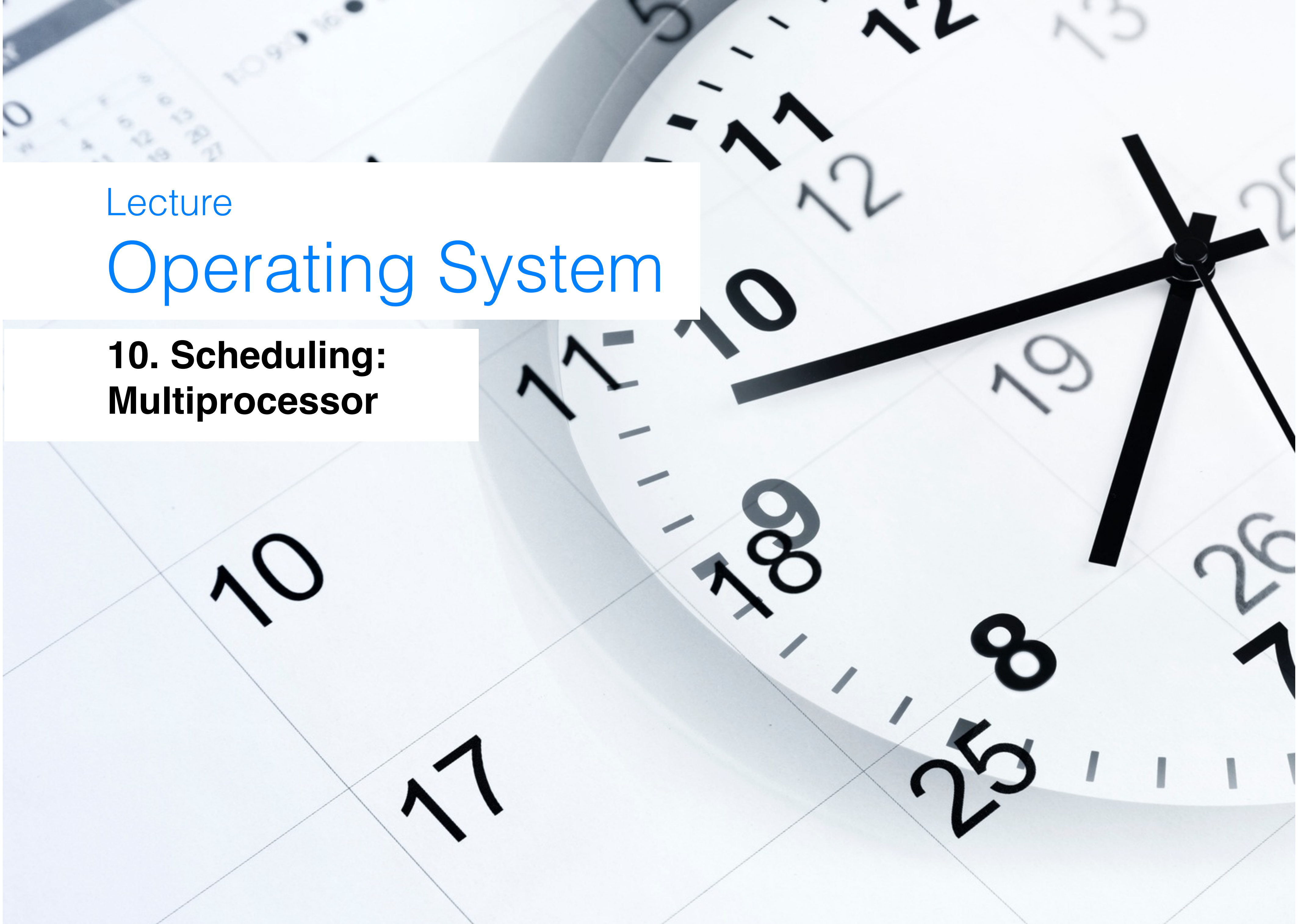


Lecture

Operating System

10. Scheduling: Multiprocessor

Professor Dr. Michael Mächtel

10. Scheduling: Multiprocessor

- **How should the OS schedule jobs on multiple CPUs?**
- **What new problems arise?**
- **Do the same old techniques work, or are new ideas required?**

