Lecture 28 —Multiprocessor and Real-Time Scheduling

Jeff Zarnett
`jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering
University of Waterloo

May 29, 2015

Multiprocessor Scheduling

If you thought scheduling for a single processor was complicated enough, well, things are about to get exponentially harder.

When we have more than one processor working on things at a time, then the complexity increases dramatically.

Multiprocessor Systems

We can classify multiprocessor systems into three major buckets:

1. **Distributed**.
2. **Functionally Specialized**.
3. **Tightly Coupled**.

The third is our focus here.

Multiprocessor: Granularity

Then we have to worry about the interactions of various processes.

Specifically, how often they plan to interact.

Grain Size	**Description**	**Interval (Instructions)**
Fine	Single instruction stream	< 20
Medium	Single application	$20 - 200$
Coarse	Multiple processes	$200 - 2000$
Very Coarse	Distributed computing	$2000 -$ 1M
Independent	Unrelated processes	N/A

Granularity

The finer-grained the parallelism, the more care and attention needs to be given to how we are going to schedule a process in a multiprocessor system.

If the processes are independent, then there is not too much to worry about.

If we are taking a single process's thread and doing different instructions on different CPUs, then we have to be very careful.

Asymmetric vs Symmetric

Asymmetric multiprocessing: a boss processor and this one alone is responsible for assigning work and managing the kernel data structures.

Symmetric multiprocessing, each processor is responsible for scheduling itself.

We will need to make use of mutual exclusion and other synchronization techniques in the kernel to prevent errors.

We do not want to have two processors trying to dequeue from the ready queue at the same time, after all.

Processor Affinity

Imagine that every processor has its own cache (e.g., the L1, L2, & L3 caches).

We want to have processor affinity.

A process will have a bunch of its data in the cache of that processor.

If the process begins executing on another processor, all the data is in the "wrong" cache and there will be a lot more cache misses.

This desire to stick with a certain processor is called processor affinity.

Processor Affinity

If the OS is just going to make an effort but not guarantee that a process runs on a given processor, that is called soft affinity.

A process can move from one processor to another, but will not do so if it can avoid it.

The alternative is hard affinity: a process will only run on a specified processor (or set of processors).

Linux, for example, has both soft and hard affinity

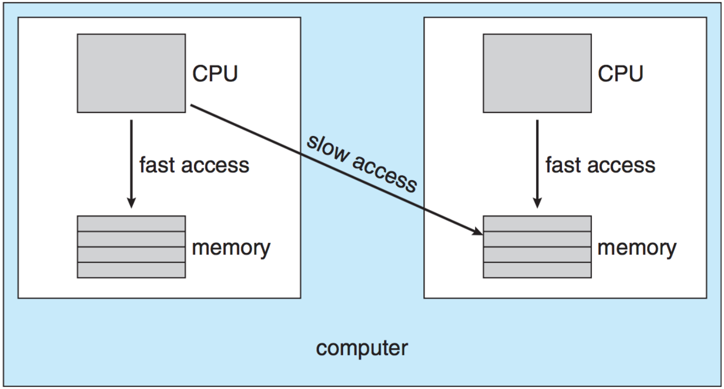
Affinity - NUMA

For the most part any memory read takes as much time as any other.

If we have one bus connecting the CPU to all of main memory, that is a safe assumption.

If the CPU can access some parts of memory faster than others, the system has non-uniform memory access (NUMA).

NUMA


CPU
fast access
memory
slow access
CPU
fast access
memory
computer

NUMA

With NUMA: choice of processor should be based on where the memory of the process is located.

The memory allocation routine should also pay attention to where to allocate memory requests.

If there is data in one of the other blocks of memory, it does not mean game over, but it means slower execution.

Load Balancing

If we have 4 processors, it is less than ideal to have one processor at 100% utilization and 3 processors sitting around doing nothing.

We want to keep the workload balanced between all the different systems.

The process for this is load balancing.

Load Balancing

Load balancing is typically necessary only where each processor has its own private queue of processes to run.

If there is a common ready queue then load balancing will tend to happen all on its own.

A processor with nothing to do will simply take the next process from the queue.

But in most of the modern operating systems we are familiar with, each processor does have a private queue, so we need to do load balancing.

Push and Pull Load Balancing

There are two, non-exclusive approaches: push and pull migration.

It is called migration because a process migrates from one processor to another (it moves homes).

Push migration: a task periodically checks how busy each processor is and then moves processes around to balance things out (to within some tolerance).

Pull migration: a processor with nothing to do “steals” a process from the queue of a busy processor.

The Linux and FreeBSD schedulers, for example, use both.

Balancing Affinity

Load balancing sometimes conflicts with processor affinity.

If a process has a hard affinity for a processor, it cannot be migrated.

If there is a soft affinity, it can be moved, but it is not our first choice.

Balancing Affinity

Should we always move a process despite the fact that it means a whole bunch of cache misses?

Should we never do so and leave processors idle?

Perhaps the best thing to do is to put a certain “penalty” on moving.

Only move a process from one queue to another if it would be worthwhile (i.e., the imbalance is sufficiently large).

Multicore Processors

Before the early 2000s, the only way to get multiple processors in the system was to have multiple physical chips.

If you open up your laptop you are likely to find one physical chip.

What gives? Multicore processors.

As far as the operating system is concerned, a quad-core chip is made of four logical processors.

Multicore Processors

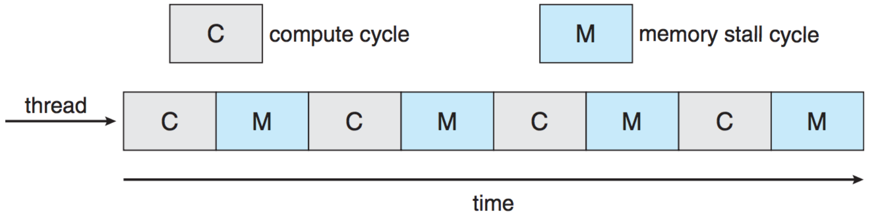
On a cache miss the CPU core can spend a lot of time (50%?) of its time waiting for that read to take place.

We might refer to periods of time where there is computation as a compute cycle, and time spent waiting for memory as a memory stall.

These tend to alternate, though the length of time for each will vary.

A question of how often memory is accessed and how many cache misses.

Compute Cycles and Memory Stalls



Memory Stall

During a memory stall, the processor core may have nothing to do.

You can sometimes move instructions around so that the memory read goes out “early”.

A few other instructions can be executed in the meantime.

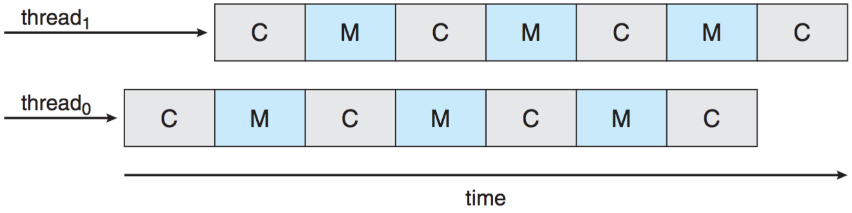
(Take CPU design / programming for performance classes.)

Hyperthreading

To offset this problem, the solution was originally called hyperthreading.

Two threads are assigned to each core; if one thread does a memory access or stalls, the code can switch to another thread with a limited penalty.

Hyperthreading



Hyper- and Multithreading

Coarse-grained multithreading: flushing the instruction pipeline.

That is expensive.

Fine-grained multithreading: alternation between two threads that are in the pipeline.

The cost of switching between these two threads is small.