Lecture 4: advanced synchronization concepts

interruption, cancellation, partitioning, privatization, replication, thread-local, ownership

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l4.pdf

In previous episodes

We know basic techniques for thread coordination

- Mutual exclusion
- Signalling: one-to-one and broadcast
- Group-level concurrency
- Pooling of computation agents and separating task from particular executor

There are plenty of concurrent data structures that implement those ideas:

- `Lock`, `Condition`, `Monitor`, `CountDownLatch`, `Semaphore`, `ReadWriteLock`

You still need to be aware of

- Safety, Liveness, Performance
- Race conditions, data races, visibility
- Progress guarantees: livelock, priority inversion, deadlock, fairness
- Logical errors related to signalling: lost signal, spurious wakeup, predicate invalidation
- Implementation-specific performance caveats: lock convoy, thundering herd, oversubscription, resource-constrained deadlocks

Teaser

Today we will add a new **dimension** for all previously studied problems.

Teaser

Today we will add a new **dimension** for all previously studied problems.
At arbitrary moment user decides to properly cancel particular concurrent task.

Teaser

Today we will add a new **dimension** for all previously studied problems.
At arbitrary moment user decides to properly cancel particular concurrent task.
Without ruining the whole system.

Teaser

Today we will add a new **dimension** for all previously studied problems.
At arbitrary moment user decides to properly cancel particular concurrent task.
Without ruining the whole system.
It could become overwhelmingly complicated, so we will study basic decomposition techniques for concurrent programs.

Teaser

Today we will add a new **dimension** for all previously studied problems.
At arbitrary moment user decides to properly cancel particular concurrent task.
Without ruining the whole system.
It could become overwhelmingly complicated, so we will study basic decomposition techniques for concurrent programs.
Main idea: control program complexity, avoid preliminary optimization.

Lecture plan

1. Cancellation
   - Problems with forced cancellation
   - Cooperative cancellation
   - Design space
   - Timeouts
2. Concurrent problem decomposition techniques
   - State machine
   - Function shipping
   - Designated thread
   - Partitioning threads
   - Partitioning data
   - Privatization
   - Replication
   - Ownership
3. Summary

Cancellation/interruption

Program initiated some task and then

- user is no more interested in the result
- expected result is known to be out-of-date
- a lot of tasks with higher priority appeared
- program intentionally started several identical tasks[1]

[1] https://cacm.acm.org/research/the-tail-at-scale/

Cancellation/interruption

Program initiated some task and then

- user is no more interested in the result
- expected result is known to be out-of-date
- a lot of tasks with higher priority appeared
- program intentionally started several identical tasks[1]

Need to cancel (interrupt) running task to save CPU/RAM/network bandwidth/...

[1] https://cacm.acm.org/research/the-tail-at-scale/

Question time

Question: Terminate running thread at arbitrary execution point: could you predict any problems?

Forced cancellation

Idea

Operation systems allow killing processes and threads:

- `kill -9 <PID>`[2]
- `int pthread_kill(pthread_t thread, int sig)`[3]

[2] https://man7.org/linux/man-pages/man1/kill.1.html

[3] https://man7.org/linux/man-pages/man3/pthread_kill.3.html

Forced cancellation

Idea

Operation systems allow killing processes and threads:

- `kill -9 <PID>`[2]
- `int pthread_kill(pthread_t thread, int sig)`[3]

Not exactly. They provide mechanism to *send a signal* to running thread which *asynchronously* "transfer" thread execution flow into signal handler.

[2] https://man7.org/linux/man-pages/man1/kill.1.html

[3] https://man7.org/linux/man-pages/man3/pthread_kill.3.html

Forced cancellation

Idea

Operation systems allow killing processes and threads:

- `kill -9 <PID>`[2]
- `int pthread_kill(pthread_t thread, int sig)`[3]

Not exactly. They provide mechanism to *send a signal* to running thread which *asynchronously* "transfer" thread execution flow into signal handler.
By default, unhandled "important" signal (e.g. `SIGKILL`) will stop thread execution.

[2] https://man7.org/linux/man-pages/man1/kill.1.html
[3] https://man7.org/linux/man-pages/man3/pthread_kill.3.html

Forced cancellation

Idea

Operation systems allow killing processes and threads:

- `kill -9 <PID>`[2]
- `int pthread_kill(pthread_t thread, int sig)`[3]

Not exactly. They provide mechanism to *send a signal* to running thread which *asynchronously* "transfer" thread execution flow into signal handler.
By default, unhandled "important" signal (e.g. `SIGKILL`) will stop thread execution.
Some concurrent systems also provide facilities for forced asynchronous thread cancellation[4].

[2] https://man7.org/linux/man-pages/man1/kill.1.html
[3] https://man7.org/linux/man-pages/man3/pthread_kill.3.html
[4] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#stop()

Forced cancellation

Problems

Program could be stopped at arbitrary point:

- In the middle of file writing (corrupted data)
- In the middle of DB connect (lost data)
- In the middle of mutex critical section (lock leakage, deadlock)

Forced cancellation

Problems

Program could be stopped at arbitrary point:

- In the middle of file writing (corrupted data)
- In the middle of DB connect (lost data)
- In the middle of mutex critical section (lock leakage, deadlock)

Common problems:

- inconsistent global state
- non-recoverable data corruption

Forced cancellation

Problems

Program could be stopped at arbitrary point:

- In the middle of file writing (corrupted data)
- In the middle of DB connect (lost data)
- In the middle of mutex critical section (lock leakage, deadlock)

Common problems:

- inconsistent global state
- non-recoverable data corruption

Additional level of complexity: cancellation **inside** cancellation handler.

