Coordination and Scheduling Part 2

Concurrent and Parallel Programming

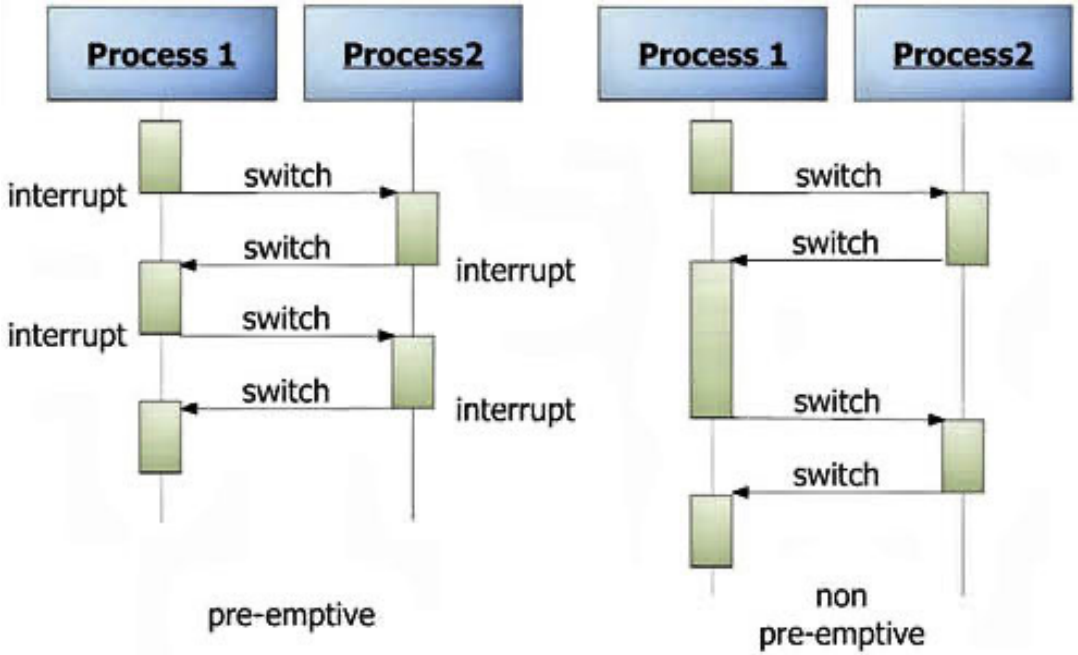
12 April 2022

Scheduling in operating systems

- Scheduling of processes and threads is one of the main functionalities of the operating system.
- The required infrastructure has a high complexity level.
- It has to determine when and how long processes and threads are allowed to run.
- In addition it is responsible for handling hardware-driven interrupts, and react to the use of synchronization primitives such as locks and waits.
- With the evolution of hardware and the introduction of multi-core processors, the scheduling infrastructure also evolved ...

Scheduling in operating systems

- Today, general-purpose operating systems implement preemptive scheduling as a solution for time-sharing.
- A specific interrupt associated with a timer wakes up the operating system scheduler at regular intervals.



Scheduling in operating systems

- In preemptive scheduling, the scheduler assigns to each thread a fraction of the execution time called the time-slice.
- At the end of each time-slice the thread execution can be suspended by the scheduler.
- When suspended, the thread is forced to pause the execution by performing a context-switch.
- The advantage of preemptive scheduling is that it is more fair to all the scheduled threads.

Scheduling in operating systems

In an operating system there are 3 main categories of schedulers:

- Long-term scheduler: responsible to select which process is accepted for execution and with which priority.
- Medium-term scheduler: temporarily removes processes from the main-memory and saves them to disk ("swap out") – in particular processes that are not running.
- Short-term scheduler (CPU scheduler): select which process/thread is allowed to execute. If it is a preemptive scheduler, it is activated at least for each time-slice or for interrupts, synchronization primitives (locks, waits, ...), OS calls or other signals.

