1. Concurrency Motivation

Multiprocessors

The **Moore's Law** is an observation by Gordon Moore which states that the number of transistors on integrated circuits doubles approximately every two years.
Single processors have fast improvement over time and the same user code runs faster in improved processors, but they could reach technological limits.
The solution is to put more processors into the same machine (**multiprocessors**), but one problem still persists: the user code conceived for single processors does not run faster on more processors.
The only possible solution is to **parallelize** the user code, making it runnable on more processors.

Amdahl's Law and Speedup

The **Amdahl's law** is a formula from Gene Amdahl which gives the theoretical speedup in latency of the execution of a task that can be expected for a system whose resources are improved.

- **Execution time** $T_n = T_s + (T_p / n)$
  - T_n → execution time on *n* processors
  - T_s → sequential component (cannot be parallelized)
  - T_p → parallel component (can benefit from multiprocessors)
- **Speedup** $S_n = T_1 / T_n$
  - compares the latency for solving the identical computational problem on one hardware unit versus on n hardware units
  - (T_1 / T_n) is limited by T_s, since if $n \to \infty$, then $T_n \to T_s$
- **Parallel efficiency** $E_n = S_n / n$
  - a program that scales linearly ($S_n = n$) has a parallel efficiency of 1

Example:

➢ Assume 30% of the runtime of a program is not parallelizable

➢ What would be the speedup when the program is executed on 2 processors, wrt. the program execution on a single processor?

➢ Solution:

$T_1 / T_2 = T_1 / (0.3 * T_1 + (0.7/2) * T_1) =$

$T_1 / (T_1 * (0.3 + (0.7/2)))=$

$1 / (0.3 + (0.7/2)) =$

$1 / 0.65 \approx 1.54$

2. Processes and Threads

Processes

A **process** is the abstraction of a running program.
The **process list** contains the information about all processes.
An entry in the process list is called **Process Control Block (PCB)** and contains:

- Process identification data
- Process state data
- Process control data

The **address space** is the set of ranges of virtual addresses that the OS makes available to a process, which has its own address space where to read and write:

- **virtual addresses**: logical addresses provided by the OS
- **physical addresses**: hardware addresses of physical memory

The **Memory Management Unit (MMU)** performs the **address translation** (virtual → physical) and stores the translations in the **Translation Lookaside Buffer (TLB)**.
The **virtual memory** is the combination of main memory and secondary memory.
Swapping is a reclamation method wherein memory contents not currently in use are swapped to secondary memory to increase the main memory available.

Concurrency and Parallelism

In **sequential execution**, each statement in the source code is executed in sequential order.
A system is said to be **concurrent** if it can support two or more operations in progress at the same time.
In **concurrent execution**, two or more threads execute statements in the source code at some time.
In **parallel execution**, the system supports two or more operations executing simultaneously with multiple computing units, so that two or more threads can execute code at the same time.
“Parallel" is a subset of “concurrent”: in the absence of multiple computing units, concurrent applications cannot run in parallel.
Sequential execution is faster than parallel execution only when the dataset size is small or when each iteration has to perform simple operations which are not expensive in terms of processors time.

Synchronous and Asynchronous Execution

The **synchronous execution** waits for an operation to complete **blocking** processes.
The **asynchronous execution** permits other operations to continue before the current one has finished, **non-blocking** processes.
With **multitasking**, systems can support multiple concurrent processes.
OSs ensure **concurrency transparency**, meaning that independent processes cannot maliciously affect the correctness of other processes.
It comes at a high price since the **context switching**, which consists in switching CPU between processes, may involve saving and loading of PCB state, modifying registers of the MMU, flushing the TLB, and swapping processes between main and secondary memory.

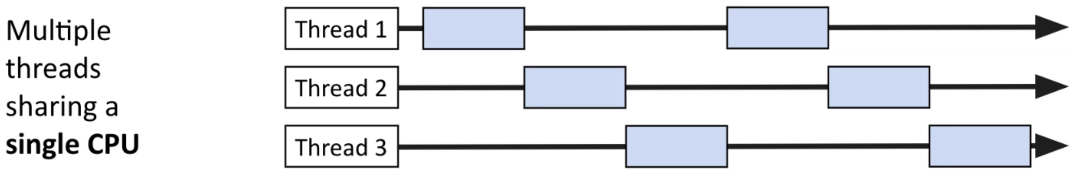
Process Communication and Execution

Related processes that cooperate for some job need to communicate and to synchronize: the **Inter Process Communication (IPC)** is a mechanism that allows processes to communicate via shared memory and message passing.
Execution:

- NEW state → the process has just been initialized
- READY state → the process is ready to start
- RUNNING state → the process is using part of the CPU
  - READY state → the process is ready to start another task
  - BLOCKED state → the process waits for some operations
- TERMINATED state → the process execution is finished

Threads of Execution

A **thread of execution** is a sequence of programmed instructions executed sequentially and managed by an OS **scheduler**.
Multiple threads can exist within one process, allowing concurrent activities within a single process but sharing a single address space.
Thread context switching is the change from one thread to another of the same process, suspending the current one and restoring the state of the thread being switched to.
The threads number can be specified, but there's no way to control the scheduler (OS controls it maximising the usage of the resources).



Why should threads be used?

- **Performance**: programs execute faster due to better resource utilization
- **Responsiveness**: increased access to device, some threads could free the others
- **Clean design**: well-organised systems

Java Threads

The **Java Virtual Machine (JVM)** allows an application to have multiple threads running concurrently:

- **user threads**: the JVM waits for user to complete
- **daemon threads**: do not prevent the JVM from shutting down when the application terminates

The first thread in execution calls `main()` and the program ends when all non-daemon threads have terminated.
There are two ways to create a new Java thread:

- implementing **`java.lang.Runnable`**interface
- extending **`java.lang.Thread`** class

3. Thread Safety

Risks of Threads

Safety Hazard

Threads share the same memory address space and can run concurrently, so they can access variables that other threads might be using.
This makes inter-thread communication effective, but introduces the need of coordinating the access to shared variables, otherwise the behaviour of a multithreaded program would be unpredictable.
Java provides **synchronization** mechanism to coordinate the access to shared variables.