Forced cancellation

Problems

Program could be stopped at arbitrary point:

- In the middle of file writing (corrupted data)
- In the middle of DB connect (lost data)
- In the middle of mutex critical section (lock leakage, deadlock)

Common problems:

- inconsistent global state
- non-recoverable data corruption

Additional level of complexity: cancellation **inside** cancellation handler.
Conclusion: forced cancellation in concurrent systems is **bad design**[5]

[5] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/doc-files/threadPrimitiveDeprecation.html

Question time

Question: Name a highly concurrent multi-agent system which is tolerant to arbitrary agent failure by design. Hint: you use it every day.

Lecture plan

Cooperative cancellation

Idea

- `Thread A` initiates cancellation
- `Thread B` performs cancellation handler
- `Thread A` finishes cancellation protocol

Cooperative cancellation

Idea

- `Thread A` initiates cancellation
- `Thread B` performs cancellation handler
- `Thread A` finishes cancellation protocol

Design challenges:

- How to pass information from one thread to another
  - reasonable response time vs. non-prohibitive throughput overheads
- When to handle cancellation request
  - immediately, even inside of "critical code path"
  - defer processing
  - ordering among received requests
  - deduplication/queueing of requests
- What to do after cancellation handling
  - Send confirmation response
  - Stop thread/throw exception/continue execution

Cancellation token

```
final class CancellationToken {
    private boolean canceled = false;
    public synchronized boolean isCanceled() { return this.canceled; }
    public synchronized void cancel() { this.canceled = true; }
}
```

Cancellation token

```
final class CancellationToken {
    private boolean canceled = false;
    public synchronized boolean isCanceled() { return this.canceled; }
    public synchronized void cancel() { this.canceled = true; }
}
```

- How to pass information from one thread/task to another: lock + state variable
- When to handle cancellation request: when thread/task is ready to cooperate
- What to do after cancellation handling: up to receiver

Cancellation token

```
final class CancellationToken {
    private boolean canceled = false;
    public synchronized boolean isCanceled() { return this.canceled; }
    public synchronized void cancel() { this.canceled = true; }
}
```

- How to pass information from one thread/task to another: lock + state variable
- When to handle cancellation request: when thread/task is ready to cooperate
- What to do after cancellation handling: up to receiver

Maybe useful in single-threaded [6] and in multi-threaded[7] environment.

[6] https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal
[7] https://learn.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken

Cancellation token

```
final class CancellationToken {
    private boolean canceled = false;
    public synchronized boolean isCanceled() { return this.canceled; }
    public synchronized void cancel() { this.canceled = true; }
}
```

- How to pass information from one thread/task to another: lock + state variable
- When to handle cancellation request: when thread/task is ready to cooperate
- What to do after cancellation handling: up to receiver

Maybe useful in single-threaded [6] and in multi-threaded[7] environment.
Important: CancellationToken works the same for `Threads` and for `Tasks`.

[6] https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal
[7] https://learn.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken

Cancellation token

- easy to implement in any language
- straightforward implementation and behaviour
- supports both **thread cancellation** and **task cancellation**
- languages **with exceptions** suffer (`CancellationException` from almost any method)

Cancellation token

- easy to implement in any language
- straightforward implementation and behaviour
- supports both **thread cancellation** and **task cancellation**
- languages **with exceptions** suffer (`CancellationException` from almost any method)
- manual polling everywhere
  - before *any* blocking operation (synchronization, I/O)
  - in long loops or deep recursion
- requires standard library/external libraries support
- bad composability (how many cancellation tokens should be passed to function `foo`?)
- languages **without exceptions** suffer (which value to `return` if cancellation requested)

Cancellation token

- easy to implement in any language
- straightforward implementation and behaviour
- supports both **thread cancellation** and **task cancellation**
- languages **with exceptions** suffer (`CancellationException` from almost any method)
- manual polling everywhere
  - before *any* blocking operation (synchronization, I/O)
  - in long loops or deep recursion
- requires standard library/external libraries support
- bad composability (how many cancellation tokens should be passed to function `foo`?)
- languages **without exceptions** suffer (which value to `return` if cancellation requested)

What if thread/task:

- not started yet, already finished, already received cancellation

Cancellation token

- easy to implement in any language
- straightforward implementation and behaviour
- supports both **thread cancellation** and **task cancellation**
- languages **with exceptions** suffer (`CancellationException` from almost any method)
- manual polling everywhere
  - before *any* blocking operation (synchronization, I/O)
  - in long loops or deep recursion
- requires standard library/external libraries support **for better responsiveness**
- bad composability (how many cancellation tokens should be passed to function `foo`?)
- languages **without exceptions** suffer (which value to `return` if cancellation requested)

What if thread/task:

- not started yet, already finished, already received cancellation

Question time

Question: Imagine that we could associate a single cancellation token with every thread. Which parts of standard library should you adapt to support such token?

