Lecture 3: advanced synchronization primitives

monitor, latch, barrier, thundering herd, semaphore, read-write lock, thread pool, executor, producer-consumer, fork-join, load balancing

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l3.pdf

In previous episode

We study communication and coordination of different threads in pre-emptive multitasking OS, execution speed is not under our control.

We know the following tools:

- `Thread.start`, `Thread.join`, `ReentrantLock`, `Lock`, `Condition`

We are focusing on the following properties:

- Safety, Liveness, Performance

For any new concept we must understand

- Granularity of coordination (race conditions, level of parallelism)
- Blocking API (deadlocks)
- Admission policy (fairness, progress)
- Locking policy (visibility issues, data races)
- Signal send-receive conditions (lost signal, spurious wakeup)

We are focusing on high-level policies rather than particular synchronization mechanisms (until Lecture 11).

Lecture plan

1. Monitor
2. Broadcast
   - CountDownLatch
   - CyclicBarrier
   - Thundering herd problem
3. Group-level concurrency
   - Semaphore
   - ReadWriteLock
4. Thread pools
5. Workload generation
6. Load balancing
7. Summary

Disclaimer

Concurrent programming is evolving branch of Computer Science.

Disclaimer

Concurrent programming is evolving branch of Computer Science.
There are many designs for key synchronization primitives:

- "Monitor classification" by Buhr et al[1]
- "Semaphores in Plan 9" by Mullender and Cox[2]
- "Futexes are tricky" by Drepper[3]

[1] https://dl.acm.org/doi/10.1145/214037.214100
[2] https://swtch.com/semaphore.pdf
[3] https://www.semanticscholar.org/paper/Futexes-Are-Tricky-Drepper/86253463161f45ea013a501f89c8de36f1a09192

Disclaimer

Concurrent programming is evolving branch of Computer Science.
There are many designs for key synchronization primitives:

- "Monitor classification" by Buhr et al[1]
- "Semaphores in Plan 9" by Mullender and Cox[2]
- "Futexes are tricky" by Drepper[3]

We will stick to the Java programming language (built-in monitors) and `java.util.concurrent` package.

[1] https://dl.acm.org/doi/10.1145/214037.214100
[2] https://swtch.com/semaphore.pdf
[3] https://www.semanticscholar.org/paper/Futexes-Are-Tricky-Drepper/86253463161f45ea013a501f89c8de36f1a09192

Disclaimer

Concurrent programming is evolving branch of Computer Science.
There are many designs for key synchronization primitives:

- "Monitor classification" by Buhr et al[1]
- "Semaphores in Plan 9" by Mullender and Cox[2]
- "Futexes are tricky" by Drepper[3]

We will stick to the Java programming language (built-in monitors) and `java.util.concurrent` package.
It is enough to illustrate design space, trade-offs, common correctness problems.

[1] https://dl.acm.org/doi/10.1145/214037.214100
[2] https://swtch.com/semaphore.pdf
[3] https://www.semanticscholar.org/paper/Futexes-Are-Tricky-Drepper/86253463161f45ea013a501f89c8de36f1a09192

Disclaimer

Concurrent programming is evolving branch of Computer Science.
There are many designs for key synchronization primitives:

- "Monitor classification" by Buhr et al[1]
- "Semaphores in Plan 9" by Mullender and Cox[2]
- "Futexes are tricky" by Drepper[3]

We will stick to the Java programming language (built-in monitors) and `java.util.concurrent` package.
It is enough to illustrate design space, trade-offs, common correctness problems.
It is **not** enough to start concurrency-oriented low-level development in other languages (e.g. C++ stdlib/boost, Golang with plan9 style sema etc).

