Lecture 5: additional topics of practical concurrency

documenting protocols and classes, checking concurrent invariants, stress testing, execution trace analysis, estimating required testing effort, static and dynamic checks, scheduling randomization, model checking

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l5.pdf

In previous episodes

Concurrent coordination concepts:

- Mutual exclusion, Signalling, Group-level concurrency, Separation of threads and tasks

Concurrent data structures:

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

Key properties of concurrent algorithm:

- Safety, Liveness, Performance

Common problems:

- Safety: race condition, data race, deadlock, lost signal, predicate invalidation
- Liveness: livelock, priority inversion, fairness
- Performance: lock convoy, thundering herd, oversubscription

Widely adopted concurrency designs that could complicate development:

- Asynchronous exceptions, Cancellation/interruption, Timeouts

Decomposition ideas:

- State machines, Partitioning, Ownership, Batching, Weakening

Pain points

- Unpredicatable speed of execution
- Arbitrary "whole program control flow"
  - `Thread.start` looks like `goto`
- Arbitrary "thread-specific control flow"
  - timeout, notification, interruption, spurious wakeup
- Limited composability of modules
  - locking policy is not clear
  - lock ordering hidden by virtual methods
  - exposed `synchronized`
  - non-documented thread-safety
- Hard to diagnose performance problems
  - livelock, priority inversion, thundering herd, scalability
- It is **too easy** to introduce a race condition
  - it is **too hard** to find it on review, during testing, debug on production system

Question time

Question: Concurrency is a very complicated programming domain. What could we do to simplify our programming life?

Lecture plan

1. Public API
   - Documentation
   - Concurrent invariants
2. Testing
   - Unit testing and stress testing
   - Execution trace analysis
   - Test design to increase bug probability
3. Tooling
   - Static checks
   - Dynamic checks
   - Monitoring
   - Chaos mode execution
4. Summary: design of reliable concurrent software
5. Formal methods: model checking

Lecture plan

1. Public API
   - Documentation
   - Concurrent invariants
2. Testing
   - Unit testing and stress testing
   - Execution trace analysis
   - Test design to increase bug probability
3. Tooling
   - Static checks
   - Dynamic checks
   - Monitoring
   - Chaos mode execution
4. Summary: design of reliable concurrent software
5. Formal methods: model checking

Documentation

Reading the docs:

- Thread-safety of a class
- How to use class to avoid inconsistent state (e.g. `try-finally` for `Lock`)
- Blocking operations
- Locking policy
- Admission policy
- Inheritance policy

Documentation

Reading the docs:

- Thread-safety of a class
- How to use class to avoid inconsistent state (e.g. `try-finally` for `Lock`)
- Blocking operations
- Locking policy
- Admission policy
- Inheritance policy

Writing the docs:

Documentation

Reading and **writing** the docs:

- Thread-safety of a class
- How to use class to avoid inconsistent state (e.g. `try-finally` for `Lock`)
- Blocking operations
- Locking policy
- Admission policy
- Inheritance policy

Documentation

Reading and **writing** the docs:

- Thread-safety of a class
- How to use class to avoid inconsistent state (e.g. `try-finally` for `Lock`)
- Blocking operations
- Locking policy
- Admission policy
- Inheritance policy

`java.util.concurrent` is a good source for inspiration

Question time

Question: Nobody reads the documentation! How could I enforce usage patterns of my precious class?

Invariants checking

Single-threaded code crashes with `ConcurrentModificationException`[1]:

```
void foo(List<X> list) {
  for (X x : list) {
    if (!x.isValid()) list.remove(x);
  }
}
```

[1] https://stackoverflow.com/questions/3184883/concurrentmodificationexception-for-arraylist

Invariants checking

Single-threaded code crashes with `ConcurrentModificationException`[1]:

```
void foo(List<X> list) {
  for (X x : list) {
    if (!x.isValid()) list.remove(x);
  }
}
```

It is intended behaviour[2]

> *The iterators returned by this class's 'iterator' ... are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own 'remove' or 'add' methods, the iterator will throw a 'ConcurrentModificationException'. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.*

[1] https://stackoverflow.com/questions/3184883/concurrentmodificationexception-for-arraylist

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/ArrayList.html

Question time

Question: How to implement such consistency check (collection is not modified while iterator is used)?

Invariants checking

Advantages:

- Contract is enforced by implementation rather by javadoc
- Fail-fast behaviour speed-ups debugging
- Type of exception and stacktrace information help to debug large systems

Disadvantages:

- Performance overheads
- Incompleteness (impossible to cover all misuses)
- Affects even single-threaded programs

Question time

Question: How could we implement lightweight or zero-cost checks?

Assertions

Assertions[3]:

```
void foo(ArrayList<String> list) {
  assert list != null;
  assert list.size() > 0;
}
```

[3] https://docs.oracle.com/javase/specs/jls/se11/html/jls-14.html#jls-14.10

Assertions

Assertions[3]:

```
void foo(ArrayList<String> list) {
  assert list != null;
  assert list.size() > 0;
}
```

Typically, assertion checking is enabled during program development and testing, and disabled for deployment, to improve performance.