Language support for cancellation points: low-level parts

Common "long" methods:

- `Thread.sleep`
- File I/O
- User input
- Networking

Language support for cancellation points: low-level parts

Common "long" methods:

- `Thread.sleep`
- File I/O
- User input
- Networking
- User-specified functions[8]

[8] https://man7.org/linux/man-pages/man3/pthread_testcancel.3.html

Language support for cancellation points: low-level parts

Common "long" methods:

- `Thread.sleep`
- File I/O
- User input
- Networking
- User-specified functions[8]

Some methods are marked as "cancellation points" or "cancelation points"[9]

[8] https://man7.org/linux/man-pages/man3/pthread_testcancel.3.html
[9] https://man7.org/linux/man-pages/man7/pthreads.7.html

Language support for cancellation points: low-level parts

Common "long" methods:

- `Thread.sleep`
- File I/O
- User input
- Networking
- User-specified functions[8]

Some methods are marked as "cancellation points" or "cancelation points"[9]
`Thread A` ready to handle cancel requests[10] and installs handlers[11].

[8] https://man7.org/linux/man-pages/man3/pthread_testcancel.3.html
[9] https://man7.org/linux/man-pages/man7/pthreads.7.html
[10] https://man7.org/linux/man-pages/man3/pthread_setcanceltype.3.html
[11] https://man7.org/linux/man-pages/man3/pthread_cleanup_push.3.html

Language support for cancellation points: low-level parts

Common "long" methods:

- `Thread.sleep`
- File I/O
- User input
- Networking
- User-specified functions[8]

Some methods are marked as "cancellation points" or "cancelation points"[9]
`Thread A` ready to handle cancel requests[10] and installs handlers[11].
`Thread B` requests cancelation[12] which will be eventually delivered for `Thread A`.

[8] https://man7.org/linux/man-pages/man3/pthread_testcancel.3.html
[9] https://man7.org/linux/man-pages/man7/pthreads.7.html
[10] https://man7.org/linux/man-pages/man3/pthread_setcanceltype.3.html
[11] https://man7.org/linux/man-pages/man3/pthread_cleanup_push.3.html
[12] https://man7.org/linux/man-pages/man3/pthread_cancel.3.html

Language support for cancellation points

Advantages:

- similar to `CancellationToken`
- responsiveness due to standard library support
  - proper API design
  - using low-level APIs (OS/pthread)
- fine-grained control of cancellation handler invocation
  - controlled frequency of polling points
  - easy to organize "uninterruptible sections"

Disadvantages:

- similar to `CancellationToken`
- still a lot of manual annotation and requires external library support

Important: low-level cancellation API works on `Thread` level.

Question time

Question: Enter into `synchronized` section should be interruptible or not interruptible? What about `Object.wait`?

Language support for cancellation points: managed language

Question time

Question: What is a key difference between managed language (e.g. Java) and unmanaged language (e.g. C++)?

Predefined cancellation points: managed language

Idea: in managed language we already have a Garbage Collection support, so any thread could be stopped if needed[13]

Let's stop thread at arbitrary execution point and `throw new CancellationException`!

[13] https://shipilev.net/jvm/anatomy-quarks/22-safepoint-polls/

Predefined cancellation points: managed language

Idea: in managed language we already have a Garbage Collection support, so any thread could be stopped if needed[13]

Let's stop thread at arbitrary execution point and `throw new CancellationException`!

Oh, this is forced cancellation ... never mind

[13] https://shipilev.net/jvm/anatomy-quarks/22-safepoint-polls/

Language support for cancellation points: Java

Common "long" methods:

- `Thread.sleep`, File I/O, User input, Networking
- User-specified "interruption state checks"[14]

[14] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#isInterrupted()

Language support for cancellation points: Java

Common "long" methods:

- `Thread.sleep`, File I/O, User input, Networking
- User-specified "interruption state checks"[14]
- Some synchronization (`Lock.lock`, `Condition.await`, `Object.wait`)

[14] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#isInterrupted()

Language support for cancellation points: Java

Common "long" methods:

- `Thread.sleep`, File I/O, User input, Networking
- User-specified "interruption state checks"[14]
- Some synchronization (`Lock.lock`, `Condition.await`, `Object.wait`)

Ability to `interrupt` executing thread[15]

[14] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#isInterrupted()
[15] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#interrupt()

Language support for cancellation points: Java

Common "long" methods:

- `Thread.sleep`, File I/O, User input, Networking
- User-specified "interruption state checks"[14]
- Some synchronization (`Lock.lock`, `Condition.await`, `Object.wait`)

Ability to `interrupt` executing thread[15]

The most unfortunate naming for method: `static boolean Thread.interrupted`[16]

- checks that **current** thread
- was interrupted and
- clears interrupted status

[14] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#isInterrupted()

[15] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#interrupt()

[16] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#interrupted()

Language support for cancellation points: Java

Common "long" methods:

- `Thread.sleep`, File I/O, User input, Networking
- User-specified "interruption state checks"[14]
- Some synchronization (`Lock.lock`, `Condition.await`, `Object.wait`)

Ability to `interrupt` executing thread[15]

The most unfortunate naming for method: `static boolean Thread.interrupted`[16]

- checks that **current** thread
- was interrupted and
- clears interrupted status

Remember, `Future` could return one of the following results:

- `Success<T>`, `Failure<Exception>`, `Interruption<Exception>`

[14] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#isInterrupted()

[15] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#interrupt()

[16] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html#interrupted()

Question time

Question: Ok, Java has great standard library with many explicit interrupted checks. But when I call `Thread.sleep(1_000)` thread goes to kernel and loses scheduling quantum. Why `Thread.interrupt` wakes up such thread? What kind of black magic is used?

