Lecture 2: basic concepts

mutex, acquisition order, reentrancy, fairness, data locking, code locking, signalling, condition variable, lost signal, spurious wakeup

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l2.pdf

In previous episode

- We study communication and coordination of different agents.
- Every agent has its own speed and scenario of execution.
- We are focusing on threads which are part of OS process and managed by scheduler.
- We expect OS to use pre-emptive multitasking (time-sharing of CPU cores).

Any concurrent task has

- parallel (independent)
- sequential (dependent)

parts so max speedup is limited by Amdahl's law.
Threads have read/write access to shared memory which leads to

- race conditions, data races, visibility problems

Threads use blocking methods which leads to

- deadlocks, priority inversion

therefore we use wait-for graph and observability API.

Lecture plan

1. Thread safety
2. Mutual exclusion
3. Mutex
   - Basics
   - Usage patterns
   - Bug prevention
4. Signalling
5. Condition variable
   - Concurrency hazards
   - Performance hazards
6. Summary

Lecture plan

Toy problem: thread-safe counter

Description

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

Question time

Question: What is "thread-safe"?

Toy problem: thread-safe counter

Description

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

Thread-safe – may be invoked from different threads simultaneously and behave "normally".

Toy problem: thread-safe counter

Description

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

Thread-safe – may be invoked from different threads simultaneously and behave "normally".
What is normal?

Toy problem: thread-safe counter

Description

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

Thread-safe – may be invoked from different threads simultaneously and behave "normally".
What is normal?

- `get` and `increment` are consistent
- no `increment` is lost

Toy problem: thread-safe counter

Description

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

How to handle race conditions?

Toy problem: thread-safe counter

Description

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

How to handle race conditions?

How to distinguish user-side misuse from library-side bug?

Toy problem: thread-safe counter

Description

```
Counter c = new Counter(0);
Thread t1 = new Thread( () -> { c.increment(); println(c.get()); });
Thread t2 = new Thread( () -> { c.increment(); println(c.get()); });
t1.start(); t2.start(); t1.join(); t2.join();
System.out.println(c.get());
```

Toy problem: thread-safe counter

Description

```
Counter c = new Counter(0);
Thread t1 = new Thread( () -> { c.increment(); println(c.get()); });
Thread t2 = new Thread( () -> { c.increment(); println(c.get()); });
t1.start(); t2.start(); t1.join(); t2.join();
System.out.println(c.get());
```

Execution 1: `t1=1 t2=2 main=2`

Toy problem: thread-safe counter

Description

```
Counter c = new Counter(0);
Thread t1 = new Thread( () -> { c.increment(); println(c.get()); });
Thread t2 = new Thread( () -> { c.increment(); println(c.get()); });
t1.start(); t2.start(); t1.join(); t2.join();
System.out.println(c.get());
```

Execution 1: `t1=1 t2=2 main=2`

Execution 2: `t1=2 t2=1 main=2`

Toy problem: thread-safe counter

Description

```
Counter c = new Counter(0);
Thread t1 = new Thread( () -> { c.increment(); println(c.get()); });
Thread t2 = new Thread( () -> { c.increment(); println(c.get()); });
t1.start(); t2.start(); t1.join(); t2.join();
System.out.println(c.get());
```

Execution 1: `t1=1 t2=2 main=2`
Execution 2: `t1=2 t2=1 main=2`
Execution 3: `t1=2 t2=2 main=2`

Concurrent consistency

- Nothing crashes

Concurrent consistency

- Nothing crashes
- When I run the program, it works as intended

Concurrent consistency

- Nothing crashes
- When I run the program, it works as intended
- All operations work "logically"

Concurrent consistency

- Nothing crashes
- When I run the program, it works as intended
- All operations work "logically"

One possible formalization: all operations could be treated as "atomic" (non-divisible, transactional) and ordered on single timeline.

Concurrent consistency

- Nothing crashes
- When I run the program, it works as intended
- All operations work "logically"

One possible formalization: all operations could be treated as "atomic" (non-divisible, transactional) and ordered on single timeline.
There are other approaches, see consistency models in Lecture 6.

Concurrent consistency

- Nothing crashes
- When I run the program, it works as intended
- All operations work "logically"

One possible formalization: all operations could be treated as "atomic" (non-divisible, transactional) and ordered on single timeline.
There are other approaches, see consistency models in Lecture 6.
Our current requirements:

- all events (method calls) could be ordered as if they executed sequentially
- in-thread events and operations are "sequential", but may be reordered up to "synchronization points"

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Question time

Question: If you have only `Thread.start` and `Thread.join` as concurrent primitives, how would you implement thread-safe counter?

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Looks like that is not enough.

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Looks like that is not enough.

When some thread executes `counter.increment`, other threads:

- allowed to execute `counter.get`?

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Looks like that is not enough.

When some thread executes `counter.increment`, other threads:

- allowed to execute `counter.get`? No, read-write data race!

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Looks like that is not enough.

When some thread executes `counter.increment`, other threads:

- allowed to execute `counter.get`? No, read-write data race!
- allowed to execute `counter.increment`?

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Looks like that is not enough.
When some thread executes `counter.increment`, other threads:

- allowed to execute `counter.get`? No, read-write data race!
- allowed to execute `counter.increment`? No, write-write data race!

