Lecture 2: mutual exclusion

thread safety, concurrent consistency, mutual exclusion, critical section, deadlock-freedom, starvation, fairness, check-then-act, reentrancy, admission policy, code locking, data locking, lock splitting, lock ordering, dining philosophers problem

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).
- Interleaving `N:1` is useful yet simplified model of concurrent execution.

Any concurrent task has

- parallel (independent) and sequential (dependent)

parts so max speedup is limited by Amdahl's law.

Threads have read/write access to shared memory which leads to

- non-determinism, race conditions, data races, visibility problems

Threads use blocking methods which leads to

- deadlocks, priority inversion

therefore we use wait-for graphs and observability API.

Lecture plan

1. Thread safety
2. Mutual exclusion
3. Mutex
   - Reentrancy
   - Admission policy
   - Visibility
4. Patterns
   - Code locking
   - Data locking
   - Lock splitting
5. Bug prevention
6. Summary

Lecture plan

1. Thread safety
2. Mutual exclusion
3. Mutex
    - Reentrancy
    - Admission policy
    - Visibility
4. Patterns
    - Code locking
    - Data locking
    - Lock splitting
5. Bug prevention
6. Summary

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”

Possible formalization: all operations could be treated as ”atomic” (non-divisible, transactional) and ordered on a single timeline.

Question time

Question:

- Concurrently consistent: all operations are ordered on a single timeline
- Interleaving model: all executed instructions are totally ordered

Does it mean any concurrent data structure is consistent if we use interleaving model?

Concurrent consistency

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

Possible formalization: all operations (methods of corresponding concurrent data structure class) 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"

Possible formalization: all operations (methods of corresponding concurrent data structure class) could be treated as "atomic" (non-divisible, transactional) and ordered on single timeline. There are other approaches, see consistency models in Lecture 7.

Toy problem: thread-safe counter

How to implement?

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?

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"

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

Mutual exclusion: no more than one thread enters code fragment (critical section)

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

Mutual exclusion

Naming

Mutual exclusion: no more than one thread enters code fragment (critical section)

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

```
interface Mutex {
    void enter();
    void exit();
}
```

Mutual exclusion

Naming

Mutual exclusion: no more than one thread enters code fragment (critical section)

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

Mutual exclusion

Usage pattern

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 pattern

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: Assume your program `locks` in one method and `unlocks` in other. Which constructs for control flow could "spoil" your locking invariants?

Exceptions are hard

https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.1/readme.markdown

Homework

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

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.

Exceptions are hard

https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.1/readme.markdown

Homework

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

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.
Warning: these tasks are hard. It usually takes 3-5 attempts to "defend" your answer.

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their "turn"

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their "turn"

What exactly should current thread do when mutex is already busy?

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their "turn"

What exactly should current thread do when mutex is already busy?

- Release current scheduling quantum

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their ”turn”

What exactly should current thread do when mutex is already busy?

- Release current scheduling quantum

When thread will be awaken?

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their ”turn”

What exactly should current thread do when mutex is already busy?

- Release current scheduling quantum

When thread will be awaken?

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

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their "turn"

What exactly should current thread do when mutex is already busy?

- Release current scheduling quantum

When thread will be awaken?

- randomly after some time period (if mutex is still busy, thread will be suspended again)
- when mutex is unlocked (but mutex may become busy before thread is "ready to go")

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their "turn"

What exactly should current thread do when mutex is already busy?

- Release current scheduling quantum

When thread will be awaken?

- randomly after some time period (if mutex is still busy, thread will be suspended again)
- when mutex is unlocked (but mutex may become busy before thread is "ready to go")
- ...

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their "turn"

What exactly should current thread do when mutex is already busy?

- Release current scheduling quantum

When thread will be awaken?

- randomly after some time period (if mutex is still busy, thread will be suspended again)
- when mutex is unlocked (but mutex may become busy before thread is "ready to go")
- ...

Actually, you do not know.

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

What should other contending threads do?

- Await their "turn"

What exactly should current thread do when mutex is already busy?

- Release current scheduling quantum

When thread will be awaken?

- randomly after some time period (if mutex is still busy, thread will be suspended again)
- when mutex is unlocked (but mutex may become busy before thread is "ready to go")
- ...

Actually, you do not know. Some time after other thread releases the mutex.