Liveness Hazard

Thread safety means "nothing bad ever happens", while liveness complementary means "something good eventually happens".
A **liveness failure** occurs when an activity gets into a state such that it is permanently unable to make progress.
Endless loops are an example of liveness hazards in sequential programs.

Performance Hazards

Liveness means "eventually progress will be made", but parallel programming is aimed at **performance**.
Multithread applications can introduce performance overheads due to frequent context switches, task scheduling, inter-thread communication, and loss of locality.

Thread Safety

Thread safe code is about managing access to state, particularly to shared mutable state.
An object's **state** is the data stored in its state variables:

- **shared**: a variable can be accessed by multiple threads concurrently
- **mutable**: the state of a variable may change during its lifetime

Whether an object needs to be thread safe depends on whether it will be accessed from multiple threads and whether it is mutable:

- an object not shared → trivially thread-safe
- an immutable object → trivially thread-safe
- a **stateless object** → trivially thread-safe
- a shared mutable object → prone to safety hazards

Atomicity

An operation is **atomic** if no other thread can see it partly executed and can be used without fear of thread interference since they cannot be interleaved.
Memory reordering is possible from hardware point of view: modern multiprocessors do not enforce global ordering of all instructions for performance reasons: most processors have pipelined architecture and can execute multiple instructions simultaneously.

execute multiple instructions simultaneously.
Each processor has a local cache and thus load and store to shared memory can become visible to other processors at different times.
Memory reordering is possible from software point of view: modern compilers do not guarantee that a global ordering of memory accesses is provided since some memory accesses may be optimized away completely.
A memory model provides minimal guarantees for the effects of memory operations such as leaving open optimisation possibilities for hardware and compiler and including guidelines for writing correct multithreaded programs.
A key to avoid memory consistency errors in Java is to understand the **happens-before relationship**, which simply guarantees that memory writes by one specific statement are visible by another specific statement.
When a statement invokes **`Thread.start()`**, every statement that has a happens-before relationship with that statement also has a happen-before relationship with every statement executed by the thread.
When a thread terminates and causes a **`Thread.join()`** in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join.
Visibility ensures that the effects of one thread can be seen by another thread.
There is no guarantee that a reading thread will see a value written by another thread on a timely basis, or even at all.
Without proper synchronization, the compiler, processor, and runtime can change the order in which operations appear to execute: attempts to reason about the order in which memory actions "must" happen will almost certainly be incorrect.

Races

A **race condition** occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime.
A **data race** arises when synchronization is not used to coordinate all access to a shared non-final field:

- when one thread might read an object field at the same moment that another thread writes the same field
- when one thread might write an object field at the same moment that another thread also writes the same field

Consider the following 2 concurrent threads:

- Thread 1:
  - A `if (x == 0)`
  - B `x = x + 1;`
- Thread 2:
  - C `if (x == 0)`
  - D `x = 10;`

What are the possible values of x after execution, assuming x = 0 initially:

- x = 1 → ABCD
- x = 10 → CDAB
- x = 11 → ACDB (not possible in sequential execution)

Can these statements cause race conditions?

- `if (x == 0)` x = 10 yes, as seen before
- `x = x + 1` yes, because first x value is loaded, then incremented, then stored in x
- `x = 100` no, writing a 32-bit variable is considered atomic

Whether an instance is null depends on unpredictable timing, so this is an example of **check-then-act** race condition.

Thread-safe classes encapsulate any needed synchronization so that clients don't need to provide their own:

- **ThreadLocal** enables associating a per-thread value with a value-holding object
- **thread confinement** makes an object accessible only by a single thread

After a thread terminates, all its thread-local instances are subject to garbage collection.

Synchronization

A **critical section** is a portion of a program that uses a shared resource.

Properties of critical sections:

- **Mutual exclusion** (= safety): only one thread can be inside the CS at a time
- **Progress** (= liveness): if some thread T is not in the CS, then T cannot prevent some other thread S from entering the CS
- **Bounded waiting** (= no starvation): if some thread T is waiting on the CS, then T will eventually enter the CS
- **No assumption on timing**: requirements must be met with any number of CPUs and must not rely on the underlying scheduler

Thread synchronization ensures that two or more concurrent threads do not simultaneously execute a critical section.

Mutual execution can be guaranteed by placing **locks** around critical sections:

- **L = NEW** creates a new lock **L** that is initially "not held"
- **ACQUIRE(L)** allows a thread to take ownership of lock **L**:
  - if lock is not held, makes lock “held” by current thread
  - if lock is held, wait until lock is “not held”
  - call **ACQUIRE** before entering the critical section
- **RELEAS(L)** releases lock **L** allowing another thread to take ownership of it:
  - makes a lock “not held”;
  - if ≥ 1 threads are waiting on **ACQUIRE**, only 1 of them will acquire the lock;
  - call **RELEASE** after leaving the critical section

4. Semaphores and Monitors

Semaphores

A **semaphore** is a variable or abstract data type that controls access to a common resource by multiple processes.
Every semaphore maintains a number of permits that restricts the number of threads accessing a shared resource.
Binary semaphores have only one permit and are used to implement locks:**`permit available = unlocked`**.
The entity holding the sole permit can access the critical section.
Semaphores implement three operations:

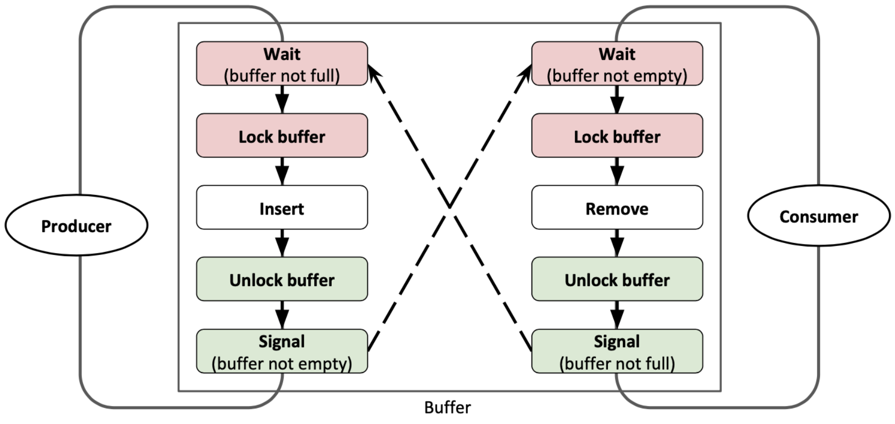
- **Initialization**: initializes a non-negative value
- **Wait**: decrements semaphore value
- **Signal**: increments the semaphore value

If the initial value is 1, the semaphore works like a lock, while if the new value is not positive, one previously blocked thread is unblocked.
The semaphore can put waiting threads in a queue, removing them in order.