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Looks like that is not enough.
When some thread executes `counter.increment`, other threads:

- allowed to execute `counter.get`? No, read-write data race!
- allowed to execute `counter.increment`? No, write-write data race!

Conclusion:

- avoid concurrent execution of the same code block by different threads (mutual exclusion)

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Looks like that is not enough.
When some thread executes `counter.increment`, other threads:

- allowed to execute `counter.get`? No, read-write data race!
- allowed to execute `counter.increment`? No, write-write data race!

Conclusion:

- avoid concurrent execution of the same code block by different threads (mutual exclusion)
- guard instruction sequences against concurrent modification (code locking)

Toy problem: thread-safe counter

How to implement?

Synchronization points we know so far:

- `Thread.start`
- `Thread.join`

Looks like that is not enough.
When some thread executes `counter.increment`, other threads:

- allowed to execute `counter.get`? No, read-write data race!
- allowed to execute `counter.increment`? No, write-write data race!

Conclusion:

- avoid concurrent execution of the same code block by different threads (mutual exclusion)
- guard instruction sequences against concurrent modification (code locking)
- guarantee that only one thread may enter some code fragment (critical section)

Lecture plan

Mutual exclusion

Naming

```
interface Lock {
    void lock();
    void unlock();
}
```

Mutual exclusion

Naming

```
interface Lock {
    void lock();
    void unlock();
}
interface Mutex {
    void enter();
    void exit();
}
```

Mutual exclusion

Naming

```
interface Lock {
    void lock();
    void unlock();
}
interface Mutex {
    void enter();
    void exit();
}
interface CriticalSection {
    void begin();
    void end();
}
```

Mutual exclusion

Usage

Lock usage[1]:

```
Lock lock = ...
lock.lock();
try {
  ...
} finally {
  lock.unlock();
}
```

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/Lock.html

Mutual exclusion

Usage

Lock usage[1]:

```
Lock lock = ...
lock.lock();
try {
  ...
} finally {
  lock.unlock();
}
```

- **If you are not using try-finally for locks – you are writing incorrect code**

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/Lock.html

Question time

Question: How to write code that `locks` in one method and `unlocks` in other?

Exceptions are hard

Homework, mail

Task 2.1.a Is it possible that some exception would happen ***inside*** `lock` *or* `unlock` *operation? Justify your answer by using precise* `chapter.section` *number from Java Language Specification.*

Exceptions are hard

Homework, mail

Task 2.1.a Is it possible that some exception would happen ***inside*** `lock` *or* `unlock` *operation? Justify your answer by using precise* `chapter.section` *number from Java Language Specification.*

Help: $\sqrt[3]{1331}$ is good magic number.

Exceptions are hard

Homework, mail

Task 2.1.a Is it possible that some exception would happen ***inside*** `lock` *or* `unlock` *operation? Justify your answer by using precise* `chapter.section` *number from Java Language Specification.*

Help: $\sqrt[3]{1331}$ is good magic number.

Homework, mail

Task 2.1.b Is it possible to design "bullet-proof" (w.r.t. exceptions) concurrency primitives in Java language? Justify your answer by using precise JDK Enhancement Proposal number.

Exceptions are hard

Homework, mail

Task 2.1.a Is it possible that some exception would happen ***inside*** `lock` *or* `unlock` *operation? Justify your answer by using precise* `chapter.section` *number from Java Language Specification.*

Help: $\sqrt[3]{1331}$ is good magic number.

Homework, mail

Task 2.1.b Is it possible to design "bullet-proof" (w.r.t. exceptions) concurrency primitives in Java language? Justify your answer by using precise JDK Enhancement Proposal number.

Help: $\sqrt{72900}$ is good magic number, too.

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do?

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do?
Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy?

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy? Subscribe to "mutex release" event, perform context switch, release current scheduling quantum. When thread will be notified?

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy? Subscribe to "mutex release" event, perform context switch, release current scheduling quantum. When thread will be notified?

- randomly after some time period (if mutex is still busy, thread will retreat again)

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy? Subscribe to "mutex release" event, perform context switch, release current scheduling quantum. When thread will be notified?

- randomly after some time period (if mutex is still busy, thread will retreat again)
- when mutex is unlocked (but mutex may become busy while thread was "awakening")

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy? Subscribe to "mutex release" event, perform context switch, release current scheduling quantum. When thread will be notified?

- randomly after some time period (if mutex is still busy, thread will retreat again)
- when mutex is unlocked (but mutex may become busy while thread was "awakening")
- when mutex owner passed ownership to this particular thread (not every algorithm allows such approach, more on this later)

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy? Subscribe to "mutex release" event, perform context switch, release current scheduling quantum. When thread will be notified?

- randomly after some time period (if mutex is still busy, thread will retreat again)
- when mutex is unlocked (but mutex may become busy while thread was "awakening")
- when mutex owner passed ownership to this particular thread (not every algorithm allows such approach, more on this later)
- when OS analysis of priority inversion problems decides to boost some threads

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy? Subscribe to "mutex release" event, perform context switch, release current scheduling quantum. When thread will be notified?

- randomly after some time period (if mutex is still busy, thread will retreat again)
- when mutex is unlocked (but mutex may become busy while thread was "awakening")
- when mutex owner passed ownership to this particular thread (not every algorithm allows such approach, more on this later)
- when OS analysis of priority inversion problems decides to boost some threads ...

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy? Subscribe to "mutex release" event, perform context switch, release current scheduling quantum. When thread will be notified?