[1] https://dl.acm.org/doi/10.1145/214037.214100
[2] https://swtch.com/semaphore.pdf
[3] https://www.semanticscholar.org/paper/Futexes-Are-Tricky-Drepper/86253463161f45ea013a501f89c8de36f1a09192

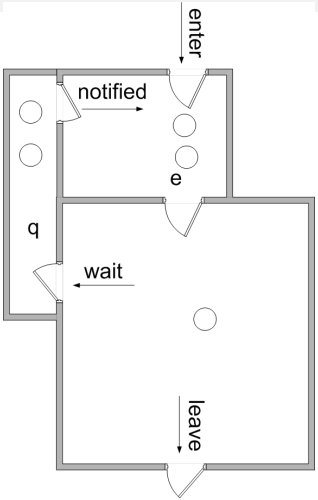
Lecture plan

Java monitor

Design

Effectively `ReentrantLock` + single `Condition`

- `MonitorEnter`
- `MonitorExit`
- `wait`
- `notify`
- `notifyAll`



Java monitor

For every object

Every Java object have associated built-in monitor.

```
Object x = new Object();
synchronized(x) {
  x.wait();
  x.notifyAll();
}
```

Special keyword: `synchronized` (no `.lock/unlock`, no try-finally, always structured)
Note: `Condition` and `ReentrantLock` are Java objects and they have built-in monitor.
Library-level synchronization and language-level synchronization are **independent**, do not mix `notify/signal` and `wait/await`.

Java monitor

For every method

Every Java method could be marked as `synchronized`.

```
public void foo() { doStuff(); }
public synchronized void bar() { doStuffUnderMutex(); }
```

It synchronizes on `this` (instance methods) or `Class<?>` (static methods):

```
public synchronized void fooSignature() { doStuff(); }
public void fooInternal() {
  synchronized(this) {
      doStuff();
  }
}
```

Question time

Question: Is it good or bad to mark methods as thread-safe (`synchronized` on publicly available signature level?)

Java monitor

For every method

Every Java method could be marked as `synchronized`.

```
public void foo() { doStuff(); }
public synchronized void bar() { doStuffunderMutex(); }
```

It synchronizes on `this` (instance methods) or `Class<?>` (static methods):

```
public synchronized void fooSignature() { doStuff(); }
public void fooInternal() {
  synchronized(this) {
      doStuff();
  }
}
```

Remember that inheritance or `synchronized(myAwesomeClassInstance)` may break your locking policy and cause a deadlock.

It is common pattern to use `private final Object lock` and do not expose it as public API.

Java monitor

Single-producer single-consumer bounded queue

```
static Dequeue<Object> buffer = ...
void producer(Object e) {
    synchronized(buffer) {
        while (buffer.size() > N) { buffer.wait(); }
        buffer.offer(e); buffer.notify();
    }
}
Object consumer() {
    synchronized(buffer) {
        while (buffer.isEmpty()) { buffer.wait(); }
        buffer.notify();
        return buffer.poll();
    }
}
```

Java monitor

Common pitfalls

Question time

Question: Name 3 key properties that you should understand for any concurrent object

Java monitor

Common pitfalls

Correctness(safety):

- Blocking `MonitorEnter`, `wait` – deadlock, locking order
- Reentrant
- Visibility/consistency – `synchronized` looks like "atomic transaction"
- Lost signal
- Predicate invalidation
- Spurious wakeup

Progress(liveness):

- Admission policy for `MonitorEnter`/`MonitorExit` is not specified
- Admission policy for `wait`/`notify` is not specified

Performance:

- Lock convoy
- Thundering herd

Java monitor

Common pitfalls

Correctness(safety):

- Blocking `MonitorEnter`, `wait` – deadlock, locking order
- Reentrant
- Visibility/consistency – `synchronized` looks like "atomic transaction"
- Lost signal
- Predicate invalidation
- Spurious wakeup

Progress(liveness):

- Admission policy for `MonitorEnter`/`MonitorExit` is not specified
- Admission policy for `wait`/`notify` is not specified

Performance:

- Lock convoy
- Thundering herd (what is it?)

Java monitor

Specific pitfalls

Unintended encapsulation breach:

- `synchronized` methods inheritance
- `synchronized` on object outside of library method

Solution: `private` guards

Java monitor

Specific pitfalls

Unintended encapsulation breach:

- `synchronized` methods inheritance
- `synchronized` on object outside of library method

Solution: `private` guards

Unintended mix of `java.util.concurrent` and built-in synchronization:

- `wait` vs `await`
- `synchronized` vs `lock/unlock`

Solution: static analysis (FindBugs/InteliJ IDEA)

Java monitor

Rules of thumb

Mutual exclusion

- If you need "just mutex" – consider using encapsulated object + `synchronized`
  - Save your time with try-finally
  - Other stuff (built-ins have better monitoring and optimizer-friendly) is debatable