Bounded Buffers

The **Bounded Buffer** problem is an example of a multi-process synchronization problem:

- assume a shared buffer of limited size
- producer puts data into a queue if space is free
- consumer removes data if queue is not empty

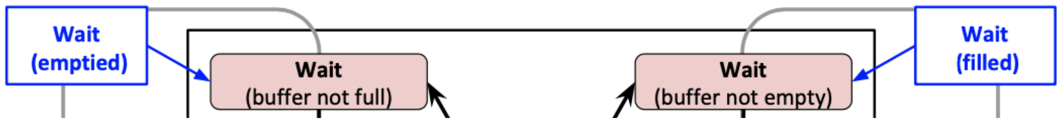


When a thread releases a semaphore, a happens-before relationship is established between that action and any subsequent calls to **`Semaphore.acquire`** on the same semaphore.
Even simple algorithms often require more than one semaphore; this may increase the complexity of some algorithms.
The use of semaphores is spread throughout the program code: the programmer has to keep track of all calls to wait and to signal the semaphore.

Monitors

A **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait for a certain condition to become true.
The goal is to decrease the probability of making mistakes while synchronizing by letting only one thread inside a monitor at a time.
Sometimes we would rather wait until something happens, not only until access is granted (be notified when something has been put inside).
Therefore, monitors can have condition variables and threads can wait for a condition.
There are two operations on a condition variable:

- **`x.wait()`**: a thread that invokes the operation is suspended, another thread can access the monitor
- **`x.signal()`**: resumes one of the threads that invoked `x.wait()`



5. Implicit Lock

Intrinsic Locks

Synchronization in Java is built around an internal entity known as the **intrinsic lock**, which play a role in both enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.
Every object has an intrinsic lock associated with it, but locking an array of objects locks the array, not the objects inside it.
To avoid race conditions, a thread that needs exclusive access to an object's fields has to acquire the object's intrinsic lock before accessing it and release it once the object is no more needed.
A thread is said to own the intrinsic lock between the time it has acquired and released it.
As long as a thread owns an intrinsic lock, no other thread can acquire the same lock, since they block when attempt to acquire it.
When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Synchronized Blocks

Block synchronization takes an argument indicating which object to lock on.
It prevents multiple threads from being inside blocks synchronized on the same object, protecting part of a method.
Entire methods can be declared synchronized, which is the same as locking the object which the method is declared on.
If some methods of an object are synchronized and others are not, only the synchronized methods are **mutually exclusive**.
Non-synchronized methods can be executed in parallel to synchronized ones.

Acquiring and Releasing Locks

A **reentrant lock** is a lock that can be acquired multiple times by the holding thread without blocking.
It “remembers” the thread that currently holds it and a count, which is set to 0 when the lock goes from not-held to held.
If the current holder calls acquire it increments the count without blocking.
On release:

- count > 0 ➝ the count is decremented
- count == 0 ➝ the lock becomes not-held

Synchronization

The **synchronized** keyword is not considered part of a method's signature: the modifier is not automatically inherited when subclasses override superclass methods.
However, a non-static inner class can lock its containing class instance via code blocks

However, a non-static inner class can lock its containing class instance via code blocks using: `synchronized(OuterClass.this){…}`.
Static synchronization means declaring a static method synchronized and employs the lock possessed by the Class object associated with the class in which static methods are declared.
This lock is different from the lock for any instance of the class, since the class lock can also be accessed inside instance methods: `synchronized(C.class){…}`.

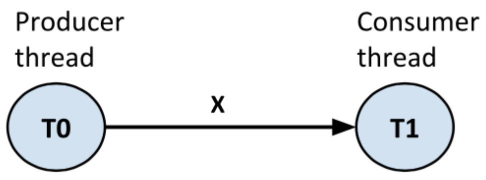
- Always lock during updates to object attributes:
  - `synchronized(point){`
  - `    point.x = 5;`
  - `    point.y = 7; }`
- Always lock while accessing possibly updated object attributes:
  - `synchronized(point){`
  - `    if (point.x > 0){…} }`
- You do not need to synchronize stateless parts of methods:
  - `synchronized void service(){`
  - `    state = …;`
  - `    operation(); }`
  - `void service(){`
  - `     synchronized (this){`
  - `        state = …; }`
  - `    operation(); }`
- Avoid locking when invoking methods on other objects:
  - `synchronized void service(){`
  - `    state = …;`
  - `    other.operation(); }`
  - `void service(){`
  - `    synchronized (this){`
  - `        state = ...; }`
  - `    other.operation(); }`

Assignments

Assigning variables is atomic: in the JVM, a single write to a non-volatile long or double value may be treated as two separate writes, one to each 32-bit half.
Threads are allowed to hold the variables values in local memory, so one thread can change the value and another may not see it.
Declaring a variable **volatile** forces it to be read/written from/to memory upon each access.
If a thread needs to read the value of a volatile variable and generate a new value for a shared volatile

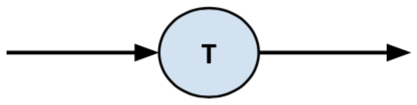
6. Object-Based Synchronization

Producer/Consumer Pattern



- T0 computes X and passes it to T1
- T1 uses X
- Synchronization is not needed for X, since at any point in time only one thread accesses it
- Synchronization is required to pass X from T0 to T1

Pipeline Node



```
while (true) {
    input = q_in.dequeue();
    output = do_something(input);
    q_out.enqueue(output);
}
```

Producer/Consumer Queues

```
q.enqueue(x1);          q.dequeue(x1);
q.enqueue(x2);     →    q.dequeue(x2);
…                       …
```