- randomly after some time period (if mutex is still busy, thread will retreat again)
- when mutex is unlocked (but mutex may become busy while thread was "awakening")
- when mutex owner passed ownership to this particular thread (not every algorithm allows such approach, more on this later)
- when OS analysis of priority inversion problems decides to boost some threads ...

Actually, you do not know.

Mutex basics

OS level

Mutex allows only one thread to enter critical section. What should other participants do? Await their "turn". Who controls active/non-active state of a thread?
OS scheduler. What exactly should current thread do when mutex is already busy? Subscribe to "mutex release" event, perform context switch, release current scheduling quantum. When thread will be notified?

- randomly after some time period (if mutex is still busy, thread will retreat again)
- when mutex is unlocked (but mutex may become busy while thread was "awakening")
- when mutex owner passed ownership to this particular thread (not every algorithm allows such approach, more on this later)
- when OS analysis of priority inversion problems decides to boost some threads ...

Actually, you do not know.

Always assume that concurrent primitive obeys the public contract and may arbitrary change internal implementation

Question time

Question: Contended `mutex.lock` operation may cause "preliminary" context switch of a thread. How scheduler could remedy the problem of "broken amortization"?

Toy problem: thread-safe counter

ReentrantLock-based[2] implementation

```
public class Counter {
    private final Lock lock = new ReentrantLock();
    private long counter;
    public Counter(long initial) { counter = initial; }
    public void increment() {
        lock.lock();
        try { counter++; } finally { lock.unlock(); }
    }
    public long get() {
        lock.lock();
        try {  return counter; } finally { lock.unlock(); }
    }
}
```

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

Toy problem: thread-safe counter

Mutex implementation: brief discussion

Any concurrent algorithm should be analyzed for the **key** properties:

- Safety (correctness)
  - Implements contract (consistency)
  - Absence of invariant violations
  - Absence of data races
  - Absence of concurrent logical errors (unfortunate race conditions)
- Liveness (progress)
  - Deadlock-freedom
  - Livelock-freedom
  - Starvation-freedom
- Performance
  - Throughput (fast-path/slow-path overheads)
  - Latency (fairness, priority inversion)
  - Scalability

Question time

Question: You have thread-safe List with methods:

- `void add(e)`
- `boolean contains(e)`

Characterize thread-safety of the following operation:

```
addIfAbsent(e) = if (!contains(e)) add(e)
```

Lecture plan

Admission policy

Contenders ordering

Contended mutex:

- single **owner**
- set of **waiters** (EnterSet)
- set of **arriving threads** (ArriveSet)

Admission policy

Contenders ordering

Contended mutex:

- single **owner**
- set of **waiters** (EnterSet)
- set of **arriving threads** (ArriveSet)

How to manage EnterSet?

Admission policy

Contenders ordering

Contended mutex:

- single **owner**
- set of **waiters** (EnterSet)
- set of **arriving threads** (ArriveSet)

How to manage EnterSet?

- last-in-first-out (LIFO)
- first-in-first-out (FIFO)
- priority queue or random choice

Admission policy

Contenders ordering

Contended mutex:

- single **owner**
- set of **waiters** (EnterSet)
- set of **arriving threads** (ArriveSet)

How to manage EnterSet?

- last-in-first-out (LIFO)
- first-in-first-out (FIFO)
- priority queue or random choice

How to manage ArriveSet?

Admission policy

Contenders ordering

Contended mutex:

- single **owner**
- set of **waiters** (EnterSet)
- set of **arriving threads** (ArriveSet)

How to manage EnterSet?

- last-in-first-out (LIFO)
- first-in-first-out (FIFO)
- priority queue or random choice

How to manage ArriveSet?

- try-lock-then-wait (ArriveSet > EnterSet)
- if-busy-then-wait (ArriveSet < EnterSet)
- random or heuristic choice

Admission policy

Contenders ordering

Contended mutex:

- single **owner**
- set of **waiters** (EnterSet)
- set of **arriving threads** (ArriveSet)

How to manage EnterSet?

- last-in-first-out (LIFO)
- first-in-first-out (FIFO)
- priority queue or random choice

How to manage ArriveSet?

- try-lock-then-wait (ArriveSet > EnterSet)
- if-busy-then-wait (ArriveSet < EnterSet)
- random or heuristic choice

Race conditions everywhere!

Admission policy

Design space

Depends on your goal:

- Max throughput: LIFO (unfair, best average case, degraded outliers)

Question time

Question: Why LIFO admission policy for mutex provides the best throughput?

Admission policy

Design space

Depends on your goal:

- Max throughput: LIFO (unfair, best average case, degraded outliers)
- Latency: FIFO (fair, guaranteed worst case)

Admission policy

Design space

Depends on your goal:

- Max throughput: LIFO (unfair, best average case, degraded outliers)
- Latency: FIFO (fair, guaranteed worst case)
- Predictability: almost FIFO or priorities (semi-fair, acceptable worst case)

Admission policy

Design space

Depends on your goal:

- Max throughput: LIFO (unfair, best average case, degraded outliers)
- Latency: FIFO (fair, guaranteed worst case)
- Predictability: almost FIFO or priorities (semi-fair, acceptable worst case)

Everything has negative side:

- Starvation
- Priority inversion
- Counter-examples to heuristics
- Deadlock probability