Java monitor

Rules of thumb

Mutual exclusion

- If you need "just mutex" – consider using encapsulated object + `synchronized`
  - Save your time with try-finally
  - Other stuff (built-ins have better monitoring and optimizer-friendly) is debatable
- Use `Lock` if you need "advanced" properties:
  - Fairness (remember, scheduling is counter-intuitive)
  - Non-structured locking (remember, exceptions are hard)
  - Polling via `tryLock` (remember, race conditions are tricky)

Java monitor

Rules of thumb

Mutual exclusion

- If you need "just mutex" – consider using encapsulated object + `synchronized`
  - Save your time with try-finally
  - Other stuff (built-ins have better monitoring and optimizer-friendly) is debatable
- Use `Lock` if you need "advanced" properties:
  - Fairness (remember, scheduling is counter-intuitive)
  - Non-structured locking (remember, exceptions are hard)
  - Polling via `tryLock` (remember, race conditions are tricky)

Signalling

Java monitor

Rules of thumb

Mutual exclusion

- If you need "just mutex" – consider using encapsulated object + `synchronized`
  - Save your time with try-finally
  - Other stuff (built-ins have better monitoring and optimizer-friendly) is debatable
- Use `Lock` if you need "advanced" properties:
  - Fairness (remember, scheduling is counter-intuitive)
  - Non-structured locking (remember, exceptions are hard)
  - Polling via `tryLock` (remember, race conditions are tricky)

Signalling. The story is absolutely the same.
Built-in monitors are "friendlier" yet less flexible.

Java monitor

Must-know tool

Monitor is basic universal building block for concurrent primitives and protocols.

Java monitor

Must-know tool

Monitor is basic universal building block for concurrent primitives and protocols.

Homework, mail

*Implement every concurrent primitive presented in this course using built-in Java monitor(s). This task is **not** graded, but your questions are welcome!*

Java monitor

Must-know tool

Monitor is basic universal building block for concurrent primitives and protocols.

Homework, mail

*Implement every concurrent primitive presented in this course using built-in Java monitor(s). This task is **not** graded, but your questions are welcome!*

Monitor combines **mutual exclusion** and **signalling** allowing you to make infinite amount of

Java monitor

Must-know tool

Monitor is basic universal building block for concurrent primitives and protocols.

Homework, mail

*Implement every concurrent primitive presented in this course using built-in Java monitor(s). This task is **not** graded, but your questions are welcome!*

Monitor combines **mutual exclusion** and **signalling** allowing you to make infinite amount of

- concurrent design mistakes

Java monitor

Must-know tool

Monitor is basic universal building block for concurrent primitives and protocols.

Homework, mail

*Implement every concurrent primitive presented in this course using built-in Java monitor(s). This task is **not** graded, but your questions are welcome!*

Monitor combines **mutual exclusion** and **signalling** allowing you to make infinite amount of

- concurrent design mistakes
- bugs

Java monitor

Must-know tool

Monitor is basic universal building block for concurrent primitives and protocols.

Homework, mail

*Implement every concurrent primitive presented in this course using built-in Java monitor(s). This task is **not** graded, but your questions are welcome!*

Monitor combines **mutual exclusion** and **signalling** allowing you to make infinite amount of

- concurrent design mistakes
- bugs
- performance regressions

Java monitor

Must-know tool

Monitor is basic universal building block for concurrent primitives and protocols.

Homework, mail

*Implement every concurrent primitive presented in this course using built-in Java monitor(s). This task is **not** graded, but your questions are welcome!*

Monitor combines **mutual exclusion** and **signalling** allowing you to make infinite amount of

- concurrent design mistakes
- bugs
- performance regressions

Make yourself a favour – learn basics and then progress on advanced stuff.

Lecture plan

Signalling: broadcast

`Lock+Condition` and `Monitor` provide `notifyAll`.

Question time

Question: Does `notifyAll` or `signalAll` actually force all waiting threads to wake-up?

Signalling: broadcast

`Lock+Condition` and `Monitor` provide `notifyAll`.
Effectively threads are "released" one-by-one, since they are forced to enter critical section.

Signalling: broadcast

`Lock+Condition` and `Monitor` provide `notifyAll`.

