Lecture 3

Process scheduling

I. Purpose of scheduling

- Historically, the CPU was allocated to one process until its completion – this strategy is known as *batch processing*.
- Then, the next strategy was to time-share the CPU among multiple processes ready to execute.
- As CPU is time-shared, processes compete for the next available time slice.

- The *scheduler* is the kernel process that executes 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 to run.
- A process takes control of the CPU by having its state restored in the CPU registers, after saving the state of the previous process.

Scheduling strategies: 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:
  - start 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.

Example: 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 increasing value 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: three processes with their execution time, a(40), b(60), c(20); T_{at} : average turnaround time

$$T_a = 40,\ T_b = 100,\ T_c = 120 \quad T_{at} = \frac{T_a + T_b + T_c}{3} = 260/3$$

In increasing value of execution time:

$$T_c = 20,\ T_a = 60,\ T_b = 120,\ T_{at} = 200/3$$

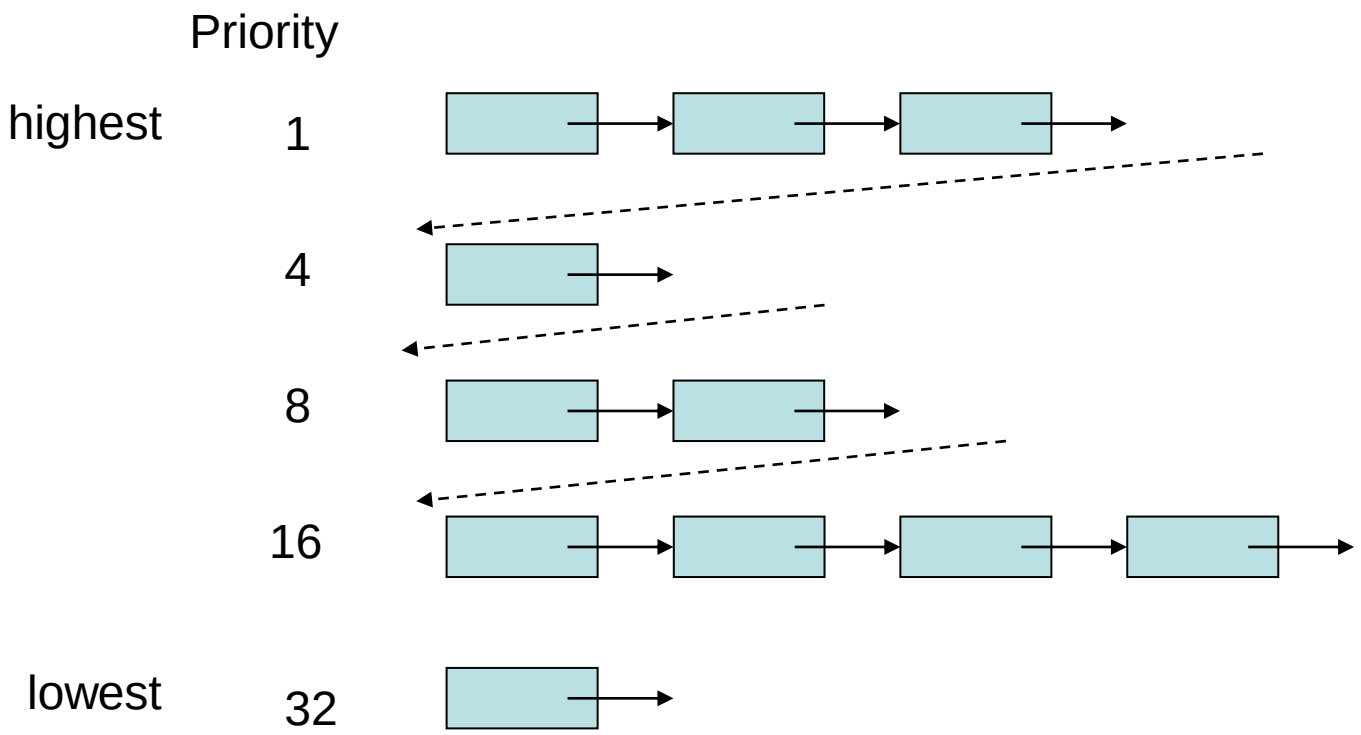
II. Priority scheduling

- Processes are different, some of them are interactive, others are computation demanding and therefore need to be dealt with differently in order to provide system responsiveness.
- One solution: assign priorities to processes. Fast, interactive processes will have higher priorities than computation strong ones.
- *Rule*: Kernel processes have higher priorities than user processes.
- The priority of user processes is given by considering the user or process attributes.
- If there are several processes with the same priority, they are scheduled in a round-robin manner.