Scheduling in operating systems

In addition, a very important role is played by the:

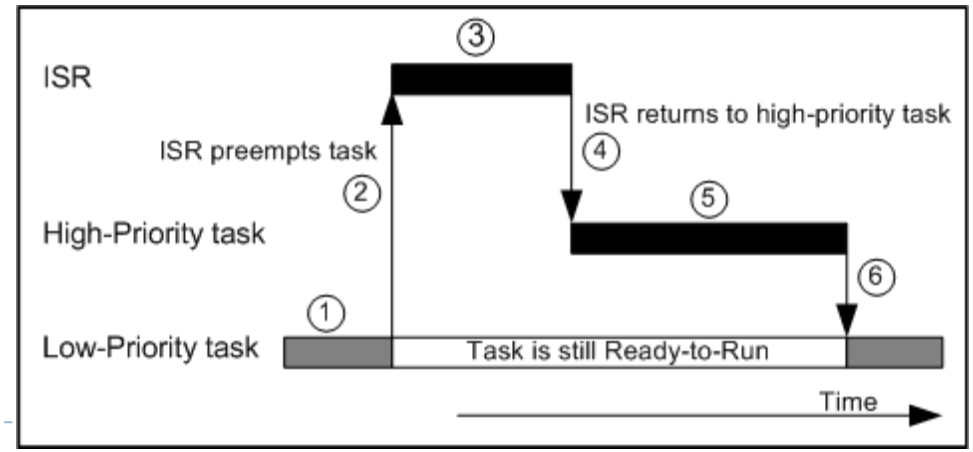
- Dispatcher: is the module that takes care of assigning the CPU control to the processes/threads. It is responsible for context-switches.

Scheduling strategies

- The short-term scheduler can be preemptive or non-preemptive (voluntary and cooperative).
- The scheduler is preempive, if it is able to suspend the execution of the processes/threads at regular intervals (time-slices).
- Today, non-preemptive scheduler are rare, probably used only in real-time operating systems for embedded processors.

Scheduling strategies

- Possible scheduling disciplines:
  - First-come, first-served (FIFO): processes/threads are scheduled in the order of arrival.
  - Round-robin: a fixed amount of time is assigned to each process/thread. Execution is performed in cycle.
  - Fixed-priority preemptive: the scheduler organizes the execution of the processes/threads according to their priorities.
  - …



