**Thread 1:**

General part

Write 2-5 lines of text for each bullet:

* When and why will we use Threads in our programs?

Det er for at give vores program en “multitasking evne” for at kunne execute flere ting samtidig. Ex: hvis du har en part A og Part B vil systemet først klar Part A også Part B bagefter hvor med Threads gøre den det samtidig.

* What is the Race Condition Problem and how can you solve it?

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Race condition: When the behaviour of a program depends on the ordering/interleaving of threads, *and shouldn’t.*

The tool needed to prevent these errors is *synchronization*

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*

Every object has an intrinsic lock associated with it.

By convention, a thread that needs exclusive access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them

 A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock

As long as a thread owns an intrinsic lock, no other thread can acquire the same lock

Other threads will block when they attempts to acquire a lock owned by a another Thread

* Explain the Producer/Consumer-problem and how to solve it in modern Java Programs

The producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-thread synchronization problem

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

The producer's job is to generate a piece of data, put it into the buffer and start again.

The consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is 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

Resolve: Use Blocking que implementation Hvis der ikke er noget at tage fra que sætter threadsen sig til at sove og vågner først op når der bliver puttet noget nyt derind. Det samme gælder Producer. Den sover når quen er fuld af lort.

* Explain what Busy Waiting is and why it's a bad thing in a modern software system.

Busy Waiting er når en process bliver ved med at tjekke om en “condition is true”. such as whether [keyboard](https://en.wikipedia.org/wiki/Computer_keyboard) input or a [lock](https://en.wikipedia.org/wiki/Lock_(computer_science)) is available.  
Busy Waiting is considered an [anti-pattern](https://en.wikipedia.org/wiki/Anti-pattern) and should be avoided,[[1]](https://en.wikipedia.org/wiki/Busy_waiting#cite_note-1) as processor time that could be used to execute a different [task](https://en.wikipedia.org/wiki/Thread_(computer_science)) is instead wasted on useless activity.

* Describe Java's *BlockingQueue* interface, relevant implementations and methods relevant for the producer consumer problem.

A BlockingQueue has 4 different sets of methods for inserting, removing and getting the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Throws Exception | Special Value | Blocks | Times Out |
| Insert | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| Remove | remove(o) | poll() | take() | poll(timeout, timeunit) |
| Get | element() | peek() |  |  |

**Throws Exception:**  
If the attempted operation is not possible immediately, an exception is thrown.

**Special Value**:   
If the attempted operation is not possible, a special value is returned (often true / false).

**Blocks:**  
If the attempted operation is not possible, the method call blocks until it is.

**Times Out:**  
If the attempted operation is not possible, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

Den relevante method for producer-consumer problemet er blocking!

Den relevante implementation i java er ArrayBlockingQue:  
 A bounded [blocking queue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) backed by an array. This queue orders elements FIFO (first-in-first-out). The *head* of the queue is that element that has been on the queue the longest time. The *tail* of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

This is a classic "bounded buffer", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the capacity cannot be changed. Attempts to put an element into a full queue will result in the operation blocking; attempts to take an element from an empty queue will similarly block.

CA or Semester Project

For a real exam exercise, this will be a small part where you are expected to talk, in about 5 minutes, about one of the semester CA’s or the semester project (related to the topic for this question).

**Thread 2:**

General part

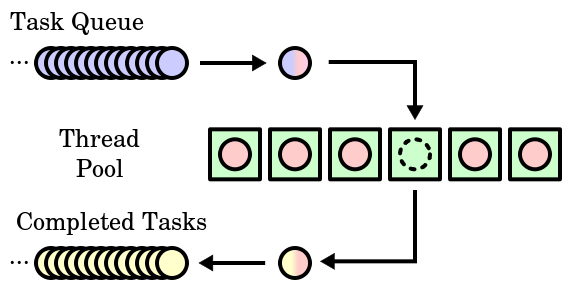
* Explain the benefits from using a Thread Pool

Thread creation and destruction overhead is negated,

Better performance and better system stability.