- Priority is denoted by a small integer, generally a smaller value indicating a higher priority.
- Priority can be changed at runtime according to the process behaviour; for example, if a process takes too long to complete, its priority will be lowered.

Priority scheduling strategy: multilevel feedback queues

- This is one implementation of *dynamic priorities*.
- There are several queues, each associated with a priority level.
- Initially, a process gets a priority that puts it on a certain level.
- If not completed, after consuming the CPU time slice, the process 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, a process gets a higher priority (priority boost).
- As seen above, during its existence, one process can have a priority that varies within a defined range.



Multilevel feedback queue

The process for power management

- 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 makes some cleaning and switches the system into sleep state(-s).

- The idle process implements the kernel power policy manager. It 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 input data, such as end-user power policy, CPU utilization, battery level, or thermal conditions and events.
- The CPU driver is used to make actual state transitions on the kernel power policy manager's behalf.

Fairness: adjusting scheduling parameters

1. Priority

   – 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.

2. Time slice size

   – 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 $2^i q$.

Priority inversion

- Priority inversion occurs when two or more processes with different priorities are in contention to be scheduled. Consider a simple case with three processes. Process 1 is high priority and becomes ready to be scheduled. Process 2, a low-priority process, is executing code in a critical section. Process 1, the high-priority process, begins waiting for a shared resource from process 2. Process 3 has medium priority. Process 3 receives all the processor time, because the high-priority process (process 1) is waiting for shared resources from the low-priority process (process 2). Process 2 will not leave the critical section, because it does not have the highest priority and will not be scheduled.

- The scheduler solves the above problem by randomly boosting the priority of the ready processes (in this case, the low priority lock-holders). The low priority processes run long enough to exit the critical section, and the high-priority thread can enter the critical section. If the low-priority process does not get enough CPU time to exit the critical section the first time, it will get another chance during the next round of scheduling.

III. 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.

IV. Real-time scheduler

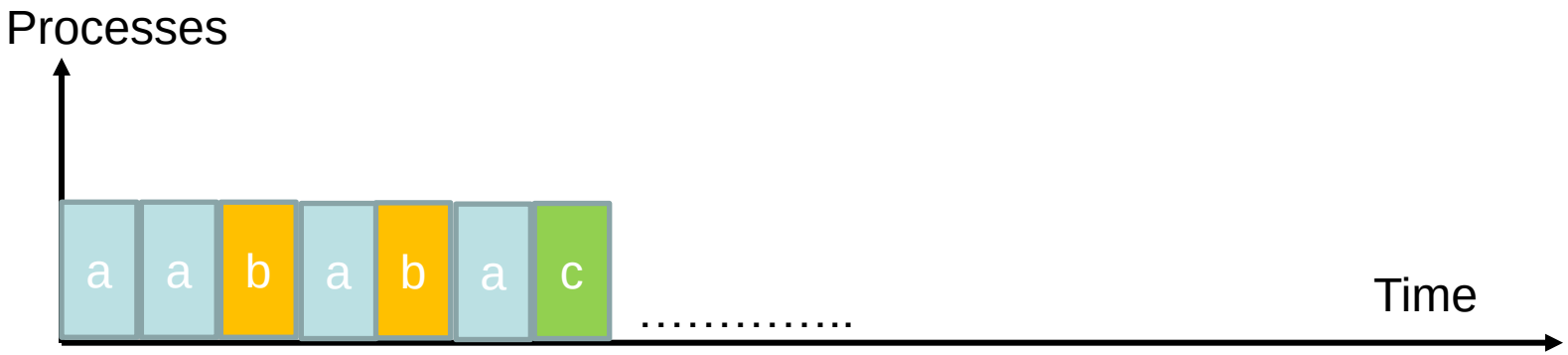
- In real-time applications, computing systems react to events signalled by interrupts. The interrupt triggers the scheduler to give control to the respective event handler process.
- If more than one occurs in the same time, priority and deadlines are considered. For example, if a new event has a higher priority or a tighter time requirement than the current running process, then the scheduler will preempt the existing one.
- One popular technique for real-time applications with deadlines 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 the attribute period to the time at which the time slice ended.

EDF example

- The time slice = 100 ms and three processes, *a* with attribute period of 300 ms, *b* with 500 ms and *c* with 1000 ms.
- All processes are ready at t = 0 and have deadlines equal to their attribute periods.

  a is selected first; after execution, its next deadline is 400 ms which makes it the next candidate again. The new deadline is 200 + 300 = 500 ms.

  b is scheduled and its new deadline is 300 + 500 = 800 ms,…



Conclusions

- Process scheduling is a key service of the kernel.
- Its algorithm is dictated by the nature of the applications run by that computer and the system configuration.
- Scheduler's parameters can sometime be modified dynamically.
- All processes need to be treated fairly.