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.
- Mutex affects thread scheduling.

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.
- Mutex affects thread scheduling.

Acquisition order, system throughput, observed latency depend on

- Mutex implementation
- OS scheduling policy
- Non-determinism of CPU timings

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.
- Mutex affects thread scheduling.

Acquisition order, system throughput, observed latency depend on

- Mutex implementation
- OS scheduling policy
- Non-determinism of CPU timings

Mutual exclusion solved with flags

Thread A only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                 // A.1
  while (is_raised_B()) {   // A.2
    continue;                // A.3
  }
  critical_section();        // A.4
  lower_A();                 // A.5
}
```

```
public void useful_B() {
  raise_B();                 // B.1
  while (is_raised_A()) {   // B.2
    lower_B();               // B.3
    while (is_raised_A());// B.4
    raise_B();               // B.5
  }
  critical_section();        // B.6
  lower_B();                 // B.7
}
```

Mutual exclusion solved with flags

Thread A only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {  // A.2
    continue;                   // A.3
  }
  critical_section();          // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();          // B.6
  lower_B();                    // B.7
}
```

Mutual exclusion solved with flags

Thread A only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {       // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {       // B.2
    lower_B();                  // B.3
    while (is_raised_A());      // B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Mutual exclusion solved with flags

Thread A only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) { // A.2
    continue;                   // A.3
  }
  critical_section();         // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) { // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();         // B.6
  lower_B();                    // B.7
}
```

Mutual exclusion solved with flags

Thread A only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()        { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()        { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                 // A.1
  while (is_raised_B()) {    // A.2
    continue;                // A.3
  }
  critical_section();        // A.4
  lower_A();                 // A.5
}
```

```
public void useful_B() {
  raise_B();                 // B.1
  while (is_raised_A()) {    // B.2
    lower_B();               // B.3
    while (is_raised_A());   // B.4
    raise_B();               // B.5
  }
  critical_section();        // B.6
  lower_B();                 // B.7
}
```

Mutual exclusion solved with flags

Thread A only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) {      // A.2
    continue;                  // A.3
  }
  critical_section();          // A.4
  lower_A();                   // A.5
}
```