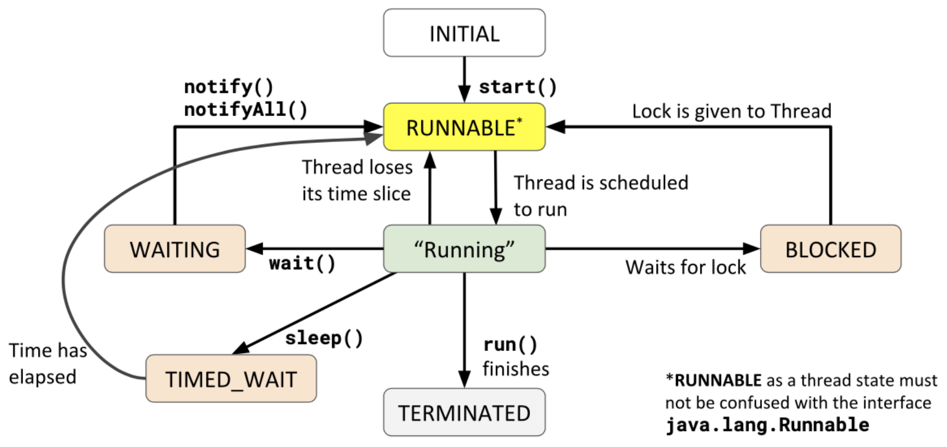
Semaphores are not sufficient to synchronize all the operations we need to handle, so we need a lock that we could temporarily escape from when waiting on a condition.

We must use **`synchronized`** and some **`Object`** methods:

- **`wait()`**:
  - current thread waits to be notified by another thread
  - waiting thread releases this object's lock
  - when notified, the current thread re-acquires the lock and resumes the execution
- **`notify()`**:
  - wakes up a single thread waiting on this object's lock
  - the choice among the threads is arbitrary
- **`notifyAll()`**:
  - wakes up all threads that are waiting on this object's lock

- wakes up all threads that are waiting on this object's lock

Thread State Transitions in Java



7. Explicit Locks

Explicit Locking

Intrinsic locks provide a good abstraction, but they have limitations:

- no fairness guarantees
- can only be used in blocks
- lock acquisition cannot be interrupted
- no timeout can be specified when waiting for a lock
- if multiple threads are waiting for different conditions, all threads need to be awakened if just one condition is met

The **`Lock`** interface supports locking disciplines that can be used in non-block-structured contexts and offers a choice of unconditional, timed, and interruptible lock acquisition, and all lock and unlock operations are **explicit**.
The main implementation of **`Lock`** is **`ReentrantLock`**, which provides the same mutual exclusion and memory visibility guarantees as **`synchronized`**.
The lock must be released in a **`finally`** block, otherwise the lock would never be released in case of throwing an exception.
The **execute-around idiom** provides a generic mechanism to perform resource allocation and cleanup operations so that the client can focus on the required functionality separating the operations implementation on the shared resource from the logics concerning resource management.

Condition and ReentrantLock

The **`Condition`** interface describes condition variables that may be associated with locks, similar to intrinsic locks but with extended capabilities.
Where a **`Lock`** replaces the use of **`synchronized`** methods and statements, a **`Condition`** replaces the use of the object-based synchronization methods.
`ReentrantLock` implements a standard mutual-exclusion lock, which means that at most one thread at a time can hold a reentrant lock.
Mutual exclusion prevents writer/writer, writer/reader, and reader/readeroverlap.
If data structures need to be modified, but most accesses involve only reading, then **read-write locks** allow a shared data structure to be accessed by multiple readers or a single writer, but not both at a time.
The **`ReadWriteLock`** interface define locks that may be shared among readers but are exclusive to writers.

8. Synchronization Utilities

Synchronizers

A **synchronizer** is an object that coordinates the control flow of threads based on its state.
A **latch** is a synchronizer that can delay the progress of coordinated threads not letting them pass until it reaches its terminal state an opens the gate.
Once the gate is open, it cannot be closed anymore: it is a **single-use object** and is used to ensure that a computation does not proceed until the resources it needs have been initialized.

CountDownLatch

A **`CountDownLatch`** is initialized with a given count and has two **decoupled** methods (can be called by different threads):

- **`countDown()`**: decrements the count of the latch, releasing all waiting threads if the count reaches zero
- **`await()`**: causes the current thread to wait until the latch has counted down to zero

Barriers

A **barrier** is a synchronizer that allows a set of threads to all wait for each other to reach a common barrier point.
It allows a fixed number of threads to meet repeatedly at the barrier point.
A **`CyclicBarrier`** is initialized with a given count, which represents the number of threads to coordinate.
The method **`await()`** decreases the counter and block until it reaches zero (both decrementing and waiting are **coupled**).
Once the counter reaches zero, the threads can proceed and the counter is reset.

9. Tasks

Work Partitioning

Work partitioning consists in splitting up the work of a single program into parallel tasks, either manually, where the user explicitly defines tasks, or automatically, where the user expresses an operation and the system does the rest.

Tasks

Tasks are abstract units of work that execute code, spawn other tasks and wait for results from other tasks.
Option 1: new thread for each task:

- thread lifecycle overhead (creation, termination)
- high resource consumption (possible *OutOfMemory*)
- reduced stability

Option 2: framework for asynchronous task execution which decouples task submission from execution policy:

- flexible execution policies
- order of task execution
- control of parallelism, preventing resource overuse

Java Executor Interface

The **`Executor`** interface provides a way of decoupling task submission from the mechanics of how each task will be run.
An **`ExecutorService`** allows a lifecycle management divided into three states: running, shutting down, and terminated.
It also provides methods to manage termination and to produce a **`Future`** for tracking progress of one or more asynchronous tasks.
An unused **`ExecutorService`** should be shut down to reclaim its resources.
The interface **`Future<T>`** can create objects acting like a placeholder for a future result, and allows a thread to wait until the result is completed and stored into it.
`FutureTask<T>` represents a cancellable asynchronous computation and implements methods to start and cancel a computation, to query to see if the computation is completed, and to retrieve the result of the computation.