Effectively threads are "released" one-by-one, since they are forced to enter critical section.

What should we do to perform "massive release"?

Signalling: broadcast

`Lock+Condition` and `Monitor` provide `notifyAll`.
Effectively threads are "released" one-by-one, since they are forced to enter critical section.
What should we do to perform "massive release"?
Use specialized tools!

Lecture plan

CountDownLatch

CountDownLatch

CountDownLatch[4]

```
CountDownLatch(int count)
void await()      // awaits counter == 0
void countDown()  // counter = max(counter - 1, 0)
long getCount()   // return counter
```

Common usage patterns:

- `CountDownLatch(1)` acts as a "gate":
  - **closed** (`counter == 1`)
  - **open** (`counter == 0`)
- `CountDownLatch(N)` acts as a "all threads ready" point:
  - **not-yet-ready** (`counter > 0`)
  - **ready** (`counter == 0`)

[4] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CountDownLatch.html

CountDownLatch

Example

```
static CountDownLatch ready = new CountDownLatch(N + 1);
static CountDownLatch start = new CountDownLatch(1);
main() {
    for (i = 0; i < N; i++) new Thread(new Worker(i)).start();
    ready.countDown(); ready.await();
    start.countDown();
}
worker() {
    prepare();
    ready.countDown();
    start.await();
}
```

Lecture plan

CyclicBarrier

`CountDownLatch` is single-use because counter monotonically decreases.
Some tasks require `N` iterations of some parallel activity before completion.

Question time

Question: Provide examples of parallelizable tasks that consist of `N` repeating steps.

CyclicBarrier

`CountDownLatch` is single-use because counter monotonically decreases.
Some tasks require `N` iterations of some parallel activity before completion.
Creating `ArrayList<CountDownLatch>` is cumbersome.

CyclicBarrier

CyclicBarrier[5]

```
CyclicBarrier(int parties, Runnable barrierAction)
int await()
int getNumberWaiting()
int getParties()
```

Common usage patterns: mostly-parallel tasks with sequential coordination

- Multiply matrices row-by-row (parallel), handle result (sequential)
- Do numerical simulation for `dt` (parallel), save intermediate result (sequential)
- Handle batch of requests (parallel), decide next batch size (sequential)

[5] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CyclicBarrier.html

CyclicBarrier

Example

They have example with matrix multiplication right in the javadoc[6]!

[6] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CyclicBarrier.html

Lecture plan

Thundering herd problem

```
static Deque<Object> data = ...
static Lock lock = ...
static CountDownLatch dataReady = new CountDownLatch(1);
main(workItems) {
    lock.lock();
    try { data.addAll(workItems); } finally { lock.unlock(); }
    dataReady.countDown();
}
worker_i() {
    dataReady.await();
    lock.lock();
    try { Object element = data.poll(); } finally { lock.unlock(); }
    process(element);
}
```

Thundering herd problem

```
static Deque<Object> data = ...
static CountDownLatch dataReady = new CountDownLatch(1);
main(workItems) {
    synchronized(data) {
        data.addAll(workItems);
    }
    dataReady.countDown();
}
worker_i() {
    dataReady.await();
    synchronized(data) {
        Object element = data.poll();
    }
    process(element);
}
```

Thundering herd problem

Scenario:

- many threads wake-up because of broadcast
- only few (e.g. single one) make progress

Thundering herd problem

Scenario:

- many threads wake-up because of broadcast
- only few (e.g. single one) make progress

Problem: inefficient utilization of scheduling quantums.

Thundering herd problem

Scenario:

- many threads wake-up because of broadcast
- only few (e.g. single one) make progress

Problem: inefficient utilization of scheduling quantums.
Really similar to lock convoy.

Thundering herd problem

Scenario:

- many threads wake-up because of broadcast
- only few (e.g. single one) make progress

Problem: inefficient utilization of scheduling quantums.
Really similar to lock convoy.
Solution:

- Use broadcasting with care
- `monitor.notifyAll` and `condition.signalAll` avoid this by moving signalled thread from one queue to another, waking up single "victim" on `MonitorExit/unlock`[7].

