Aalto University
School of Science

Synchronization

CCS-E4110 Concurrent Programming
Autumn 2021 (II), week 3

Vesa Hirvisalo

Week 3 in a nutshell

- Monitors
  - Waiting and notifying those that wait
- Semaphores
  - Synchronization with the ability to count
- Messages
  - Use communication channels instead of shared memory
- Some classic (textbook) problems
  - The producers–consumers problem
  - The readers–writers problem
  - The dining philosophers problem

Towards the Mini-Project

Theory and Practice

- Understanding the relationships between the theory and practical programming is important
  - Concurrent programming are often hard to debug
- Remember to read to course textbook
  - Weekly reading: Chapters 1 & 2 (basic/concurrency)
  - Weekly reading: Chapters 3 & 5 (Critical Section)
    - Check also Ch4, but deepen your understanding with it later on
  - Weekly reading Chapters 7 & 6 (Monitors, etc.)
  - The rest: read selectively toward the end of the course
    - There is some theory useful with rest of the exercises
    - Some useful considering the exam
      - The need depends much on the grade you are targeting
      - Note the technical material pointed in the slides (not in the book)

The mini-project

- The mini-project is a real programming challenge
  - Not trivial: you have to think (and understand the theory!)
  - It is based on the **reactor pattern**
    - A common concurrent programming programming pattern
    - An event handling pattern for handling service requests delivered concurrently to a service
- Will be available from week 3 on
  - To be done in three phases
  - Submit in proper order
  - Read the instructions carefully
    - They **include definitions** (and definitions for concurrent programs are very often hard to understand!)

More on Monitors

Monitors: A generic view (beyond Scala)

- A monitor is a data abstraction construct
  - A monitor consist of one or more methods (operations of the data abstraction), an initialization sequence and local data
  - Local data is accessible only by the monitor's methods
  - A thread enters the monitor by invoking one of its methods
  - Only one methods may be executing the monitor at a time
- Synchronization is done with condition variables
  - Threads can exchange signals
    - Only **within** the monitor (simplifies the use)
  - cwait(c) suspends the execution the caller
  - csignal(c) resumes the execution a thread waiting for signal "c"
    - If there are none, do nothing (the signal is "lost")
    - If there are several threads waiting, one is chosen by the monitor
- Note: **Scala monitors are simpler** than in many languages

Waiting in Scala Monitors

- In Scala, waiting makes it possible to wait for generic notifications on an object instance
- The `wait()` method
  - When executing the `wait()` method of an object, the thread

  1. adds itself to the object’s wait set, and
  2. releases the object’s lock (fully, even if multiply acquired), ensuring all variable values written will be visible to the thread that acquires the lock.
  3. If the thread holds multiple locks, only the one for which `wait()` is called is resumed.

Waking up from `wait()`

- A waiting thread will continue execution:
  - When another thread notifies the waiting thread with `notify()` or `notifyAll()` on the same object.
    - The waiting thread must then reacquire the object's lock, just like when entering a synchronized block (and restore re-entrancy state). Values written by the last holder of the lock are ensured to be visible.
    - The thread may have to compete with other threads to obtain the lock, because it does not receive any form of special priority.
  - When the thread is interrupted with `Thread.interrupt()`
  - For no apparent reason (spurious wakeup)

Spurious wake-ups

- Scala is affected by POSIX
  - POSIX allows spurious behavior because condition variables are otherwise hard to implement efficiently on some multiprocessor systems
  - Linux also has a habit of generating spurious wakeups when processes are signaled.
- From a performance point of view, you can usually assume that spurious wakeups are rare
- The behavior is easy to mitigate: always use a while loop to check the wait condition

Notifying

- The monitor semantics in Scala are rather loose
  - Some systems, the consequences of signaling are very strictly defined (more control but harder to implement)
- The `notify()` method.
  - The `notify()` method wakes up one thread waiting on an object's wait set. If there are no threads waiting at the moment, then `notify()` does nothing
  - If there are multiple threads waiting, the thread is chosen arbitrarily
  - The notified thread is removed from the object's wait set
  - `notify()` does not give up ownership of the object's lock

Resuming the execution

- A notified thread can resume execution only after it re-obtains ownership of the object's lock.
  - May happen only after the notifying thread has given up the lock
  - The notified thread may have to compete with other threads to obtain the lock
- The `notifyAll()` method
  - The `notifyAll()` method works just like `notify()`, but instead of one thread, it wakes up all threads waiting on the object's wait set
  - The notified threads must compete to reobtain lock ownership
- Note the difference between resuming and waking up

Interrupting

- Waiting threads can be interrupted using by calling `interrupt()` on the thread
- `wait()`, `join()` and such throw an `InterruptedException` when interrupted
- If the thread is not waiting, its interrupt flag is set.
  - The methods above also check the interrupt flag before they start waiting and are interrupted if it is set
  - The interrupt flag is read and cleared (atomically) by the static `interrupted()` method
  - The interrupt flag is also cleared when throwing and does `InterruptedException` above
  - The thread instance method `isInterrupted()` does not affect the interrupt status

Semaphores

Overview of semaphores

- In classical concurrency literature a semaphore is a shared, non-negative integer variable operated exclusively with two atomic operations: wait and signal
- Calls to wait and signal are written as P(s) and V(s), where s is a semaphore
- 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

