10. Multiprocessor Scheduling (Advanced)

Operating System: Three Easy Pieces
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**.
By keeping data in cache, the system can make slow memory appear to be a fast one.
Cache coherence

- Coherence of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory

1. CPU0 reads a data at address 1.
2. $D$ is updated and CPU1 is scheduled.

CPU 0

CPU 1

Cache

$D'$

Cache

Bus

Memory

$D$

$0$

$1$

$2$

$3$

3. CPU1 re-reads the value at address A

CPU 0

CPU 1

Cache

$D'$

Cache

$D$

Bus

Memory

$D$

$0$

$1$

$2$

$3$

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**.

```c
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 (Cont.)

Solution

```c
1      pthread_mutex_t m;
2      typedef struct __Node_t {
3              int value;
4            struct __Node_t *next;
5          } Node_t;
6
7      int List_Pop() {
8          lock(&m)
9        Node_t *tmp = head;       // remember old head ...
10       int value = head->value;  // ... and its value
11      head = head->next;        // advance head to next pointer
12      free(tmp);                // free old head
13     unlock(&m)
14    return value;              // return value at head
15     }
```

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 runs, 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.
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
<table>
<thead>
<tr>
<th>A</th>
<th>E</th>
<th>D</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
</table>
CPU1
<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>E</th>
<th>D</th>
<th>C</th>
</tr>
</thead>
</table>
CPU2
<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
<th>A</th>
<th>E</th>
<th>D</th>
</tr>
</thead>
</table>
CPU3
| D | C | B | A | E |
```

... (repeat) ...
Scheduling Example with Cache affinity

Queue: A → B → C → D → E → NULL

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

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

```
<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Q0</strong></td>
<td><strong>Q1</strong></td>
</tr>
<tr>
<td>Q0 → A → C</td>
<td>Q1 → B → D</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>
```

MQMS provides more **scalability** and cache **affinity**.
Load Imbalance issue of MQMS

- After job C in Q0 finishes:

  Q0 → A

  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:

  Q0 →

  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:

    ![Diagram showing load imbalance and migration]

    The OS moves one of B or D to CPU 0

    ![Diagram showing different migration scenarios]
How to deal with load imbalance? (Cont.)

- A more tricky case:

- A possible migration pattern:
  - Keep switching jobs

```
<table>
<thead>
<tr>
<th>CPU0</th>
<th>B</th>
<th>D</th>
<th>B</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>
```

Migrate B to CPU0
Migrate A to CPU1
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)
  - 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) (current mainline)
  - Deterministic proportional-share approach
  - Based on Staircase Deadline (fairness is the focus)
  - Red-black tree for scalability
BF Scheduler (BFS) (Not in the mainline)
- A single queue approach
- Proportional-share
- Based on Earliest Eligible Virtual Deadline First (EEVDF)
- Focus on interactive (not scale well with cores). Superseded by MuQSS to fix that

The battle of schedulers: Kolivas (SD) vs Molnar (CFS)
“And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy.”

Linus Torvalds, 2001 [43]

Scheduling is not easy!, E.g:

“The Linux Scheduler: a Decade of Wasted Cores”
Disclaimer: This lecture slide set is used in AOS course at University of Cantabria. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)