[3] https://docs.oracle.com/javase/specs/jls/se11/html/jls-14.html#jls-14.10

Assertions

Assertions[3]:

```
void foo(ArrayList<String> list) {
  assert list != null;
  assert list.size() > 0;
}
```

> *Typically, assertion checking is enabled during program development and testing, and disabled for deployment, to improve performance.*
>
> *Because assertions may be disabled, programs must not assume that the expressions contained in assertions will be evaluated. Thus, these boolean expressions should generally be free of side effects.*

[3] https://docs.oracle.com/javase/specs/jls/se11/html/jls-14.html#jls-14.10

Assertions

Assertions[3]:

```
void foo(ArrayList<String> list) {
  assert list != null;
  assert list.size() > 0;
}
```

> *Typically, assertion checking is enabled during program development and testing, and disabled for deployment, to improve performance.*
>
> *Because assertions may be disabled, programs must not assume that the expressions contained in assertions will be evaluated. Thus, these boolean expressions should generally be free of side effects.*
>
> *In light of this, assertions should not be used for argument checking in public methods. Argument checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled.*

[3] https://docs.oracle.com/javase/specs/jls/se11/html/jls-14.html#jls-14.10

Assertions

- Could be enabled/disabled without recompilation of Java program[4]
  - Many guidelines enforce `java -ea -esa -jar production.jar ...`
- Should be used for checking **internal consistency** (violation of programmer's intent), not **external consistency** (e.g validation of user input)
- Perfect choice to make concurrent programs fail-fast
  - Use `-enablesystemassertions` to fail-fast on subtle data races inside improperly used classes from standard library (e.g. unsynchronized access to `ArrayList`)

[4] https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

Optional heavyweight checks

```
class MyLinkedList {
  private int cachedLength;
  private Node head;
  private int length() {
    int size = 0;
    for (Node n = this.head; n != null; n = n.next, size++) {}
    return size;
  }
  public void add(Node e) {
    e.next = this.head;
    this.head = e;
    elementCount++;
    assert cachedLength == length();
  }
}
```

Optional heavyweight checks

```
class MyLinkedList {
  private int cachedLength;
  private Node head;
  private int length() {
    int size = 0;
    for (Node n = this.head; n != null; n = n.next, size++) {}
    return size;
  }
  public void add(Node e) {
    e.next = this.head;
    this.head = e;
    elementCount++;
    assert heavyChecksDisabled() || cachedLength == length();
  }
}
```

Optional heavyweight checks

- Could find non-local data inconsistency
- May be prohibitively slow for production workloads
- Distort concurrent execution patterns (change probability of race condition)
- May be designed to provide detailed information about failure

Summary: invariants

- Invariants help to make better contracts and design more reliable programs
- Report failed external contracts with detailed exceptions
- Check internal contracts with lightweight checks (e.g. optional `asserts`)
- Consider using heavyweight checks for special modes of execution (e.g. debug builds)

Summary: invariants

- Invariants help to make better contracts and design more reliable programs
- Report failed external contracts with detailed exceptions
- Check internal contracts with lightweight checks (e.g. optional `asserts`)
- Consider using heavyweight checks for special modes of execution (e.g. debug builds)

You can not start checking for correctness until you understand

- allowed
- forbidden

result of your algorithm or data structure.

Summary: concurrent invariants

Some validity rules are obvious and trivial:

- Does not fail with `AssertionError`, `NullPointerExcepiton` ...
- Deadlock never happens, data race never happens ...
- ...

Summary: concurrent invariants

Some validity rules are obvious and trivial:

- Does not fail with `AssertionError`, `NullPointerExcepiton` ...
- Deadlock never happens, data race never happens ...
- ...

Describing behaviour in multithreaded environment requires additional effort:

- After data is added to thread-safe collection, some other thread eventually could `remove` it

Summary: concurrent invariants

Some validity rules are obvious and trivial:

- Does not fail with `AssertionError`, `NullPointerExcepiton` ...
- Deadlock never happens, data race never happens ...
- ...

Describing behaviour in multithreaded environment requires additional effort:

- After data is added to thread-safe collection, some other thread eventually could `remove` it
- only once

Summary: concurrent invariants

Some validity rules are obvious and trivial:

- Does not fail with `AssertionError`, `NullPointerExcepiton` ...
- Deadlock never happens, data race never happens ...
- ...

Describing behaviour in multithreaded environment requires additional effort:

- After data is added to thread-safe collection, some other thread eventually could `remove` it
- only once

Concurrent consistency could be hard to explain and check[5]:

> *... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

[5] https://en.wikipedia.org/wiki/Sequential_consistency

Summary: concurrent invariants

Some validity rules are obvious and trivial:

- Does not fail with `AssertionError`, `NullPointerExcepiton` ...
- Deadlock never happens, data race never happens ...
- ...

Describing behaviour in multithreaded environment requires additional effort:

- After data is added to thread-safe collection, some other thread eventually could `remove` it
- only once

Concurrent consistency could be hard to explain and check[5]:

> *... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

Lecture 6 and Lecture 7 will heavily use complicated math to formalize *consistency*.

