Lecture 28 —Multiprocessor and Real-Time Scheduling

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 18, 2016

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

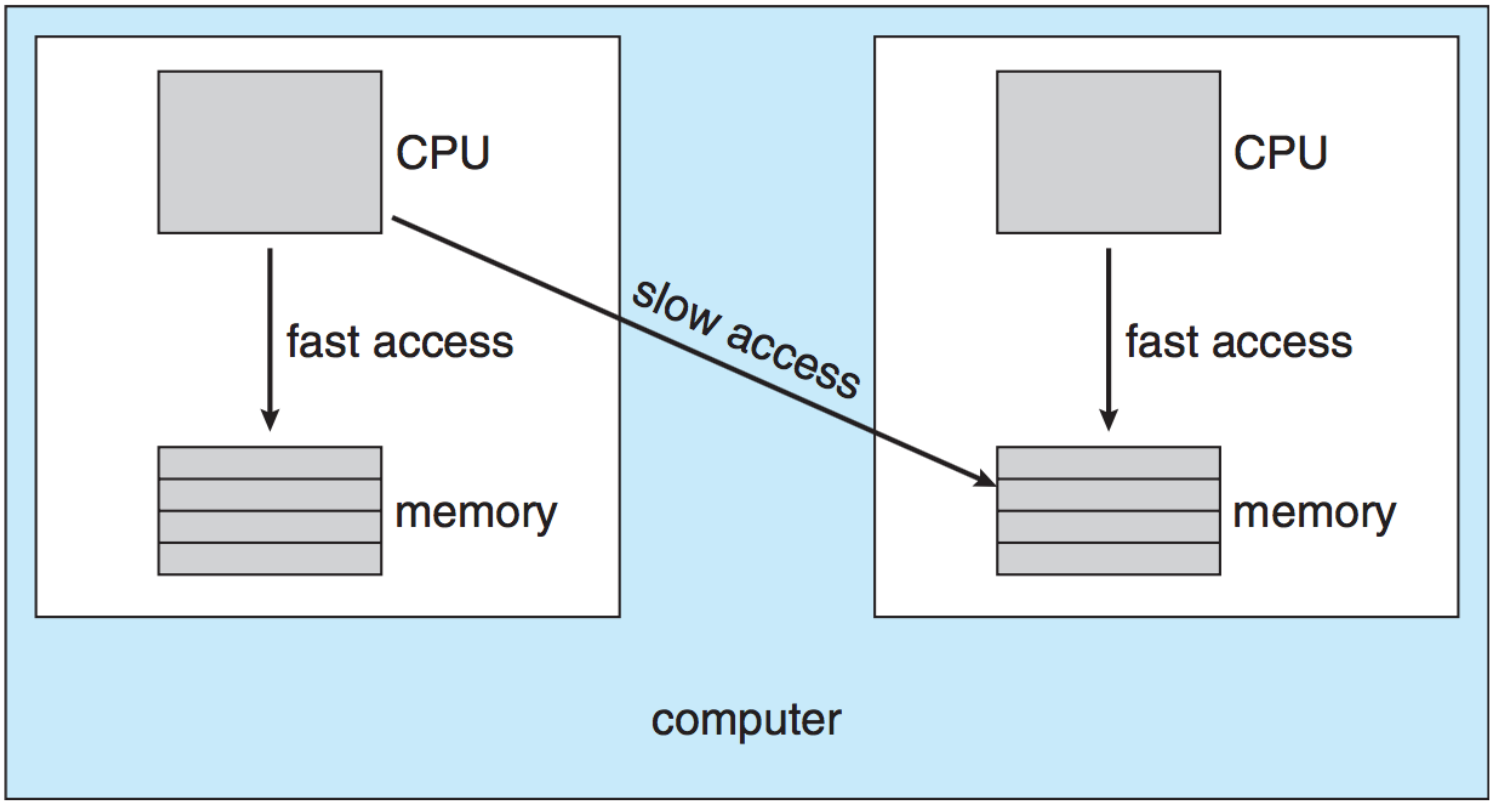
Affinity - NUMA

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

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.