```
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) {      // B.2
    lower_B();                 // B.3
    while (is_raised_A());     // B.4
    raise_B();                 // B.5
  }
  critical_section();          // B.6
  lower_B();                   // B.7
}
```

Mutual exclusion solved with flags

Thread B only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {       // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {       // B.2
    lower_B();                  // B.3
    while (is_raised_A());      // B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Mutual exclusion solved with flags

Thread B only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Thread B only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) { // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) { // B.2
    lower_B();                // B.3
    while (is_raised_A());// B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Thread B only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) { // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) { // B.2
    lower_B();                // B.3
    while (is_raised_A());// B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Thread B only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()        { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()        { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Thread B only

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) { // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) { // B.2
    lower_B();                // B.3
    while (is_raised_A());// B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()        { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()        { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                     // A.1
  while (is_raised_B()) {        // A.2
    continue;                    // A.3
  }
  critical_section();            // A.4
  lower_A();                     // A.5
}
```

```
public void useful_B() {
  raise_B();                     // B.1
  while (is_raised_A()) {        // B.2
    lower_B();                   // B.3
    while (is_raised_A());       // B.4
    raise_B();                   // B.5
  }
  critical_section();            // B.6
  lower_B();                     // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {  // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                // B.3
    while (is_raised_A());// B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                      // A.1
  while (is_raised_B()) {  // A.2
    continue;                     // A.3
  }
  critical_section();         // A.4
  lower_A();                      // A.5
}
```

```
public void useful_B() {
  raise_B();                          // B.1
  while (is_raised_A()) {      // B.2
    lower_B();                        // B.3
    while (is_raised_A());// B.4
    raise_B();                        // B.5
  }
  critical_section();             // B.6
  lower_B();                          // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                      // A.1
  while (is_raised_B()) {  // A.2
    continue;                     // A.3
  }
  critical_section();           // A.4
  lower_A();                      // A.5
}
```

```
public void useful_B() {
  raise_B();                      // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                    // B.3
    while (is_raised_A());// B.4
    raise_B();                    // B.5
  }
  critical_section();           // B.6
  lower_B();                      // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()        { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()        { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                // A.1
  while (is_raised_B()) { // A.2
    continue;               // A.3
  }
  critical_section();       // A.4
  lower_A();                // A.5
}
```

```
public void useful_B() {
  raise_B();                // B.1
  while (is_raised_A()) { // B.2
    lower_B();              // B.3
    while (is_raised_A());// B.4
    raise_B();              // B.5
  }
  critical_section();       // B.6
  lower_B();                // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()           { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()           { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) {  // A.2
    continue;                  // A.3
  }
  critical_section();          // A.4
  lower_A();                   // A.5
}
```

```
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                 // B.3
    while (is_raised_A()); // B.4
    raise_B();                 // B.5
  }
  critical_section();          // B.6
  lower_B();                   // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) { // A.2
    continue;                 // A.3
  }
  critical_section();        // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) { // B.2
    lower_B();                // B.3
    while (is_raised_A());// B.4
    raise_B();                // B.5
  }
  critical_section();        // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()        { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()        { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {       // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {       // B.2
    lower_B();                  // B.3
    while (is_raised_A());      // B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                // A.1
  while (is_raised_B()) {  // A.2
    continue;               // A.3
  }
  critical_section();       // A.4
  lower_A();                // A.5
}
```

```
public void useful_B() {
  raise_B();                // B.1
  while (is_raised_A()) {  // B.2
    lower_B();              // B.3
    while (is_raised_A());// B.4
    raise_B();              // B.5
  }
  critical_section();       // B.6
  lower_B();                // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()       { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()       { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) {      // A.2
    continue;                  // A.3
  }
  critical_section();          // A.4
  lower_A();                   // A.5
}
```

```
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) {      // B.2
    lower_B();                 // B.3
    while (is_raised_A());     // B.4
    raise_B();                 // B.5
  }
  critical_section();          // B.6
  lower_B();                   // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()         { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()         { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                 // A.1
  while (is_raised_B()) {    // A.2
    continue;                // A.3
  }
  critical_section();        // A.4
  lower_A();                 // A.5
}
```

```
public void useful_B() {
  raise_B();                 // B.1
  while (is_raised_A()) {    // B.2
    lower_B();               // B.3
    while (is_raised_A());   // B.4
    raise_B();               // B.5
  }
  critical_section();        // B.6
  lower_B();                 // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {  // A.2
    continue;                   // A.3
  }
  critical_section();          // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();          // B.6
  lower_B();                    // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) {      // A.2
    continue;                  // A.3
  }
  critical_section();          // A.4
  lower_A();                   // A.5
}
```

```
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) {      // B.2
    lower_B();                 // B.3
    while (is_raised_A());     // B.4
    raise_B();                 // B.5
  }
  critical_section();          // B.6
  lower_B();                   // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()        { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()        { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {       // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {       // B.2
    lower_B();                  // B.3
    while (is_raised_A());      // B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                      // A.1
  while (is_raised_B()) {         // A.2
    continue;                     // A.3
  }
  critical_section();             // A.4
  lower_A();                      // A.5
}
```

```
public void useful_B() {
  raise_B();                      // B.1
  while (is_raised_A()) {         // B.2
    lower_B();                    // B.3
    while (is_raised_A());        // B.4
    raise_B();                    // B.5
  }
  critical_section();             // B.6
  lower_B();                      // B.7
}
```

Mutual exclusion solved with flags

Contention

```
static boolean A_flag = false, B_flag = false;
static Lock l = ...;
void raise_X()          { l.lock(); try { X_flag = true;  } finally { l.unlock();
void lower_X()          { l.lock(); try { X_flag = false; } finally { l.unlock();
boolean is_raised_X() { l.lock(); try { return X_flag;  } finally { l.unlock();
```

```
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) { // A.2
    continue;                  // A.3
  }
  critical_section();      // A.4
  lower_A();                   // A.5
}
```

```
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) { // B.2
    lower_B();                 // B.3
    while (is_raised_A());// B.4
    raise_B();                 // B.5
  }
  critical_section();      // B.6
  lower_B();                   // B.7
}
```

Mutual exclusion and deadlock-freedom

https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.2/readme.markdown

Homework

Task 2.2.a Prove that algorithm on previous slide guarantees mutual exclusion for 2 threads. Assume it is not and get a contradiction.