Example

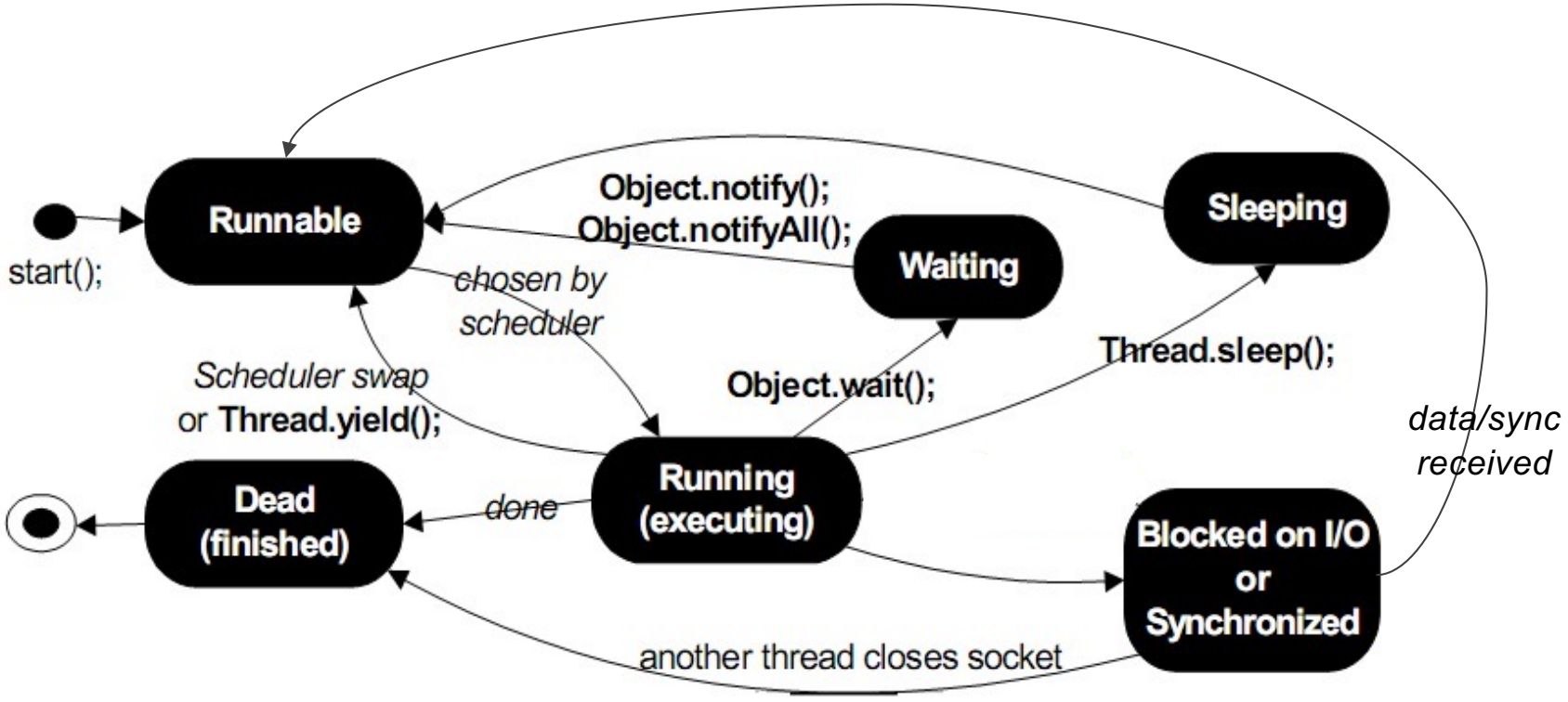
- Microsoft Windows uses a multi-level scheduler with feedback queue. A combination of first-in first-out, round-robin and fixed-priority preemptive scheduling.
- It allows to dynamically increase/decrease the priority of each process/thread depending on the waiting time.

Scheduling and Java

- If the Java Virtual-Machine is executed with a modern operating systems, there's no imposition on any specific approach for thread scheduling. The task is delegated to the operating system.
- Therefore, the behaviour of scheduling in Java is platform dependent!
- However, Java provides an abstraction, independent from the operating system, to manage the state of threads: tools such has sleep(), wait(), notify(), ...

States of threads in Java

- Possible states of threads in Java are:



Scheduling and Java

- As for any other program, the scheduling of threads in a Java program is performed based on time-slices, interrupts and other events triggered by the operating system.
- But it's not mandatory that a thread always completely consumes the assigned time-slice.
- In particular, scheduling also occurs when threads are suspended for instructions like sleep() and synchronization primitives (such as locks).
- yield() is an instruction that allows to activate the scheduler in an explicit way.

yield() and scheduling

yield() is a static method that interrupts (temporarily) the current thread and start the OS scheduler.

Depending on the used operating system:

- No time-slicing and no yield() calls: just one thread runs to completion at a time.
- No time-slicing and yield() calls: cooperative scheduling, threads voluntarily exchange the execution control.
- Time-slicing and no yield() calls: threads are scheduled preemptively and at locks, sleep(), ...
- Time-slicing and yield() calls: similar to the previous one, but with additional scheduling possibilities at yield(). yield() is used to facilitate the scheduling.

