Advanced Scheduling Schemes

Proportional Share Scheduler

- Fair-share scheduler
  - Guarantee that each job obtain *a certain percentage* of CPU time
  - Not optimized for turnaround or response time

Lottery scheduling

- Tickets
  - Represent the share of a resource that a process should receive
  - The percent of tickets represents its share of the system resource in question

- Example
  - There are two processes, A and B
    - Process A has 75 tickets → receive 75% of the CPU
    - Process B has 25 tickets → receive 25% of the CPU

Lottery scheduling

- The scheduler picks <u>a winning ticket</u>
  - Load the state of that *winning process* and runs it
- Example
  - There are 100 tickets
    - Process A has 75 tickets: 0 ~ 74
    - Process B has 25 tickets: 75 ~ 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Resulting scheduler:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

Ticket Mechanisms

- Ticket currency
  - A user allocates tickets among their own jobs in whatever currency they would like
  - The system converts the currency into the correct global value
  - Example
    - There are 200 tickets (Global currency)
    - Process A has 100 tickets
    - Process B has 100 tickets

User A → *500* (A's currency) to A1 → *50* (global currency)
→ *500* (A's currency) to A2 → *50* (global currency)

User B → *10* (B's currency) to B1 → *100* (global currency)

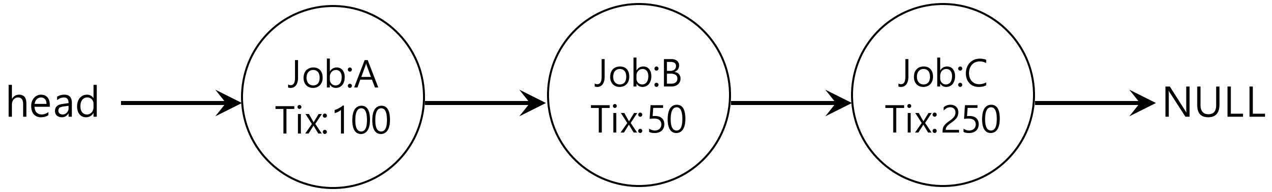
Ticket Mechanisms

- Ticket transfer
  - A process can temporarily hand off *its tickets* to another process

- Ticket inflation
  - A process can temporarily raise or lower the number of tickets it owns
  - If any one process needs *more CPU time*, it can boost its tickets

Implementation

- Example: There are three processes, A, B, and C.
  - Keep the processes in a list:



