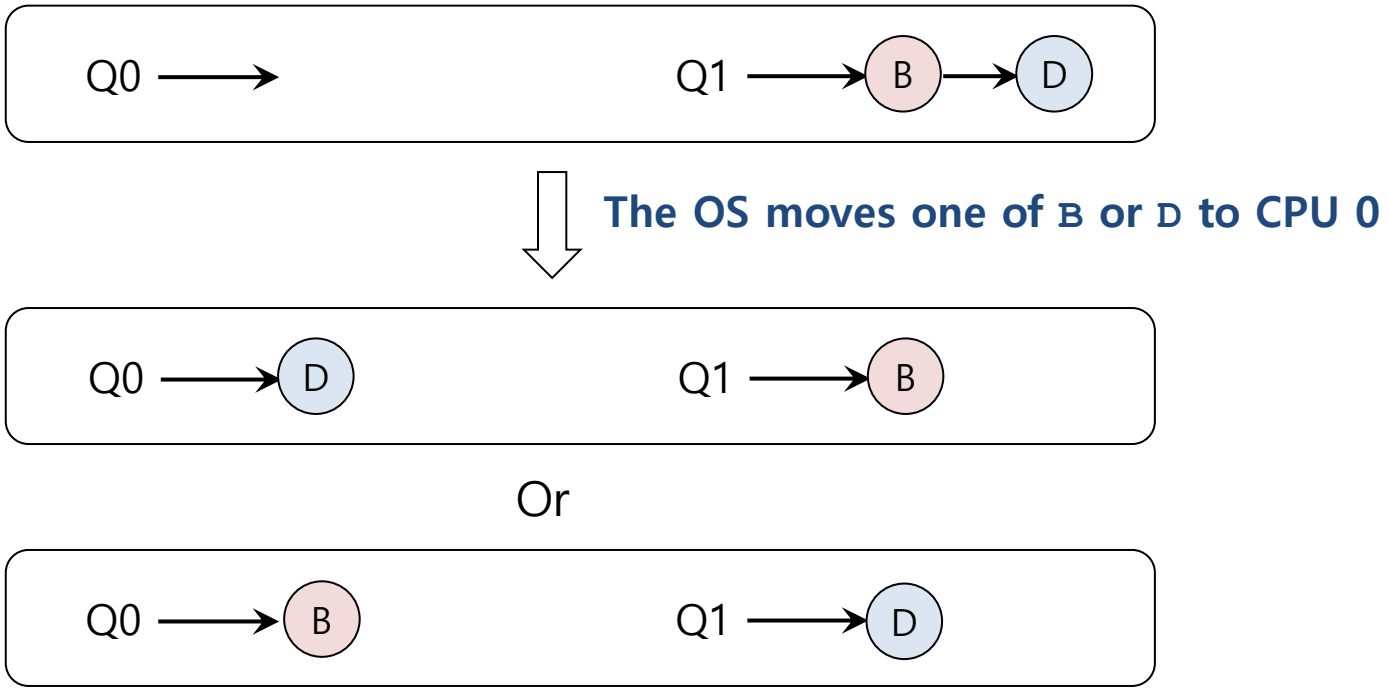
- After job A in Q0 finishes:




**CPU0 will be left idle!**

# How to deal with load imbalance?

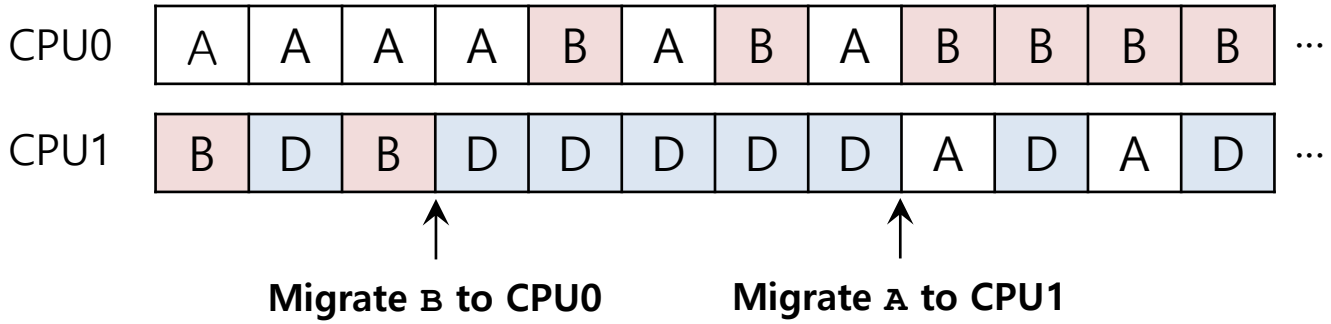
- The answer is to move jobs (**Migration**).
  - Example:



# How to deal with load imbalance? (Cont.)

- A more tricky case:


- A possible migration pattern:
  - Keep switching jobs



# Work Stealing

- Move jobs between queues
  - Implementation:
    - A source queue that is low on jobs is picked.
    - The source queue occasionally peeks at another target queue.
    - If the target queue is more full than the source queue, the source will "**steal**" one or more jobs from the target queue.
  - Cons:
    - *High overhead* and trouble *scaling*