10. Scheduling: Multiprocessor

1. Multiprocessor Architecture

2. Multi-queue Multiprocessor Scheduling (MQMS)

3. Example: Linux

10. Scheduling: Multiprocessor

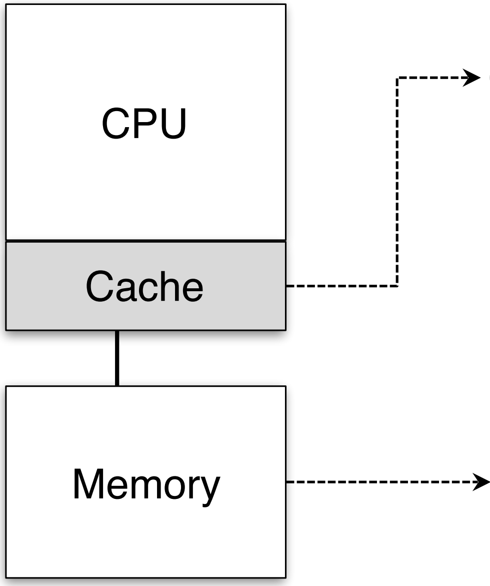
1. **Multiprocessor Architecture**
2. **Multi-queue Multiprocessor Scheduling (MQMS)**
3. **Example: Linux**