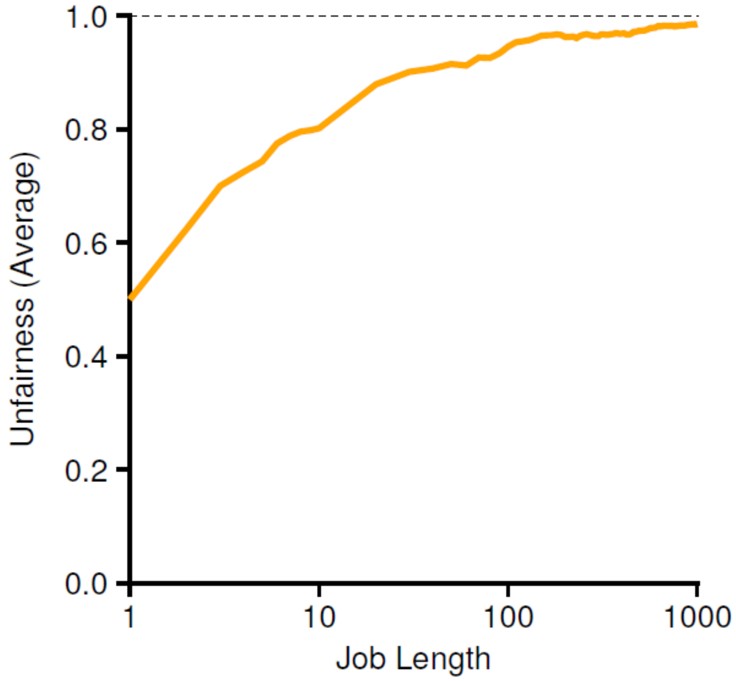

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
        counter = counter + current->tickets;
        if (counter > winner)
                break; // found the winner
        current = current->next;
}
// 'current' is the winner: schedule it...
```

Implementation

- U: unfairness metric
  - The time the first job completes divided by the time that the second job completes
- Example:
  - There are two jobs, each job has runtime 10
    - First job finishes at time 10
    - Second job finishes at time 20
  - $U = \frac{10}{20} = 0.5$
  - U will be close to 1 when both jobs finish at nearly the same time

Lottery Fairness Study

- There are two jobs
  - Each jobs has the same number of tickets (100)



**When the job length is not very long,
average unfairness can be quite severe**

Stride Scheduling

- Stride of each process
  - (A large number) / (the number of tickets of the process)
  - Example: A large number = 10,000
    - Process A has 100 tickets → stride of A is 100
    - Process B has 50 tickets → stride of B is 200

- A process runs, increment a counter(=pass value) for it by its stride
  - Pick the process to run that has the lowest pass value