Interruptible vs. uninterruptible methods

Homework, mail

Task 4.1 Write a program that empirically checks if the following methods are interruptible:

- `Thread.sleep`
- `ReentrantLock.lock`
- *monitor enter in* `synchronized`
- `Condition.await`
- `Object.wait`
- `System.in.read`
- `FileWriter.write`

Cancellation of tasks vs. thread interruption

Short summary:

- User-defined `CancellationToken`
  - works for threads and for tasks
  - has limited responsiveness
- Language-designed thread cancellation points
  - work only for threads
  - allow to quickly respond from almost any code

Cancellation of tasks vs. thread interruption

Short summary:

- User-defined `CancellationToken`
  - works for threads and for tasks
  - has limited responsiveness
- Language-designed thread cancellation points
  - work only for threads
  - allow to quickly respond from almost any code

Let's implement task cancellation on top of thread cancellation!

Cancellation of tasks vs. thread interruption

Short summary:

- User-defined `CancellationToken`
  - works for threads and for tasks
  - has limited responsiveness
- Language-designed thread cancellation points
  - work only for threads
  - allow to quickly respond from almost any code

Let's implement task cancellation on top of thread cancellation!

- Running task should know current "carrier" thread (`Executor` internals)
- `InterruptedException` should be properly cleared from worker thread to not affect subsequent tasks
- No race conditions on interruptions allowed (`spurious interrupts` are forbidden)
- Handle interruption status for not-yet-started and already-finished tasks
- Be ready for successfully finished interrupted tasks

Cancellation of tasks vs. thread interruption

Short summary:

- User-defined `CancellationToken`
  - works for threads and for tasks
  - has limited responsiveness
- Language-designed thread cancellation points
  - work only for threads
  - allow to quickly respond from almost any code

Let's implement task cancellation on top of thread cancellation!

- Running task should know current "carrier" thread (`Executor` internals)
- `InterruptedException` should be properly cleared from worker thread to not affect subsequent tasks
- No race conditions on interruptions allowed (`spurious interrupts` are forbidden)
- Handle interruption status for not-yet-started and already-finished tasks
- Be ready for successfully finished interrupted tasks

Remember: making your own `Executor`/`ThreadPool` is not easy

Language support for cancellation points: other managed languages

Different languages use different policies for cancellation in standard library.

[17] https://go.dev/blog/context

[18] https://kotlinlang.org/docs/cancellation-and-timeouts.html#asynchronous-timeout-and-resources

Language support for cancellation points: other managed languages

Different languages use different policies for cancellation in standard library.

The most elaborate and widely adopted designs belong to languages with lightweight threading:

- Go [17], managed language without exceptions
- Kotlin [18], managed JVM-based language with compiler-driven CPS transformation (`suspend` methods)

[17] https://go.dev/blog/context

[18] https://kotlinlang.org/docs/cancellation-and-timeouts.html#asynchronous-timeout-and-resources

Language support for cancellation points: other managed languages

Different languages use different policies for cancellation in standard library.

The most elaborate and widely adopted designs belong to languages with lightweight threading:

- Go [17], managed language without exceptions
- Kotlin [18], managed JVM-based language with compiler-driven CPS transformation (`suspend` methods)

We will cover goroutines/koroutines/coroutines in Lecture 12.

[17] https://go.dev/blog/context

[18] https://kotlinlang.org/docs/cancellation-and-timeouts.html#asynchronous-timeout-and-resources

Language support for cancellation points: other managed languages

Different languages use different policies for cancellation in standard library.

The most elaborate and widely adopted designs belong to languages with lightweight threading:

- Go [17], managed language without exceptions
- Kotlin [18], managed JVM-based language with compiler-driven CPS transformation (`suspend` methods)

We will cover goroutines/koroutines/coroutines in Lecture 12.

Basics you should know:

- Forced cancellation leads to corruption and should not be used
- Cooperative cancellation:
  - could be totally user-designed (`CancellationToken`) with responsiveness trade-off
  - "embedded" into standard library (`Thread` interruption status) with complexity trade-off
  - requires additional effort to guarantee responsiveness and consistency
  - widely adopted but slightly different in modern production languages

[17] https://go.dev/blog/context

[18] https://kotlinlang.org/docs/cancellation-and-timeouts.html#asynchronous-timeout-and-resources

Question time

Question: It looks like managed and unmanaged languages could implement cooperative cancellation in the same manner: provide thread-specific `CancellationToken` with `.cancel` and `.isCancelled`. Could Garbage Collection simplify something here?

Lecture plan

Propagating cancellation

`Thread A` started `Thread B` and `Thread C`.

- `Thread A` was interrupted, should `B` and `C` be interrupted too? (parent -> child)
- `Thread C` was interrupted, should `A` be interrupted too? (child -> parent)
- `Thread C` was interrupted, should `B` be interrupted too? (sibling -> sibling)

Propagating cancellation

`Thread A` started `Thread B` and `Thread C`.

- `Thread A` was interrupted, should `B` and `C` be interrupted too? (parent -> child)
- `Thread C` was interrupted, should `A` be interrupted too? (child -> parent)
- `Thread C` was interrupted, should `B` be interrupted too? (sibling -> sibling)

It is up to you!
Use structured concurrency techniques and life cycle management (e.g. `ExecutorService.shutdown`).

Propagating cancellation

`Thread A` started `Thread B` and `Thread C`.