Admission policy

Design space

Depends on your goal:

- Max throughput: LIFO (unfair, best average case, degraded outliers)
- Latency: FIFO (fair, guaranteed worst case)
- Predictability: almost FIFO or priorities (semi-fair, acceptable worst case)

Everything has negative side:

- Starvation
- Priority inversion
- Counter-examples to heuristics
- Deadlock probability

Your concurrent data structures should document admission policy and livelock scenarios for blocking methods

Reentrancy

NonReentrantLock

Mutex implementation = `{ boolean busy }`

Reentrancy

NonReentrantLock

Mutex implementation = { `boolean busy` }

Single-threaded deadlock:

```
nonReentrantLock.lock();
nonReentrantLock.lock();
```

Reentrancy

Motivation

```
public class Counter {
    public void increment() {
        lock.lock();
        try {
            counter = get() + 1;
        } finally { lock.unlock(); }
    }
    public long get() {
        lock.lock();
        try {
            return counter;
        } finally { lock.unlock(); }
    }
}
```

Reentrancy

ReentrantLock

- ReentrantLock, ReentrantMutex
- RecursiveLock, RecursiveMutex

Reentrancy

ReentrantLock

- ReentrantLock, ReentrantMutex
- RecursiveLock, RecursiveMutex

Mutex implementation = `{ Owner owner, int count }`
Not every `unlock` actually releases ownership

Reentrancy

ReentrantLock

- ReentrantLock, ReentrantMutex
- RecursiveLock, RecursiveMutex

Mutex implementation = `{ Owner owner, int count }`
Not every `unlock` actually releases ownership
Important concepts:

- Structured locking: every `lock` paired with `unlock`
- Ownership: unique Thread ID (`Thread.currentThread()`[3]) to distinguish owners

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

Question time

Question: Concurrency is hard! Why would anybody use NonReentrantMutex?

Visibility and consistency

- all lock and `unlock` operations of **particular mutex** are totally ordered
- intra-thread lock and `unlock` operations of **all mutexes** are totally ordered

Visibility and consistency

- all lock and `unlock` operations of **particular mutex** are totally ordered
- intra-thread lock and `unlock` operations of **all mutexes** are totally ordered

Partial orders are tricky[4]

[4] `https://en.wikipedia.org/wiki/Partially_ordered_set`

Visibility and consistency

- all lock and `unlock` operations of **particular mutex** are totally ordered
- intra-thread lock and `unlock` operations of **all mutexes** are totally ordered

Partial orders are tricky[4]

Synchronization points you know so far:

- `Thread.start`
- `Thread.join`
- `Lock.lock`
- `Lock.unlock`

[4] `https://en.wikipedia.org/wiki/Partially_ordered_set`

Question time

Question: It would be **much** easier to say that all critical sections (code between `lock` and `unlock`) of **all** mutexes have a strict total order.
Why do we use much weaker partial ordering?

Visibility and consistency

Insufficient ordering

```
static int x, y;
void threadA() {
    lock.lock(); try { x = 1; y = 1; } finally { lock.unlock(); }
}
void threadB() {
    lock.lock(); try { x = 2; y = 2; } finally { lock.unlock(); }
}
void threadC() {
    System.out.println(x);
    System.out.println(y);
}
```

Visibility and consistency

Insufficient ordering

```
static int x, y;
void threadA() {
    lock.lock(); try { x = 1; y = 1; } finally { lock.unlock(); }
}
void threadB() {
    lock.lock(); try { x = 2; y = 2; } finally { lock.unlock(); }
}
void threadC() {
    System.out.println(x);
    System.out.println(y);
}
```

Possible result: x=2 y=0

Mutex basics

Conclusion

- Mutual exclusion maintains "order of execution" for code fragment, one thread a time
- Implicit control flow (e.g. exceptions) may violate consistency of concurrent primitive
- Performance depends on OS (scheduling quantum, scheduling policy, context switch overheads, priority) and particular implementation (admission policy)
- There are different flavours of locking primitives (reentrancy, fairness)

Locks help to solve some problems:

- avoid data race
- prevent race condition
- implement thread-safety

but may introduce new challenges:

- deadlock
- livelock/starvation/unfairness
- sequential part of execution (see Amdahl's law in Lecture 1)

Lecture plan

Using mutexes

Code locking

```
enum Grade { A, B, C, FAIL }
static long grades[] = new long[Grade.values().length()];
public static void gradeStudent(Grade g) {
    grades[g.ordinal()]++;
}
```

How to make gradeStudent thread-safe?

Using mutexes

Code locking

```
enum Grade { A, B, C, FAIL }
static long grades[] = new long[Grade.values().length()];
public static void gradeStudent(Grade g) {
    grades[g.ordinal()]++;
}
```

How to make gradeStudent thread-safe?

```
static Lock lock = new ReentrantLock();
public static void gradeStudent(Grade g) {
    lock.lock();
    try { grades[g.ordinal()]++; }
    finally { lock.unlock(); }
}
```

Using mutexes

Data locking

```
public static void gradeStudent(Grade g) {
    lock.lock();
    try {
        grades[g.ordinal()]++;
    } finally {
        lock.unlock();
    }
}
```

How to make program more scalable?

