Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής & Τηλεπικοινωνιών

Προηγμένες Μέθοδοι Προγραμματισμού

ΠΜΣ (M135.CS1E, M135.CS23B, M135.IC1E, παλαιό: M117)

Concurrent programming (1)

Δρ. Κώστας Σαΐδης (saiko@di.uoa.gr)

Processes

- A central concept in operating systems.
- Independently running programs that are (almost) isolated from each other:
  - A process has a self-contained execution environment (stack, registers, program counter).
  - A process has its own memory space.
  - A process <= application (an application may incorporate more than one processes).

Threads

- Threads are independent execution paths within the process, executing simultaneously and asynchronously to each other.
- Also called lightweight processes
    - similar to processes, threads have their own execution environment).
    - creating a new thread requires fewer resources than creating a new process.
- Threads exist/live in a process, sharing the same memory space.
- Every process has at least one thread.

The JVM Main Thread

- Every java application starts with one thread, called the main thread.
- Behind the scenes, the JVM spawns additional internal, housekeeping threads -- e.g., for garbage collection or for signal handling. Such threads are not visible to the developer.
- The main thread is responsible for running the main method: `public static void main(String[] args)`.
- Such a method should be available in the main class, provided as an argument during Java invocation (or manifested by the jar file specified by the -jar argument)

The Thread and the Runnable

- `java.lang.Thread` : Represents a thread of execution (a separate execution path in the application).
- `java.lang.Runnable` : Represents what a thread should be executing.
- The Thread is-a Runnable: the `Thread` class implements the `Runnable` interface. Yet its implementation of the `run()` method is empty (it should be overriden).

Creating threads

- Almost as simple as creating an instance of the `Thread` class; yet you should provide the code that will be run by the thread.
- There are two ways to do this:
  - Create an implementation of `Runnable` and pass it to the `Thread` constructor.
  - Subclass the `Thread` and override its `run()` method.

Example 01 (Runnable)

Example 02 (Thread Subclassing)

Example 03 (Thread Subclassing inline)

Starting and Stopping a thread

- Starting a thread involves a single method call: calling the `start()` method of a constructed `Thread` object.
- Stopping a thread, in many cases, is harder than starting it!
- We can't call the `stop()` method of the `Thread` object (unsafe, the caller cannot know whether the thread is ready to be stopped).
- To make the thread stop, try to implement `run()` to exit.

The `Runnable.run()` method

- The `run()` method is an ordinary java method (no magic).
- You can invoke it in the current thread -- without spawing a new one (that's why using `Runnable` is more flexible).

Example 04 (Reuse runnable)

The Threads do the Magic

- Threads have states.
- Threads have priorities.
- Threads can be daemons.
- Threads can sleep.
- Threads can be interrupted.

Thread scheduling

- Threads are scheduled in a non-deterministic manner (their execution does not provide any ordering guarantees).
- Thread priorities: threads with higher priority are executed in preference to threads with lower priority.
- The `Thread` class has `getPriority()`, `setPriority()` methods (1-10).

Thread States

- NEW: Created but not started.
- RUNNABLE: Started.
- BLOCKED: Waiting for the acquisition of a monitor/lock (discussed later).
- WAITING: Waiting indefinetely for another thread to perform a particular task.
- TIMED_WAITING: As above, but waits for a specific period of time (not indefinetely).
- TERMINATED: The thread's run method has exited.

Daemon vs User Threads

- A thread can be either a daemon or a user thread. By default all threads being created are user threads.
- A thread can be marked as a daemon through the `setDaemon(boolean)` method of the `Thread` class (should be called before the thread is started).
- The internal threads run by the JVM are daemon threads (e.g., the garbage collector).

When does a Java app terminate?

- The Java Virtual Machine exits when the only threads running are all daemon threads.

Thread sleep() and interrupt()

- The static method `Thread.sleep(long)` causes the current thread (the execution path that performs the call) to suspend execution for the specified period.
- The method throws an `InterruptedException`: when another thread interrupts a sleeping thread, the latter is awaken from the sleep by throwing a exception.
- Thread interruption is performed by the `interrupt()` method (interrupt the thread from whatever it is doing).