Example

```
public class ThreadDemo implements Runnable {
    private final Thread t;

    ThreadDemo(String str) {
        t = new Thread(this, str);
        t.start();
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            // yields control to another thread every 5 iterations
            if ((i % 5) == 0) {
                System.out.println(Thread.currentThread().getName()
                                   + " yielding control...");

                /* causes the currently executing thread to temporarily
                   pause and allow other threads to execute */
                Thread.yield();
            }
        }

        System.out.println(Thread.currentThread().getName()
                           + " has finished executing.");
    }

    public static void main(String[] args) {
        new ThreadDemo("Thread 1");
        new ThreadDemo("Thread 2");
        new ThreadDemo("Thread 3");
    }
}
```

Threads Priority in Java

- The priority of a thread is specified with an integer between 1 (lowest priority) and 10 (highest priority). It is also possible to use the constants Thread.MIN_PRIORITY and Thread.MAX_PRIORITY.
- The default priority is 5 = Thread.NORM_PRIORITY.

```
Thread aThread = Thread. currentThread();
int currentPriority = aThread.getPriority();
aThread.setPriority(currentPriority + 1);
```

Threads Priority in Java

- The effect of the priority may differ depending on the execution platform.
- Usually, larger priorities are used for threads that block frequently (sleeping or waiting for I/O), and medium/low priorities for threads that are CPU-intensive (to avoid saturation of the cores).

Stopping threads

- A Java application only completes when all its (non-deamon) threads have completed.
- But, if there is an interruption, how is it possible to inform the threads that the execution has to be prematurely stopped?

Stopping threads

- Java provides a specific infrastructure to manage the interruption of threads.
- With the interrupt() method, it is possible to assign 'true' to a boolean flag called interrupt status.
- The value of this flag can be queried by threads at regular intervals with the interrupted() or isInterrupted() methods.
- As a consequence, threads have the possibility to stop their execution.

Stopping threads

- Thread.interrupted() is a static method. Returns the value of the flag for the thread that is active at that moment (current thread) and clears the flag (set the value to 'false'). At the practical level, is used by the running thread to understand if an interruption has been requested and to stop the execution if needed.
- isInterrupted() is an instance method that leaves the value of the flag unmodified. Can be used by other threads to understand if an interruption has been requested for a specific thread.

Stopping threads

- But, ... when an interruption is needed, how is it possible to manage threads that are suspended because of a sleep() or a wait()? Is it possible to wake them up and stop their execution?
- Yes! Every call to the interrupt() method for a thread waiting or sleeping, forces the thread to exit the waiting/sleeping phase by generating an InterruptedException.
- Similarly to the Thread.interrupted() method, the InterruptedException clears the interrupt status flag.

Example (1/3)

```
class ThreadInterruptionDemo {
    public static void main(String[] args) {
        ThreadB thdb = new ThreadB();
        thdb.setName("B");
        ThreadA thda = new ThreadA(thdb);
        thda.setName("A");
        thdb.start();
        thda.start();
    }
}

class ThreadA extends Thread {
    private final Thread theOther;

    ThreadA(Thread theOther) {
        this.theOther = theOther;
    }

    @Override
    public void run() {
        final int sleepTime = (int) (Math.random() * 10000);
        System.out.println(getName() + " sleeping for " + sleepTime + " milliseconds.");
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
        }
        System.out.println(getName() + " waking up, interrupting other "
                                     + "thread and terminating.");
        theOther.interrupt();
    }
}
```

Example (2/3)

```
class ThreadB extends Thread {
    int count = 0;

    @Override
    public void run() {
        while (!Thread.interrupted()) {
            try {
                Thread.sleep((int) (Math.random() * 10));
            } catch (InterruptedException e) {
                System.out.println(getName() + " about to terminate...");
                // Because the Boolean flag in the consumer thread's thread
                // object is cleared, we call interrupt() to set that flag again.
                // As a result, the next consumer thread call to isInterrupted()
                // retrieves a true value, which causes the while loop statement
                // to terminate.
                interrupt();
            }
            System.out.println(getName() + " " + count++);
        }
    }
}
```

Example (3/3)

```
B 613
B 614
B 615
B 616
B 617
B 618
B 619
B 620
B 621
A waking up, interrupting other thread and terminating.
B about to finish...
B 622
```