Multiprocessor

- 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

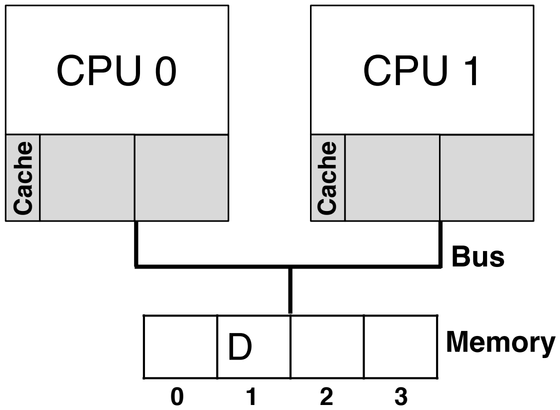
Main Memory

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

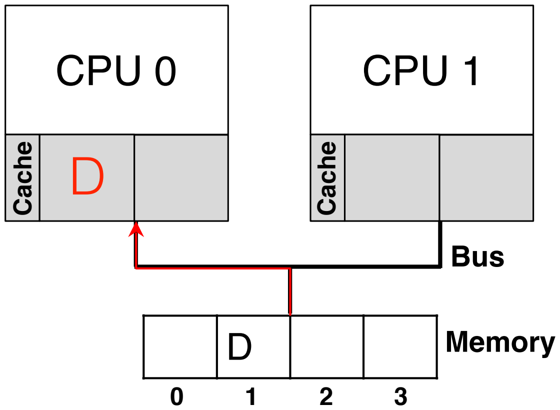
Cache coherence

- Consistency of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory

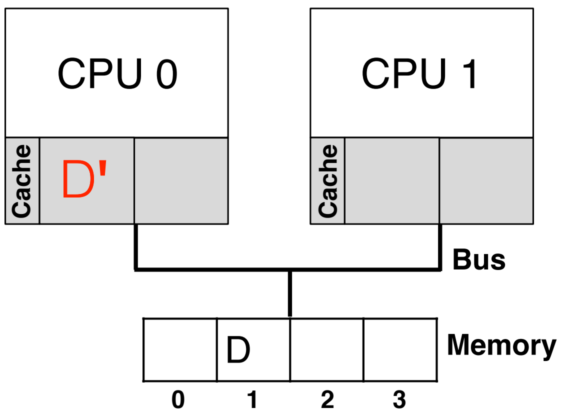


1. CPU0 reads a data at address 1.

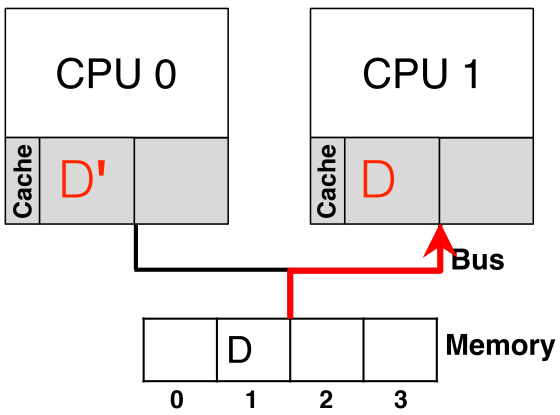


Cache coherence

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



3. **CPU1** re-reads the value at address 1



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

Cache coherence solution

- **Bus snooping**
  - Each cache pays attention to memory updates by **observing the bus.**
  - When a CPU sees an update for a data item it holds in its cache, it will notice the change and either *invalidate* its copy or *update* it.

Don’t forget synchronization

- When accessing shared data across CPUs, mutual exclusion primitives should likely be used to **guarantee correctness**.
- Simple list delete code:

```
typedef struct __Node_t {
    int value;
    struct __Node_t *next;
} Node_t;

int List_Pop() {
    Node_t *tmp = head;         // remember old head ...
    int value = head->value;    // ... and its value
    head = head->next;          // advance head to next pointer
    free(tmp);                  // free old head
    return value;               // return value at head
}
```

Don’t forget synchronization

- Solution

```
pthread_mtuex_t m;
typedef struct __Node_t {
    int value;
    struct __Node_t *next;
} Node_t;

int List_Pop() {
    lock(&m)
    Node_t *tmp = head;         // remember old head ...
    int value = head->value;    //  ... and its value
    head = head->next;          // advance head to next pointer
    free(tmp);                  // free old head
    unlock(&m)
    return value;               // return value at head
}
```

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.

10. Scheduling: Multiprocessor

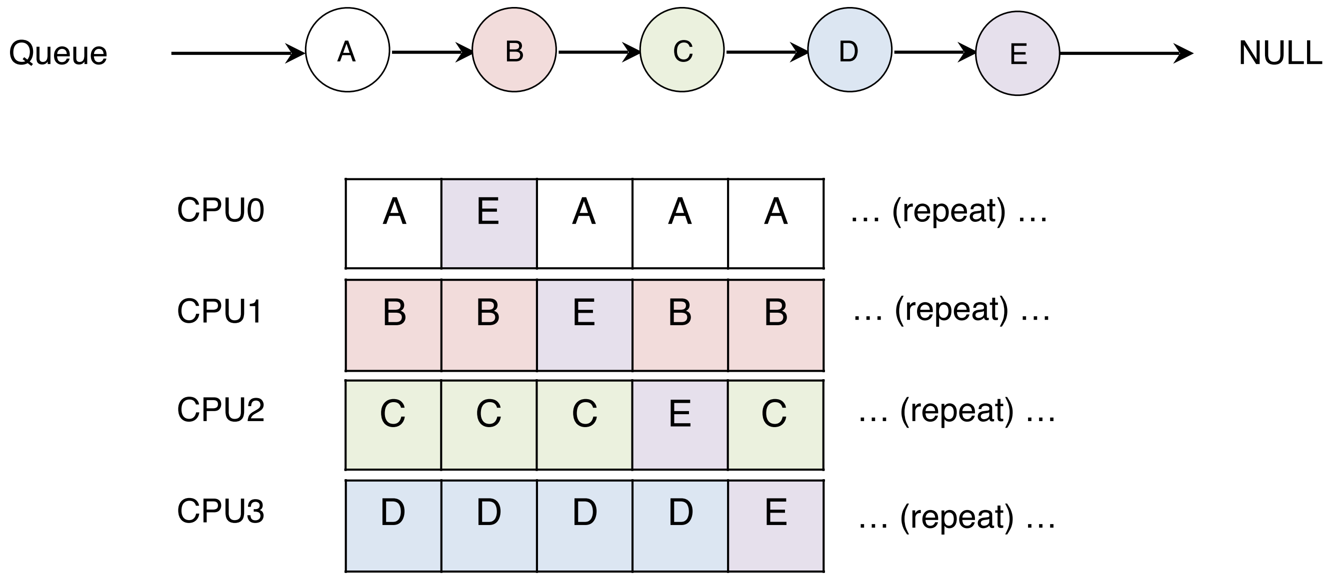
1. Multiprocessor Architecture
2. **Multi-queue Multiprocessor Scheduling (MQMS)**
3. Example: Linux