[7] This is called wait morphing

Thundering herd problem

Scenario:

- many threads wake-up because of broadcast
- only few (e.g. single one) make progress

Problem: inefficient utilization of scheduling quantums.
Really similar to lock convoy.
Solution:

- Use broadcasting with care
- `monitor.notifyAll` and `condition.signalAll` avoid this by moving signalled thread from one queue to another, waking up single "victim" on `MonitorExit/unlock`[7].

Warm reminder: built-in primitives are good and friendly[8]

[7] This is called wait morphing
[8] High-level overview: "Compact Java Monitors" by Dice, Kogan `https://arxiv.org/pdf/2102.04188`

Question time

Question: I am confused, could I observe thundering herd problem when using monitors or could not I?

Lecture plan

1. Monitor
2. Broadcast
   - CountDownLatch
   - CyclicBarrier
   - Thundering herd problem
3. Group-level concurrency
   - Semaphore
   - ReadWriteLock
4. Thread pools
5. Workload generation
6. Load balancing
7. Summary

Group level concurrency

Mutex divides threads into "owner" and "others".
Condition divides threads into "signalled" and "others".
`CountDownLatch`/`CyclicBarrier` help to coordinate start/stop for group of threads.
Could we do more?

Group level concurrency

Mutex divides threads into "owner" and "others".
Condition divides threads into "signalled" and "others".
`CountDownLatch`/`CyclicBarrier` help to coordinate start/stop for group of threads.
Could we do more?

- No more than N threads in critical section

Group level concurrency

Mutex divides threads into "owner" and "others".
Condition divides threads into "signalled" and "others".
`CountDownLatch`/`CyclicBarrier` help to coordinate start/stop for group of threads.
Could we do more?

- No more than N threads in critical section (semaphore)

Group level concurrency

Mutex divides threads into "owner" and "others".
Condition divides threads into "signalled" and "others".
`CountDownLatch`/`CyclicBarrier` help to coordinate start/stop for group of threads.
Could we do more?

- No more than N threads in critical section (semaphore)
- One exclusive writer and N independent readers

Group level concurrency

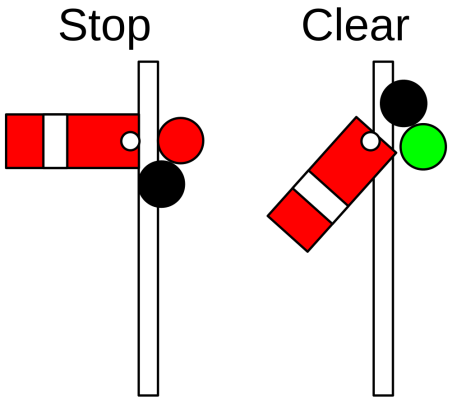
Mutex divides threads into "owner" and "others".
Condition divides threads into "signalled" and "others".
`CountDownLatch`/`CyclicBarrier` help to coordinate start/stop for group of threads.
Could we do more?

- No more than N threads in critical section (semaphore)
- One exclusive writer and N independent readers (read-write lock)

Lecture plan

Semaphore

https://en.wikipedia.org/wiki/Railway_semaphore_signal



Semaphore

Semaphore[9]

```
Semaphore(int permits)
void acquire() // acquire permit, blocks until available
void release() // release permit
```

Important notes:

- Perfect to
  - limit number of active threads (parallelism control)
  - bound shared resource usage (concurrent access control)
- No ownership – any thread could `release` permits
- *Binary semaphore* (`permits == 1`) is very similar to `NonReentrantLock`

[9] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Semaphore.html

Semaphore

Example

```
static Semaphore sema = new Semaphore(N);
static Set<Object> items = ...;
useItem() {
    sema.acquire();
    try {
        synchronized(items) {
            Object item = selectRandom(items);
            items.remove(item);
        }
        use(item);
        synchronized(items) { items.add(item); }
    } finally { sema.release(); }
}
```

Lecture plan

ReadWriteLock

ReadWriteLock[10]

```
Lock readLock();
Lock writeLock();
```

Read-mostly data structures:

- Concurrent readers, Exclusive writer

Design decisions:

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

Question time

Question: What are the main design decisions for any mutual exclusion primitive?