Homework

Task 2.2.b Prove that algorithm on previous slide is free of deadlocks.

Suggested reading: use companion slides for "Herlihy, Shavit: The Art of Multiprocessor Programming"[2], Lecture slides, Chapter 01, slides 43-72.

[2] https://booksite.elsevier.com/9780123973375

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.
- Mutex affects thread scheduling.
- At least one contending thread enters critical section. **Deadlock-freedom**.

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {       // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {       // B.2
    lower_B();                  // B.3
    while (is_raised_A());      // B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

```
Thread A acquisitions: 0, Thread B acquisitions: 0
```

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {  // A.2
    continue;                    // A.3
  }
  critical_section();       // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();       // B.6
  lower_B();                    // B.7
}
```

Thread A acquisitions: 0, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) { // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) { // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Thread A acquisitions: 0, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                      // A.1
  while (is_raised_B()) {  // A.2
    continue;                    // A.3
  }
  critical_section();       // A.4
  lower_A();                      // A.5
}
```

```
public void useful_B() {
  raise_B();                      // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                   // B.3
    while (is_raised_A());// B.4
    raise_B();                   // B.5
  }
  critical_section();       // B.6
  lower_B();                      // B.7
}
```

Thread A acquisitions: 0, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) { // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) { // B.2
    lower_B();                // B.3
    while (is_raised_A());// B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Thread A acquisitions: 0, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                     // A.1
  while (is_raised_B()) {  // A.2
    continue;                    // A.3
  }
  critical_section();            // A.4
  lower_A();                     // A.5
}
```

```
public void useful_B() {
  raise_B();                     // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                   // B.3
    while (is_raised_A());// B.4
    raise_B();                   // B.5
  }
  critical_section();            // B.6
  lower_B();                     // B.7
}
```

Thread A acquisitions: 0, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {       // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {       // B.2
    lower_B();                  // B.3
    while (is_raised_A());      // B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Thread A acquisitions: 0, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {  // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Thread A acquisitions: 0, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {  // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Thread A acquisitions: 0, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

```
Thread A acquisitions: 1, Thread B acquisitions: 0
```

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {  // A.2
    continue;                   // A.3
  }
  critical_section();          // A.4
  lower_A();                     // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();          // B.6
  lower_B();                     // B.7
}
```

Thread A acquisitions: 1, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) {  // A.2
    continue;                  // A.3
  }
  critical_section();          // A.4
  lower_A();                   // A.5
}
```

```
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                 // B.3
    while (is_raised_A());// B.4
    raise_B();                 // B.5
  }
  critical_section();          // B.6
  lower_B();                   // B.7
}
```

Thread A acquisitions: 1, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```java
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) {      // A.2
    continue;                  // A.3
  }
  critical_section();          // A.4
  lower_A();                   // A.5
}
```

```java
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) {      // B.2
    lower_B();                 // B.3
    while (is_raised_A());     // B.4
    raise_B();                 // B.5
  }
  critical_section();          // B.6
  lower_B();                   // B.7
}
```

Thread A acquisitions: 1, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) { // A.2
    continue;                  // A.3
  }
  critical_section();          // A.4
  lower_A();                   // A.5
}
```

```
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) { // B.2
    lower_B();                 // B.3
    while (is_raised_A());// B.4
    raise_B();                 // B.5
  }
  critical_section();          // B.6
  lower_B();                   // B.7
}
```

Thread A acquisitions: 1, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Thread A acquisitions: 1, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```java
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```java
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Thread A acquisitions: 1, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {  // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                // B.3
    while (is_raised_A());// B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Thread A acquisitions: 1, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {       // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {       // B.2
    lower_B();                  // B.3
    while (is_raised_A());      // B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Thread A acquisitions: 1, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                    // A.1
  while (is_raised_B()) {  // A.2
    continue;                   // A.3
  }
  critical_section();           // A.4
  lower_A();                    // A.5
}
```

```
public void useful_B() {
  raise_B();                    // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                  // B.3
    while (is_raised_A());// B.4
    raise_B();                  // B.5
  }
  critical_section();           // B.6
  lower_B();                    // B.7
}
```

Thread A acquisitions: 2, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Thread A acquisitions: 2, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                   // A.1
  while (is_raised_B()) {  // A.2
    continue;                  // A.3
  }
  critical_section();          // A.4
  lower_A();                   // A.5
}
```