Using mutexes

Data locking

```
public static void gradeStudent(Grade g) {
    lock.lock();
    try {
        grades[g.ordinal()]++;
    } finally {
        lock.unlock();
    }
}
```

How to make program more scalable?

```
static Counter grades[] = new Counter[Grade.values().length];
public static void gradeStudent(Grade g) {
    grades[g.ordinal()].increment();
}
```

Question time

Question: In Java, every method is attached to the object instance which is actually data. Should not we always use "Data locking" terminology?

Question time

Question: In Java, every method is written in bytecode so effectively we are guarding some code fragment. Should not we always use "Code locking" terminology?

Using mutexes

Lock splitting

```
static int passedExams[] = new int[StudentList.size()]; // millions!
static Lock lock = new Lock();
public static void pass(Student s) {
    lock.lock();
    try {
        passedExams[s.number()]++;
    } finally {
        lock.unlock();
    }
}
```

Using mutexes

Lock splitting

```
static int passedExams[] = new int[StudentList.size()]; // millions!
static Lock lock = new Lock();
public static void pass(Student s) {
    lock.lock();
    try {
        passedExams[s.number()]++;
    } finally {
        lock.unlock();
    }
}
```

We cannot afford to allocate millions of `Counter` instances.

Using mutexes

Lock splitting

```
static int passedExams[] = new int[StudentList.size()]; // millions!
static Lock lock = new Lock();
public static void pass(Student s) {
    lock.lock();
    try {
        passedExams[s.number()]++;
    } finally {
        lock.unlock();
    }
}
```

We cannot afford to allocate millions of `Counter` instances.
Divide-and-conquer using arbitrary granularity.

Using mutexes

Lock splitting

```
static int passedExams[] = new int[StudentList.size()]; // millions!
static Lock[] locks = new Lock[1 + (passedExams.length / 1_000)];
public static void pass(Student s) {
    int sNum = s.number();
    int lockNum = sNum / 1_000;
    Lock lock = locks[lockNum];
    lock.lock();
    try {
        passedExams[sNum]++;
    } finally {
        lock.unlock();
    }
}
```

Lecture plan

Inevitable evil

"I would never do it"

```
threadA() {
    l1.lock();
    try {
        l2.lock();
        try { ... } finally { l2.unlock(); }
    } finally { l1.unlock(); }
}
threadB() {
    l2.lock();
    try {
        l1.lock();
        try { ... } finally { l1.unlock(); }
    } finally { l2.unlock(); }
}
```

Inevitable evil

"Oops!... I did it again"

```
void transfer(long sum, Account a, Account b) {
    a.lock.lock();
    try {
        b.lock.lock();
        try {
            if (a.withdraw(sum)) {
                b.add(sum)
            }
        } finally { b.lock.unlock(); }
    } finally { a.lock.unlock(); }
}
```

Inevitable evil

"Oops!... I did it again"

```
void transfer(long sum, Account a, Account b) {
    a.lock.lock();
    try {
        b.lock.lock();
        try {
            if (a.withdraw(sum)) {
                b.add(sum)
            }
        } finally { b.lock.unlock(); }
    } finally { a.lock.unlock(); }
}
threadA() { transfer(1, A, B); }
threadB() { transfer(1, B, A); }
```

Deadlock prevention

Ultimate deadlock prevention weapon

Deadlock prevention

Ultimate deadlock prevention weapon
Do not use blocking methods ;)

Deadlock prevention

Ultimate deadlock prevention weapon
Do not use blocking methods ;)
Alternative approaches also have drawbacks

Deadlock prevention

Minimizing attack surface:

- Use single lock in the program

Deadlock prevention

Minimizing attack surface:

- Use single lock in the program
- Use recursive locks

Deadlock prevention

Minimizing attack surface:

- Use single lock in the program
- Use recursive locks
- Use single thread in the program

Deadlock prevention

Minimizing attack surface:

- Use single lock in the program
- Use recursive locks
- Use single thread in the program
- Use message passing (copy and transfer) instead of shared mutable state

Deadlock prevention

Minimizing attack surface:

- Use single lock in the program
- Use recursive locks
- Use single thread in the program
- Use message passing (copy and transfer) instead of shared mutable state
- Use high-level abstractions (`stream.parallel.map.collect`) instead of low-level ones (`mutex.lock/unlock`)

Deadlock prevention

Minimizing attack surface:

- Use recursive locks

Question time

Question: You have `private Lock` instance and public `foo` method. How should you use lock inside method to avoid deadlocks on this instance?

Deadlock prevention

Minimizing attack surface:

- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section (”leaf” locking)

Deadlock prevention

Minimizing attack surface:

- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section ("leaf" locking)

Specialized techniques:

- Lock ordering (`lockA` < `lockB`). First sort, then lock!

Deadlock prevention

Minimizing attack surface:

- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section ("leaf" locking)

Specialized techniques:

- Lock ordering (`lockA` < `lockB`). First sort, then lock!
- Locking hierarchies (`lockA.num = 1`, `lockB.num = 2`). Always lock greater numbers!

Deadlock prevention

Minimizing attack surface:

- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section ("leaf" locking)

Specialized techniques:

- Lock ordering (`lockA` < `lockB`). First sort, then lock!
- Locking hierarchies (`lockA.num = 1`, `lockB.num = 2`). Always lock greater numbers!

Requires thinking at design time.

Question time

Question: Locking hierarchy reports deadlock in run-time, on erroneous attempt to "get lower lock". Actually it preliminary crashes the program, even if deadlock was not supposed to happen. Why do we call it "deadlock prevention" software development technique?