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: Queue → A → B → C → D → E → NULL
    - 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



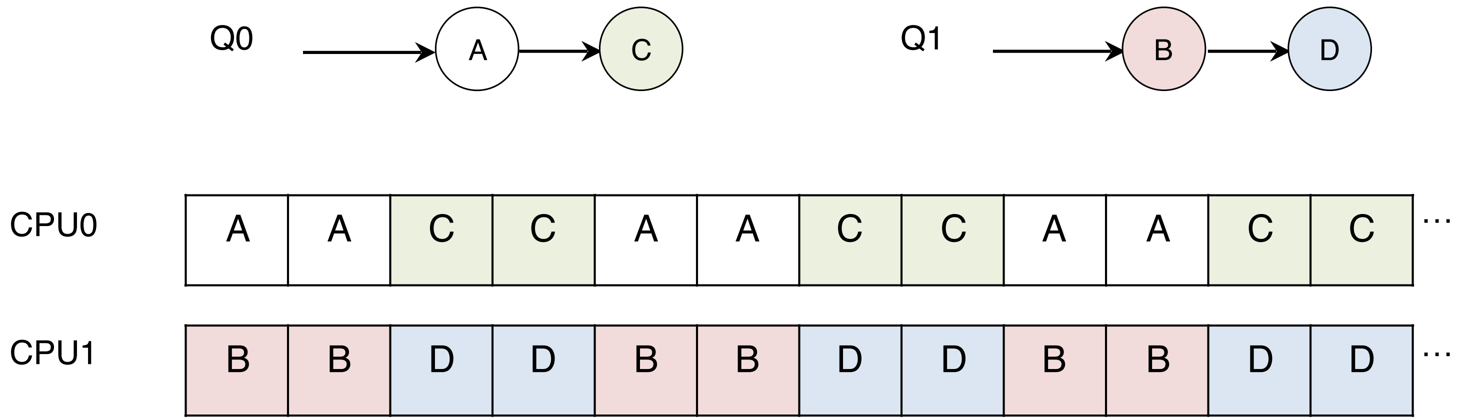
- Preserving affinity for most
  - Jobs A through D are not moved across processors.
  - Only job e Migrating from CPU to CPU.
- Implementing such a scheme can be **complex**.

MQMS Multi-queue Multiprocessor Scheduling

- 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.
  - Avoid the problems of *information sharing* and *synchronization*.

