Lecture 4

Process scheduling in multi-core systems

Multi-core systems

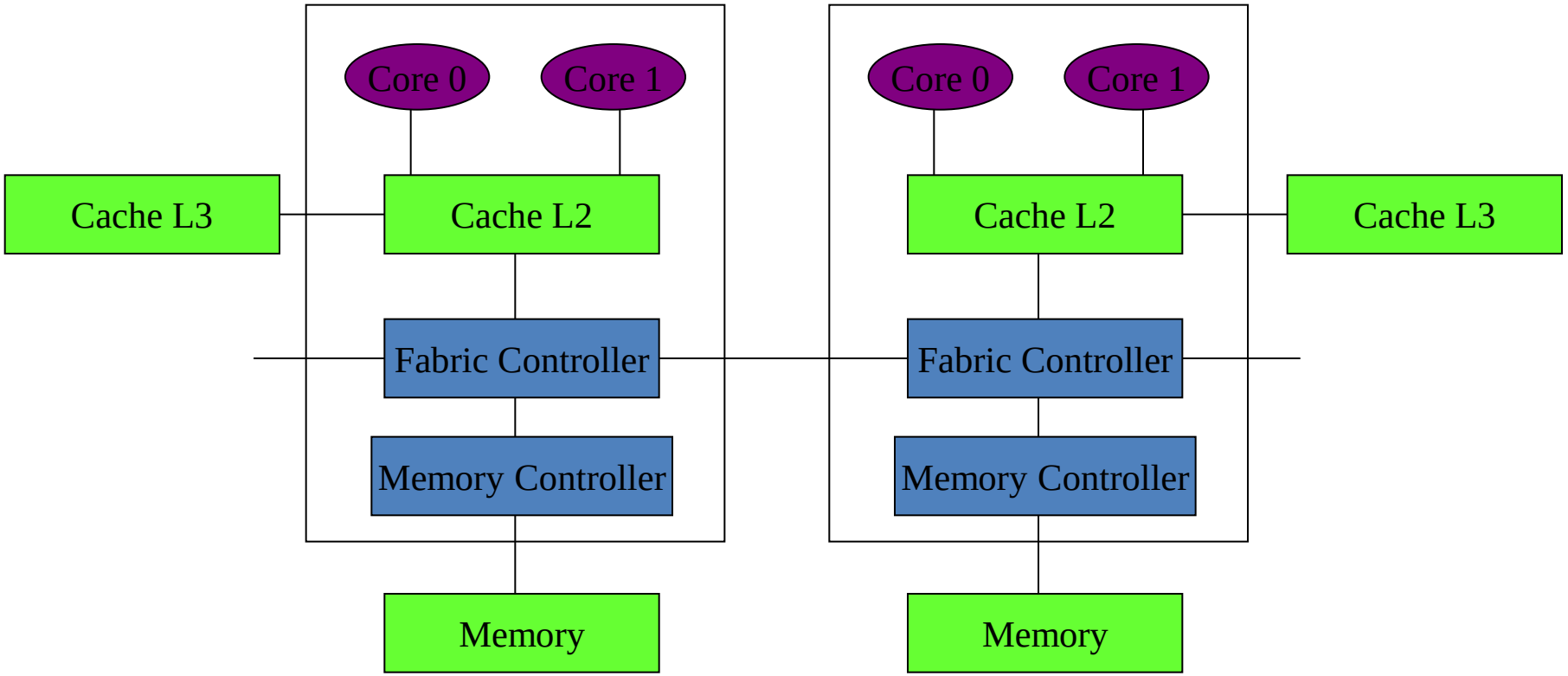
- Multi-cores architectures implement real parallelism: multiple processes/threads can run in the same time on different cores.

- Scheduler's main challenge: to identify and predict the resource needs of each process and schedule them in a fashion that will *minimize shared resource contention*, *maximize shared resource utilization*, and exploit the advantage of shared resources between cores.

- To achieve this, the process scheduler needs to be aware of
  - multi-core, shared resource topology,
  - resource requirements of processes, and
  - the inter-relationships between the processes.

- Example: 2-packages, dual-core each package system. If the application has two processes (or threads), they can be allocated to different packages, minimising contentions. However, power saving would require to have both allocated to the same package.

Multi-core topologies: IBM Power5 model



This is an example of SMP – symmetric multi-processor.

Aspects to consider

- Memory contention (L2) and its impact on performance depend on the resources shared, the number of active processes, and the memory access patterns of the individual processes.

- Heterogeneous data access patterns of *memory-intensive processes* running on the cores sharing caches can lead to cache contention and sub-optimal performance.

- Approaches:
  1. If processes share data, it makes sense to schedule them on same package cores. Otherwise, scheduling on one package will lead to L2 contention; the main benefit is power saving from idle packages that can switch both cores and L2 cache to sleep states.

  2. Scheduling processes on cores of different packages maximises execution speed but it's not optimal in respect to energy.

I. Process group scheduling

- If all processes are resource intensive, the challenge before the scheduler is to identify the processes that *share data* and schedule them on the cores sharing the L2 cache. This will also result in efficient data communication between processes that share data.

- The system software has some inherent knowledge about data sharing between processes:
  - threads belonging to a process share the same address space and as such share everything (text, data, heap, etc.);
  - similarly, processes attached to the same shared memory segment will share the data in that segment.