Lock convoy

First thread arrives to mutex.

Lock convoy

First thread arrives to mutex. Owns it.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex. Waits for it. Quantum lost.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex. Waits for it. Quantum lost.
First thread leaves mutex.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex. Waits for it. Quantum lost.
First thread leaves mutex. Second thread gets ownership.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex. Waits for it. Quantum lost.
First thread leaves mutex. Second thread gets ownership.
Third thread arrives to mutex.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex. Waits for it. Quantum lost.
First thread leaves mutex. Second thread gets ownership.
Third thread arrives to mutex. Waits for it. Quantum lost.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex. Waits for it. Quantum lost.
First thread leaves mutex. Second thread gets ownership.
Third thread arrives to mutex. Waits for it. Quantum lost.
First thread arrives to mutex.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex. Waits for it. Quantum lost.
First thread leaves mutex. Second thread gets ownership.
Third thread arrives to mutex. Waits for it. Quantum lost.
First thread arrives to mutex. Waits for it. Quantum lost.

Lock convoy

First thread arrives to mutex. Owns it.
Second thread arrives to mutex. Waits for it. Quantum lost.
First thread leaves mutex. Second thread gets ownership.
Third thread arrives to mutex. Waits for it. Quantum lost.
First thread arrives to mutex. Waits for it. Quantum lost.
Second thread leaves mutex ...

Design challenges

- When you call some code, it could acquire/release arbitrary locks
- When your code is invoked by some thread, that thread could already own arbitrary locks

Design challenges

- When you call some code, it could acquire/release arbitrary locks
- When your code is invoked by some thread, that thread could already own arbitrary locks

Composability hell.

Design challenges

- When you call some code, it could acquire/release arbitrary locks
- When your code is invoked by some thread, that thread could already own arbitrary locks

Trust no one

- Before calling external code, release all locks
- Avoid using external locks
- Do not expose internal locks
- Start computation in special "clean" thread

Design challenges

- When you call some code, it could acquire/release arbitrary locks
- When your code is invoked by some thread, that thread could already own arbitrary locks

Trust no one

- Before calling external code, release all locks
- Avoid using external locks
- Do not expose internal locks
- Start computation in special "clean" thread

But be friendly

- Document locking policy inside class
- Document locking policy for users

Mutual exclusion

Conclusion

Mutual exclusion is one of the simplest approaches to guarantee thread-safety

There exist many mutex implementations featuring different properties:

- reentrancy, admission policy, fairness

You could use different ways to structure your program and enjoy locking benefits:

- code locking, data locking, lock splitting

Mutex is a blocking primitive, so additional care needed to prevent deadlocks:

- preference to recursive locks, encapsulation, enforcing lock acquisition order, avoiding external code invocation inside critical section

Default progress problems (deadlock, starvation, priority inversion) accompanied by lock convoy

Homework

Homework, mail

Task 2.2 Open `https://deadlockempire.github.io`, *pass all "Locks" levels.*

Dining philosophers problem

Homework, code

Task 2.3 `https://github.com/Suazars/parallel-programming/blob/main/hw/block1/lec2/dining-philosophers/dining-philosophers/readme.md`

Not critical task.

Dining philosophers problem

Homework, code

Task 2.3 `https://github.com/Svazars/parallel-programming/blob/main/hw/block1/lec2/dining-philosophers/dining-philosophers/readme.md`

Not critical task. But your first programming assignment.

Dining philosophers problem

Homework, code

Task 2.3 `https://github.com/Svazars/parallel-programming/blob/main/hw/block1/lec2/dining-philosophers/dining-philosophers/readme.md`

Not critical task. But your first programming assignment.
Three levels of difficulty (required time grows exponentially).

Dining philosophers problem

Homework, code

Task 2.3 `https://github.com/Suazars/parallel-programming/blob/main/hw/block1/lec2/dining-philosophers/dining-philosophers/readme.md`

Not critical task. But your first programming assignment.
Three levels of difficulty (required time grows exponentially).
Task is simple (could be checked on practical lesson)

Dining philosophers problem

Homework, code

Task 2.3 `https://github.com/Suazars/parallel-programming/blob/main/hw/block1/lec2/dining-philosophers/dining-philosophers/readme.md`

Not critical task. But your first programming assignment.
Three levels of difficulty (required time grows exponentially).
Task is simple (could be checked on practical lesson) , but let's imagine it is a hard one.

Dining philosophers problem

Homework, code

Task 2.3 `https://github.com/Svazars/parallel-programming/blob/main/hw/block1/lec2/dining-philosophers/dining-philosophers/readme.md`

Not critical task. But your first programming assignment.
Three levels of difficulty (required time grows exponentially).
Task is simple (could be checked on practical lesson) , but let's imagine it is a hard one.

- Fork the repository (do **not** commit/pull-request anything to my repo)
- Create branch `homework/task-2.3` (do not commit to main)
- Create **single** commit with your final solution (do not spam with `wip` and `fixup!`)
- Whenever you fix reviewer remarks, provide **single** commit with good commit message

All work-in-progress mess should be kept in separate development branch, not in public branch.
Fixing/refactoring stuff which was not asked by reviewer is a red flag.

Lecture plan

Signalling

Motivation

Thread `A` offloads some tasks to other threads and waits for completion.