MQMS Example

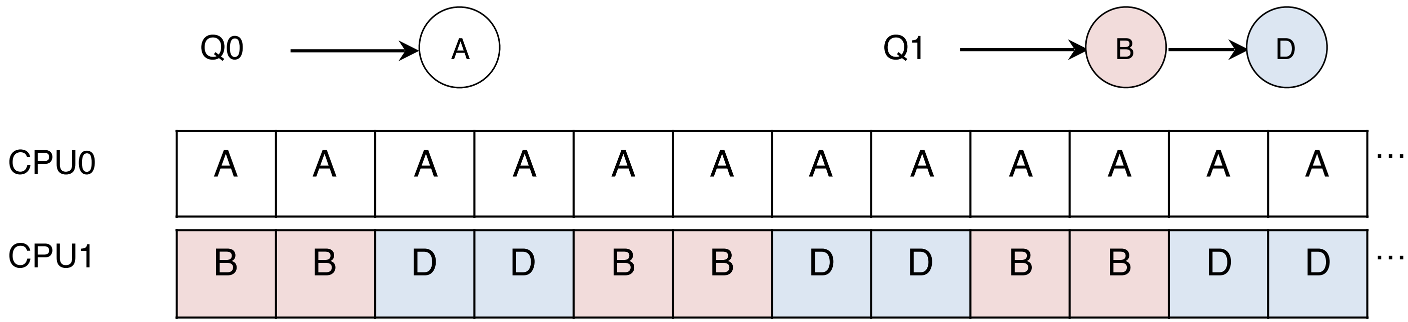
- With **round robin**, the system might produce a schedule that looks like this:



MQMS provides more scalability and cache affinity.

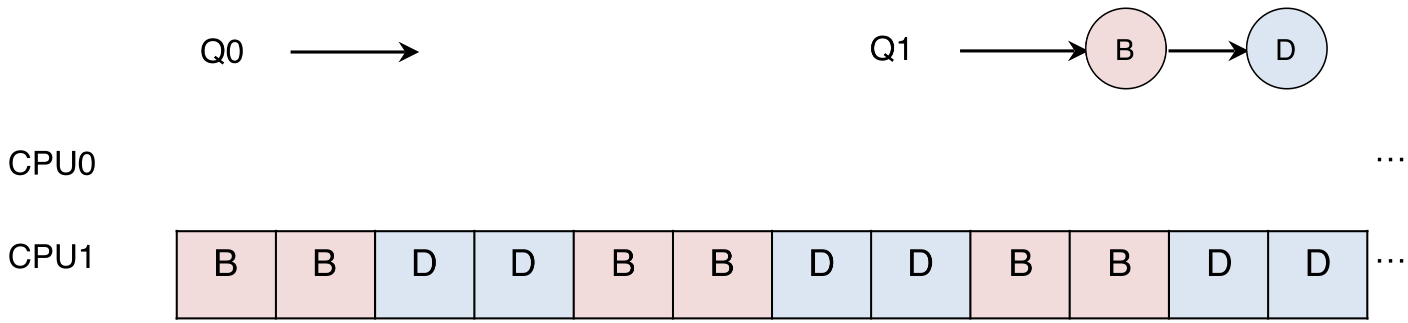
Load Imbalance issue of MQMS

- After job C in Q0 finishes:



A gets twice as much CPU as B and D.