```
public void useful_B() {
  raise_B();                   // B.1
  while (is_raised_A()) {  // B.2
    lower_B();                 // B.3
    while (is_raised_A());// B.4
    raise_B();                 // B.5
  }
  critical_section();          // B.6
  lower_B();                   // B.7
}
```

```
Thread A acquisitions: 2, Thread B acquisitions: 0
```

Mutual exclusion solved with flags

Starvation

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Thread A acquisitions: N, Thread B acquisitions: 0

Mutual exclusion solved with flags

Starvation

Starvation: user of concurrent object *could* be delayed for *arbitrary* time if there are other users of the same object.

```
public void useful_A() {
  raise_A();                 // A.1
  while (is_raised_B()) {  // A.2
    continue;                // A.3
  }
  critical_section();        // A.4
  lower_A();                 // A.5
}
```

```
public void useful_B() {
  raise_B();                 // B.1
  while (is_raised_A()) {  // B.2
    lower_B();               // B.3
    while (is_raised_A());// B.4
    raise_B();               // B.5
  }
  critical_section();        // B.6
  lower_B();                 // B.7
}
```

Mutual exclusion solved with flags

Starvation

Starvation: user of concurrent object *could* be delayed for *arbitrary* time if there are other users of the same object. **Unfair mutex**: current thread starves, whole system progresses.

```
public void useful_A() {
  raise_A();                  // A.1
  while (is_raised_B()) {     // A.2
    continue;                 // A.3
  }
  critical_section();         // A.4
  lower_A();                  // A.5
}
```

```
public void useful_B() {
  raise_B();                  // B.1
  while (is_raised_A()) {     // B.2
    lower_B();                // B.3
    while (is_raised_A());    // B.4
    raise_B();                // B.5
  }
  critical_section();         // B.6
  lower_B();                  // B.7
}
```

Question time

Question: What is the difference between **deadlock-freedom** and **starvation-freedom**?

Mutex basics

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

- Only one contending thread enters critical section. **Mutual exclusion**.
- Mutex affects thread scheduling.
- At least one contending thread enters critical section. **Deadlock-freedom**.
- Some contending threads could lag. **No starvation-freedom/fairness by default**.

Check-then-act

```
class ThreadSafeContainer {
    List a = new ArrayList<>();
    Lock l = ... ;
    public void add(Object o) {
        l.lock(); try { a.add(o); } finally { l.unlock(); }}
    public boolean contains(Object o) {
        l.lock(); try { return a.contains(o); } finally { l.unlock(); }}
    public void addIfAbsent(Object o) {
        if (!contains(o)) add(o); }
}
```

Check-then-act

```
class ThreadSafeContainer {
    List a = new ArrayList<>();
    Lock l = ... ;
    public void add(Object o) {
        l.lock(); try { a.add(o); } finally { l.unlock(); }}
    public boolean contains(Object o) {
        l.lock(); try { return a.contains(o); } finally { l.unlock(); }}
    public void addIfAbsent(Object o) {
        if (!contains(o)) add(o); }
}
```

No data race.

Check-then-act

```
class ThreadSafeContainer {
    List a = new ArrayList<>();
    Lock l = ... ;
    public void add(Object o) {
        l.lock(); try { a.add(o); } finally { l.unlock(); }}
    public boolean contains(Object o) {
        l.lock(); try { return a.contains(o); } finally { l.unlock(); }}
    public void addIfAbsent(Object o) {
        if (!contains(o)) add(o); }
}
```

No data race. Inconsistent behaviour due to race condition.

Check-then-act

```
class ThreadSafeContainer {
    List a = new ArrayList<>();
    Lock l = ... ;
    public void add(Object o) {
        l.lock(); try { a.add(o); } finally { l.unlock(); }}
    public boolean contains(Object o) {
        l.lock(); try { return a.contains(o); } finally { l.unlock(); }}
    public void addIfAbsent(Object o) {
        if (!contains(o)) add(o); }
}
```

No data race. Inconsistent behaviour due to race condition.

Remember: state of concurrent system may have changed since your last inspection

Mutex basics

Summary

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

- Only one contending thread enters critical section. **Mutual exclusion**.
- Mutex affects thread scheduling.
- At least one contending thread enters critical section. **Deadlock-freedom**.
- Some contending threads could lag. **No starvation-freedom/fairness by default**.
- Mutex helps to avoid data races, **does not** magically solves all race conditions.

Question time

Question: How to fix `addIfAbsent(e) = if (!contains(e)) add(e);`?

Question time

Question: How to fix `addIfAbsent(e) = if (!contains(e)) add(e);`?
It it OK to grab the lock in `addIfAbsent` and then again in `contains`?

Lecture plan

NonReentrantLock