Signalling

Motivation

Thread A offloads some tasks to other threads and waits for completion.
Naive solution: periodically inspect task status (polling).

```
public static void main(String... args) {
    Future<Object> task1 = startAsync( () -> { return jobA(); } );
    Future<Object> task2 = startAsync( () -> { return jobB(); } );
    boolean done1 = false; boolean done2 = false;
    while (true) {
        if (!done1 && task1.isReady()) { consume(task1); done1 = true; }
        if (!done2 && task2.isReady()) { consume(task2); done2 = true; }
        if (done1 && done2) break;
    }
}
```

Question time

Question: `while (true) { if(task.isReady()) { process(task);} ...`
Could you spot **performance** problem here?

Signalling

Motivation

Thread `A` offloads some tasks to other threads and waits for completion.
Naive solution: periodically inspect task status (polling).
Polling could be CPU intensive. How to backoff?

Signalling

Motivation

Thread `A` offloads some tasks to other threads and waits for completion.
Naive solution: periodically inspect task status (polling).
Polling could be CPU intensive. How to backoff?
Use `Thread.sleep`! How to select time period?

Signalling

Motivation

Thread `A` offloads some tasks to other threads and waits for completion.
Naive solution: periodically inspect task status (polling).
Polling could be CPU intensive. How to backoff?
Use `Thread.sleep`! How to select time period?
"Time-to-react" (latency) vs. "unproductive time" (throughput).

Signalling

Motivation

Thread `A` offloads some tasks to other threads and waits for completion.
Naive solution: periodically inspect task status (polling).
Polling could be CPU intensive. How to backoff?
Use `Thread.sleep`! How to select time period?
"Time-to-react" (latency) vs. "unproductive time" (throughput).
We need to

- subscribe on completion event
- notify subscribers (other threads) in efficient way

We need some support from OS!

Signalling

Idea

Blocking API:

- `wait` (a.k.a. `await`, `subscribe` ...)

Non-blocking API:

- `notify` (`notify_one`, `signal` ...)
- `notifyAll` (`signalAll`, `broadcast` ...)

Signalling

Idea

Blocking API:

- `wait` (a.k.a. `await`, `subscribe` ...)

Non-blocking API:

- `notify` (`notify_one`, `signal` ...)
- `notifyAll` (`signalAll`, `broadcast` ...)

What are we waiting for?

- Some task to be completed (execution flow control)
- Some data to be available (data flow control)

Must avoid race conditions and data races!

Question time

Question: Name concurrency primitive that helps to achieve thread-safety via blocking API

Lecture plan

Condition variable

API

Operations[5], creation[6]:

```
interface Condition {
    void await();
    void signal();
    void signalAll();
}
interface Lock {
    Condition newCondition();
}
```

[5] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/Condition.html
[6] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/locks/Lock.html#newCondition()

Condition variable

Usage rules

Illegal operations:

- `await` out of critical section
- `signal` out of critical section

Important details:

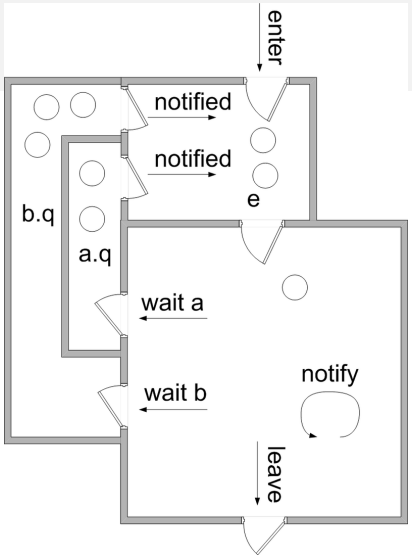
- `await` leaves mutex and starts listening for notification **atomically**
- `await` wake-ups (consumes notification) and **starts** mutex acquisition

Condition variable

Overview

One of possible high-level designs[7]:

- Mutex: EnterSet, ArriveSet
- Condition: WaitSet, NotifiedSet



[7] https://en.wikipedia.org/wiki/Monitor_(synchronization)#/media/File:Monitor_(synchronization)-Mesa.png

Lecture plan

Concurrency hazards

Lost signal (race condition)

Short description: `WaitSet` is empty due to race condition

```
threadA() {
    lock.lock();
    try {
        ready = true; condition.signal();
    } finally { lock.unlock(); }
}
threadB() {
    lock.lock();
    try {
        condition.await()
        if (ready) doComputation();
    } finally { lock.unlock(); }
}
```

Concurrency hazards

Lost signal (race condition)

Short description: `WaitSet` is empty due to race condition
Solution: never expect that your `signal` will reach already waiting receiver.
Sender thread:

- mark ready
- then `signal`

Receiver thread:

- check ready
- then `await`

Question time

Question: Changing two consecutive lines of code transforms program-with-bugs-in-signalling into default-concurrent-pattern. Name other concurrent primitive that has the same property.

