# Lecture 2

## Process scheduling

- How does a process get the control of the CPU ?
- What strategies are in use for scheduling ?
- Can the scheduling parameters be modified dynamically ?
- What means multi-core scheduling ?

## Purpose of scheduling

- Historically, the CPU was allocated to one process until its completion – known as batch processing. Then, the CPU was time-shared by multiple processes ready to execute.
- As CPU is time-shared, processes compete for the next available time slot.
- The scheduler implements an algorithm that decides which process gets the CPU next.
- The scheduling process needs to be fair to all processes.
- Processes ready to execute are organized in a queue from where the scheduler selects the next one.
- A process takes control of the CPU by having its state restored, while the state of the previous process is saved.

## First-come first-served / round robin

- FCFS is the simplest algorithm: processes are getting CPU control in their order in the ready-to-execute queue.
- One possibility is to have the control of the CPU until the process finishes - non-preemptive execution. This may lead to starvation of other processes.
- Therefore, the best solution is to time-share the CPU.
- If a process is not finished during its time slice, it will be returned at the end of the queue.
- Other possibilities to be switched from the running state are:
  - an I/O operation that will put the process in the blocked queue;
  - it suspends itself until a certain event occurs;
  - a higher priority process requires control.

# Shortest process first

- If the CPU is not time-shared, the order in which processes are scheduled is important.
- Processes can be ordered according to their execution time.
- If processes get control in the decreasing order of their execution time, the average turnaround time is better than in the random order.
- The turnaround time is the time consumed from the moment the process is ready for execution until its completion.
- Example: 3 processes, a(35), b(40), c(15).
  - $t_a = 35$, $t_b = 75$, $t_c = 90$ $t_{aa} = 200/3$
  - In decreasing order of execution time: $t_c = 15$, $t_a = 50$, $t_b = 90$, $t_{aa} = 155/3$

## Priority scheduling

- Processes are different, some of them are interactive, others are computing demanding and therefore need to be dealt with differently in order to provide system responsiveness.
- One solution is to assign priorities to processes. Fast, interactive processes will have higher priorities than computation strong ones.
- Moreover, priority can be changed at runtime according to the process behaviour; if a process takes too long to complete, its priority will be lowered.
- Priority is denoted by a small integer, generally a smaller value indicating a higher priority.
- Obviously, kernel processes have higher priorities than user processes.
- The priority of user processes is given considering the user or process attributes.
- If there are several processes with the same priority, they are scheduled in a round robin manner.
- Many systems have an idle process, which has the lowest priority. When there is no other process to execute, the CPU is given to the idle process that switches the system into sleep state(-s).