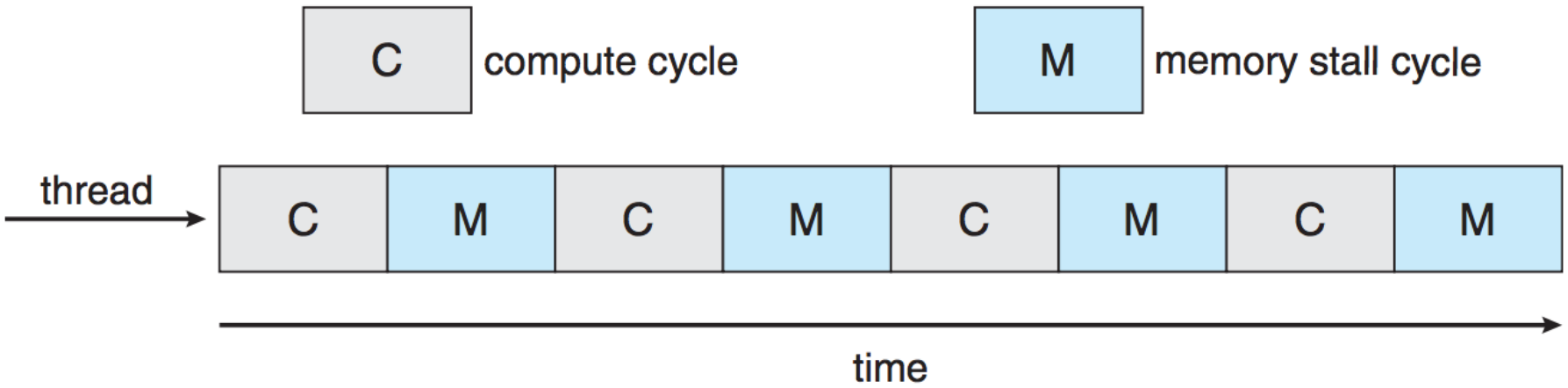
On a cache miss the CPU core can spend a lot of time (50%?) of its time waiting for that read to take place.

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.

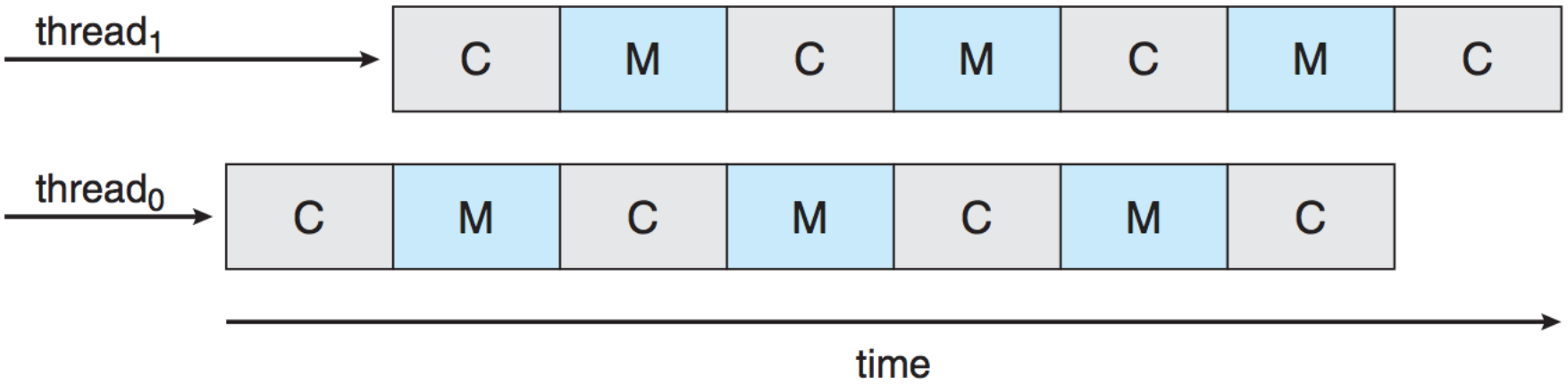
(Take CPU design / programming for performance classes.)

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 2 threads in the pipeline.

The cost of switching between these two threads is small.

Two Levels of Scheduling

So now we have two different levels of scheduling.

1. Assigning a process or thread to a processor (job of the operating system)

2. When to swap between the two threads in the core (job of the hardware).

Real-Time Scheduling

Real-Time scheduling is just scheduling for real-time systems.

But what is a real-time system?

Supposed to respond to events within some real (wall-clock) time.

There are deadlines, and there are consequences for missing deadlines.

Fast is not as important as predictable.

Hard and Soft Real-Time

Hard real-time: it has a deadline that must be met to prevent an error, prevent some damage to the system, or for the answer to make sense.

If a task is attempting to calculate the position of an incoming missile, a late answer is no good.

A soft real-time task has a deadline that is not, strictly speaking, mandatory; missing the deadline degrades the quality of the response, but it is not useless.

Hard Real-Time Failure

If a task is hard real-time, there are two scenarios in which it might not complete before its deadline.

The first is that it is scheduled too late; like an assignment that will take two hours to complete being started one hour before the deadline.

If that is the case, the system will likely reject the request to start the task, or perhaps never schedule the task to run at all.

Why waste computation time on a task that will not finish in time?

Hard Real-Time Failure

The second scenario is that at the time of starting, completion was possible.

For whatever reason (e.g., other tasks with higher priority have occurred) it is no longer possible to meet the deadline.