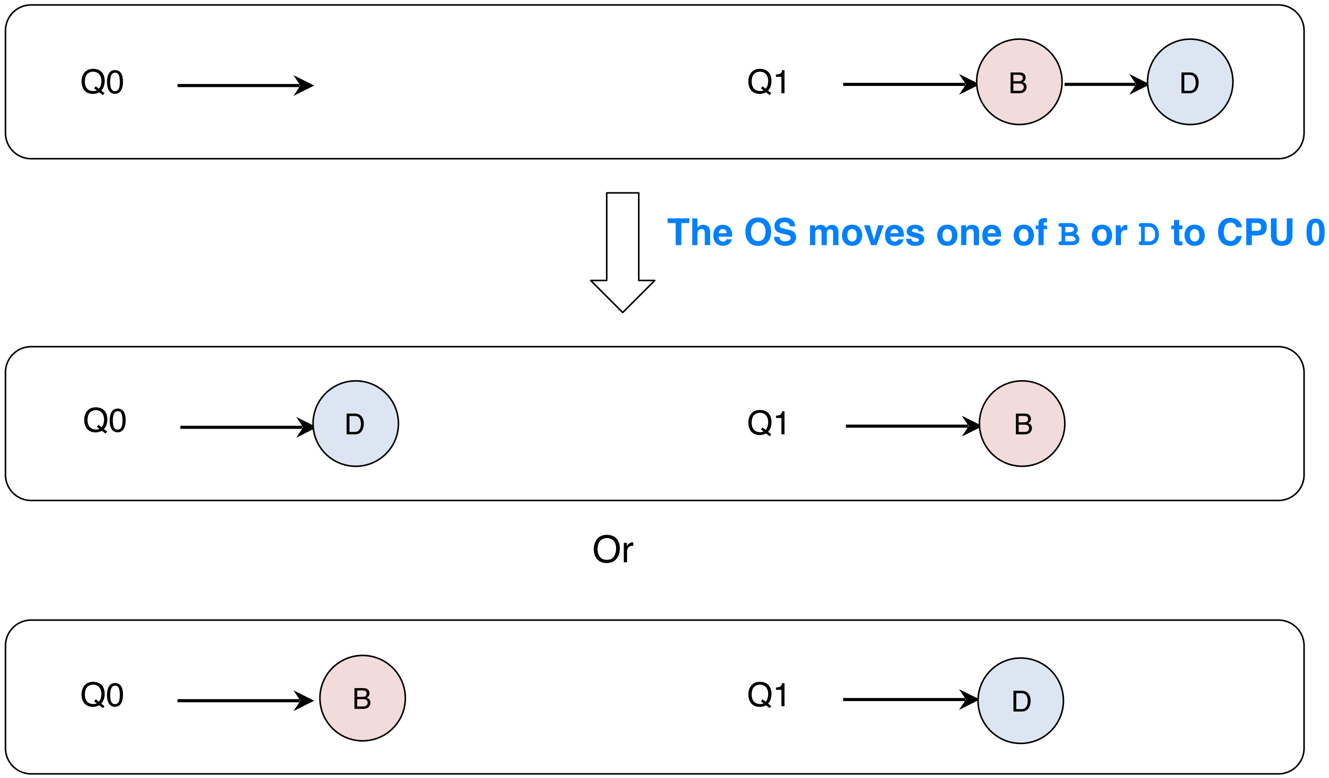
- After job A in Q0 finishes:



CPU0 will be left idle!

How to deal with load imbalance?

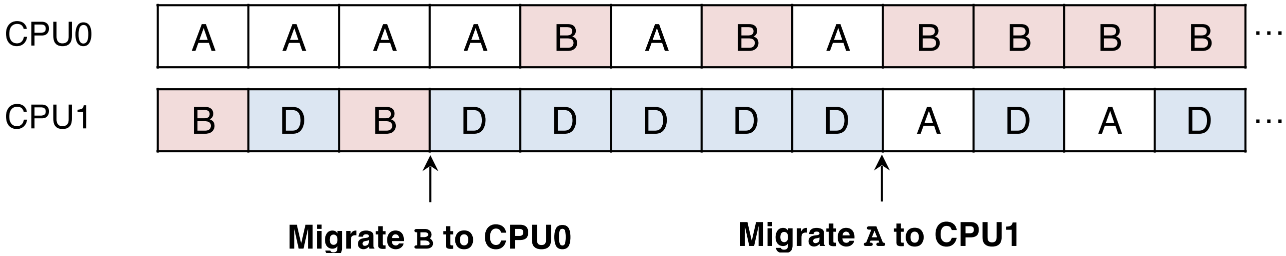
- The answer is to move jobs (Migration).
  - Example:



How to deal with load imbalance?