- In a scenario where all the shared resources and packages are busy, the process scheduler needs to minimize the resource contention. For example, *grouping CPU-intensive and memory-intensive processes* onto the cores sharing the same L2 cache will result in minimized cache contention.

Prediction of resource use

- Process characteristics and behaviour can be predicted using the *micro-architectural history* of a process by using performance counters. In the absence of such micro-architectural information, the system software can also use some heuristics to estimate the resource requirements.
- For example, one can use the number of physical pages that are accessed (using the Accessed bit in the page tables that manage virtual to physical address translation in x86 architecture) for certain intervals or use the processes memory Resident Set Size (RSS).
- The process scheduler can use this information and group processes on the cores residing in a physical package with the goal of minimizing shared resource contention.

II. Scheduling domains

- Load balancing: in multi-core systems, one goal of the scheduler is to balance the cores' load such that there is no idle core while other cores are overloaded.

- The *domain-based scheduler* aims to solve this problem by way of a new data structure which describes the system's structure and scheduling policy in sufficient detail that good decisions can be made:
-
  - a *scheduling domain* (struct sched_domain) is a set of cores which share properties and scheduling policies, and which can be balanced against each other. *Scheduling domains are hierarchical*; a multi-level system will have multiple levels of domains.
  - each domain contains one or more *core groups* (struct sched_group) which are treated as a single unit. When the scheduler tries to balance the load within a domain, it tries to even out the load carried by each core group without worrying directly about what is happening within the group.

Domain policies

- Each scheduling domain contains policy information which controls how decisions are made at that level of the hierarchy. The policy parameters include:
  - how often attempts should be made to balance loads across the domain,
  - how far the loads on the component processors are allowed to get out of sync before a balancing attempt is made,
  - how long a process can sit idle before it is considered to no longer have any significant cache affinity, and
  - various policy flags.

Policy examples

1. When a process calls exec() to run a new program, its current cache affinity is lost. At that point, it may make sense to move it elsewhere. So the scheduler works its way up the domain hierarchy looking for the highest domain which has the SD_BALANCE_EXEC flag set. The process will then be shifted over to the core within that domain with the lowest load. Similar decisions are made when a process forks.
2. If a processor becomes idle, and its domain has the SD_BALANCE_NEWIDLE flag set, the scheduler will go looking for processes to move over from a busy processor within the domain.
3. If one processor in a shared pair is running a high-priority process, and a low-priority process is trying to run on the other processor, the scheduler will actually idle the second processor for a while. In this way, the high-priority process is given better access to the shared package.

1. if exec() is invoked then
   - determine the highest domain with load balancing flag set;
   - move the process to the least loaded core of the domain;

2. if core idle then
   - determine overloaded core;
   - move load to the idle core;

3. if priorities of processes running on the two core are high/low then
   - stop the low priority process;

III. Active balancing

- The last component of the domain scheduler is *the active balancing code*, which moves processes within domains when things get too far out of balance.

- Every scheduling domain has an interval which describes how often balancing efforts should be made; if the system tends to stay in balance, that interval will be allowed to grow. The scheduler "rebalance tick" function runs out of the clock interrupt handler; it works its way up the domain hierarchy and checks each one to see if the time has come to balance things out. If so, it looks at the load within each core group in the domain; if the loads differ by too much, the scheduler will try to move processes from the busiest group in the domain to the least busy group. In doing so, it will take into account factors like the cache affinity time for the domain.

Heterogeneous multi-cores

- While Intel and AMD use 4/8 *identical* cores, ARM uses cores of different designs, e.g., a cluster of Cortex-A72 (high performance core) and a cluster of Cortex-A53 (energy efficient core). This configuration is known as the big.LITTLE architecture, where big cores (A72) work with little ones (A53).

- The scheduler starts by using LITTLE cores and when the workload increases, it starts to use big cores. LITTLE cores go to sleep, until the workload starts to decrease, and they return.

- Strategy: in the big.LITTLE configuration, the scheduler picks the right core for the right task.

Questions and problems

1. What can be the load of a core, the number of processes ready to run, or %CPU? How frequently should the core load be assessed?
2. Consider a homogenous 8-core computer. The basic domain unit is a 2-core package with L2. Draw the binary-tree model of the domain hierarchy up to the root level that includes all cores. Define policies for each domain, considering energy saving and load balancing as the main criteria. Write the domain scheduler that works with this model using pseudo-code.
3. Choose a load parameter and write in pseudo-code a scheduler that manages heterogeneous cores, little and big, when the load varies from low to high and back.

Conclusions

- Multi-core schedulers are more complex as they consider in their decisions data about the system topology and processes behaviour information.
- Group scheduling and domain scheduling are designed to work for multi-cores.
- In multi-cores, scheduling is extended with load balancing.
- Heterogeneous multi-cores use more elaborate scheduling strategies – energy saving is key.
- Critical aspects are: performance (minimize cache collisions and maximize sharing) and energy saving.

References

- http://www.intel.com/technology/itj/2007/v11i4/9-process/2-intro.htm
- http://lwn.net/Articles/80911/
- https://www.androidauthority.com/fact-or-fiction-android-apps-only-use-one-cpu-core-610352/