ReadWriteLock

ReadWriteLock[11]

```
Lock readLock()
Lock writeLock()
```

Read-mostly data structures:

- Concurrent readers, Exclusive writer

Design decisions:

- Reader preference vs. Writer preference
- Lock upgrade (promotion): readLock -> writeLock
- Lock downgrade: writeLock -> readLock
- Reentrancy for writers, reentrancy for readers

Advanced concurrency control: split threads into two types.

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

ReadWriteLock

Example

```
static Map<K, V> kv = ...
static ReadWriteLock rw = ..
V get(K k) {
    rw.readLock().lock();
    try {
        return kv.get(k);
    } finally { rw.readLock().unlock(); }
}
void put(K k, V v) {
    rw.writeLock().lock();
    try {
        kv.put(k, v);
    } finally { rw.writeLock().unlock(); }
}
```

Lecture plan

Motivation

- `Thread` is valuable and scarce OS resource.
  - Caching could help.
- Too many `Threads` may slow-down each other (oversubscription).
  - Soft and hard limits could help.
- Most of the computations do not care which thread is used.
  - Decoupling tasks from carriers could help.
- `Threads` may have purpose and context (name, priority, task-specific limits).
  - Should be properly reflected in observability API.

ThreadFactory

ThreadFactory[12]

```
Thread newThread(Runnable r)
```

Key concept: abstracting `Thread` creation, deletion and maintenance.
Smart words: decoupling `Thread` life cycle management from business logic.

[12] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ThreadFactory.html

Question time

Question: `ThreadFactory` that returns "previously used" `Thread` instance: what are pros and cons?

Executor

Executor[13]

```
void execute(Runnable command)
```

Key concept: abstracting task execution.
Smart words: decoupling business logic (**what** to do) from implementation (**how** and **where** to do) to simplify resource management (**who** will do).

[13] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executor.html

Question time

Question: `Executor` does not return `Future`. Describe valid `Executor` implementation that indeed does not require `Futures`.

ExecutorService

ExecutorService[14]

```
void execute(Runnable command) // implements Executor
<T> Future<T> submit(Callable<T> task)
<T> T invokeAny(Collection<Callable<T>> tasks) // returns any successful
void shutdown() // no new tasks
List<Runnable> shutdownNow() // try stop all executing, return all pending
boolean awaitTermination(long timeout, TimeUnit unit)
```

Structured concurrency:

- Asynchronous result (`Future`)
- Conditional execution (`invokeAny`)
- Life cycle management (`submit`, `shutdown`, `awaitTermination`)

[14] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html

Question time

Question: What if somebody `submit` task with unhandled exception to `ExecutorService`? What about other ways to "destroy" thread?

ScheduledExecutorService

ScheduledExecutorService[15]

```
// Value-returning one-shot task that becomes enabled after the given delay
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay,
                                TimeUnit unit);
// Periodic action that becomes enabled first after the given initial delay
// Executions will start after `initialDelay`, then `initialDelay + period` ...
ScheduledFuture<?>  scheduleAtFixedRate(Runnable command, long initialDelay,
                                        long period, TimeUnit unit);
```

Delay and periodic schedule control.
Hint: could be useful for testing, stress measurements and simulating "workload bursts".

[15] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ScheduledExecutorService.html

ThreadPools API: summary

- `ThreadFactory` – create threads
- `Executor` – execute tasks
- `ExecutorService`/`ScheduledExecutorService` – control task execution (order and resources)

ThreadPools API: summary

- `ThreadFactory` – create threads
- `Executor` – execute tasks
- `ExecutorService`/`ScheduledExecutorService` – control task execution (order and resources)

Bad news: you do not know which thread will execute your code.

- blocking methods, unexpected lock order, deadlocks, race conditions, data races

ThreadPools API: summary

- `ThreadFactory` – create threads
- `Executor` – execute tasks
- `ExecutorService`/`ScheduledExecutorService` – control task execution (order and resources)

Bad news: you do not know which thread will execute your code.

- blocking methods, unexpected lock order, deadlocks, race conditions, data races

Good news: you do not know which thread will execute your code.