```
current = remove_min(queue);          // pick client with minimum pass
schedule(current);                    // use resource for quantum
current->pass += current->stride;     // compute next pass using stride
insert(queue, current);               // put back into the queue
```

A pseudo code implementation

Stride Scheduling Example

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	…

If new job enters with pass value 0,
It will monopolize the CPU!

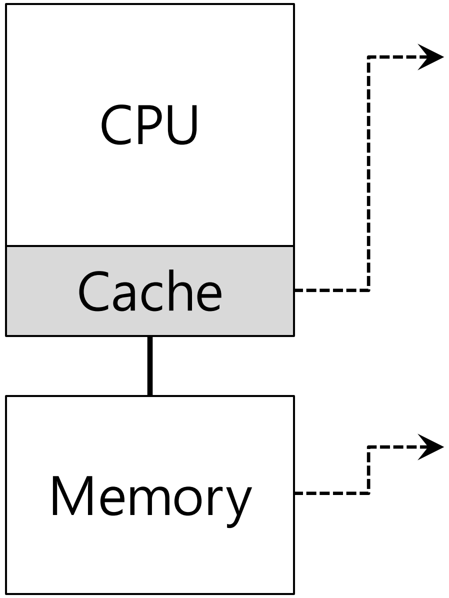
Multiprocessor Scheduling

- 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 <u>popular</u> data that is found in the main memory
- Utilize *temporal* and *spatial* locality

Main Memory

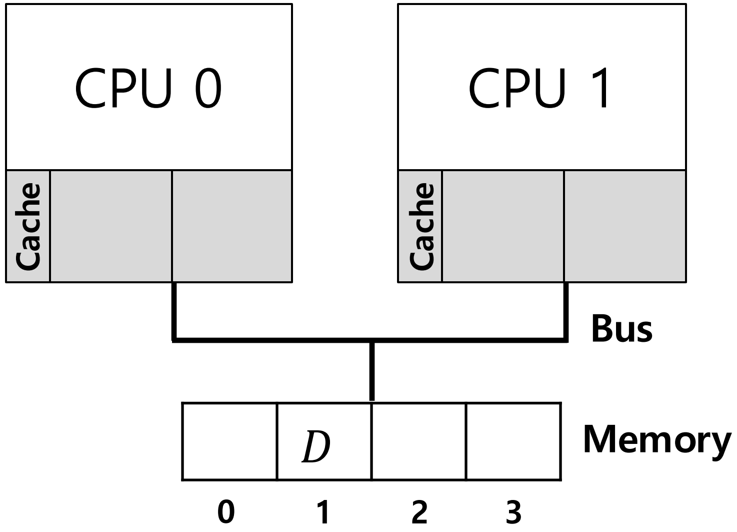
- Holds all of the data
- Access to main memory is slower than cache

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

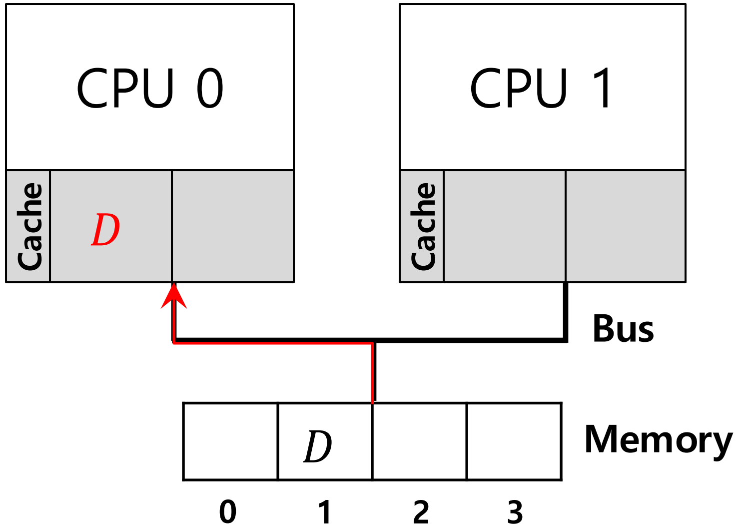
Cache Coherence

- Consistency of shared resource data stored in multiple caches

0. Two CPUs with caches sharing memory

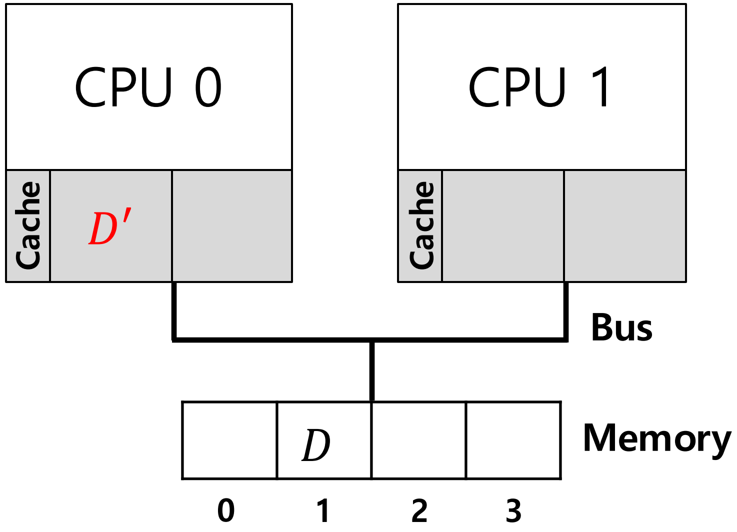


1. CPU0 reads a data at address 1.

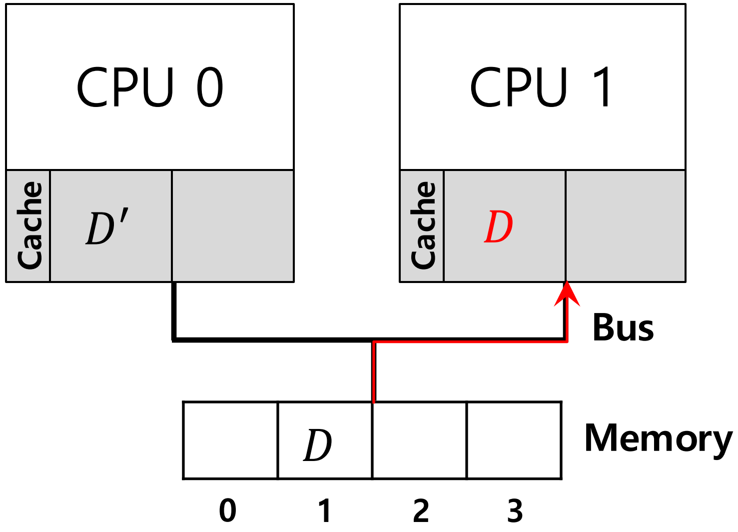


Cache Coherence

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 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 <u>invalidate</u> its copy or <u>update</u> it

Don’t Forget Synchronization

- When accessing shared data across CPUs, mutual exclusion primitives should likely be used to guarantee correctness.