Task Patterns

Java supports two main patterns related to tasks:

- **thread pools**: manage a homogeneous pool of worker threads, bound to a **work queue** holding tasks waiting to be executed
- **form-join pools**: a framework that allows developers to easily execute divide-and-conquer algorithms

10. Synchronized Collections

Synchronized Collections

A **collection** is **synchronized** if it archives thread-safety by allowing only one thread at time to access its state.
Synchronization wrappers add automatic synchronization to a collection.
Iteration is accomplished via single atomic operation calls into the collection, and user must manually synchronize on the collection when iterating over it.
Locking an entire collection during iteration may be undesirable, since other threads that need to access it are blocked for the entire iteration.
An alternative approach is to clone the collection and iterate on the copy instead, since it would be thread-confined and no other thread can modify it during iteration, but the collection still must be locked during the clone operation itself and such an approach can lead to inconsistent views of a collection.

Concurrent Collections

Concurrent collections are improved thread-safe collections not governed by a single lock.
Replacing synchronized collections with concurrent ones can offer big scalability improvements.
`BlockingQueue<T>` extends **`Queue<T>`** to add blocking insertion and retrieval.
The queue eliminates the random-access requirements of lists, and retrieval from an empty queue returns **`null`**.
Methods in four forms:

	Throws exception	Special value	Blocks	Times out
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	`(not applicable)`	`(not applicable)`

`BlockingDeque<T>` is a double-ended queue that extends **`Queue<T>`** and supports access to the elements at both ends of the queue.
Deques can also be used as **Last-In-First-Out stacks**:

- **`push(e) → addFirst(e);`**
- **`pop() → removeFirst();`**
- **`peek() → peekFirst();`**

Implementations:

- **`ArrayBlockingQueue<T>`**: bounded `BlockingQueue<T>` backed by an array
- **`LinkedBlockingQueue<T>`** - **`LinkedBlockingDeque<T>`**: optionally-bounded blocking queue/deque
- **`PriorityBlockingQueue<T>`**: unbounded blocking queue based on a priority heap

`ConcurrentLinkedQueue<T>` extends **`Queue<T>`** and is an unbounded, thread-safe queue based on linked nodes.

11. Double Checked Locking

Definition

The **double checked locking** is a software idiom used to reduce the overhead of acquiring a lock by first testing the locking criterion without actually acquiring the lock and limiting synchronization to the rare case of computing the field's value or constructing a new instance.

Correct single-threaded version using lazy initialization:

```
final class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
}
```

Correct multithreaded version using synchronization:

```
final class Foo {
    private Helper helper = null;
    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
    }
}
```

Correct multithreaded version using double-checked locking:

```
final class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized (this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
    }
}
```

12. Liveness

Deadlocks

Deadlock: system is blocked because two or more competing actions are waiting for the other to finish, causing a circular wait
Livelock: system is blocked because two or more competing actions are constantly changing, impeding the progress of each other
Starvation: system is not blocked, but one action is perpetually denied right or resources to execute
A process is deadlocked when it is waiting for an event that will never occur:

- no other process was set up to trigger the event
- process responsible for the trigger is blocked

Necessary Conditions

- Mutual exclusion: at least one resource is not shareable
- Hold and wait: a process is holding some resource while waiting for another
- No preemption: processes cannot take resources away
- Circular wait: P_0 waits for P_1 , ... , P_n waits P_0

Strategies Against Deadlocks

- Prevention: break at least one necessary condition
- Avoidance: avoid unsafe states
- Detection and recovery: detect and fix

Breaking Necessary Conditions

- Mutual exclusion:
  - resources cannot be always shared in this way
- Hold and wait:
  - allow requests only when holding no resources
  - under-utilized resources
  - may not be able to anticipate all resources in advance
  - could starve waiting for popular resource
- No preemption
  - take resource back from waiting process
  - for some devices this is difficult
  - must design processes to be able to recover
- Circular wait
  - impose total order on resources
  - can only request resources with higher value in order

PF3 cheat sheet

Semaphores

- Usually used to solve producer-consumer type problems.

  Problem : To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

```
//CODICE PRESO DA GEEKSFORGEEKS (anche il testo qui sopra), che tradotto vuol dire:
occhio a non copiare troppo senò ti gamano
import java.util.concurrent.Semaphore;

class Q {
    // an item
    int item;

    // semCon initialized with 0 permits
    // to ensure put() executes first
    static Semaphore semCon = new Semaphore(0);  //RICORDA DI CAMBIARE IL NOME DEL
SEMAPHORE

    static Semaphore semProd = new Semaphore(1); //RICORDA DI CAMBIARE IL NOME DEL
SEMAPHORE

    // to get an item from buffer
    void get()
    {
        try {
            // Before consumer can consume an item,
            // it must acquire a permit from semCon
            semCon.acquire();
        }
        catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        finally{
          // consumer consuming an item
```

```
            System.out.println("Consumer consumed item : " + item);

            // After consumer consumes the item,
            // it releases semProd to notify producer
            semProd.release();
          }
      }

      // to put an item in buffer
      void put(int item)
      {
          try {
              // Before producer can produce an item,
              // it must acquire a permit from semProd
              semProd.acquire();
          }
          catch (InterruptedException e) {
              System.out.println("InterruptedException caught");
          }
          finally{

            // producer producing an item
            this.item = item;

            System.out.println("Producer produced item : " + item);

            // After producer produces the item,
            // it releases semCon to notify consumer
            semCon.release();
          }
      }
  }
```

Reentrant Locks

- Example producer-consumer

```
public final class ShopReentrantLock implements Shop {
  private final Queue<Product> warehouse = new LinkedList<Product>();
  private final Lock lock = new ReentrantLock(); //LOCK
  private final Condition notEmpty = lock.newCondition(); //LOCK CONDITION

  public void buy(Product product) throws InterruptedException {
    processOrder();
    lock.lock();

    try{
      warehouse.add(product);
      notEmpty.signalAll();
    } finally{
      lock.unlock();
    }
  }

  public Product sell() throws InterruptedException {
    Product productToSell;
    processOrder();
    lock.lock();

    try {
```

```java
      while(warehouse.size() == 0) {
        notEmpty.await();
      }
      productToSell = warehouse.poll();
    }
    finally {
      lock.unlock();
    }
    processOrder();
    return productToSell;
  }

  private static void processOrder() throws InterruptedException {
    // simulating waiting time
    Thread.sleep(100);
  }
}
```

