Aalto University
School of Science

CS-E4110
Concurrent Programming

Week 3 – Exercise session

2021-11-19

14:15 – 16:00

Jaakko Harjuhahto

Today

- Semaphores
- Standard APIs
- Reactor assignment
- General Q&A

Semaphores

Semaphore Semantics

- A semaphore is a compound type of a non-negative integer and a set of waiting processes
  - The value of the integer is often called the number of permits or the number of tickets
- The two operations `wait` and `signal` are atomic and mutually exclusive
  - With P(s) a process decrements s (> 0) by one. If s is 0, the process is put to wait “in P(s)” until s is positive so that s can be decremented and the process may proceed
  - With a V(s) a process increments s by one. If s was 0, and there are one or more processes waiting “in P(s)”, one of them may complete P(s) and proceed

Terminology

Decrement	Increment	
`wait(s)`	`signal(s)`	Standard synchronization terminology
`P(s)`	`V(s)`	Classic shorthand from Dijkstra
Prolaag	Verhoog	E.W.Dijkstra's original terminology
`s.acquire()`	`s.release()`	Java API termilogy
`down(s)`	`up(s)`	Linux kernel

Even more variants exist...

Example: Simple Critical Section

- Since the semaphore only has a single permit available, if a thread is inside the critical section, other threads encountering the `wait()` will be placed in the wait set
  - `signal()` will wake a thread, which can then acquire a permit and proceed

```
val sem = new Semaphore(1)


sem.wait()    //permits -1
//critical section
sem.signal()  //permits +1
```

Example: Semaphore With JVM Monitors

```
class MySemaphore(var tickets: Int) {

  def wait(): Unit = ???

  def signal(): Unit = ???

}
```

Gotchas

- The semaphore can be an object with methods
  - E.g. `semInstance.wait()`
- The semaphore can be a struct that is operated on by static methods
  - E.g. `sem_wait(semInstance)`
- The initial ticket count of a general semaphore is not an upper limit for tickets or a maximum capacity
  - Example: It is valid to initialize a semaphore with a ticket count of 2 and then increase it to 5 by making three calls to `signal()`

Standard APIs

Scala & Java Standard APIs

- Most of the standard library is **not** thread safe
  - Default assumption: calling API methods from multiple threads will break something, especially if calls happen concurrently
    - Lack of mutual exclusion
    - Lack of happens-before
  - The few thread safe exceptions are documented
- We can always implement a thread safe variant of an API class by writing a new class that wraps the methods of the original class in synchronized methods

Standard Library

- On this course, we **implement** concurrent code using threads, JVM monitors and other language level primitives to **learn** concurrent programming
- For practical day-to-day usage outside this course, the standard library offers APIs for concurrency
  - `java.util.concurrent`
  - `scala.concurrent`
- The same logic applies to other languages: there is a good chance that a standard library or a common framework might already have what you need

Examples

- Concurrent collections (lists, queues, maps, sets etc.)
- Atomic variables
  - e.g. `AtomicInteger, AtomicReference`
- Explicit locks
  - e.g. `ReadWriteLock`
- Executors
  - Don’t explicitly create threads, instead submit items of work, e.g. tasks, to a service that will take care of executing these items
- Synchronization constructs
  - `CyclicBarrier`, `CountDownLatch`, `Phaser`
- We will discuss parallel collections, futures and promises in more detail later on during this course

Reactor Assignment

Reactor Assignment

- Course mini-project in three phases
  - Objective: implement a small networked game server
  - Problem: network programming is inherently concurrent, as we must listen to multiple clients concurrently
  - Solutions: use a design pattern to separate concurrency issues from the game logic
- A step up in complexity compared to A+ exercises
- Tasks define an API and behaviour
  - You may choose how to implement your own solution
  - Several good solution archtypes for the problems exist

The Reactor Pattern

- Design patterns are re-usable solutions to common problems in software design
- The reactor is a mechanism for changing an inherently concurrent problem into an asynchronous but sequential problem
  - All the necessary threading and locking is implemented inside the reactor and the application built on top of the reactor can be written as sequentially executed event handlers
  - Our use case: responding to multiple network clients
- Similar designs: JavaScript interpreter event loop, UI toolkit dispatchers, etc…