- `Thread A` was interrupted, should `B` and `C` be interrupted too? (parent -> child)
- `Thread C` was interrupted, should `A` be interrupted too? (child -> parent)
- `Thread C` was interrupted, should `B` be interrupted too? (sibling -> sibling)

It is up to you!
Use structured concurrency techniques and life cycle management (e.g. `ExecutorService.shutdown`).
Kotlin/Go contexts are really well-designed!

Ignoring cancellation

- Intended policy of task designer
- Task designer was not aware about interruption scenarios

Ignoring cancellation

- Intended policy of task designer
- Task designer was not aware about interruption scenarios

In general, it is good to write `interruption-aware` code to make it reusable and modular.

Ignoring cancellation

- Intended policy of task designer
- Task designer was not aware about interruption scenarios

In general, it is good to write `interruption-aware` code to make it reusable and modular. However, correct programs may ignore cancellation as soon as it affects performance only, not safety nor progress.

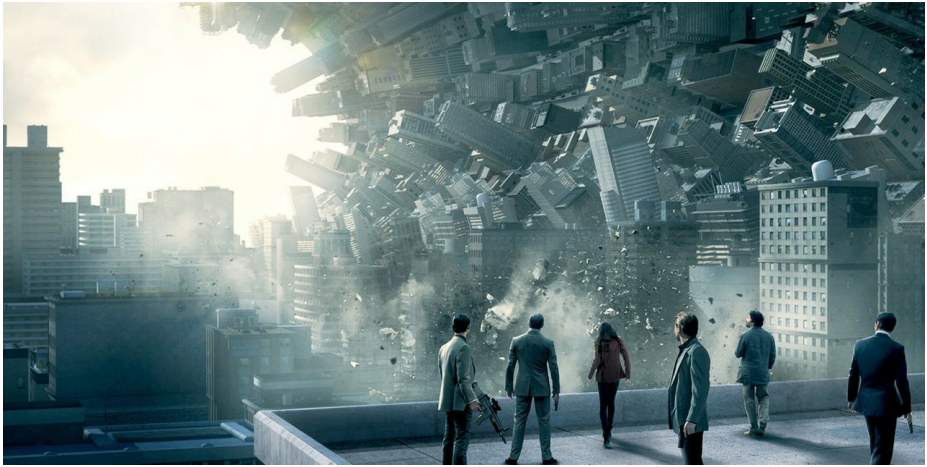
Cancelling cancellation

Cancelling cancellation

Do not play mind games!

Cancelling cancellation

Do not play mind games!



Notification and Interruption ordering

```
static Object monitor = new Object();
worker1()     { synchronized(monitor) { monitor.wait();   } }
worker2()     { synchronized(monitor) { monitor.wait();   } }
notifier()    { synchronized(monitor) { monitor.notify(); } }
interrupter() { getWorker1().interrupt(); }
```

Notification and Interruption ordering

```
static Object monitor = new Object();
worker1()     { synchronized(monitor) { monitor.wait();   } }
worker2()     { synchronized(monitor) { monitor.wait();   } }
notifier()    { synchronized(monitor) { monitor.notify(); } }
interrupter() { getWorker1().interrupt(); }
```

Language-level guarantees

- Delivered notify or delivered interrupt, no "overriding"
- Delivered interrupt could change `Thread` interruption status only (no immediate `InterruptedException`)

Notification and Interruption ordering

```
static Object monitor = new Object();
worker1()     { synchronized(monitor) { monitor.wait();   } }
worker2()     { synchronized(monitor) { monitor.wait();   } }
notifier()    { synchronized(monitor) { monitor.notify(); } }
interrupter() { getWorker1().interrupt(); }
```

Language-level guarantees

- Delivered notify or delivered interrupt, no "overriding"
- Delivered interrupt could change `Thread` interruption status only (no immediate `InterruptedException`)

It starts to be really hard!

Your tasks should be always aware of interruption requests. It helps to ensure your code is correct, responsiveness is not main goal here.

Lecture plan

Timeouts

Many blocking methods have optional `timeout` parameter:

- `Thread.join`
- `Lock.tryLock`
- `Condition.await`
- `Object.wait`

Timeouts

Many blocking methods have optional `timeout` parameter:

- `Thread.join`
- `Lock.tryLock`
- `Condition.await`
- `Object.wait`

Timeouts could "race" with timings, notifications and interruptions.

Timeouts

Many blocking methods have optional `timeout` parameter:

- `Thread.join`
- `Lock.tryLock`
- `Condition.await`
- `Object.wait`

Timeouts could "race" with timings, notifications and interruptions.
Unlike notifications (`I am done!`/`I failed!`) or interruptions (`I stopped!`), timeouts do not mean that executor stopped to consume resources (CPU, RAM, file handles...).

Timeouts

Many blocking methods have optional `timeout` parameter:

- `Thread.join`
- `Lock.tryLock`
- `Condition.await`
- `Object.wait`

Timeouts could "race" with timings, notifications and interruptions.
Unlike notifications (`I am done!`/`I failed!`) or interruptions (`I stopped!`), timeouts do not mean that executor stopped to consume resources (CPU, RAM, file handles...).
Remember, `Future.get(timeout)` could return one of the following results:

- `Success<T>`
- `Failure<Exception>`
- `Interruption<Exception>`
- `Timeout<Exception>`

Cancellation/interruption

Conclusion

- Requires cooperation for correctness and consistency
- May be implemented by user, would lack responsiveness in some scenarios
- May be supported on language level, standard library level or framework level
- Tricky because of many corner cases
  - Normal completion
  - Erroneous completion
  - Cancellation
  - Notification
  - Timeout
  - Concurrent repeated cancellation

Extremely useful design that helps to save resources and provide fine-grained control of multi-component systems.