Thread interruption

- When a thread receives an interrupt its interrupted status is set (`isInterrupted()` will return true)
- When the interrupted thread is blocked/waiting by/for:
  - the `wait()` methods of the `Object` class,
  - the `join()` methods of the `{Thread}` class,
  - the `sleep()` methods of the `Thread` class,
  - then it will receive an `InterruptedException` and its interrupted status will be cleared (`isInterrupted()` will retu rn false).

Dealing with interruption

- It's your decision how a thread should respond to an interruption:
    - it is very common for the thread to just terminate (exit the `run()` method).
    - in our previous examples, the threads kept on running after an interrupt!

Note

- The semantics of interrupting a thread, depends on what the thread is doing.
- You should implement your thread with the possibility of interruption in mind and act accordingly, reflecting the business-logic of your application.
- A thead can always interrupt itself :-)

Example 05 (Sum of Factorials)

Threads can wait for other threads

- The `join()` method causes the caller's thread to wait for this thread to die (terminate).
- Consider two threads, `t1` and `t2`, where `t1` contains a call to `t2.join()` in its `run()` method.
  - When the call is reached, `t1` will pause and wait for `t2` to terminate.
  - When `t2` terminates, `t1` resumes its execution.
- There are `join()` variations that accept timeouts.

Example 06 (Impatient)

Thread Synchronization

- Threads can efficiently communicate with each other by sharing access to objects and fields (remember: threads operate on a common heap).
- However, two types of errors may occur when multiple threads access and modify common data:
    - Thread interference
    - Memory inconsistency
- Synchronization prevents these errors from happening.

Thread interference

Two operations, running in different threads, but acting on the same data, interleave (two operations that consist of multiple steps, overlap each other's sequences of steps).

Memory inconsistency

Two threads have inconsistent views of what should be the same data. This may happen during data transfer from/to the CPU registers/caches, the main memory and the thread-local registers/caches.

The fundamental Java synchronization Idioms

- The `synchronized` keyword (for methods and blocks of statements): allow only a signle thread to execute a block of code at a time.
- The `volatile` keyword, offering atomic reads and writes to variables.

Atomic Operations

- An atomic operation effectively happens all at once.
- It cannot stop in the middle: it either happens completely, or it doesn't happen at all.
- No side effects of an atomic operation are visible until the operation is complete.

Non-atomic Read/Update/Write operations

Such as

```
count++
```

or

```
count--
```

The synchronized keyword

- Its overall purpose is to allow only one thread at a time into a particular section of code (the critical section).
- Every instantiated Java object, including every Class loaded, has an associated lock (or monitor).
- Putting code inside a synchronized block makes the compiler append instructions to acquire the lock on the specified object before executing the code, and release it afterwards (either because the code finishes normally or abnormally).

Re-entrant locking

- Between acquiring the lock and releasing it, a thread is said to own the lock.
- At the point of Thread B wanting to acquire the lock, if Thread A already owns it, then Thread B must wait for Thread A to release it.
- Threads are allowed to acquire a lock they already own (re-entrant locking).

Example 07 (Counter & Synchronized Counter)

Volatile variables

- Declare a variable as volatile to indicate that it is expected to be modified by different threads.
- The value of a volatile variable will never be cached by threads locally: all reads and writes will go straight to main memory.
- In other words, the volatile keyword indicates that reads and writes to the variable's value should be atomic. Changes to a volatile variable are always visible to other threads.

When to use a volatile

- Use volatiles when the set of valid values that can be written to a variable is independent of any other program state, including the variable's current state.
    - E.g. when you use a boolean variable as a thread exit flag.
- Don't use volatile for:
    - arrays: only the reference to the array becomes volatile, not the array's items.
    - read-update-write operations (such as incrementing a counter).
    - complex operations that need to prevent access to a variable for the duration of the operation (use