Common Terminology

- Handles represent sources of data
  - Reading the handle will produce an element of data
  - May block while reading, until data becomes available
    - E.g. a network client sends data
- Events contain an element of data and a handler
- Handlers are code which accepts the element of data and does something with it, typically changes program state
  - E.g. ”when a new player connects (= the event), register the player with the game, so that they will get updates from the game server”

Task A – BlockingEventQueue

- Theme: JVM monitors and synchronization
- Implement a FIFO bounded buffer for storing objects
  - Blocks if trying to get from an empty buffer
  - Blocks if trying to put into a full buffer
  - The buffer contains events of an generic type A
    - I.e. `class BlockingEventQueue[A]`
- The base API defines four methods:
  - `enqueue()`, `dequeue()`, `getSize()` and `getCapacity()`
    - Only `enqueue()` and `dequeue()` are necessary for Task-B
- Two optional features give extra points
  - `getAll()` method and notification efficient design

Task-A Degrees of Difficulty

Features	Points	Difficulty
Base API	60	★
API + getAll	60 + 15	★★
API + notification efficient	60 + 15	★★
API + getAll + notification efficient	60 + 15 + 15	★★★

Beware: Implementing the optional goals can break the base API implementation and introduce new bugs

Task B – Dispatcher

- Theme: thread management and design patterns
- The Dispatcher maintains a registry of handles
- It is responsible for starting threads to read handles when they are registered
  - Also, must ensure that the started threads terminate correctly
- Two fundamental requirements:
  - The dispatcher must be able to read all the registered handlers concurrenctly
  - The event loop must processes events from handles sequentially, one at a time, in a single thread

Task C – The Game

- Theme: asynchronous/callback programming
- Implement the Hangman game on top of the reactor
  - Wikipedia: https://en.wikipedia.org/wiki/Hangman_(game)
  - The ruleset is implemented for you in the template
- You must implement game logic by defining handlers that change game state when certain events occur
  - Events are, for example: player joins the game, player makes a guess, etc
  - As the game is a network server, you don't know the order incoming of events

The Template

- A ZIP package with Scala classes you must implement
  - Includes the necessary API classes
  - Utility classes you can freely use as a part of your Task-C
- The template contains minimal unit tests
  - Only check that basic semantics work, the tests do not exhaustively test concurrency behavior
  - You can of course implement more tests…
- The same instructions as in A+

Assignment Practicalities

Submission

- Each task is returned to A+ as a single `.scala` file
  - If your solution needs multiple classes, implement them in the same source file
- Distinct deadline for each ask
  - Task-A Monday 29th of November at 14:00
  - Task-B Monday 6th of December at 14:00
  - Task-C Monday 13th of December at 14:00
- Only the last submission at the time of the DL is graded
  - You can make up to 100 submissions before the DL

Grading (1/2)

- The assignment submissions are graded manually
  - The target for publishing the grading is the next Friday
- After the grading, you receive feedback and points
  - The feedback gives you a breakdown of how the point score is composed and refers to specific issues with the submission
- Each task is worth 100 points
  - Up to 90 points for the implementation code
  - Up to 10 points for documentation (= comments)

Grading (2/2)

- For each part, the submission starts with a base level of points
  - Base is 60 points for Task-A and 90 points for Tasks B and C
  - For Task-A, optional features give +15 and +15 points
- Errors, bugs, design flaws and deviations from the specification **deduct** points, depending on the severity
  - For example, if the design can potentially deadlock: -25 points
  - Beware stacking bugs

Documentation

- Part of the grading criteria
  - The comments should explain why your implementation is correctly synchronized as defined in the Java Language Specification
  - Help the TAs understand your solution
  - In borderline cases, the documentation is used to determine if a concurrency problem was the result of a coding error (minor point penalty) or a design flaw (larger penalty)
- Documentation gives up to 10 points for each Task

Q&A