Java 꽉 잡아!
- JVM부터 GC, 스레드 동기화까지!

[2-2] 동시성 처리를 위한 스레드 동기화에 대해 살펴봅니다.

김재녕 (2023-06-13)

[1] 스레드 동기화와 동시성

스레드 동기화와 동시성

Program

- 컴퓨터가 실행 가능한 프로그래밍 언어로 작성된 인스트럭션(명령)의 셋 또는 시퀀스를 의미
- 일반적으로 소프트웨어에 속하는 컴포넌트

스레드 동기화와 동시성

Process

- 일반적으로 프로그램이 메모리에 로딩되어 실행되는 인스턴스
  이 프로세스는 하나 이상의 스레드에 의해 실행됨
  - 하나의 프로그램이 여러 프로세스가 될 수도 있음
- 대부분의 프로세스는 다음을 포함하고 있음
  - 프로그램 내에 코드
    할당 받은 시스템 리소스
    물리적 또는 논리적 권한
    초기 자료구조
- OS에 따라 인스트럭션을 동시에 실행하기 위한 멀티 스레드 환경으로 구성될 수 있음
- 멀티 태스킹은 여러 프로세스가 프로세서, 시스템 리소스 등을 공유하는 것
  즉 일정 기간동안 여러 프로세스를 동시에 실행
- 프로세스는 일반적으로 리소스를 공유하지 않게 설계되어 있음
  하지만 IPC(Inter-Process Communication)를 통해 가능하긴 함

스레드 동기화와 동시성

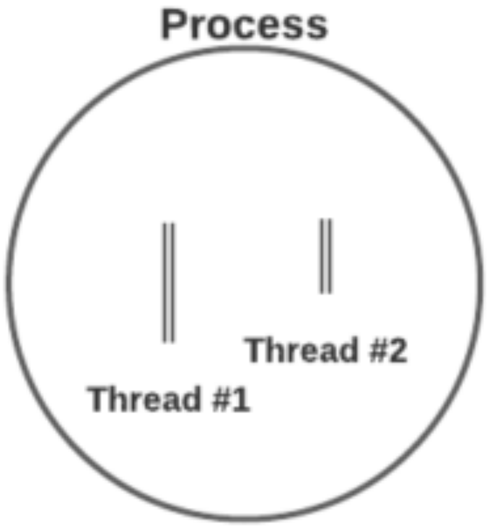
Thread

- OS의 일부인 스케줄러에 의해 독립적으로 관리되는 프로그래밍 인스트럭션의 최소 시퀀스
  즉 프로세스를 실행시키기 위한 최소 시퀀스이자 일종의 프로세스 실행 단위라고 할 수 있음
- 일반적으로 스레드는 프로세스의 구성 요소이며 OS 마다 스레드의 구현이 다름
- 멀티 스레딩 기능을 통해 스레드 간 메모리를 포함한 시스템 리소스를 공유할 수 있음
  실행 코드, 변수에 동적으로 할당된 값, 전역 변수 등을 언제든 공유할 수 있음

스레드 동기화와 동시성

Process vs Thread

- 프로세스는 일종의 프로그램의 실행을 의미
  프로그램 자체와 데이터, 리소스, 프로세스 관련 정보 등을 포함
  - 일반적으로 프로세스 간 다른 메모리 영역을 소유
  - 즉 격리된 실행 개체로 자체 스택, 메모리 소유
  - 멀티 프로세스 생성은 별도의 시스템 콜이 필요함
  - 프로세스 간 통신은 IPC 필요 (시스템 콜 호출 횟수 증가)
- 스레드는 일종의 세미 프로세스
  자체 스택을 갖고 코드를 실행
  - 일반적으로 스레드 간 메모리 영역을 공유
  - 한 번의 시스템 콜을 통해 2개 이상의 스레드 생성 가능
  - 스레드 간 통신을 위해 특별한 기술이 필요 없음
  - 스레드 관리 시에도 별도의 시스템 콜이 거의 필요 없음