- A more tricky case:

- A possible migration pattern:
  - Keep switching jobs



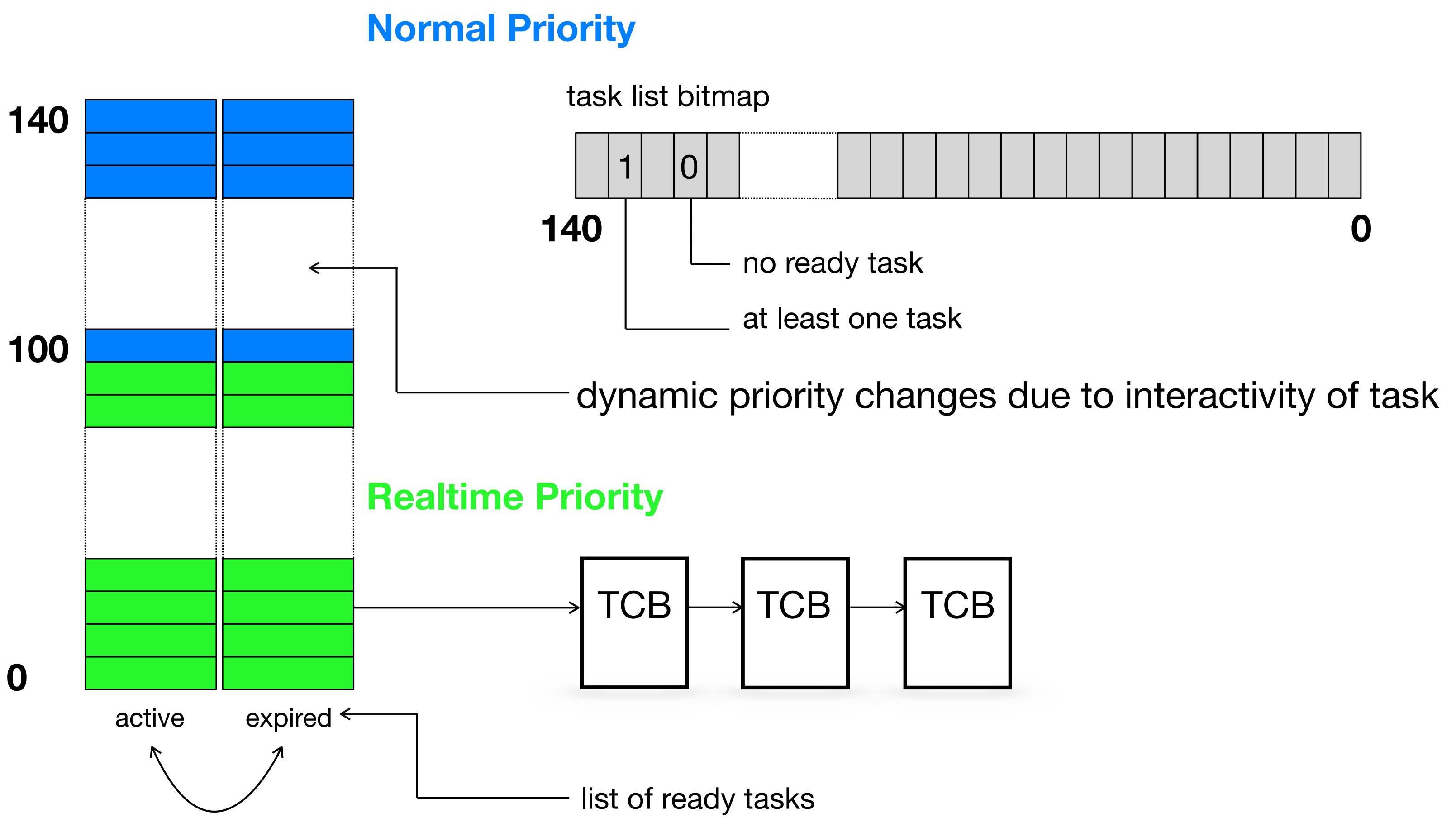
Work Stealing

- 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*

10. Scheduling: Multiprocessor

1. Multiprocessor Architecture
2. Multi-queue Multiprocessor Scheduling (MQMS)
3. **Example: Linux**

Linux O1 Scheduling



Linux Multiprocessor Schedulers

- O(1)
  - A Priority-based scheduler
  - Use Multiple queues
  - Change a process's priority over time
  - Schedule those with highest priority
  - Interactivity is a particular focus