```
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
}
```

Simple List Delete Code

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
}
```

Simple List Delete Code with lock

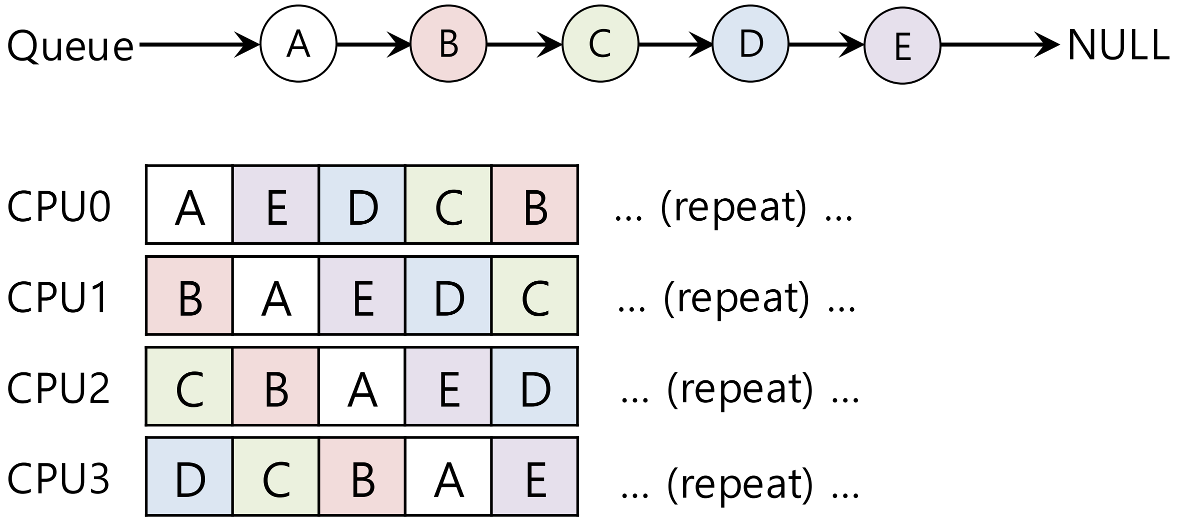
Cache Affinity

- 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

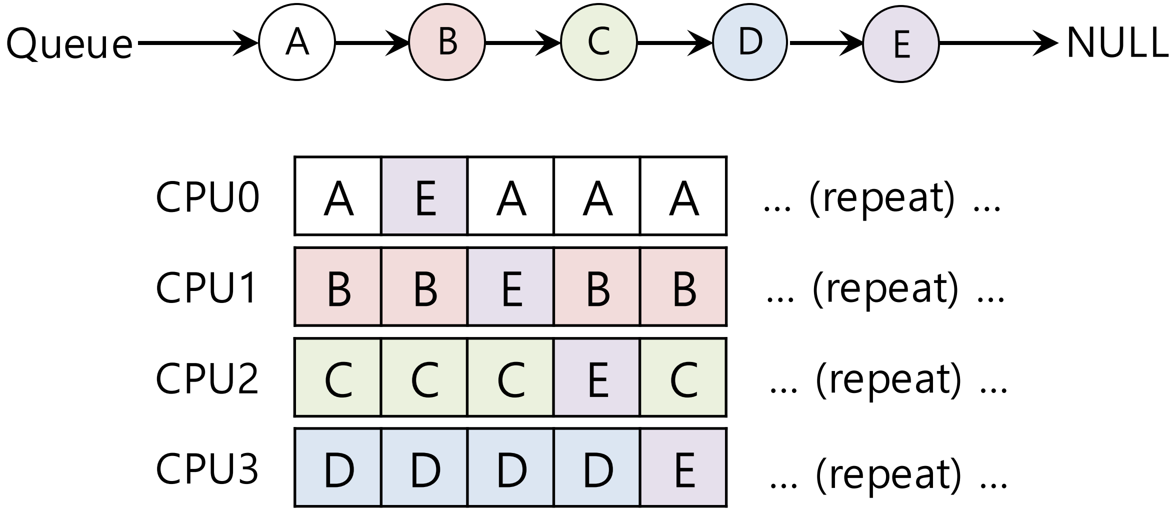
A multiprocessor scheduler should consider cache affinity when making its scheduling decision.