[5] https://en.wikipedia.org/wiki/Sequential_consistency

Lecture plan

1. Public API
   - Documentation
   - Concurrent invariants
2. Testing
   - Unit testing and stress testing
   - Execution trace analysis
   - Test design to increase bug probability
3. Tooling
   - Static checks
   - Dynamic checks
   - Monitoring
   - Chaos mode execution
4. Summary: design of reliable concurrent software
5. Formal methods: model checking

Unit testing

Focus on safety (correctness).

Unit testing

Focus on safety (correctness).

- All single-threaded scenarios
- Basic multi-threaded scenarios
- Advanced multi-threaded scenarios with custom scheduling (insertion of `Thread.sleep`)

Unit testing

Focus on safety (correctness).

- All single-threaded scenarios
- Basic multi-threaded scenarios
- Advanced multi-threaded scenarios with custom scheduling (insertion of `Thread.sleep`)

Goals:

- coverage
- regression tests
- supplementary documentation for intended and forbidden usages

Unit testing

Focus on safety (correctness).

- All single-threaded scenarios
- Basic multi-threaded scenarios
- Advanced multi-threaded scenarios with custom scheduling (insertion of `Thread.sleep`)

Goals:

- coverage
- regression tests
- supplementary documentation for intended and forbidden usages

Tools:

- JUnit[6]
- Java Concurrency Stress (jcstress)[7]

[6] https://junit.org
[7] https://github.com/openjdk/jcstress

Unit testing

Focus on safety (correctness).

- All single-threaded scenarios
- Basic multi-threaded scenarios
- Advanced multi-threaded scenarios with custom scheduling (insertion of `Thread.sleep`)

Goals:

- coverage
- regression tests
- supplementary documentation for intended and forbidden usages

Tools:

- JUnit[6]
- Java Concurrency Stress (jcstress)[7]

Caveats:

- there are few Quality Assurance (QA) engineers that are aware about concurrency pitfalls

[6] https://junit.org

[7] https://github.com/openjdk/jcstress

Homework: JCStress

Check out few solutions for Dining Philosophers Problem in JCStress repo[8]
Checkout repo, run example. Modify example with some incorrect solution, observe deadlock and see how it is reported by the framework.

Homework, mail

Task 5.1 Send two screenshots:

- *incorrect solution to dining philosophers problem written in JCStress DSL*
- *console output when you run this sample via JCStress harness*

[8] https://github.com/openjdk/jcstress/blob/master/jcstress-samples/src/main/java/org/openjdk/jcstress/samples/problems/classic/Classic_01_DiningPhilosophers.java

Stress testing

Focus on performance (resource utilization).

Stress testing

Focus on performance (resource utilization).

- Undersubscription because of critical sections
- Oversubscription because of misconfigured thread pools
- Inefficient contended execution path
- Progress problems (livelock, priority inversion, starvation)

Stress testing

Focus on performance (resource utilization).

- Undersubscription because of critical sections
- Oversubscription because of misconfigured thread pools
- Inefficient contended execution path
- Progress problems (livelock, priority inversion, starvation)

Key metrics:

- Throughput
- Latency distribution
- Resource leaks (objects, thread-local memory, thread handles, caches ...)
- Scalability

Stress testing

Focus on performance (resource utilization).
Key metrics:

- Throughput, Latency, Resource utilization, Scalability

Stress testing

Focus on performance (resource utilization).
Key metrics:

- Throughput, Latency, Resource utilization, Scalability

Tools:

- Java Microbenchmark Harness (JMH)[9]
- Lincheck[10]

[9] https://github.com/openjdk/jmh
[10] https://github.com/JetBrains/lincheck

Stress testing

Focus on performance (resource utilization).
Key metrics:

- Throughput, Latency, Resource utilization, Scalability

Tools:

- Java Microbenchmark Harness (JMH)[9]
- Lincheck[10]

Caveats:

- high-quality performance analysis requires years of low-level expertise
- "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"

[9] https://github.com/openjdk/jmh
[10] https://github.com/JetBrains/lincheck

Lecture plan

Execution trace analysis: idea

```
Timestamp: 1000400, Thread A: X.lock   (success)
Timestamp: 1000405, Thread B: Y.lock   (success)
Timestamp: 1000408, Thread A: X.lock   (success)
Timestamp: 1000411, Thread B: X.lock   (block)
Timestamp: 1000415, Thread A: Y.lock   (block)
```

Execution trace analysis: idea

```
Timestamp: 1000400, Thread A: X.lock   (success)
Timestamp: 1000405, Thread B: Y.lock   (success)
Timestamp: 1000408, Thread A: X.lock   (success)
Timestamp: 1000411, Thread B: X.lock   (block)
Timestamp: 1000415, Thread A: Y.lock   (block)
```

Thread A owns `X`, awaits for `Y`
Thread B owns `Y`, awaits for `X`

Examples of nontrivial consistency

```
Timestamp: 1, Thread A: list.add(User(1))
Timestamp: 2, Thread B: list.add(User(2))
Timestamp: 3, Thread C: list.removeAny(), result = User(2)
Timestamp: 3, Thread D: list.removeAny(), result = null
Timestamp: 5, Thread A: list.removeAny(), result = User(2)
```

Examples of nontrivial consistency

```
Timestamp: 1, Thread A: list.add(User(1))
Timestamp: 2, Thread B: list.add(User(2))
Timestamp: 3, Thread C: list.removeAny(), result = User(2)
Timestamp: 3, Thread D: list.removeAny(), result = null
Timestamp: 5, Thread A: list.removeAny(), result = User(2)
```

Where is `User(1)`?