In that case, execution of the task may be terminated partway through so that no additional effort is wasted on a task that cannot be completed.

Desktop OSes and Real-Time

Most of the operating systems you are familiar with (standard Desktop/Server Linux, Mac OS, Windows) are not very suitable to real time systems.

They make few guarantees, if any, about service.

When there are consequences for missing deadlines, this kind of thing matters.

Remember Java's "stop the world" garbage collector scenario.

Timeline Scheduling

If a process/task recurs at regular intervals, it is periodic.

Periodic tasks are very common: check a sensor, decode and display a frame of video to a screen, keep the wifi connection alive, etc.

Periodic Task

Consider a periodic task to have two attributes:

1. τ_k, the period (how often the task occurs); and

2. c_k, the computation time (how long, in the worst case, the task might run).

Pessimism

In real-time systems we are usually pessimists and care almost exclusively about the worst case scenario.

We can calculate the processor utilization of periodic tasks according to the following formula, to get the long-term average of processor utilization U:

$$U = \sum_{k=1}^{n} \frac{c_k}{\tau_k}$$

If $U > 1$, it means the system is overloaded.

Overload

If overloaded: there are too many periodic tasks and we cannot guarantee that the system can execute all tasks and meet the deadlines.

Otherwise, we will be able to devise a schedule that makes it all work.

Periodic Tasks

If the only tasks in the system are periodic ones, create a fixed schedule.

This is what the university administrators do when they create the schedule of classes for a given term.

Every Monday, for example, from 13:30-14:50, course ECE 254 (a “process”, if you will) takes place in classroom EIT 1015 (a resource).

If there are more requests for room reservations than rooms and time slots available, it means some requests cannot be accommodated.

Living in a Periodic World

A world in which all the tasks are periodic and behave nicely is, well, a very orderly world (and that has its appeal).

Unfortunately, the real world is not so accommodating most of the time.

So we will need to deal with tasks that are not periodic, which we can categorize as aperiodic or sporadic.

Aperiodic Tasks

Aperiodic tasks are ones that respond to events that occur irregularly.

There is no minimum time interval between two events.

Therefore it would be very difficult to make a guarantee that we will finish them before the next one occurs, so they are rarely hard real-time (hard deadlines).

Aperiodic Tasks

Tasks like this should be scheduled in such a way that that they do not prevent another task with a hard deadline from missing its deadline.

As an example, if we expect an average arrival rate of 3 requests per second, there is still a 1.2% chance that eight or more requests appear within 1 second.

If so, there is not much we can do to accommodate them all (most likely).

Sporadic Tasks

Sporadic tasks are aperiodic, but they require meeting deadlines.

To make such a promise we need a guarantee that there is a minimum time τ_k between occurrences of the event.

Sporadic tasks can overload the system if there are too many of them, and if that is the case, we must make decisions.

If we know a task will not make its deadline, we will likely not even bother to schedule it (why waste the time on a task where the answer will be irrelevant?)

Aperiodic, Sporadic, and Timeline

So, these aperiodic and sporadic tasks really mess with timeline scheduling.

This unpredictability makes it hard to create a simple timetable and follow it.

If we have pre-emptive scheduling, then we can examine 2 optimal alternatives.

They are called optimal because they will ensure a schedule where all tasks meet their deadlines, not because they are the ideal algorithms.

Earliest Deadline First

The earliest deadline first algorithm is, presumably, very familiar to students.

Assignment due today, an assignment due next Tuesday, and an exam next month, then you may choose to schedule these things by their deadlines.

Do the assignment due today first.

After completing an assignment, decide what to do next (probably the new assignment, but perhaps a new task has arrived in the meantime?) and start.

Earliest Deadline First

The principle is the same for the computer.

Choose the task with the soonest deadline; if there is a tie, then random selection between the two will be sufficient (or other criteria may be used).

If there exists some way to schedule all the tasks such that all deadlines are met, this algorithm will find it.

If a task is executing and another task arrives with a sooner deadline, the currently executing task should be suspended and the new task scheduled.

This may mean a periodic task being preempted by an aperiodic/sporadic task.

Least Slack First

A similar algorithm to earliest deadline first, is least slack first.

Slack: how long a task can wait before it must be scheduled to meet a deadline.

If a task will take 10 ms to execute and its deadline is 50 ms away, then there are (50 - 10) = 40 ms of slack time remaining.

Start the task before 40 ms are expired if we want to be sure that it will finish.

Least Slack First

This does not mean, however, that we necessarily want to wait 40 ms before starting the task (even though many students tend to operate on this basis).

All things being equal, we prefer tasks to start and finish as soon as possible.

It does, however, give us an indication of what tasks are in most danger of missing their deadlines and should therefore have priority.