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:



Scheduling Example with Cache affinity



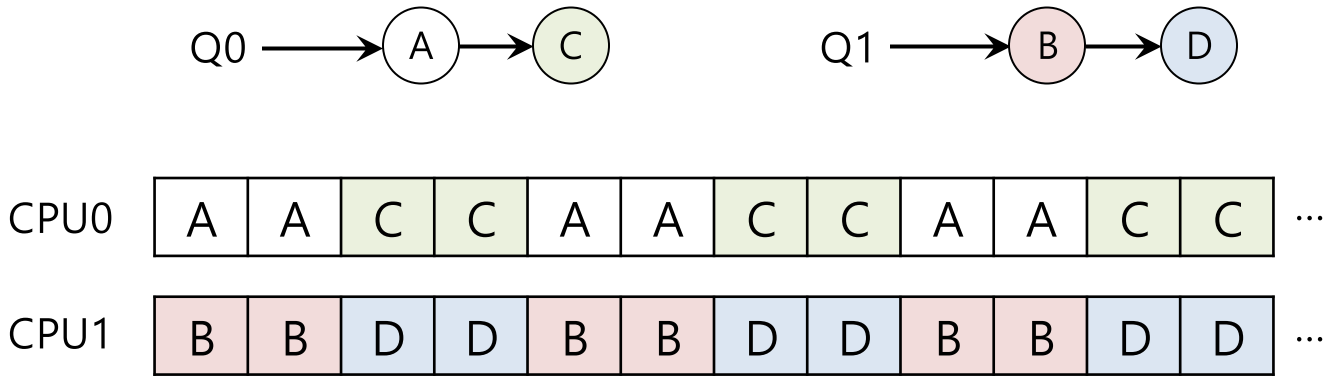
- Preserving affinity for most
  - Jobs A through D are not moved across processors
  - Only job e Migrating from CPU to CPU
- Implementing such a scheme can be **complex**

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
  - 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