Examples of nontrivial consistency

```
Timestamp: 1, Thread A: list.add(User(1))
Timestamp: 2, Thread B: list.add(User(2))
Timestamp: 3, Thread C: list.removeAny(), result = User(2)
Timestamp: 3, Thread D: list.removeAny(), result = null
Timestamp: 5, Thread A: list.removeAny(), result = User(2)
```

Where is `User(1)`?
Why `User(2)` was removed twice?

Execution trace analysis

- Use **monotonic** stamps[11]

[11] CLOCK_MONOTONIC https://linux.die.net/man/3/clock_gettime

Execution trace analysis

- Use **monotonic** stamps[11]
  - `System.currentTimeMillis()` is **not** monotonic, be ready to time travel[12]

[11] `CLOCK_MONOTONIC` https://linux.die.net/man/3/clock_gettime

[12] https://shipilev.net/blog/2014/nanotrusting-nanotime/

Execution trace analysis

- Use **monotonic** stamps[11]
  - `System.currentTimeMillis()` is **not** monotonic, be ready to time travel[12]
  - `static synchronized long stamp() { return ++counter; }` is safe approximation

[11] `CLOCK_MONOTONIC` https://linux.die.net/man/3/clock_gettime

[12] https://shipilev.net/blog/2014/nanotrusting-nanotime/

Execution trace analysis

- Use **monotonic** stamps[11]
  - `System.currentTimeMillis()` is **not** monotonic, be ready to time travel[12]
  - `static synchronized long stamp() { return ++counter; }` is safe approximation
- Define obvious consistency violations
  - Thread-safe collection cannot return `null` in moment `t2` if element was added at `t1 < t2`
  - Thread-safe collection cannot remove the same element twice if it was added only once
  - ...

[11] `CLOCK_MONOTONIC` https://linux.die.net/man/3/clock_gettime

[12] https://shipilev.net/blog/2014/nanotrusting-nanotime/

Execution trace analysis

- Use **monotonic** stamps[11]
  - `System.currentTimeMillis()` is **not** monotonic, be ready to time travel[12]
  - `static synchronized long stamp() { return ++counter; }` is safe approximation
- Define obvious consistency violations
  - Thread-safe collection cannot return `null` in moment `t2` if element was added at `t1 < t2`
  - Thread-safe collection cannot remove the same element twice if it was added only once
  - ...
- Collect execution trace (e.g. log file) from some scenario and check validity

[11] `CLOCK_MONOTONIC` https://linux.die.net/man/3/clock_gettime

[12] https://shipilev.net/blog/2014/nanotrusting-nanotime/

Execution trace analysis

- Use **monotonic** stamps[11]
  - `System.currentTimeMillis()` is **not** monotonic, be ready to time travel[12]
  - `static synchronized long stamp() { return ++counter; }` is safe approximation
- Define obvious consistency violations
  - Thread-safe collection cannot return `null` in moment `t2` if element was added at `t1 < t2`
  - Thread-safe collection cannot remove the same element twice if it was added only once
  - ...
- Collect execution trace (e.g. log file) from some scenario and check validity
- You could implement simple "reference solution" and check that it provides the same result on given execution trace[13]

[11] `CLOCK_MONOTONIC` https://linux.die.net/man/3/clock_gettime

[12] https://shipilev.net/blog/2014/nanotrusting-nanotime/

[13] https://www.amazon.science/publications/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3

Question time

Question: How could I check that requirements for my data structure are complete, consistent, strong enough/weak enough?

Question time

Question: How could I check that requirements for my data structure are complete, consistent, strong enough/weak enough?
Reconsider your answer after Lecture 7.

Lecture plan

1. Public API
   - Documentation
   - Concurrent invariants
2. Testing
   - Unit testing and stress testing
   - Execution trace analysis
   - Test design to increase bug probability
3. Tooling
   - Static checks
   - Dynamic checks
   - Monitoring
   - Chaos mode execution
4. Summary: design of reliable concurrent software
5. Formal methods: model checking

Common errors

Common errors

- Use delays (e.g. `Thread.sleep(100)`) to ensure that "other thread done something"
  - use synchronized state variables, `CountDownLatch` or monitors

Common errors

- Use delays (e.g. `Thread.sleep(100)`) to ensure that "other thread done something"
  - use synchronized state variables, `CountDownLatch` or monitors
- Use too strict assertions that trigger false-positive test failures
  - formalize invariants, drop irrelevant constraints

Common errors

- Use delays (e.g. `Thread.sleep(100)`) to ensure that "other thread done something"
  - use synchronized state variables, `CountDownLatch` or monitors
- Use too strict assertions that trigger false-positive test failures
  - formalize invariants, drop irrelevant constraints
- Test scenarios are missing execution patterns so could not detect some race conditions
  - use tests with custom scheduling and aim to 100% coverage