- A variant of the reentrantlock is the **ReentrantReadWriteLock**

  In many cases, data structures need to be modified, but most accesses involve only reading. In these cases, *read-write locks* allow a shared data structure to be accessed by multiple readers or a single writer, but not both at a time.

```java
//Codice dell'assignment 4
public final class CatalogReadWriteLock extends CatalogImpl {

  final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

  public CatalogReadWriteLock(int numberOfGames) {
    super(numberOfGames);
  }

  public void publish(Game newGame) throws InterruptedException {
    rwl.writeLock().lock();
    try{
      super.publish(newGame);
    }
    finally{
      rwl.writeLock().unlock();
    }
  }

  public void play(int index) throws InterruptedException, GameNotAvailableException {
    rwl.readLock().lock();
    try{
      super.play(index);
    }
    finally{
      rwl.readLock().unlock();
    }
  }
}
```

Cyclic Barrier

- A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called *cyclic* because it can be re-used after the waiting threads are released.
- A `CyclicBarrier` supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This *barrier action* is useful for updating shared-state before any of the parties continue.

```
//Javadoc
class Solver {
   final int N;
   final float[][] data;
   final CyclicBarrier barrier; //the barrier

   class Worker implements Runnable {
     int myRow;
     Worker(int row) { myRow = row; }
     public void run() {
       while (!done()) {
         processRow(myRow);

         try {
           barrier.await(); //the threads will wait here
         } catch (InterruptedException ex) {
           return;
         } catch (BrokenBarrierException ex) {
           return;
         }
       }
     }
   }

   public Solver(float[][] matrix) {
     data = matrix;
     N = matrix.length;
     barrier = new CyclicBarrier(N,
                                 new Runnable() { //the runnable for the barrier
                                   public void run() {
                                     mergeRows(...);
                                   }
                                 });
     for (int i = 0; i < N; ++i)
       new Thread(new Worker(i)).start();

     waitUntilDone();
   }
 }
```

```
//CODICE COPIATO DA INTERNET OCCHIO A COPIARE!!!!
//
public class CyclicBarrierExample {
    //Runnable task for each thread
    private static class Task implements Runnable {

        private CyclicBarrier barrier;

        public Task(CyclicBarrier barrier) {
            this.barrier = barrier;
        }

        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + " is waiting on
barrier");
                barrier.await(); //ALL THE THREADS WILL WAIT HERE WHILE THE OTHER THREADS
FINISH
                System.out.println(Thread.currentThread().getName() + " has crossed the
barrier");
            } catch (InterruptedException ex) {
                //Error
            } catch (BrokenBarrierException ex) {
               //Error
            }
        }
    }

  public static void main(String args[]) {

        //creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()
        final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){
            @Override
            public void run(){
                //This task will be executed once all thread reaches barrier
                System.out.println("All parties are arrived at barrier, lets play");
            }
        });

        //starting each of thread
        Thread t1 = new Thread(new Task(cb), "Thread 1");
        Thread t2 = new Thread(new Task(cb), "Thread 2");
        Thread t3 = new Thread(new Task(cb), "Thread 3");

        t1.start();
        t2.start();
        t3.start();

    }
}

Output:
Thread 1 is waiting on the barrier
Thread 3 is waiting on the barrier
Thread 2 is waiting on the barrier
All parties have arrived at the barrier, lets play
Thread 3 has crossed the barrier
Thread 1 has crossed the barrier
Thread 2 has crossed the barrier
```

Wait & Notify

- The `wait()` and `notify()` methods are designed to provide a mechanism to allow a thread to block until a specific condition is met.

  Example:

```
//Codice copiato da StackOverflow
class MyHouse {
    private boolean pizzaArrived = false;

    public void eatPizza(){
        synchronized(this){
            while(!pizzaArrived){  //Always use while(condition) when using wait and
notify
                wait();
            }
        }
        System.out.println("yumyum..");
    }

    public void pizzaGuy(){
        synchronized(this){
             this.pizzaArrived = true;
             notifyAll();
        }
    }
}
```

  Example producer-consumer **locked on THIS**:

```
public final class ShopWaitNotify implements Shop {
  private final Queue<Product> warehouse = new LinkedList<Product>();

  public void buy(Product product) throws InterruptedException {
    processOrder();
    synchronized(this){
      warehouse.add(product);
      notifyAll(); //notify when one product is added
    }
  }

  public Product sell() throws InterruptedException {
    Product productToSell;
    processOrder();
    synchronized(this){
      while(warehouse.size() == 0) {
        wait();
        // waiting for a product
      }
      productToSell = warehouse.poll();
    }
    processOrder();
    return productToSell;
  }

  private static void processOrder() throws InterruptedException {
    // simulating waiting time
    Thread.sleep(100);
    }
}
```

Example producer-consumer **locked on WAREHOUSE**

```
public final class ShopWaitNotify implements Shop {
    private final Queue<Product> warehouse = new LinkedList<Product>();

    public void buy(Product product) throws InterruptedException {
        processOrder();
        synchronized (warehouse) {
            warehouse.add(product);
            warehouse.notify();
        }
    }

    public Product sell() throws InterruptedException {
        Product productToSell;
        processOrder();
        synchronized (warehouse) {
            while (warehouse.size() == 0) {
                warehouse.wait();
            }
            productToSell = warehouse.poll();
        }
        processOrder();
        return productToSell;
    }

    private static void processOrder() throws InterruptedException {
        // simulating waiting time
        Thread.sleep(100);
    }
}
```

Count down latch

- **Sample usages:**
    - Here is a pair of classes in which a group of worker threads use two countdown latches:

      startSignal is a start signal that prevents any worker from proceeding until the driver is ready for them to proceed;

      doneSignal is a completion signal that allows the driver to wait until all workers have completed.