Lecture plan

1. Cancellation
   - Problems with forced cancellation
   - Cooperative cancellation
   - Design space
   - Timeouts
2. Concurrent problem decomposition techniques
   - State machine
   - Function shipping
   - Designated thread
   - Partitioning threads
   - Partitioning data
   - Privatization
   - Replication
   - Ownership
3. Summary

Concurrency pitfalls you know so far

How to check your concurrent system against problems?

Concurrency pitfalls you know so far

How to check your concurrent system against problems?

- Imagine that almost any line of code may be executed
  - with arbitrary speed
  - by arbitrary utility thread
- Imagine that almost any blocking method may fail spuriously
  - timeout
  - interruption
  - notification

Decomposition techniques

Tools to simplify the design:

- Encapsulation (e.g. `interface Lock`)
- Weakening (e.g. allow reentrancy for locks)
- Decoupling (e.g. threads and tasks)
- State machine (e.g. `Executor is ready/shutdown/terminated`)
- Concurrent programming patterns
  - Producer-consumer
  - Publisher-subscriber
  - Work arbitrage, work dealing, work stealing
- Partitioning:
  - Locking responsibility (data locking, code locking, lock splitting)
  - Access rights (`ReadWriteLock`, `Semaphore`)

Decomposition techniques

Tools to simplify the design:

- Encapsulation (e.g. `interface Lock`)
- Weakening (e.g. allow reentrancy for locks)
- Decoupling (e.g. threads and tasks)
- State machine (e.g. `Executor is ready/shutdown/terminated`)
- Concurrent programming patterns
  - Producer-consumer
  - Publisher-subscriber
  - Work arbitrage, work dealing, work stealing
- Partitioning:
  - Locking responsibility (data locking, code locking, lock splitting)
  - Access rights (`ReadWriteLock`, `Semaphore`)
  - Data usage (replication, privatization, ownership)
- Batching (e.g. `N` operations per critical section)

Decomposition techniques

Tools to simplify the design:

- Encapsulation (e.g. `interface Lock`)
- Weakening (e.g. allow reentrancy for locks)
- Decoupling (e.g. threads and tasks)
- State machine (e.g. `Executor is ready/shutdown/terminated`)
- Concurrent programming patterns
  - Producer-consumer
  - Publisher-subscriber
  - Work arbitrage, work dealing, work stealing
- Partitioning:
  - Locking responsibility (data locking, code locking, lock splitting)
  - Access rights (`ReadWriteLock`, `Semaphore`)
  - Data usage (replication, privatization, ownership)
- Batching (e.g. `N` operations per critical section)

And many more ideas.

Lecture plan

Concurrent state machines

```
enum ExecutorState { ACCEPTING_TASKS, SHUTDOWN, TERMINATED }
class Executor {  private ExecutorState state;
  public synchronized Future<T> submit(Cancellable<T> task) {
    if (state != ACCEPTING_TASKS) throwException("Submit after shutdown");
    ...
  }
  public synchronized void shutdown() {
    if (state != ACCEPTING_TASKS) throwException("Double shutdown");
    state = SHUTDOWN;
  }
  public synchronized boolean isTerminated() { return state == TERMINATED; }
  private synchronized void onFinishedTask() {
    if (pendingTasks.isEmpty() && state == SHUTDOWN) state = TERMINATED;
    ...
  }
```

Concurrent state machines

- At any moment, state of concurrent object is explicitly defined
- All transitions happen atomically (e.g. under `Lock`)
- Every state defines **allowed** and **forbidden** operations

Concurrent state machines

- At any moment, state of concurrent object is explicitly defined
- All transitions happen atomically (e.g. under `Lock`)
- Every state defines **allowed** and **forbidden** operations

Advantages:

- Easy to use
- Simple to implement
- Trivial to debug using logging

Concurrent state machines

- At any moment, state of concurrent object is explicitly defined
- All transitions happen atomically (e.g. under `Lock`)
- Every state defines **allowed** and **forbidden** operations

Advantages:

- Easy to use
- Simple to implement
- Trivial to debug using logging

Disadvantages:

- *Could* limit parallelism
- Some concurrent objects have infinite amount of states

Concurrent state machines

- At any moment, state of concurrent object is explicitly defined
- All transitions happen atomically (e.g. under `Lock`)
- Every state defines **allowed** and **forbidden** operations

Advantages:

- Easy to use
- Simple to implement
- Trivial to debug using logging

Disadvantages:

- *Could* limit parallelism
- Some concurrent objects have infinite amount of states

Tricks:

- use equivalence classes as "states" (concurrent queue is `EMPTY`, `NON_EMPTY`, `FULL`)
- create a separate "control structure" for accessing your inner data (`Executor.submit` vs. `Executor.taskQueue.add`)

Lecture plan

Function shipping

Imagine that you have millions of Terabytes of numerical data distributed over millions of computers all over the world with total power exceeding 10 petaFLOPS

- You are looking for aliens[19] or trying to cure cancer[20]

[19] https://en.wikipedia.org/wiki/SETI@home
[20] https://en.wikipedia.org/wiki/Folding@home

Function shipping

Imagine that you have millions of Terabytes of numerical data distributed over millions of computers all over the world with total power exceeding 10 petaFLOPS

- You are looking for aliens[19] or trying to cure cancer[20]

How to apply complicated computations over distributed data samples?