Common errors

- Use delays (e.g. `Thread.sleep(100)`) to ensure that "other thread done something"
  - use synchronized state variables, `CountDownLatch` or monitors
- Use too strict assertions that trigger false-positive test failures
  - formalize invariants, drop irrelevant constraints
- Test scenarios are missing execution patterns so could not detect some race conditions
  - use tests with custom scheduling and aim to 100% coverage
- Test scenarios are missing some essential consistency checks (e.g. because of mocking)
  - enable some lightweight fail-fast checks on production system
  - validate execution trace from production system

Common errors

- Use delays (e.g. `Thread.sleep(100)`) to ensure that "other thread done something"
  - use synchronized state variables, `CountDownLatch` or monitors
- Use too strict assertions that trigger false-positive test failures
  - formalize invariants, drop irrelevant constraints
- Test scenarios are missing execution patterns so could not detect some race conditions
  - use tests with custom scheduling and aim to 100% coverage
- Test scenarios are missing some essential consistency checks (e.g. because of mocking)
  - enable some lightweight fail-fast checks on production system
  - validate execution trace from production system
- Ignorance of tools
  - use test generators (property testing, fuzzers, lincheck)
  - use properly designed concurrency stress systems (JMH, jcstress)
  - do not use tools that you do not understand

Common errors

- Use delays (e.g. `Thread.sleep(100)`) to ensure that "other thread done something"
  - use synchronized state variables, `CountDownLatch` or monitors
- Use too strict assertions that trigger false-positive test failures
  - formalize invariants, drop irrelevant constraints
- Test scenarios are missing execution patterns so could not detect some race conditions
  - use tests with custom scheduling and aim to 100% coverage
- Test scenarios are missing some essential consistency checks (e.g. because of mocking)
  - enable some lightweight fail-fast checks on production system
  - validate execution trace from production system
- Ignorance of tools
  - use test generators (property testing, fuzzers, lincheck)
  - use properly designed concurrency stress systems (JMH, jcstress)
  - do not use tools that you do not understand (ChatGPT may block your skill growth)

Common errors

- Use delays (e.g. `Thread.sleep(100)`) to ensure that "other thread done something"
  - use synchronized state variables, `CountDownLatch` or monitors
- Use too strict assertions that trigger false-positive test failures
  - formalize invariants, drop irrelevant constraints
- Test scenarios are missing execution patterns so could not detect some race conditions
  - use tests with custom scheduling and aim to 100% coverage
- Test scenarios are missing some essential consistency checks (e.g. because of mocking)
  - enable some lightweight fail-fast checks on production system
  - validate execution trace from production system
- Ignorance of tools
  - use test generators (property testing, fuzzers, lincheck)
  - use properly designed concurrency stress systems (JMH, jcstress)
  - do not use tools that you do not understand (ChatGPT may block your skill growth)
- Obsessive perfectionism
  - formalize QA criteria and limit investments into testing

Common errors: sample v1

```
static int x = 0; static Object lock = new Object();
void thread_1() {
  synchronized(lock) {
    assert x == 0;
    lock.wait();
    assert x == 1;
  }
}
void thread_2() {
  sleep(100); // allow thread_1 to grab lock
  synchronized(lock) {
    x++;
    lock.notify();
  }
}
```

Common errors: sample v2

```
static int x = 0; static Object lock = new Object();
void thread_1() {
  synchronized(lock) {
    assert x == 0;
    lock.wait();
    assert x == 1;
  }
}
void thread_2() {
  // sleep(100);
  synchronized(lock) {
    x++;
    lock.notify();
  }
}
```

Common errors: sample v3

```
static int x = 0; static Object lock = new Object();
void thread_1() {
  synchronized(lock) {
    // assert x == 0;
    lock.wait();
    assert x == 1;
  }
}
void thread_2() {
  // sleep(100);
  synchronized(lock) {
    x++;
    lock.notify();
  }
}
```

Common errors: sample v4

```
static int x = 0; static Object lock = new Object();
void thread_1() {
  synchronized(lock) {
    // assert x == 0;
    if (x == 0) lock.wait();
    assert x == 1;
  }
}
void thread_2() {
  // sleep(100);
  synchronized(lock) {
    x++;
    lock.notify();
  }
}
```

Validation: required mindset

"Is parallel programming hard, and, if so, what can you do about it?"(a.k.a. perfbook)[14]
Chapter 11 "Validation":

[14] https://github.com/paulmckrcu/perfbook

Validation: required mindset

"Is parallel programming hard, and, if so, what can you do about it?"(a.k.a. perfbook)[14]
Chapter 11 "Validation":

- The only bug-free programs are trivial programs

[14] https://github.com/paulmckrcu/perfbook

Validation: required mindset

"Is parallel programming hard, and, if so, what can you do about it?"(a.k.a. perfbook)[14]
Chapter 11 "Validation":

- The only bug-free programs are trivial programs
- A reliable program has no known bugs

[14] https://github.com/paulmckrcu/perfbook

Validation: required mindset

"Is parallel programming hard, and, if so, what can you do about it?"(a.k.a. perfbook)[14]
Chapter 11 "Validation":

- The only bug-free programs are trivial programs
- A reliable program has no known bugs

Conclusion:

[14] https://github.com/paulmckrcu/perfbook

Validation: required mindset

"Is parallel programming hard, and, if so, what can you do about it?"(a.k.a. perfbook)[14] Chapter 11 "Validation":

- The only bug-free programs are trivial programs
- A reliable program has no known bugs

Conclusion: any reliable non-trivial program contains at least one bug that you do not know about.