- under-the-hood synchronization "just works", nothing to implement

ThreadPools API: implementations

Executors[16]

```
static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory tFactory)
static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory);
static ExecutorService newCachedThreadPool(ThreadFactory threadFactory);
```

[16] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executors.html

[17] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html

ThreadPools API: implementations

Executors[16]

```
static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory tFactory)
static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory);
static ExecutorService newCachedThreadPool(ThreadFactory threadFactory);
```

Remember:

- Fixed-size thread pools may cause resource deadlock
- Single-thread executor not necessarily use the same thread
- Caching can cause memory leaks or unintentionally reuse resources (e.g. native-lib-specific resources)

Full overview and arbitrary customization available[17]

[16] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executors.html

[17] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html

ThreadPools API: now you know it

From now on, you **must** avoid using `new Thread()` in your solutions.

- Use existing `ExecutorServices`
- Create custom `ThreadPoolExecutors`
- Encapsulate thread configuration in `ThreadFactory`

ThreadPools API: now you know it

From now on, you **must** avoid using `new Thread()` in your solutions.

- Use existing `ExecutorServices`
- Create custom `ThreadPoolExecutors`
- Encapsulate thread configuration in `ThreadFactory`

Homework, code

Task 3.1 Rewrite `DiningTable.java`[a] using `Executors` to simplify life cycle management (`boolean started`, `boolean shouldStop`).

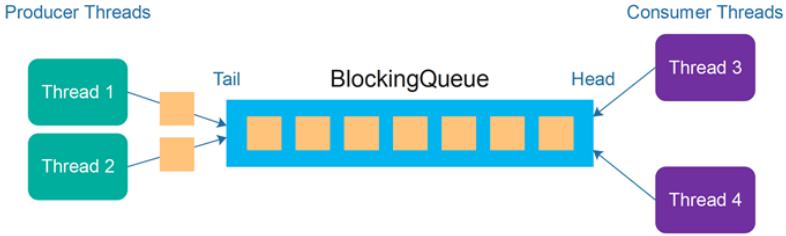
[a] https://github.com/Svazars/parallel-programming/blob/main/hw/block1/lec2/dining-philosophers/dining-philosophers/src/main/java/org/nsu/syspro/parprog/base/DiningTable.java

This task is **independent** from dining philosophers problem, you should refactor existing concurrent code, not invent new one.

Lecture plan

Producer-consumer

- N threads generate work items
- M threads process work items



BlockingQueue[18]

[18] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/BlockingQueue.html

Question time

Question: How to ensure that producers and consumers are "balanced"?

Producer-consumer

- X worker threads

Depending on heuristic, every thread either

- generate work item
- process existing work item

Producer-consumer

- X worker threads

Depending on heuristic, every thread either

- generate work item
- process existing work item

Common problems:

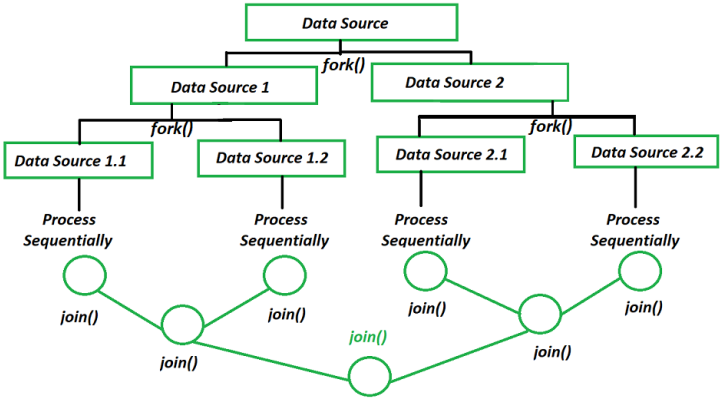
- Fixed-size buffer deadlock:
  - buffer capacity is X
  - every thread produces no more than 2 elements per iteration
- Empty-buffer deadlock:
  - buffer is empty
  - every thread heuristically decided to be consumer

Question time

Question: How to ensure that producers and consumers reach "consistent progress"?

Fork-join

Fork-join model[19]



[19] https://en.wikipedia.org/wiki/Fork-join_model

Question time

Question: How to decide when to stop `fork`?