```
NonReentrantLock = { boolean busy }
```

NonReentrantLock

`NonReentrantLock = { boolean busy }`

Single-threaded deadlock №1:

```
void foo() { l.lock(); l.lock(); }
```

NonReentrantLock

`NonReentrantLock = { boolean busy }`

Single-threaded deadlock №1:

```
void foo() { l.lock(); l.lock(); }
```

Single-threaded deadlock №2:

```
void add(Object o) {
    l.lock(); try { a.add(o); } finally { l.unlock(); }}
boolean contains(Object o) {
    l.lock(); try { return a.contains(o); } finally { l.unlock(); }}
void addIfAbsent(Object o) {
    l.lock(); try { if (!contains(o)) add(o); } finally { l.unlock(); }}
```

ReentrantLock

- ReentrantLock, ReentrantMutex
- RecursiveLock, RecursiveMutex

ReentrantLock

- ReentrantLock, ReentrantMutex
- RecursiveLock, RecursiveMutex

`ReentrantLock = { Owner owner, int count }`

Not every `unlock` actually releases ownership

ReentrantLock

- ReentrantLock, ReentrantMutex
- RecursiveLock, RecursiveMutex

`ReentrantLock = { 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()`

ReentrantLock

- ReentrantLock, ReentrantMutex
- RecursiveLock, RecursiveMutex

`ReentrantLock = { 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

`https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.3/readme.markdown`

Homework

Task 2.3 Implement reentrant mutex using non-reentrant one.

[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?

Toy problem: thread-safe counter

ReentrantLock-based[4] 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(); }
    }
}
```

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

Toy problem: thread-safe counter

ReentrantLock-based 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(); }
    }
}
```

Toy problem: thread-safe counter

ReentrantLock-based 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(); }
    }
}
```

- Data races?

Toy problem: thread-safe counter

ReentrantLock-based 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(); }
    }
}
```

- Data races? Race conditions?

Toy problem: thread-safe counter

ReentrantLock-based 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(); }
    }
}
```

- Data races? Race conditions? Concurrently consistent?

Toy problem: thread-safe counter

ReentrantLock-based 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(); }
    }
}
```

- Data races? Race conditions? Concurrently consistent?
- Deadlock-freedom?

Toy problem: thread-safe counter

ReentrantLock-based 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(); }
    }
}
```

- Data races? Race conditions? Concurrently consistent?
- Deadlock-freedom? Starvation-freedom?