[14] https://github.com/paulmckrcu/perfbook

Enough is enough: estimate bug probability

We will discuss the simplest discrete test scenario:

- Individual test run either fails or successfully finishes[15]

[15] Continuous test example: run 1 minute under heavy contention, ensure no exception happen

Enough is enough: estimate bug probability

We will discuss the simplest discrete test scenario:

- Individual test run either fails or successfully finishes[15]

Probability of single failure f (e.g. 0.1, 10%)

[15] Continuous test example: run 1 minute under heavy contention, ensure no exception happen

Enough is enough: estimate bug probability

We will discuss the simplest discrete test scenario:

- Individual test run either fails or successfully finishes[15]

Probability of single failure f (e.g. 0.1, 10%)
Probability of single success $1 - f$ (e.g. 0.9, 90%)

[15] Continuous test example: run 1 minute under heavy contention, ensure no exception happen

Enough is enough: estimate bug probability

We will discuss the simplest discrete test scenario:

- Individual test run either fails or successfully finishes[15]

Probability of single failure f (e.g. 0.1, 10%)
Probability of single success $1 - f$ (e.g. 0.9, 90%)
Probability n repeats will succeed $S_n = (1 - f)^n$

[15] Continuous test example: run 1 minute under heavy contention, ensure no exception happen

Enough is enough: estimate bug probability

We will discuss the simplest discrete test scenario:

- Individual test run either fails or successfully finishes[15]

Probability of single failure f (e.g. 0.1, 10%)
Probability of single success $1 - f$ (e.g. 0.9, 90%)
Probability n repeats will succeed $S_n = (1 - f)^n$
Probability of failure $F_n = 1 - (1 - f)^n$

[15] Continuous test example: run 1 minute under heavy contention, ensure no exception happen

Enough is enough: estimate bug probability

We will discuss the simplest discrete test scenario:

- Individual test run either fails or successfully finishes[15]

Probability of single failure f (e.g. 0.1, 10%)
Probability of single success $1 - f$ (e.g. 0.9, 90%)
Probability n repeats will succeed $S_n = (1 - f)^n$
Probability of failure $F_n = 1 - (1 - f)^n$
"How many times should we run the test to cause the probability of failure to rise above 99%?"

[15] Continuous test example: run 1 minute under heavy contention, ensure no exception happen

Enough is enough: estimate bug probability

We will discuss the simplest discrete test scenario:

- Individual test run either fails or successfully finishes[15]

Probability of single failure f (e.g. 0.1, 10%)
Probability of single success $1 - f$ (e.g. 0.9, 90%)
Probability n repeats will succeed $S_n = (1 - f)^n$
Probability of failure $F_n = 1 - (1 - f)^n$
"How many times should we run the test to cause the probability of failure to rise above 99%?"

$$n = \frac{log(1 - F_n)}{log(1 - f)}$$

[15] Continuous test example: run 1 minute under heavy contention, ensure no exception happen

Be careful. It is real world out there.

As of 2017 Linux kernel was estimated to have more than 20 billion instances running throughout the world.
Bug that occurs once every million years on a single system will be encountered more than

Be careful. It is real world out there.

As of 2017 Linux kernel was estimated to have more than 20 billion instances running throughout the world.
Bug that occurs once every million years on a single system will be encountered more than 50 times per day.

Be careful. It is real world out there.

As of 2017 Linux kernel was estimated to have more than 20 billion instances running throughout the world.
Bug that occurs once every million years on a single system will be encountered more than 50 times per day.
Any mission-critical software for

- health[16]
- aerospace[17]
- military[18]

is also not satisfied with "just testing". We will talk a little bit about this problem at the very end of this Lecture.

[16] https://en.wikipedia.org/wiki/Therac-25
[17] https://en.wikipedia.org/wiki/Ariane_flight_V88
[18] https://devblogs.microsoft.com/oldnewthing/20180228-00/?p=98125

Lecture plan

1. Public API
   - Documentation
   - Concurrent invariants
2. Testing
   - Unit testing and stress testing
   - Execution trace analysis
   - Test design to increase bug probability
3. Tooling
   - Static checks
   - Dynamic checks
   - Monitoring
   - Chaos mode execution
4. Summary: design of reliable concurrent software
5. Formal methods: model checking

Static code analysis

- Java FindBugs `https://findbugs.sourceforge.net`
  - `https://findbugs.sourceforge.net/bugDescriptions.html`, search for "synchronized" or "concurrent"
  - "Finding Concurrency Bugs In Java"by David Hovemeyer and William Pugh[19]
- IneliJ IDEA inspections and analysis, interactive debugger[20]
- "Java Concurrency in Practice"(JCIP)
  `https://jcip.net/annotations/doc/index.html`

[19] https://www.cs.jhu.edu/~daveho/pubs/csjp2004.pdf

[20] https://www.jetbrains.com/help/idea/detect-concurrency-issues.html

Dynamic checks

- Assertions
- Configurable heavyweight checks
- Configurable loggers and other dependency injection ideas
- Using compile-time program rewriting[21]
- Using run-time program transformation[22]

