**Design Principles and Patterns for Highly Concurrent Applications**

https://www.baeldung.com/concurrency-principles-patterns

**Concurrent modules** – how to create?

- **Process**: An instance of a running program that is isolated from other processes

Each process has its own isolated time and space. Not possible to share memory between processes.

Communicate by passing messages

- **Thread**: A segment of a process. Can share same memory space. Each thread has a unique stack and priority. Can be native (by OS) or green (by runtime)

**How concurrent models interact**

- **Shared memory**

**- Message passing**

**Problems in Concurrent Programming**

- **Mutual exclusion** - threads need to have exclusive access to shared state or memory to ensure the correctness of programs

**Synchronization primitives** - lock, monitor, semaphore, or mutex.

- **Context Switching (Heavyweight Threads)** - threads are frequently switched between different states. Current state needs to be saved and resumed

**Design Patterns for High Concurrency**

**1.** **Actor-Based Concurrency**

A mathematical model of concurrent computation that basically treats everything as an actor. Eliminates one of the fundamental problems with concurrent programming — shared memory. Actors communicate through messages, and each actor processes messages from its exclusive mailboxes sequentially. However, we execute actors over a thread pool.

**2. Event-Based Concurrency (e.g. JavaScript)**

the event loop blocks on the event provider and dispatches an event to an event handler on arrival

**3. Non-Blocking Algorithms** - an algorithm that blocks on a

thread obviously brings down the throughput significantly and prevents us from building highly concurrent applications.

***Fibers in Java*** *- Project Loom - introduce continuations together with fibers, which may change the way we write concurrent applications*

***Cache Layer*** *– reduce hit in database*

*Hazelcast is a distributed, cloud-friendly, in-memory object store and compute engine that supports a wide variety of data structures such as Map, Set, List, MultiMap, RingBuffer, and HyperLogLog.*

***Redis*** *is an in-memory data structure store that we primarily use as a cache. It provides an in-memory key-value database with optional durability*

*.*

**Multithreading Desing Patterns**

**1. Producer-consumer pattern** - Two type of threads: producers and consumers, Useful when multiple sources of data and multiple workers in parallel

**2. Worker pool pattern** – Fixed number of threads to process tasks from a shared queue. Useful when limit the number of threads and reuse for different tasks

**3. Pipeline pattern** – Multiple stages of processing for each task.

**4. Reactor pattern** – Single thread to handle multiple events from multiple sources. High number of short-lived and non-blocking events.

**5. Future/Promise pattern** – abstractions to represent results of asynchronous operations. **Future** – hold the state and value of an operation that will be completed in the future. **Promise** – an object that can fulfil or reject a future. Useful for readable code

**6. Actor model** – actors communicate and coordinate with each other. Actors exchange messages each other.

**Coroutines vs Threads**

|  |  |
| --- | --- |
| **Threads** | **Coroutines** |
| Data from one thread to another thread will take time. Also, callback introduces complexity | Eliminate callbacks method |
| Creating and stopping – expensive. (it is on stack) | Coroutines do not have their own stack |
| Threads are blocking. When thread sleeps some duration then entire threads get blocked | Suspendable when they are delayed for some other works it will do another work. |
| Threads involves blocking and context switching and is slower as compared to coroutines | Coroutines are high level concurrency |
| Threads can be switched when the jobs get over | Coroutines can change any time, as they are suspendable |
| Managed by OS | Managed by users |

**Multithreading in Java**

Executing multiple threads simultaneously

Thread – lightweight sub-process. Smallest unit of processing. Both used to achieve multitasking.

**Multitasking**

Executing multiple tasks simultaneously.

1. Process-based Multitasking (Multiprocessing)

2. Thread-based Multitasking (Multithreading)

**Create thread using**

**1. Extending the Thread**

Class cannot extend any other class.

Achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc.

that are not available in Runnable interface.

class MultithreadingDemo extends Thread {

public void run()

{

try {

System.out.println("Thread " + Thread.currentThread().getId()+ " is running");

}

catch (Exception e) {

}

}

}

MultithreadingDemo object = new MultithreadingDemo();

object.start();

**2. Implementing the Runnable**

Give you an object that can be shared amongst multiple threads.

class MultithreadingDemo implements Runnable {

public void run()

{

try {

System.out.println("Thread " + Thread.currentThread().getId() + " is running");

}

catch (Exception e) {

}

}

}

**States of Thread**

New / Runnable / Blocked / Waiting / Timed waiting / Terminated

**Types of thread**

- User thread: created by devs.

- Daemon thread: low priority. eg garbage collection.

isDaemon , setDaemon

**Thread priority**: from 1 - 10

The default priority is set to 5 as excepted.

Minimum priority is set to 0.

Maximum priority is set to 10.

**Deadlock**

- First In First Out

- Last In First Out

- Round Robin Scheduling

If we do use threadT1.start() then this method will look for the run() method to create a new thread.

In case of theadT1.run() method will be executed just likely the normal method

|  |  |
| --- | --- |
| **Method** | **Description** |
| start() | Initiates the execution of the thread. |
| run() | Contains the code to be executed by the thread. You override this method to define the thread's behavior. |
| sleep(long millis) | Causes the thread to sleep for the specified number of milliseconds. |
| join() | Waits for the thread on which it is called to die. |
| isAlive() | Returns true if the thread has been started and not yet terminated. |
| interrupt() | Interrupts the thread, causing it to throw an InterruptedException. |
| isInterrupted() | Returns true if the thread has been interrupted; otherwise, returns false. |
| getName() | Gets the name of the thread. |
| setName(String name) | Sets the name of the thread. |
| yield() | Suggests to the JVM that the current thread is willing to yield its current use of a processor. |

**Multithreading in Python**

import threading

import os

def task1():

print("Task 1 assigned to thread: {}".format(threading.current\_thread().name))

print("ID of process running task 1: {}".format(os.getpid()))

if \_\_name\_\_ == "\_\_main\_\_":

t1 = threading.Thread(target=task1, name='t1')

t1.start()

t1.join()

thread pool using

import concurrent.futures

**Multithreading** - multiple threads are spawned by a process to do different tasks, at about the same time, just one after the other. (illusion that the threads are running in parallel). In Python, the **Global Interpreter Lock (GIL)** prevents the threads from running simultaneously.

**Multiprocessing** is a technique where parallelism in its truest form is achieved. Multiple processes are run across multiple CPU cores, which do not share the resources among them.

*import time, os*

*from threading import Thread, current\_thread*

*from multiprocessing import Process, current\_process*