      ```
      //from javadoc
       class Driver { // ...
         void main() throws InterruptedException {
           CountDownLatch startSignal = new CountDownLatch(1);
           CountDownLatch doneSignal = new CountDownLatch(N);

           for (int i = 0; i < N; ++i) // create and start threads
             new Thread(new Worker(startSignal, doneSignal)).start();
      ```

```
      doSomethingElse();            // don't let run yet
      startSignal.countDown();      // let all threads proceed
      doSomethingElse();
      doneSignal.await();           // wait for all to finish
    }
  }

  class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
       this.startSignal = startSignal;
       this.doneSignal = doneSignal;
    }
    public void run() {
       try {
         startSignal.await();
         doWork();
         doneSignal.countDown();
       } catch (InterruptedException ex) {} // return;
    }

    void doWork() { ... }
  }
```

- Another typical usage would be to divide a problem into N parts, describe each part with a Runnable that executes that portion and counts down on the latch, and queue all the Runnables to an Executor. When all sub-parts are complete, the coordinating thread will be able to pass through await

```
//from javadoc
class Driver2 { // ...
   void main() throws InterruptedException {
     CountDownLatch doneSignal = new CountDownLatch(N);
     Executor e = ...

     for (int i = 0; i < N; ++i) // create and start threads
       e.execute(new WorkerRunnable(doneSignal, i));

     doneSignal.await();           // wait for all to finish
   }
 }

 class WorkerRunnable implements Runnable {
   private final CountDownLatch doneSignal;
   private final int i;
   WorkerRunnable(CountDownLatch doneSignal, int i) {
      this.doneSignal = doneSignal;
      this.i = i;
   }
   public void run() {
      try {
        doWork(i);
        doneSignal.countDown();
      } catch (InterruptedException ex) {} // return;
   }

   void doWork() { ... }
 }
```

Threads

- Threads executing **runnables**

```
public long processAllFiles() throws InterruptedException {
  AtomicLong totalLineCount = new AtomicLong(0);

  // TODO 1 : Create and start one thread per file
  //          passing a FileProcessorRunnable as argument

  List<Thread> threads = new ArrayList<Thread>();

  for(File f: files){
    Thread t = new Thread(new FileProcessorRunnable(f, totalLineCount));
                          //THIS IS A RUNNABLE
    threads.add(t); //save a reference to the new thread
    t.start();
  }

  // TODO 2 : Use the join method to ensure that all files
  //          have been processed

  for(Thread t: threads){
    t.join(); //join all the threads together
  }

  //HERE ALL THE THREADS ARE DONE EXECUTING THE RUNNABLE

  // TODO 3 : Return the value encapsulated by totalLineCount
  // return 0;
  return totalLineCount.get();
}
```

- Threads executing **callables**

```
@Override
public long processAllFiles() throws InterruptedException, ExecutionException {
      List<Future<Long>> futures = new LinkedList<>();

  // TODO 1 : Create and start one thread per file
  //          passing a FutureTask as argument.
  //          The FutureTask must be created from a FileProcessorCallable.

  for(File f: files){
    FutureTask<Long> ft = new FutureTask<>(new FileProcessorCallable(f));
                                          //CALLABLE
    futures.add(ft); //save a reference to the new FutureTask
    // ft.run();
    Thread t = new Thread(ft);
    t.start();
  }

  long totalCount = 0;

  // TODO 2 : Obtain the partial result from each thread and
  //          add it to the total by waiting until
  //          the result in each future is ready.

  for(Future<Long> f: futures){
    totalCount += f.get(); //wait fot the FutureTask(s) to finish and get the result
  }

  return totalCount;
}
```

Thread Pools

- ### Create a thread pool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Executor;


public class ExampleThreadPool{

  final ExecutorService threadPool;

  ExampleThreadPool(int nThreads){
    threadPool = Executors.newFixedThreadPool(nThreads);
  }

  //Example
  public void executeRunnable(Runnable runnable){
    pool.execute(runnable);
  }

  //....
}

public class ExampleThreadPool_2{
  private final ThreadPoolExecutor threadPool;

  ExampleThreadPool_2(int nThreads){
    threadPool = (ThreadPoolExecutor)Ececutors.newFixedThreadPool(nThreads);
  }

  //Example
  public void executeRunnable(Runnable runnable){
    pool.execute(runnable);
  }

  //....
}
```

- ### Thread pools with **runnables**

```
  public long processAllFiles() throws InterruptedException {
    AtomicLong totalLineCount = new AtomicLong(0);

    // TODO 2 : Submit one FileProcessorRunnable per file
    //          to the thread pool
    for(File f: files){
      threadPool.submit(new FileProcessorRunnable(f, totalLineCount));
    }                    //THIS IS A RUNNABLE

    // TODO 3 : Shutdown the thread pool

    threadPool.shutdown();

    // TODO 4 : Block until all tasks have completed execution

    threadPool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
    //WAIT FOR ALL THE THREADS IN THE POOL TO FINISH

    // TODO 5 : Return the value encapsulated by totalLineCount
    return totalLineCount.get();
  }
```

- Thread Pools with **callables**

```
  public long processAllFiles() throws InterruptedException, ExecutionException {
    List<Future<Long>> results = new ArrayList<>();

    // TODO 2 : Submit one FileProcessorCallable per file
    //          to the thread pool
    //          Store the returned Future into the results List

    for(File f: files){
      results.add(threadPool.submit(new FileProcessorCallable(f)));
                                    //THIS IS A CALLABLE
    }

    // TODO 3 : Shutdown the thread pool
    threadPool.shutdown();

    long totalLineCount = 0;

    for(Future<Long> f: results){
      totalLineCount += f.get(); //WAITS for that task to finish ad returns the result
    }
    // TODO 4 : Obtain the partial result from each thread and
    //          add it to the total

    return totalLineCount;
  }
```

Synchronization

- Inner classes:

```
public class OuterClass{
  class InnerClass{
    public static synchronized void a(){ //synchronized on OuterClass.InnerClass.class
      //...
    }
    public static void b(){
      synchronize(InnerClass.class){ // synchronized on OuterClass.InnerClass.class
        //...
      }
    }
    public void c(){
      synchronized(this.getClass()){ // synchronized on OuterClass.InnerClass.class
        //...
      }
    }
    public synchronized void d(){  //synchronized OuterClass.InnerClass.this
      //...
    }
  }

  public static synchronized void x(){ //synchronized on OuterClass.class
    //...
  }
  public synchronized void y(){ //synchronized on OuterClass.this
    //...
  }
}
```

- Objects

```
public class Class{

  // If x == o == this than every method is synchronized on the same object ==> parallel
  // If x == o b() and c() will run sequentially.