Toy problem: thread-safe counter

ReentrantLock-based 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(); }
    }
}
```

- Data races? Race conditions? Concurrently consistent?
- Deadlock-freedom? Starvation-freedom? Scalability?

Homework: thread-safe counters

https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.4/readme.markdown

Homework

Task 2.4

- *Implement different kinds of thread-safe counters*
- *Analyze scalability using JMH*[a]
- *Find inconsistencies in "highly-distributed" counters implementations*

[a] https://github.com/openjdk/jmh

Lecture plan

Fairness

- **Mutual exclusion**: no more than one thread enters
- **Deadlock-freedom**: some thread eventually enters
- **Starvation-freedom**: this thread eventually enters

Fairness

- **Mutual exclusion**: no more than one thread enters
- **Deadlock-freedom**: some thread eventually enters
- **Starvation-freedom**: this thread eventually enters

If there are `N` contending threads, what is "distribution of enters"?

Fairness

- **Mutual exclusion**: no more than one thread enters
- **Deadlock-freedom**: some thread eventually enters
- **Starvation-freedom**: this thread eventually enters

If there are `N` contending threads, what is "distribution of enters"?

- Arbitrary

Fairness

- **Mutual exclusion**: no more than one thread enters
- **Deadlock-freedom**: some thread eventually enters
- **Starvation-freedom**: this thread eventually enters

If there are `N` contending threads, what is "distribution of enters"?

- Arbitrary
- Priority-based

Fairness

- **Mutual exclusion**: no more than one thread enters
- **Deadlock-freedom**: some thread eventually enters
- **Starvation-freedom**: this thread eventually enters

If there are `N` contending threads, what is "distribution of enters"?

- Arbitrary
- Priority-based
- Even

Fairness

- **Mutual exclusion**: no more than one thread enters
- **Deadlock-freedom**: some thread eventually enters
- **Starvation-freedom**: this thread eventually enters

If there are `N` contending threads, what is "distribution of enters"?

- Arbitrary
- Priority-based
- Even

We could empirically measure this: `https://en.wikipedia.org/wiki/Fairness_measure`

Fairness

- **Mutual exclusion**: no more than one thread enters
- **Deadlock-freedom**: some thread eventually enters
- **Starvation-freedom**: this thread eventually enters

If there are `N` contending threads, what is "distribution of enters"?

- Arbitrary
- Priority-based
- Even

We could empirically measure this: `https://en.wikipedia.org/wiki/Fairness_measure`

In our course we will use "all-or-nothing" approach: thread **could starve** or **never starves**.

Question time

Question: assume some thread starves. Does it mean that throughput of the whole program will be low?

Fairness and performance

LIFO

A work: 0

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 0

Fairness and performance
LIFO

A work: 0

B work 0

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B context switch 0

Fairness and performance

LIFO

A work: 0

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 0

Fairness and performance

LIFO

A work: 0

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 0

Fairness and performance

LIFO

A work: 0

B work 0

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B context switch 1

Fairness and performance

LIFO

A work: 1

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

LIFO

A work: 1

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

LIFO

A work: 1

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance
LIFO

A work: 2

B work 0

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B context switch 1

Fairness and performance

LIFO

A work: 2

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

LIFO

A work: 2

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

LIFO

A work: 3

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

LIFO

A work: 3

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

LIFO

A work: 3

```
public void threadA() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    unfairMutex.lock();
    doWork();
    unfairMutex.unlock();
  }
}
```

B context switch 1

- Unfair locks allow some threads to better utilize scheduling quantum. Better throughput.
- Unfair locks could cause starvation. Higher latency.

Fairness and performance
FIFO

A work: 0

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 0

Fairness and performance

FIFO

A work: 0

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 0

Fairness and performance

FIFO

A work: 0

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 0

Fairness and performance
FIFO

A work: 0

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 0

Fairness and performance

FIFO

A work: 0

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

FIFO

A work: 1

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

FIFO

A work: 1

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 0

B work 0

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

FIFO

A work: 1

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 1

B work 0

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

FIFO

A work: 1

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 1

B work 1

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

FIFO

A work: 1

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 1

B work 1

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 1

Fairness and performance

FIFO

A work: 1

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 1

B work 1

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 2

Fairness and performance

FIFO

A work: 2

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 1

B work 1

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 2

Fairness and performance

FIFO

A work: 2

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 1

B work 1

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 2

Fairness and performance

FIFO

A work: 2

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 2

B work 1

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 2

Fairness and performance

FIFO

A work: 2

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 2

B work 2

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 2

Fairness and performance

FIFO

A work: 2

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 2

B work 2

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 2

Fairness and performance

FIFO

A work: 2

```
public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

A context switch: 2

B work 2

```
public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}
```

B context switch 3

Fairness and performance

FIFO

```
A work: 2

public void threadA() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}

A context switch: 2
```

```
B work 2

public void threadB() {
  while (true) {
    fairMutex.lock();
    doWork();
    fairMutex.unlock();
  }
}

B context switch 3
```

- Fair locks could trigger many context switches. Lower utilization.
- Fair locks encourage better responsiveness. Lower latency.
- Fair locks provide more guarantees on lock ordering. Better predictability.

Admission policy

Design space

Depends on your goal:

Admission policy

Design space

Depends on your goal:

- Max throughput:

Admission policy

Design space

Depends on your goal:

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

Admission policy

Design space

Depends on your goal:

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

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:

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, Throughput, 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, Throughput, Deadlock probability

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

Admission policy

java.util.concurrent.ReentrantLock

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

Admission policy

java.util.concurrent.ReentrantLock

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

```
ReentrantLock(boolean fair)
```

Admission policy

java.util.concurrent.ReentrantLock

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

```
ReentrantLock(boolean fair)
```

When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order.