Concurrency hazards

Lost signal (logic error)

Short description: `signal` delivered message to wrong waiter, `signalAll` needed

```
producer1() { lock.lock(); try { while (true) {
    if (data != null) condition.await();
    data = new Data1(); condition.signal();
}} finally { lock.unlock(); }
}
producer2() { lock.lock(); try { while (true) {
    if (data != null) condition.await();
    data = new Data2(); condition.signal();
}} finally { lock.unlock(); }
}
consumer() { lock.lock(); try { while (true) {
    if (data != null) { process(data); data = null; condition.signal(); }
    condition.await();
}} finally { lock.unlock(); }
}
```

Concurrency hazards

Lost signal (logic error)

Short description: `signal` delivered message to wrong waiter, `signalAll` needed

```
producer1() { loop-under-lock {
    if (data != null) condition.await();
    data = new Data1(); condition.signal();
}}
producer2() { loop-under-lock {
    if (data != null) condition.await();
    data = new Data2(); condition.signal();
}}
consumer() { loop-under-lock {
    if (data != null) { process(data); data = null; condition.signal(); }
    condition.await();
}}
```

Concurrency hazards

Lost signal (logic error)

Short description: `signal` delivered message to wrong waiter, `signalAll` needed
Solution:

- `signalAll` is always safer, yet could be less performant. **Not always correct**.
- better divide responsibility across threads (one-to-one communication via personal `Conditions`? other sync primitives?)

Concurrency hazards

Predicate invalidation

Short description: thread awaits for predicate and **does not recheck** it upon wakeup

```
threadA() { lock.lock(); try {
        while (true) {
            if (data == null) condition.await();
            System.out.println(data.hashCode())
        }
    } finally { lock.unlock(); }
}
threadB() { while (true) {
        lock.lock(); try {
            if (random.nextBoolean()) { data = null; }
            else { data = new Data(); condition.signal(); }
        } finally { lock.unlock(); }
    }
```

Concurrency hazards

Predicate invalidation

Short description: thread `awaits` for predicate and **does not recheck** it upon wakeup
Solution: **always** re-check `await` predicate. If it is invalidated, wait again.
Ensure you are not suffering from missing signal bug.

Concurrency hazards

Spurious wakeup

Short description: thread wake-ups for no reason[8][9]

```
main() {
    lock.lock();
    try {
        condition.await();
        System.out.println("Should not reach here");
    } finally { lock.unlock(); }
}
```

[8] https://linux.die.net/man/3/pthread_cond_wait

[9] https://devblogs.microsoft.com/oldnewthing/20180201-00/?p=97946

Concurrency hazards

Spurious wakeup

Short description: thread wake-ups for no reason[8][9]

```
main() {
    lock.lock();
    try {
        condition.await();
        System.out.println("Should not reach here");
    } finally { lock.unlock(); }
}
```

Solution:

- always wrap `await` into `while` loop with predicate re-check
- always expect that `await` is implemented as `unlock(); sleep(1); lock();`

[8] https://linux.die.net/man/3/pthread_cond_wait

[9] https://devblogs.microsoft.com/oldnewthing/20180201-00/?p=97946

Question time

Question: Concurrency is hard by itself! Why would library designers allow such a counter-intuitive behaviour as "spurious wakeup"?

Homework

Homework, mail

Task 2.4 Open `https://deadlockempire.github.io`, *pass "Condition Variables" level.*

Performance hazards

Fairness/Starvation

Short description: fair mutex + unfair `WaitSet` => unfair system

```
threadA() { loop-under-lock { System.out.println("A");
                              condition.signal(); condition.await();
}}
threadB() { loop-under-lock { System.out.println("B");
                              condition.signal(); condition.await();
}}
threadC() { loop-under-lock { System.out.println("C");
                              condition.signal(); condition.await();
}}
```

Performance hazards

Fairness/Starvation

Short description: fair mutex + unfair `WaitSet` => unfair system

```
threadA() { loop-under-lock { System.out.println("A");
                              condition.signal(); condition.await();
}}
threadB() { loop-under-lock { System.out.println("B");
                              condition.signal(); condition.await();
}}
threadC() { loop-under-lock { System.out.println("C");
                              condition.signal(); condition.await();
}}
```

A, B, A, B, A, B ...

Performance hazards

Fairness/Starvation

Short description: fair mutex + unfair `WaitSet` => unfair system

`WaitSet` admission policy have similar throughput/latency trade-offs as for `EnterSet/ArriveSet` in mutex

General note: if your algorithm have a choice "whom to wake-up", you have an admission policy design space

Condition variable

Conclusion

- Cross-thread message passing/coordination **could** be implemented via mutex + polling
- Signalling saves computation resources for something more useful than busy looping

Mutex + ConditionalVariable allow to build custom synchronization protocols. Be aware of

- Lost signal problem
- Predicate invalidation
- Spurious wakeup
- Fairness considerations

Summary

Some data races and race conditions could be avoided by mutual exclusion.
`Mutex` (a.k.a. `lock`, a.k.a. `critical section`) provides easy-to-use and simple API.
Do not forget to keep an eye on

- safety/correctness, liveness/progress guarantees, visibility/consistency, performance

Use suitable locking policies for your use-cases

- code locking, data locking, lock splitting

Remember to document locking policy to keep your program modular/reusable.
Do not forget to read documentation of thread-safe classes you use.
Condition variable allows to replace polling with OS-level signalling.
Signalling protocols must be aware of

- lost signals, predicate invalidation, spurious wakeups, fairness

Summary: homework

- find out if asynchronous exceptions are compatible with concurrent primitives (Task 2.1)
- `https://deadlockempire.github.io/` "Locks" (Task 2.2)
- solve Dining philosophers problem (Task 2.3)
- `https://deadlockempire.github.io/` "Condition Variables" (Task 2.4)