[21] https://en.wikipedia.org/wiki/Aspect-oriented_programming

[22] https://docs.oracle.com/en/java/javase/11/docs/api/java.instrument/java/lang/instrument/package-summary.html

Dynamic checks

- Assertions
- Configurable heavyweight checks
- Configurable loggers and other dependency injection ideas
- Using compile-time program rewriting[21]
- Using run-time program transformation[22]

Other languages also support dynamic checks, popular term is "sanitizers" or "race detectors":

- `https://valgrind.org`
- `https://valgrind.org/docs/manual/hg-manual.html`
- `https://go.dev/doc/articles/race_detector`
- `https://docs.kernel.org/dev-tools/ubsan.html`
- `https://clang.llvm.org/docs/AddressSanitizer.html`

[21] https://en.wikipedia.org/wiki/Aspect-oriented_programming

[22] https://docs.oracle.com/en/java/javase/11/docs/api/java.instrument/java/lang/instrument/package-summary.html

Monitoring

- https://docs.oracle.com/en/java/javase/11/tools/jstack.html
- https://docs.oracle.com/en/java/javase/11/tools/jmap.html
- https://docs.oracle.com/en/java/javase/11/tools/jcmd.html
- https://eclipse.dev/mat
- https://axiomjdk.ru/announcements/2024/10/22/jdk-flight-recorder
- infinite number of commercial tools

Monitoring

- https://docs.oracle.com/en/java/javase/11/tools/jstack.html
- https://docs.oracle.com/en/java/javase/11/tools/jmap.html
- https://docs.oracle.com/en/java/javase/11/tools/jcmd.html
- https://eclipse.dev/mat
- https://axiomjdk.ru/announcements/2024/10/22/jdk-flight-recorder
- infinite number of commercial tools

Using `ThreadFactories` and `Executors` allows you to debug component-by-component

- Inefficient but correct implementation
- Implementation with extensive logging

Lecture plan

1. Public API
   - Documentation
   - Concurrent invariants
2. Testing
   - Unit testing and stress testing
   - Execution trace analysis
   - Test design to increase bug probability
3. Tooling
   - Static checks
   - Dynamic checks
   - Monitoring
   - Chaos mode execution
4. Summary: design of reliable concurrent software
5. Formal methods: model checking

Pain points

- Unpredicatable speed of execution
- Arbitrary "whole program control flow"
  - `Thread.start` looks like `goto`
- Arbitrary "thread-specific control flow"
  - timeout, notification, interruption, spurious wakeup
- Limited composability of modules
  - locking policy is not clear
  - lock ordering hidden by virtual methods
  - exposed `synchronized`
  - non-documented thread-safety
- Hard to diagnose performance problems
  - livelock, priority inversion, thundering herd, scalability
- It is **too easy** to introduce a race condition
  - it is **too hard** to find it on review, during testing, debug on production system

Need for speed

- Unpredicatable speed of execution

Need for speed

- Unpredicatable speed of execution
- Hard to detect bugs
- Hard to reproduce problems
- Hard to verify issue is fixed

Question time

Question: Could we change timings of multi-threaded program?

Scheduling control: language level

Custom scheduling using language primitives:

- Arbitrary `Thread.sleep` here and there
  - Dependency injection/inheritance/interfaces
  - AspectJ
  - Java agents and bytecode transformers

Scheduling control: language level

Custom scheduling using language primitives:

- Arbitrary `Thread.sleep` here and there
  - Dependency injection/inheritance/interfaces
  - AspectJ
  - Java agents and bytecode transformers
- Guided insertion of randomized delay for synchronization operations
  - Efficient Scalable Thread-Safety-Violation Detection[23]

[23] https://rohan.padhye.org/files/tsvd-sosp19.pdf

Scheduling control: language level

Custom scheduling using language primitives:

- Arbitrary `Thread.sleep` here and there
  - Dependency injection/inheritance/interfaces
  - AspectJ
  - Java agents and bytecode transformers
- Guided insertion of randomized delay for synchronization operations
  - Efficient Scalable Thread-Safety-Violation Detection[23]
- Analysis of communication points (memory locations)
  - Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis[24]

[23] https://rohan.padhye.org/files/tsvd-sosp19.pdf

[24] https://sishuaigong.github.io/pdf/sosp21-snowboard.pdf

Question time

Question: Who controls **all** timings of multi-threaded program?