  public void a(Object x){ //synchronized on Object o
    synchronized(x){
      //...
    }
  }

  public void b(Object o){ //synchronized on Object o
    synchronized(o){
      //...
    }
  }

  public void c(){
    synchronized(this){ //synchronized on Class.class
      //...
    }
  }
}
```

Classic Implementations

Counters

- Synchronized Blocks

  ```
  public class SynchronizedBlockCounter implements Counter {

    private int value;

    @Override
    public void increment() {
      synchronized (this) {  //Synchronized on SynchronizedBlockCounter.this
        value++;
      }
    }

    @Override
    public int value() {
      synchronized (this) {  //Synchronized on SynchronizedBlockCounter.this
        return value;
      }
    }

  }
  ```

- Synchronized methods

  ```
  public class SynchronizedCounter implements Counter {

      private int value;

      @Override
      synchronized public void increment() {  //Synchronized on SynchronizedCounter.this
          value++;
      }

      @Override
      synchronized public int value() {   //Synchronized on SynchronizedCounter.this
          return value;
      }
  }
  ```

- Atomic Counter

  ```
  public class AtomicCounter implements Counter {

    private final AtomicInteger value = new AtomicInteger();

    @Override
  ```

```
  public void increment() {
    value.incrementAndGet();
  }

  @Override
  public int value() {
    return value.get();
  }

}
```

Queues

- Semaphore queue buffer

```
//ESEMPIO PRESO DA ASSIGNMENT MIO CAMBIA CODICE
package buffer;

import java.util.concurrent.Semaphore;

public class SemaphoreQueueBuffer<T> implements BoundedBuffer<T>{

    private final Semaphore full;
    private final Semaphore empty;
    private final Semaphore lock;

    private final T[] elements;

    private int firstFree;
    private int firstFull;

    public SemaphoreQueueBuffer(final int size){
        this.lock = new Semaphore(1);
        this.full = new Semaphore(0);
        this.empty = new Semaphore(size);
        this.elements = (T[]) new Object[size]; //the queue
        firstFree = 0;
        firstFull = 0;
    }

    public void put(T element) throws InterruptedException {
        empty.acquire();

        lock.acquire();

        final int slot = firstFree++;
        if(firstFree >= elements.length){
            firstFree = 0;
        }
        elements[slot] = element;

        lock.release();

        full.release();
    }
    public T take() throws InterruptedException {

        full.acquire();

        lock.acquire();

        final int slot = firstFull++;
        if(firstFull >= elements.length){
            firstFull = 0;
        }

        final T result = elements[slot];

        lock.release();

        empty.release();
        return result;
    }

}
```

- Reentrant Lock queue buffer

```
//ESEMPIO PRESO DA ASSIGNMENT MIO CAMBIA CODICE
package buffer;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public final class ArrayBlockingQueueReentrantLock<T> implements BoundedBuffer<T> {


    private final T[] elements;

    private Lock lock;
    private Condition notFull;
    private Condition notEmpty;

    private int firstFree;
  private int firstFull;

    private int elementCount;


    public ArrayBlockingQueueReentrantLock(final int capacity){
        this.elements = (T[]) new Object[capacity];
        this.lock = new ReentrantLock();
        this.notFull = lock.newCondition();;
        this.notEmpty = lock.newCondition();;
        this.elementCount = 0;
        firstFree = 0;
        firstFull = 0;
    }

    @Override
    public T take() throws InterruptedException {
        lock.lock();
        while(elementCount == 0){
            notEmpty.await();
        }

        final int slot = firstFull++;
        if(firstFull >= elements.length){
            firstFull = 0;
        }

        final T result = elements[slot];

        elementCount--;

        notFull.signalAll();

        lock.unlock();

        return result;
    }

    @Override
    public void put(T element) throws InterruptedException {
        lock.lock();

        while(elementCount == elements.length){
            notFull.await();
        }

        final int slot = firstFree++;
        if(firstFree >= elements.length){
            firstFree = 0;
        }
        elements[slot] = element;

        elementCount++;

        notEmpty.signalAll();

        lock.unlock();
    }

}
```

- Wait & Notify queue buffer

```
//ESEMPIO PRESO DA ASSIGNMENT MIO CAMBIA CODICE
package buffer;

public final class ArrayBlockingQueueWaitNotify<T> implements BoundedBuffer<T> {


    private final T[] elements;
    private int firstFree;
    private int firstFull;

    private int elementCount;
    public ArrayBlockingQueueWaitNotify(final int capacity) {
        this.elements = (T[]) new Object[capacity];
        this.elementCount = 0;
        firstFree = 0;
        firstFull = 0;
    }


    @Override
    public synchronized void put(T element) throws InterruptedException {
        while(elementCount == elements.length){
            wait();
        }

        final int slot = firstFree++;
        if(firstFree >= elements.length){
            firstFree = 0;
        }
        elements[slot] = element;
        elementCount++;
        notifyAll();
    }

    @Override
    public synchronized T take() throws InterruptedException {
        while(elementCount == 0){
            wait();
        }

        final int slot = firstFull++;
        if(firstFull >= elements.length){
            firstFull = 0;
        }

        final T result = elements[slot];
        elementCount--;
        notifyAll();
        return result;
    }

}
```

- Thread Pool

```
//assignment 4
public class Ex1cSimulator extends Ex1bSimulator {

  public static void main(String[] args) throws InterruptedException {
    new Ex1cSimulator().run();
  }

  private final ExecutorService devThreadPool =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
  private final ExecutorService gamerThreadPool =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());


  // TODO: get the number of available processors (N) in your system and
  // instantiate two thread pools, one for developers and one for
  // gamers, both of them with N threads

  @Override
  protected void addDeveloper(Developer developer) {
    Runnable runnable = new MonitoredRunnable(developer);
    devThreadPool.execute(runnable);
    // TODO: execute the runnable in the developers pool
  }

  @Override
  protected void addGamer(Gamer gamer) {
    Runnable runnable = new MonitoredRunnable(gamer);
    gamerThreadPool.execute(runnable);
    // TODO: execute the runnable in the gamers pool
  }

  @Override
  protected void waitWorkers() throws InterruptedException {
    // TODO: shutdown the thread pools
    devThreadPool.shutdown();
    gamerThreadPool.shutdown();
    super.waitWorkers();
  }
}
```