It is better to use a thread pool in cases where the number of threads is very large.

You do not have to create, manage, schedule, and terminate your thread, the thread pool class do all of this for you.



* Explains ways to handle "returned values" from Threads

You can use a local final variable array. The variable needs to be of non-primitive type, so you can use an array. You also need to synchronize.

public void test() throws InterruptedException, ExecutionException  
{   
 ExecutorService executor = Executors.newSingleThreadExecutor();  
 **Callable<Integer> callable = new Callable<Integer>() {  
 @Override  
 public Integer call() {  
 return 2;  
 }  
 };** Future<Integer> future = executor.submit(callable);  
 // future.get() returns 2 or raises an exception if the thread dies, so safer  
 executor.shutdown();  
}

public class Foo implements **Runnable** {  
  **private volatile int value;**  
  
  **@Override  
 public void run() {  
 value = 2;  
 }  
  
 public int getValue() {  
 return value;  
 }** }

* Explain the use of the Callable interface and how to use it.

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. **A Runnable, however, does not return a result and cannot throw a checked exception.**

* A Callable needs to implement call() method while a Runnable needs to implement run() method.
* A Callable can return a value but a Runnable cannot.
* A Callable can throw checked exception but a Runnable cannot.
* A Callable can be used with ExecutorService#invokeXXX(Collection<? extends Callable<T>> tasks) methods but a Runnable cannot be.
* **public interface Runnable {  
   void run();  
  }**
* **public interface Callable<V> {  
   V call() throws Exception;  
  }**

CA or Semester Project

For a real exam exercise, this will be a small part where you are expected to talk, in about 5 minutes, about one of the semester CA’s or the semester project (related to the topic for this question).

**Thread 3:**

General part

Explain about Thread Programming including:

* When and why we will use Threads in our programs?

Det er for at give vores program en “multitaskin evne” for at kunne execute flere ting samtidig. Ex: hvis du har en part A og Part B vil systemt først klar Part A også Part B bagefter hvor med Threads gøre den det samtidig.

* Explain about the Race Condition Problem and ways to solve it in Java

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Race condition: When the behaviour of a program depends on the ordering/interleaving of threads, *and shouldn’t.*

The tool needed to prevent these errors is *synchronization*

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*

Every object has an intrinsic lock associated with it.

By convention, a thread that needs exclusive access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them

 A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock

As long as a thread owns an intrinsic lock, no other thread can acquire the same lock

Other threads will block when they attempts to acquire a lock owned by a another Thread

* Explain how we can write reusable non-blocking Java Controls using Threads

DET ANER JEG IKKE EN SKIDE OM OG KAN IKKE FINDE NOGET LORT OM DET!

* Explain about deadlocks, how to detect them and ways to solve the Deadlock Problem

Deadlock occurs when two (or more) threads have created a situation where they are all blocking each other

The processes involved need to share resources that must be used under mutual exclusion.

Processes hold on to resources already allocated to them while awaiting the granting of additional ones.

The system has no way of pre-empting (forcibly withdrawing) resources from processes once they have been allocated to them.

A circular chain of requests and allocations exist.

Detect DeadLock:

Abort processes one by one until the deadlock is no longer present.

Pre-empt resources one by one from processes until the deadlock is removed.

Interface ThreadMXBean er et interface der kan detecte dem for dig.

Prevent Deadlocks:

Eliminate one or more of the conditions necessary for a deadlock to occur:

The processes involved need to share resources that must be used under mutual exclusion.

Processes hold on to resources already allocated to them while awaiting the granting of additional ones.

The system has no way of pre-empting (forcibly withdrawing) resources from processes once they have been allocated to them.

A circular chain of requests and allocations exist.

CA or Semester Project

For a real exam exercise, this will be a small part where you are expected to talk, in about 5 minutes, about one of the semester CA’s or the semester project (related to the topic for this question).

→ TBD