Scheduling control: OS level

OS scheduler:

- Scheduling policy (round-robin, shortest remaining time ...)
- Thread priorities, Scheduling quantum, Context switch, Fairness, Real-time

Scheduling control: OS level

OS scheduler:

- Scheduling policy (round-robin, shortest remaining time ...)
- Thread priorities, Scheduling quantum, Context switch, Fairness, Real-time

Pinning/affinity[25]:

- `Thread A` could be executed only by `Processor 1` or `Processor 7`
- `taskset`[26], `numactl`[27]

[25] https://en.wikipedia.org/wiki/Processor_affinity
[26] https://man7.org/linux/man-pages/man1/taskset.1.html
[27] https://linux.die.net/man/8/numactl

Scheduling control: OS level

OS scheduler:

- Scheduling policy (round-robin, shortest remaining time ...)
- Thread priorities, Scheduling quantum, Context switch, Fairness, Real-time

Pinning/affinity[25]:

- `Thread A` could be executed only by `Processor 1` or `Processor 7`
- `taskset`[26], `numactl`[27]

Replace/record OS scheduling decisions to enable deterministic execution:

- Reproducible container[28]
- Lightweight record-replay[29]
- Time-travel debugging[30]

[25] https://en.wikipedia.org/wiki/Processor_affinity
[26] https://man7.org/linux/man-pages/man1/taskset.1.html
[27] https://linux.die.net/man/8/numactl
[28] https://github.com/facebookexperimental/hermit
[29] https://rr-project.org/
[30] https://en.wikipedia.org/wiki/Time_travel_debugging

Chaos mode

- Multi-threaded program executed by single processor (pre-emptive multitasking)
- Every scheduling decision is
  - Reproducible
  - Customizable – pseudo-random with some seed

[31] Probabilistic Guarantees of Finding Bugs https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/asplos277-pct.pdf

Chaos mode

- Multi-threaded program executed by single processor (pre-emptive multitasking)
- Every scheduling decision is
  - Reproducible
  - Customizable – pseudo-random with some seed

Advantages:

- Helps to find/reproduce race conditions, data races, deadlocks

[31] Probabilistic Guarantees of Finding Bugs https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/asplos277-pct.pdf

Chaos mode

- Multi-threaded program executed by single processor (pre-emptive multitasking)
- Every scheduling decision is
  - Reproducible
  - Customizable – pseudo-random with some seed

Advantages:

- Helps to find/reproduce race conditions, data races, deadlocks

Disadvantages:

- Subtle concurrency problems (word tearing, visibility) are not detected
- Heuristic search of "interesting" scheduling decisions[31]

[31] Probabilistic Guarantees of Finding Bugs https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/asplos277-pct.pdf

Lecture plan

1. Public API
   - Documentation
   - Concurrent invariants
2. Testing
   - Unit testing and stress testing
   - Execution trace analysis
   - Test design to increase bug probability
3. Tooling
   - Static checks
   - Dynamic checks
   - Monitoring
   - Chaos mode execution
4. Summary: design of reliable concurrent software
5. Formal methods: model checking

Reliable concurrent software: chasing the horizon

Any error must manifest itself as soon as possible:

- Readable and complete documentation, locking policy, inheritance suggestions
- External invariants (exceptions) + internal invariants (assertions)

Use as many validation techniques as you could:

- Unit testing
- Stress testing
- Execution trace analysis
- Design tests to avoid false-positives
- Estimate bug propability and required post-fix testing effort

Master tools:

- Static checks
- Dynamic checks
- Monitoring
- Scheduling randomization

Lecture plan

1. Public API
   - Documentation
   - Concurrent invariants
2. Testing
   - Unit testing and stress testing
   - Execution trace analysis
   - Test design to increase bug probability
3. Tooling
   - Static checks
   - Dynamic checks
   - Monitoring
   - Chaos mode execution
4. Summary: design of reliable concurrent software
5. Formal methods: model checking

Algorithm correctness

Use pen and paper to prove some properties.

Algorithm correctness

Use pen and paper to prove some properties.
We will discuss required mathematics in Lecture 6 and Lecture 7.

Machine-assisted deductive verification

Provers and verifiers:

- Coq `https://coq.inria.fr`
- Agda `https://wiki.portal.chalmers.se/agda/pmwiki.php`
- PVS `https://pvs.csl.sri.com/`
- Z3 `https://github.com/Z3Prover/z3`

Machine-assisted deductive verification

Provers and verifiers:

- Coq `https://coq.inria.fr`
- Agda `https://wiki.portal.chalmers.se/agda/pmwiki.php`
- PVS `https://pvs.csl.sri.com/`
- Z3 `https://github.com/Z3Prover/z3`

There are dedicated courses on formal verification in NSU or Computer Science Center. Which are N times harder than "just programming".

Machine-assisted deductive verification

Provers and verifiers:

- Coq `https://coq.inria.fr`
- Agda `https://wiki.portal.chalmers.se/agda/pmwiki.php`
- PVS `https://pvs.csl.sri.com/`
- Z3 `https://github.com/Z3Prover/z3`

There are dedicated courses on formal verification in NSU or Computer Science Center. Which are N times harder than "just programming".
Software foundations[32] is N times harder than SICP[33].

[32] https://softwarefoundations.cis.upenn.edu/

[33] https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs

Model checking

This part is optional part of the course. There will be no questions related to model checking on final exam.

Model checking

This part is optional part of the course. There will be no questions related to model checking on final exam.

- Hydraconf, "Java Path Finder: going to Mars without bugs and deadlocks" `https://youtu.be/dgHbSL_aDs0?si=vi81xieQ4zKRECQG`
- Heisenbug, "Java Path Finder: летим на Марс без багов и дедлоков" `https://youtu.be/sQSwShW_IlI?si=ZMlKKLQxMZYhk1T7`

Summary: homework

Task 5.1: JCStress basics.