CPU0	A	A	A	A	A	A	A	A	A	A	A	A	…
CPU1	B	B	D	D	B	B	D	D	B	B	D	D	…

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

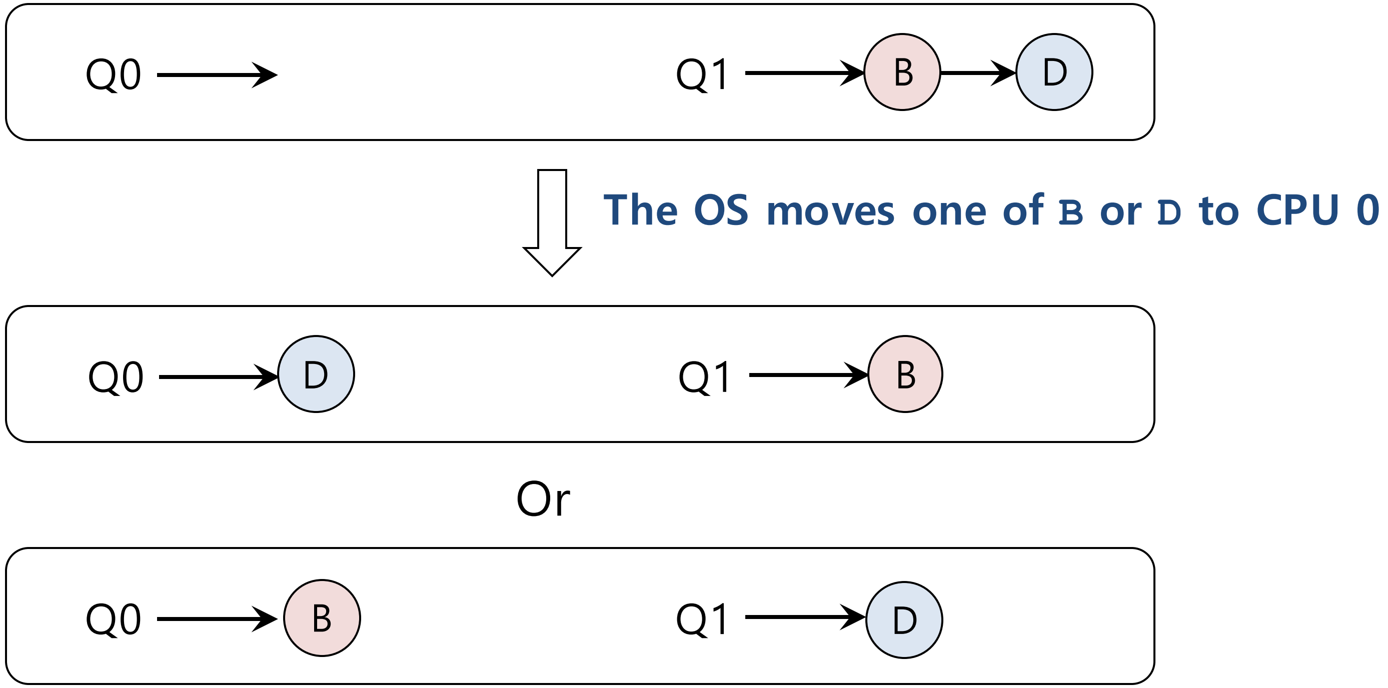
- After job A in Q0 finishes

CPU0													…
CPU1	B	B	D	D	B	B	D	D	B	B	D	D	…

CPU0 will be left idle!

How to Deal with Load Imbalance?

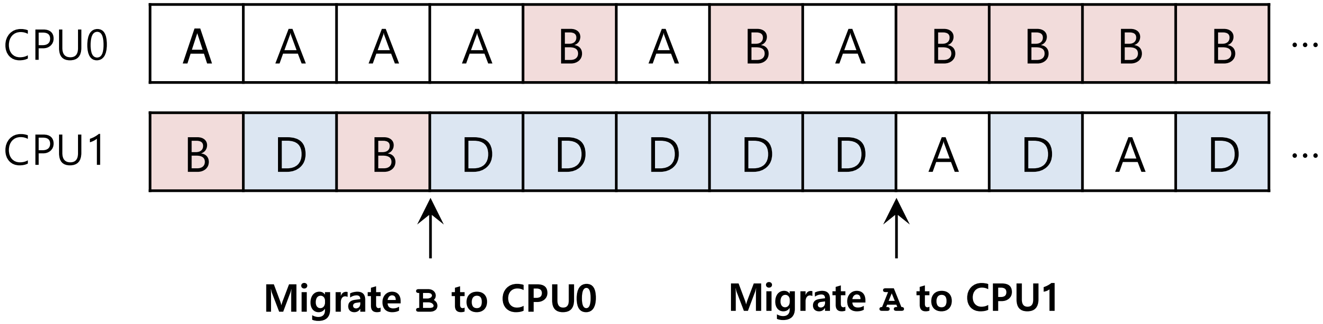
- The answer is to move jobs (**Migration**)
  - Example:



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*

Linux Multiprocessor Schedulers

- O(1) – Linux 2.6.0 (2003)
  - A Priority-based scheduler
  - Use Multiple queues
  - Change a process's priority over time
  - Schedule those with highest priority
  - Interactivity is a particular focus

- Completely Fair Scheduler (CFS) – Linux 2.6.23 (2007)
  - Deterministic proportional-share approach
  - Multiple queues

- EEVDF (Earliest Eligible Virtual Deadline First) – Linux 6.6 (2023)
  - Latency-aware scheduler
  - Proportional-share with virtual deadlines
  - Balances fairness and responsiveness