[19] https://en.wikipedia.org/wiki/SETI@home
[20] https://en.wikipedia.org/wiki/Folding@home

Function shipping

Imagine that you have millions of Terabytes of numerical data distributed over millions of computers all over the world with total power exceeding 10 petaFLOPS

- You are looking for aliens[19] or trying to cure cancer[20]

How to apply complicated computations over distributed data samples?
Transfer your algorithm to the data servers!

[19] https://en.wikipedia.org/wiki/SETI@home
[20] https://en.wikipedia.org/wiki/Folding@home

Function shipping

Imagine that you have millions of Terabytes of numerical data distributed over millions of computers all over the world with total power exceeding 10 petaFLOPS

- You are looking for aliens[19] or trying to cure cancer[20]

How to apply complicated computations over distributed data samples?
Transfer your algorithm to the data servers!
Usual design: data arrive to function (e.g. as a parameter)

[19] https://en.wikipedia.org/wiki/SETI@home
[20] https://en.wikipedia.org/wiki/Folding@home

Function shipping

Imagine that you have millions of Terabytes of numerical data distributed over millions of computers all over the world with total power exceeding 10 petaFLOPS

- You are looking for aliens[19] or trying to cure cancer[20]

How to apply complicated computations over distributed data samples?
Transfer your algorithm to the data servers!
Usual design: data arrive to function (e.g. as a parameter)
Function shipping design: function arrive to data (e.g. as a parameter)

[19] https://en.wikipedia.org/wiki/SETI@home
[20] https://en.wikipedia.org/wiki/Folding@home

Function shipping

Imagine that you have millions of Terabytes of numerical data distributed over millions of computers all over the world with total power exceeding 10 petaFLOPS

- You are looking for aliens[19] or trying to cure cancer[20]

How to apply complicated computations over distributed data samples?
Transfer your algorithm to the data servers!
Usual design: data arrive to function (e.g. as a parameter)
Function shipping design: function arrive to data (e.g. as a parameter)

- MapReduce framework[21]
- Resilient Distributed Dataset framework[22]

[19] https://en.wikipedia.org/wiki/SETI@home
[20] https://en.wikipedia.org/wiki/Folding@home
[21] https://hadoop.apache.org
[22] https://spark.apache.org/

Function shipping

Imagine that you have millions of Terabytes of numerical data distributed over millions of computers all over the world with total power exceeding 10 petaFLOPS

- You are looking for aliens[19] or trying to cure cancer[20]

How to apply complicated computations over distributed data samples?
Transfer your algorithm to the data servers!
Usual design: data arrive to function (e.g. as a parameter)
Function shipping design: function arrive to data (e.g. as a parameter)

- MapReduce framework[21]
- Resilient Distributed Dataset framework[22]
- OS signal handlers could be used to perform function shipping to particular threads

[19] https://en.wikipedia.org/wiki/SETI@home
[20] https://en.wikipedia.org/wiki/Folding@home
[21] https://hadoop.apache.org
[22] https://spark.apache.org/

Lecture plan

Designated thread

`N` philosophers have serious problems with eating together and not starving to death.

Designated thread

`N` philosophers have serious problems with eating together and not starving to death.
They ask external authority (a.k.a. `waiter`) to solve all the serving problems.

Designated thread

`N` philosophers have serious problems with eating together and not starving to death. They ask external authority (a.k.a. `waiter`) to solve all the serving problems.

- Requests for operations (`I need forkA and forkB`) passed to designated thread
- Operations (`grab forks, serve dish`) done by single designated thread
- Confirmation response (`here are your forks, dish and pasta`) returned asynchronously

Designated thread

`N` philosophers have serious problems with eating together and not starving to death.
They ask external authority (a.k.a. `waiter`) to solve all the serving problems.

- Requests for operations (`I need forkA and forkB`) passed to designated thread
- Operations (`grab forks, serve dish`) done by single designated thread
- Confirmation response (`here are your forks, dish and pasta`) returned asynchronously

Optional homework: prepare list of advantages and disadvantages of this design before the exam.

Lecture plan

Efficient counter: case study

Thread-safe counter

```
public class Counter {
    private long cnt;
    public Counter(long initial) { ... }
    public void increment() { ... }
    public long get() { ... }
}
```

- Concurrent `increment`
- Concurrent `get`
- "Consistency" of reads and updates

Locked counter

```
public class LockedCounter {
    private long cnt;
    public LockedCounter(long initial) { this.cnt = initial; }
    public synchronized void increment() { cnt++; }
    public synchronized long get() { return cnt; }
}
```

Advantages

- Easy to implement and debug

Disadvantages

- Bad scalability

Question time

Question: `LockedCounter` uses `synchronized` modifier on public methods. Please, remind, why could it be a bad idea?