Specifying semaphores

- A semaphore is a compound data type of: the integer value of the semaphore and the set of processes waiting on the semaphore
- The value of the semaphore is denoted as S.V
- The wait set is denoted as S.L
- If the value of the semaphore can take arbitrary non-negative values the semaphore is a general semaphore
- If the value can be only 0 or 1, the semaphore is a binary semaphore (also called a mutex)

Generic semaphore

Assumes atomic execution of code inside wait and signal

General semaphore initalized with the value k	
semaphore S $\leftarrow$ (k, $\emptyset$)	
wait(S)	signal(S)

if S.V > 0
    S.V $\leftarrow$ S.V - 1
else
    S.L $\leftarrow$ S.L $\cup$ { p }
    p.state $\leftarrow$ blocked
	if S.L = $\emptyset$
	    S.V $\leftarrow$ S.V + 1
	else
	    for an arbitrary q $\in$ S.L
	    S.L $\leftarrow$ S.L $\setminus$ { q }
	    q.state $\leftarrow$ ready

Binary semaphore

Assumes atomic execution of code inside wait and signal

Binary semaphore	
binary semaphore S $\leftarrow$ (1, $\emptyset$)	
wait(S)	signal(S)

if S.V > 0
    S.V $\leftarrow$ S.V - 1
else
    S.L $\leftarrow$ S.L $\cup$ { p }
    p.state $\leftarrow$ blocked
	if S.V = 1
	    // undefined
	else if S.L = $\emptyset$
	    S.V $\leftarrow$ 1
	else
	    for an arbitrary q $\in$ S.L
	    S.L $\leftarrow$ S.L $\setminus$ { q }
	    q.state $\leftarrow$ ready

On semaphore semantics

- How to choose the thread (q) from S.L?
  - Depending how the thread q is selected from among the blocked threads affects how the semaphore will behave
  - If S.L is a set and q is chosen arbitrarily (as with the previous implementations) the semaphore is *weak* and starvation is possible
  - If S.L is a first-in-first-out (FIFO) queue, the semaphore is *strong* and offers freedom from starvation
- Is S.L prioritized over new entries?
  - The set of threads waiting on a semaphore is well defined when signal(s) is invoked
    - Since the operations are atomic and mutually exclusive, S.L stays well defined during signal(s) and q is selected from S.L

Message passing

Introduction to message passing

- In message passing, message are sent and received
  - Asynchronous operation
    - Messages are buffered
  - Different interfaces, e.g.,
    - send(destination, message)
    - receive(source, message)
  - There can be filtering or peeking (selective receive)
- Exchange of messages imply order of events
  - Can be used in distributed systems
    - i.e., does not require shared memory
- Synchronous messaging
  - This is rather different (and not so often used)

On the messages

- Typical contents
  - Header: type, destination addr, source addr, length, control info
  - Body: data
- The data can differ
  - What data types are supported
  - Fixed or variable length size?
- Pass-by-reference or pass-by-value?
  - if a large amount of data to be send, send a reference instead
    - However, sharing data this way is complicated
- Data representation
  - Data formats may differ (marshalling, unmarshalling)

Messages, locks, spinlocks, mutexes, …

- Distributed systems typically use messages
  - Sharing a communication channel (instead of sharing a memory)
- There are a number of synchronization abstractions
  - Some words have more fixed meaning (e.g., semaphore, monitor)
  - Some are more vague (e.g., lock, mutex)
  - Many of them have several variations
    - Understand the semantics before using!
- Usually (the concepts like process, thread, etc. can affect)
  - Locks have ownership (a semaphore does not)
    - Semaphore can be locked by one thread and released by another while a mutex or lock cannot
    - Locks are often programming language features (monitors in practice)
  - Mutexes are systemwide (lock is typically local)
  - Spinlocks are aggressive locks (no voluntary suspension)
- And note: there can be faults, exceptions, etc.

Classical examples

The textbook classics

- Concurrent programming is not easy
  - The correctness of concurrent programs is hard to check
  - But also, it is rather hard compactly define problems
- The typical pedagogical solution
  - Use the “classical examples”
  - They illustrate programming, abstractions, and implementations
- Some of the usual ones
  - Producers/consumers
  - Readers/writers
  - Dining philosophers

The producers–consumers problem

- A shared buffer in the memory
  - Therefore also called “the bounded buffer problem”
- One or more producers
  - Generating data and adding it the buffer
- One or more consumers
  - Taking data from the buffer
- The synchronization problem
  - Only one process can access the buffer at a time
  - A consumer cannot remove data from an empty buffer
  - Finite buffers
    - Producers cannot write to a full buffer

The readers–writers problem

- There is some shared data that can be written and read
  - E.g., a contents of a file
- Concurrent access
  - Any number of readers can simultaneously read the data
  - Only one writer may write the data
  - If a writer is writing the file, no reader can read it
- Not the same as producers–consumers
  - The is no joint buffer
    - A consumer gets items from any of the producers
- Not the same as generic mutual exclusion problem
  - Readers–writers problem does always not call for total exclusion

Dining philosophers

- There is
  - A round table with N philosophers and N forks
    - N plates with spaghetti (but actually, this is irrelevant)
  - Forks are placed between philosophers
    - Each philosopher access only the forks immediately left of right
- A philosopher
  - Picks and puts down forks, needs two forks to eat
- The problem
  - They should be able to eat concurrently
  - Obviously, not all of them can eat at the same time
  - But, actually `floor(N/2)` can eat (and should eat)!
- This problem demonstrates many concurrency issues

Related readings and exercises

=> Check A+