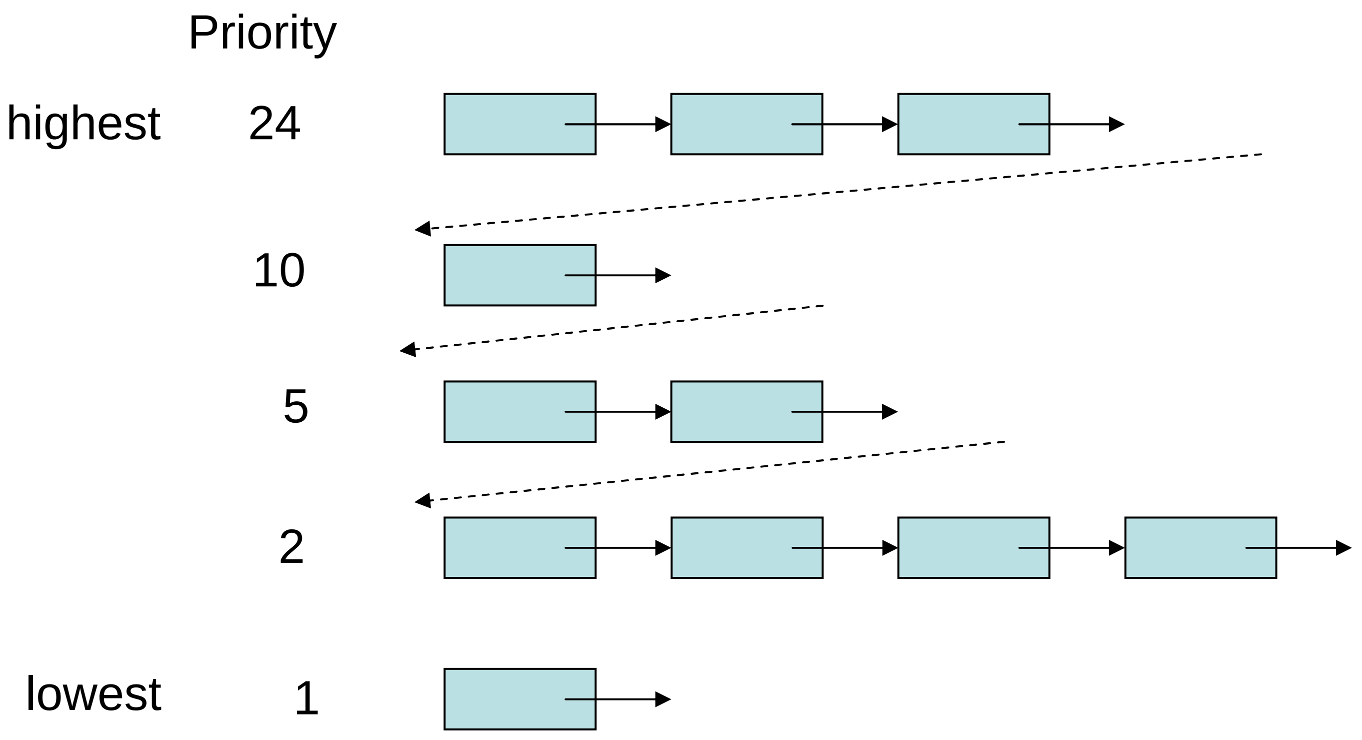
## A process for power management

- The kernel power policy manager owns the decision-making and the set of rules used to determine the appropriate frequency/voltage operating state. It may make decisions based on several inputs, such as end-user power policy, processor utilization, battery level, or thermal conditions and events.

- The processor driver is used to make actual state transitions on the kernel power policy manager’s behalf.

# Priority scheduling : multilevel feedback queues

- This is one implementation of dynamic priorities.
- Initially, a process gets a priority that puts it on a certain level.
- After each time slice, the priority is lowered to the next level, until it reaches the lowest acceptable priority. At that level, the strategy is round robin.
- However, after being blocked, the process gets a higher priority (priority boost). Consequently, during its existence, one process can have a priority that varies within a defined range.



Multilevel feedback queue

## Adjusting scheduling parameters

- Dynamic priorities allow to avoid process starvation when, for example, a medium-level priority process is computation strong and never blocks. Lower priorities processes will starve waiting for their time slice. In this case, their priorities can be raised at the medium or even higher level.
- The time slice (quantum) can be different for each priority level. For example, the highest priority level will have the shortest time slice, and then this can be increased exponentially for lower level priorities; if the base quantum is q, level I will have the time slice $2^i q$.

## Two-level scheduling

- Sometimes, there are too many processes that can’t fit in the main memory in the same time. Therefore some will have to be stored on the disk. However the process of restoring the process in the main memory while other(-s) are saved on the disk is time consuming (can lead to the thrashing phenomenon).
- One solution is to use two-level scheduling:
  – a higher-level, long-term scheduler that runs more slowly will select the subset of processes resident in the main memory;
  – these processes are then managed by a different scheduler, lower-level and short-term.

# Real-time scheduler

- In real-time applications, computing systems react to events signalled by interrupts. The interrupt triggers the scheduler to give control to the respective event handler.
- If more than one occurs in the same time, priority and deadlines are considered. For example, if a new event has a higher priority or a tighter time requirement than the current running process, then the scheduler will preempt the exiting one.
- One popular techniques is earliest deadline first (EDF). For each process, there is an attribute value that indicates the time by which it should be completed.
- The scheduler always selects the process with the earliest deadline. After the process used its time slice, the deadline is updated and the process is added to the ready list.
- Generally, the new deadline is computed by adding  a constant period to the time at which the time slice ended.
- Example: time slice = 100 ms and three processes, a with const period of 300 ms, b with 500 ms and c with 1000ms.

  All are ready at t = 0 and have deadlines equal to their periods.