Partitioning: threads

```
private long cnt;
private final ReadWriteLock rw = ...;
public Counter(long initial) { this.cnt = initial; }
public void increment() {
  rw.writeLock().lock(); try { cnt++; } finally { rw.writeLock().unlock(); }
}
public long get() {
  rw.readLock().lock(); try { return cnt; } finally { rw.readLock().unlock(); }
}
```

Partitioning: threads

```
private long cnt;
private final ReadWriteLock rw = ...;
public Counter(long initial) { this.cnt = initial; }
public void increment() {
  rw.writeLock().lock(); try { cnt++; } finally { rw.writeLock().unlock(); }
}
public long get() {
  rw.readLock().lock(); try { return cnt; } finally { rw.readLock().unlock(); }
}
```

- Easy to implement
- Great scalability for read-mostly scenarios
- Bad scalability for write-intensive scenarios

Lecture plan

Partitioning data

```
private final LockedCounter c1, c2;
public Counter(long initial) { c1 = new LockedCounter(initial);
                               c2 = new LockedCounter(0); }
public void increment() {
  var c = (Thread.currentThread().hashCode() % 2 == 0) ? c1 : c2;
  c.increment();
}
public long get() {
  synchronized(c1) { synchronized(c2) { return c1.get() + c2.get(); }}
}
```

Partitioning data

```
private final LockedCounter c1, c2;
public Counter(long initial) { c1 = new LockedCounter(initial);
                               c2 = new LockedCounter(0); }
public void increment() {
  var c = (Thread.currentThread().hashCode() % 2 == 0) ? c1 : c2;
  c.increment();
}
public long get() {
  synchronized(c1) { synchronized(c2) { return c1.get() + c2.get(); }}
}
```

- Better scalability for write-mostly scenarios
- Bad scalability for read-intensive scenarios
- Unknown optimal level of partitioning (`c1, c2, c3 ... cN`)

Lecture plan

Privatization

```
public void increment() {
  if (currentThreadOwnsCounter()) { cnt++; }
  else { incrementSlowPath(); }
}
public long get() {
  if (currentThreadOwnsCounter()) { return cnt; }
  else { return getSlowPath(); }
}
```

- Close to optimal for uncontended scenario
- How to pass "ownership" from one thread to another?

Privatization + batching

```
private long ops;
private void checkOwnershipTransfer() {
  if (curThreadIsOwner() && ((++ops) > 1_000)) { ops=0; tryPassOwnership(); }
}
public void increment() {
  checkOwnershipTransfer();
  if (curThreadIsOwner()) { cnt++; } else { incrementSlowPath(); }
}
public long get() {
  checkOwnershipTransfer();
  if (curThreadIsOwner()) { return cnt; } else { return getSlowPath(); }
}
```

- Close to optimal for uncontended scenarios with large batch size
- Non-owners could wait for arbitrary long time

Lecture plan

Replication + batching

ThreadLocal[23]

```
static final ThreadLocal<Long> privateCounter = new ThreadLocal<Long>();
static final LockedCounter globalCounter = new LockedCounter();
public long get() {  return privateCounter.get(); }
public void increment() {
  privateCounter.set(privateCounter.get() + 1);
  ops++;
  if (ops > 1_000) {
    globalCounter.add(ops);
    privateCounter.set(globalCounter.get());
    ops = 0;
  }
}
```

[23] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/ThreadLocal.html

Replication + batching

```
public long get() {  return privateCounter.get(); }
public void increment() {
  privateCounter.set(privateCounter.get() + 1); ops++;
  if (ops > 1_000) {
    globalCounter.add(ops);
    privateCounter.set(globalCounter.get());
    ops = 0;
  }
}
```

- Most operations do not require synchronization
- In-thread `get`/`increment` are consistent
- Cross-thread operations are partially inconsistent
  - global state "lags" but monotonically grows
  - thread-private states "lag" but monotonically grow

Lecture plan

Ownership

- Exclusive data ownership
  - only owner may access data for read/write
- Read-only data ownership
  - several owners, allowed only to read data
- Exclusive code ownership
  - only owner may execute code fragment
- Temporary ownership
  - owner may change, but it is always an atomic transaction
- Thread-local/thread-private data
  - always owned by the same thread

Question time

Question: Provide an example of thread-private data that exists in any programming language

Partitioning types and flavours

You could find partitioning almost in any pattern:

- Thread access (code locking)
- Lock instances (lock splitting)
- Place of code execution (function shipping)
- Responsibility of code execution (designated thread)
- Spatially partition data ownership (data locking)
- Temporal partitioning of data access (privatization, mutex-guarded ownership)
- Duplicate data (replication)
- Number of operations performed by thread locally (batching)

Partitioning types and flavours

You could find partitioning almost in any pattern:

- Thread access (code locking)
- Lock instances (lock splitting)
- Place of code execution (function shipping)
- Responsibility of code execution (designated thread)
- Spatially partition data ownership (data locking)
- Temporal partitioning of data access (privatization, mutex-guarded ownership)
- Duplicate data (replication)
- Number of operations performed by thread locally (batching)

It is useful to use ownership (thread confinement) variations:

- Temporal thread confinement (critical section)
- Automatic thread confinement (local variables)
- User-intended thread confinement (`ThreadLocal`)
- Group-level thread confinement (`Semaphore`, `ReadWriteLock`)

Summary

- Cancellation of concurrently running tasks is useful to save computation resources
- Proper cancellation requires cooperation
- There exists language-independent cancellation designs and low-level language-specific ones
- Task cancellation could be built on top of thread cancellation
- Timeouts could be treated as a variation of "self-interruption"
- Developing safe and efficient concurrent systems is hard, consider using decomposition techniques
  - State machines
  - Partitioning
  - Ownership
  - Batching
  - Weakening

Summary: homework

Homework, mail

Task 4.1 Write a program that empirically checks if the following methods are interruptible:

- `Thread.sleep`
- `ReentrantLock.lock`
- `monitor enter` in `synchronized`
- `Condition.await`
- `Object.wait`
- `System.in.read`
- `FileWriter.write`