Reactive Streams

`https://openjdk.org/jeps/266`

> *The main goal of Reactive Streams is to govern the exchange of stream data across an asynchronous boundary—think passing elements on to another thread or thread-pool — while ensuring that the receiving side is not forced to buffer arbitrary amounts of data. In other words, back pressure is an integral part of this model in order to allow the queues which mediate between threads to be bounded.*

`Flow.Publisher<T>`, `Flow.Subscriber<T>`[20]

[20] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Flow.html

Workload generation: takeaways

- Producer-consumer: "push" based concurrent system
- Fork-join: divide-and-conquer, "helping hand" approach
- Reactive: "propagate backpressure" design

More features – harder to implement, monitor and control.

Lecture plan

Load balancing

ThreadPool-based design uses classic timesharing approach:

- there are `X` tasks (functions)
- `Y` "virtual" executors (threads) which are mapped to
- `Z` "real" executors (cores)

Load balancing

ThreadPool-based design uses classic timesharing approach:

- there are `X` tasks (functions)
- `Y` "virtual" executors (threads) which are mapped to
- `Z` "real" executors (cores)

There are well-known corner cases:

- `X` < `Y`: insufficient concurrency (coarse-grained problem)
- `X` » `Y`: excessive coordination overheads (too fine-grained approach or too aggressive sub-task creation)
- `Y` < `Z`: undersubscription (underutilization)
- `Y` » `Z`: oversubscription

Load balancing

ThreadPool-based design uses classic timesharing approach:

- there are `X` tasks (functions)
- `Y` "virtual" executors (threads) which are mapped to
- `Z` "real" executors (cores)

There are well-known corner cases:

- `X` < `Y`: insufficient concurrency (coarse-grained problem)
- `X` ≫ `Y`: excessive coordination overheads (too fine-grained approach or too aggressive sub-task creation)
- `Y` < `Z`: undersubscription (underutilization)
- `Y` ≫ `Z`: oversubscription

In any scenario, `Y` threads have to coordinate task execution:

- Eventual progress: every task will be executed
- Correctness: every task is executed exactly once
- Performance: keep coordination overheads minimal

Load balancing: global queue

Work arbitrage

Load balancing design:

- Single shared thread-safe task queue
- Every available `Worker` gets tasks one-by-one

Counter-examples:

- Many short tasks
- `Workers` of varying speeds (energy-efficient and usual cores)

Load balancing: global queue + batching

Work arbitrage

Load balancing design:

- Single shared thread-safe task queue
- Every available `Worker` gets tasks in batches (N per request)

Counter-examples:

- Mix of short and long tasks
- `Workers` of varying speeds (energy-efficient and usual cores)

Load balancing: global queue + batching

Work dealing

Load balancing design:

- Single shared thread-safe task queue
- Every available `Worker` gets tasks in batches (N per request)
- Overloaded `Worker` moves fraction of tasks to global queue

Counter-examples:

- Few long tasks
- Overloaded thread spends CPU for auxiliary coordination

Load balancing: work-stealing

Load balancing design:

- Several shared thread-safe task queues (e.g. 1 per `Worker`)
- `Worker` get tasks in batches (N per request) from some queue ("steals" from neighbour)

Not that easy to implement and fine-tune[21]. Nontrivial termination protocol.
ForkJoinPool[22]

[21] https://shipilev.net/#forkjoin

[22] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ForkJoinPool.html

Summary

Mutual exclusion and signalling combined into single universal primitive: monitor
Java language has special support for built-in monitors: `synchronized` keyword
Bulk thread notification:

- `CountDownLatch`, `CyclicBarrier`
- Thundering herd problem

Tools for fine-grained group-level thread control:

- `Semaphore`, `ReadWritelock`

Universal API for thread/task management:

- `ThreadFactory`, `Executor`, `ExecutorService`

Different ways to generate workload concurrently:

- Producer-consumer, Fork-join, Reactive Streams

Designs for efficient concurrent load balancing:

- Work arbitrage, Work dealing, Work stealing

Summary: homework

Homework, mail

*Implement every concurrent primitive presented in this course using built-in Java monitor(s). This task is **not** graded, but your questions are welcome!*

Homework, code

Task 3.1 Rewrite `DiningTable.java`[a] using `Executors` to simplify life cycle management (`boolean started`, `boolean shouldStop`).

[a] https://github.com/Svazars/parallel-programming/blob/main/hw/block1/lec2/dining-philosophers/dining-philosophers/src/main/java/org/nsu/syspro/parprog/base/DiningTable.java