Admission policy

java.util.concurrent.ReentrantLock

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

```
ReentrantLock(boolean fair)
```

When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order.

Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Admission policy

java.util.concurrent.ReentrantLock

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

```
ReentrantLock(boolean fair)
```

When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order.

Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Note however, that fairness of locks does not guarantee fairness of thread scheduling ... Also note that the untimed tryLock() method does not honor the fairness setting.

Admission policy

java.util.concurrent.ReentrantLock

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

```
ReentrantLock(boolean fair)
```

When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order.

Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Note however, that fairness of locks does not guarantee fairness of thread scheduling ... Also note that the untimed tryLock() method does not honor the fairness setting.

Task 2.4, Easy level: https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.4/readme.markdown

Lecture plan

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[5]

[5] `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[5]

Synchronization points you know so far:

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

[5] 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

Mutual exclusion

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

Lecture plan

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?

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();
    }
}
```

Data locking

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

How to make program more scalable?

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

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();
    }
}
```

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();
    }
}
```

Assume we cannot afford to allocate millions of `ThreadSafeCounter` instances.

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();
    }
}
```

Assume we cannot afford to allocate millions of `ThreadSafeCounter` instances.
Divide-and-conquer using arbitrary granularity.

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();
    }}
```

Lock splitting

```java
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();
    }}
```

Task 2.4, Medium level: https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.4/readme.markdown

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

Wishful thinking

Minimize attack surface:

- Use single lock in the program

Deadlock prevention

Wishful thinking

Minimize attack surface:

- Use single lock in the program
- Use recursive locks

Deadlock prevention

Wishful thinking

Minimize attack surface:

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

Deadlock prevention

Wishful thinking

Minimize 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

Wishful thinking

Minimize 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

Practice

Minimize attack surface:

- Use recursive locks
- Use high-level abstractions and thread-safe classes

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 high-level abstractions and thread-safe classes
- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section (use ”leaf” locking)

Deadlock prevention

Minimizing attack surface:

- Use high-level abstractions and thread-safe classes
- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section (use "leaf" locking)

Specialized techniques:

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

Deadlock prevention

Minimizing attack surface:

- Use high-level abstractions and thread-safe classes
- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section (use "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 high-level abstractions and thread-safe classes
- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section (use ”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.

Deadlock prevention

Minimizing attack surface:

- Use high-level abstractions and thread-safe classes
- Use recursive locks
- Do not publish internal locks
- Avoid blocking calls inside critical section (use "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.

https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.5/readme.markdown

Homework

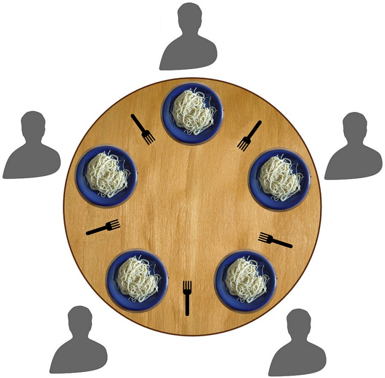
Task 2.5 Open `https://deadlockempire.github.io`, pass all "Locks" levels.

Dining philosophers problem

https://github.com/Svazars/parallel-programming/blob/main/hw/block1/2.6/readme.md

Task 2.6 Help them or they will starve to death!

- Three levels of difficulty
- Required time grows exponentially



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

Summary

- Mutual exclusion helps to achieve thread-safety
- `Mutex` (`lock`, `critical section`) provides easy-to-use and simple API. Key concepts:
  - deadlock-freedom, starvation-freedom, reentrancy, admission policy, fairness
- There are different ways to structure concurrent programs with locks:
  - code locking, data locking, lock splitting
- Mutex is a blocking primitive, so be aware of possible deadlocks:
  - recursive locks, encapsulation, enforcing lock acquisition order, avoiding external code invocation inside critical section
- Documenting locking policy is a key to modular and reliable concurrent software
- Do not forget to read documentation of thread-safe classes you use

Summary: homework

`Lecture 2, Tasks 2.*:` `https://github.com/Svazars/parallel-programming/blob/main/hw/block1`