Linux Multiprocessor: Load Balancer

- Goal: No CPU must have too many or too few tasks to run
- When is load balancer called (on each cpu)?
  - If ready queue of cpu is empty
  - If cpu is idle
  - Every 200 ms
- Source Code:

```
struct runqueue {
    spinlock_t lock;
    unsigned long nr_running;
    #ifdef CONFIG_SMP
    unsigned long cpu_load[3];
    #endif
    unsigned long long nr_switches;
     ...
```

How does the Load Balancer proceed?

1. Lock all queues (ordering matters)
2. Find_busiest_queue() of other cpus
   - at least 25% more
3. Loop
   - Select highest priorty task from the expired queue
   - Check if selection is:
     - not cache hot
     - not pinned to cpu by processor affinity
   - exit loop or start over with second highest priorty task ...
4. Move selection to own cpu
5. Repeat 2-4, until all queues are balances
6. Unlock all queues

Linux: /proc/schedstat (1)

```
    873          sys_sched_yield()
    163( 18.67%) found (only) expired queue empty on current cpu
    657( 75.26%) found both queues empty on current cpu
     53(  6.07%) found neither queue empty on current cpu

3743792173          schedule()
3739822176( 99.89%) switched active and expired queues
         0(  0.00%) used existing active queue
1096525548( 29.29%) scheduled no process (left cpu idle)

2372524899              try_to_wake_up()
2014921771( 84.93%)     task being awakened was last on same cpu as waker
 357603128( 15.07%)     task being awakened was last on different cpu than waker
   4609165(  1.29%)     moved that task to the waking cpu because it was
                        cache-cold
352993960 ( 98.71%)     didn't move that task
0.09/0.04               avg runtime/latency over all cpus (ms)

7228571          tasks pulled by pull_task()
 120681(  1.67%) pulled from hot cpu while still cache-hot and idle
5582178( 77.22%) pulled from cold cpu while idle
    157(  0.00%) pulled from hot cpu while still cache-hot and busy
  36615(  0.51%) pulled from cold cpu while busy
  22657(  0.31%) pulled from hot cpu while still cache-hot and newly idle
1466283( 20.28%) pulled from cold cpu when newly idle
```

Linux: /proc/schedstat (2)

```
3218396223                load_balance()
2115369831( 65.73%)       called while idle
   2569563(  0.12%)       tried but failed to move any tasks
       656(  0.00%)       found no busier queue
2107297807( 99.62%)       found no busier group
   5501805(  0.26%)       succeeded in moving at least one task
                          (average imbalance:   152.1)
5041850(  0.16%)           called while busy
   6882(  0.14%)          tried but failed to move any tasks
5000026( 99.17%)          found no busier group
  34942(  0.69%)          succeeded in moving at least one task
                          (average imbalance:   185.1)
1097984542( 34.12%)       called when newly idle
  10838096(  0.99%)       tried but failed to move any tasks
       291(  0.00%)       found no busier queue
1085687222( 98.88%)       found no busier group
   1458933(  0.13%)       succeeded in moving at least one task
                          (average imbalance:   0.407)

74479                     active_load_balance() was called
74479                     active_load_balance() tried to push a task
74057                     active_load_balance() succeeded in pushing a task
```

Linux Multiprocessor Schedulers

- Completely Fair Scheduler (CFS)
  - Deterministic proportional-share approach
  - Multiple queues
- BF Scheduler (BFS)
  - A single queue approach
  - Proportional-share
  - Based on Earliest Eligible Virtual Deadline First(EEVDF)

Thanks

Questions?