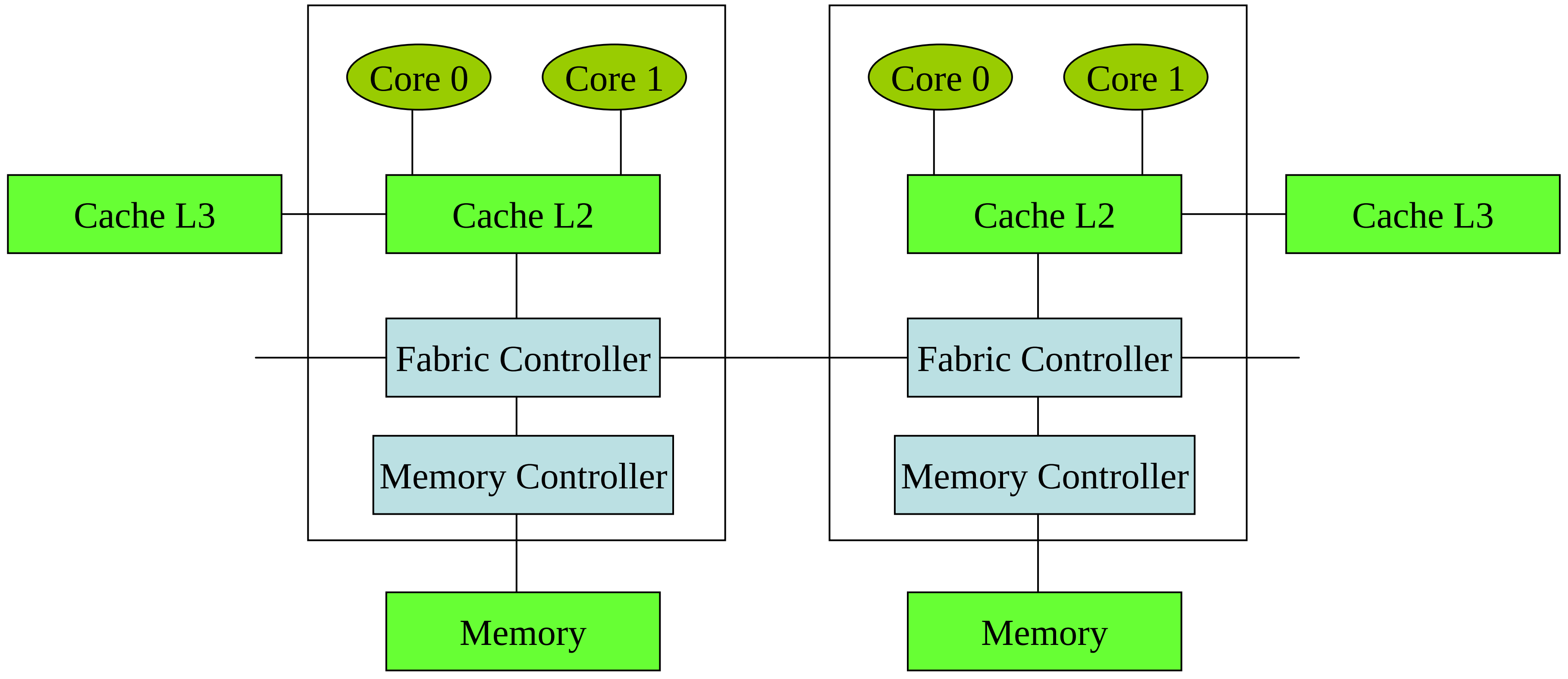
  a is selected first, its next deadline is 400ms which makes it the next candidate. Then, the deadline is 200 + 300 = 500 ms, after b. b is scheduled and its new deadline is 300 + 500 = 800 ms,…

# Multi-core systems

- The main challenge before the scheduler is 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. For example, the 2.6 Linux* kernel process scheduler introduced a concept called *scheduling domains* to incorporate the platform topology information into the process scheduler.

- 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

- Contention and its impact on performance depend on the resources shared, the number of active processes, and the 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.
- A fair amount of CPU time allocated to each process by the process scheduler will not essentially translate into efficient and fair usage of the shared resources.
- 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 benefit is power saving from idle packages that can switch both cores and L2 cache to sleep states.
- Scheduling processes on cores of different packages maximises execution speed but it’s not optimal in respect to energy.

# Process group scheduling

- If all the running processes are resource intensive, the challenge before the process scheduler is to identify the processes that share data and schedule them on the cores sharing the L2 cache. This will help minimize the shared resource contentions and shared data duplication. This will also result in efficient data communication between the processes that share data.
- The system software has some inherent knowledge about data sharing between processes. For example, 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.
- The process scheduler can do optimizations such as grouping threads belonging to a process or grouping processes attached to the same shared memory segment and co-schedule them in the cores sharing the package resources.
- In a scenario where all the shared resources and packages are busy, the process scheduler needs to minimize the resource contention for exploiting optimal performance. 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 schedule processes on the cores residing in a physical package with the goal of minimizing shared resource contention.

# Scheduling domains

- In a multi-core system, 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 by the domain. 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.
- 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

- 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 CPU within that domain with the lowest load. Similar decisions are made when a process forks.
- 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.
- 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.
- 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 CPU 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 most idle group. In doing so, it will take into account factors like the cache affinity time for the domain.

# Active balancing

- Active balancing is especially necessary when CPU-hungry processes are competing for access to a hyperthreaded processor. The scheduler will not normally move running processes, so a process which just cranks away and never sleeps can be hard to dislodge. The balancing code, by way of the migration threads, can push the CPU hog out of the processor for long enough to allow it to be moved and spread the load more widely.

- When the system is trying to balance loads across processors, it also looks at a parameter kept within the sched_group structure: the total "CPU power" of the group. Hyperthreaded processors look like independent CPUs, but the total computation power of a pair of hyperthreaded processors is far less than that of two separate packages. Two separate processors would have a "CPU power" of two, while a hyperthreaded pair would have something closer to 1.1. When the scheduler considers moving a process to balance out the load, it looks at the total amount of CPU power currently being exercised. By maximizing that number, it will tend to spread processes across physical processors and increase system throughput.

## Conclusions

- Scheduling is one of the most important functions of the kernel.
- Its algorithm is dictated by the nature of the applications run by that computer.
- Scheduler’s parameters can sometime be modified dynamically.
- All processes need to be treated fairly.
- Multi-core schedulers are more complex software that consider system topology and processes behaviour.

## References

- Brian L. Stuart, Principles of Operating Systems, 2009, Thomson Learning.
- http://www.intel.com/technology/itj/2007/v11i4/9-process/2-intro.htm
- http://lwn.net/Articles/80911/