https://www.baeldung.com/cs/process-vs-thread

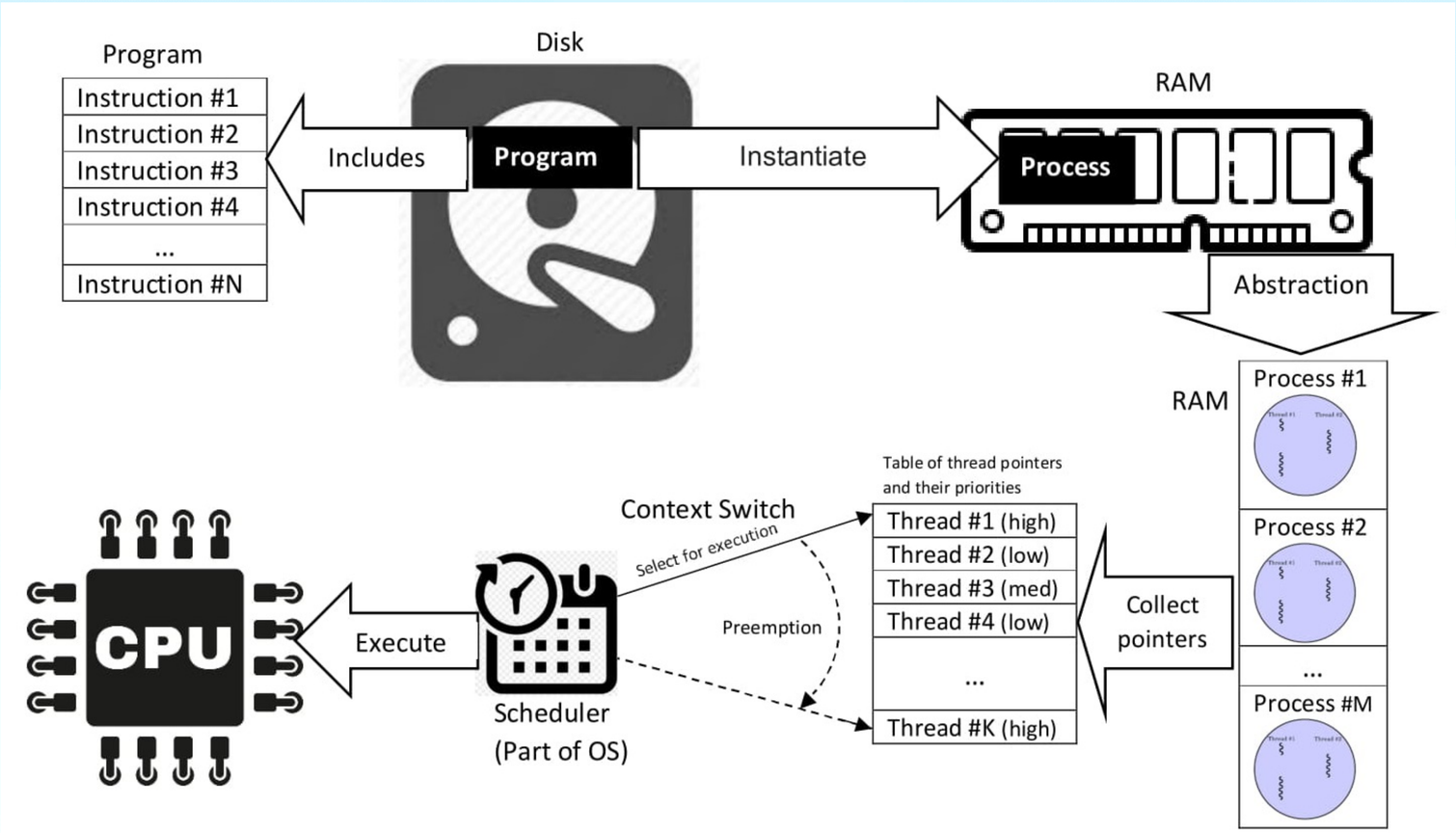
스레드 동기화와 동시성

Process 사용과 비교했을 때 Thread 사용 장단점

- 장점
  - 프로세스 그룹보다 메모리 등 리소스 공유가 원활함
  - 실제 멀티코어인 경우 멀티 스레드는 실제로 병렬 처리가 가능
  - 스레드 간 컨텍스트 스위칭이 프로세스 컨텍스트 스위칭보다 훨씬 빠름
- 단점
  - 스레드 그룹은 리소스 공유로 인해 하나의 스레드가 오작동을 일으키면 다른 스레드도 영향을 받음
    프로세스 그룹은 OS 레벨에서 격리되어 있어 스레드 그룹보다 안전함
  - 서로 다른 시스템에서 실행될 수 없음

스레드 동기화와 동시성

Program & Process & Thread



https://en.wikipedia.org/wiki/Thread_%28computing%29

스레드 동기화와 동시성

Java Thread

- `java.lang.Thread` 클래스를 통해 생성 가능
- 모든 스레드는 이름을 가지고 있으며 여러 스레드가 같은 이름을 가질 수 있음
- 모든 스레드는 우선순위를 갖고 있으며 높은 경우에 우선적으로 실행됨
- 특정 스레드 내에 실행되는 코드로 인해 새로운 스레드 객체가 생성되면
  새로 생성된 스레드는 생성한 기존 스레드와 동일한 우선순위를 가짐
- 스레드 중에는 데몬 스레드가 존재하며 생성 시 설정 가능
- Java 스레드는 상태를 가지고 있음
  이 상태는 OS 상태와 무관한 JVM 내에 Java 스레드 상태임
- JVM이 스레드를 종료시키는 조건
  - `Runtime.getRuntime().exit()` 메서드가 호출된 경우
  - 모든 `User Thread`가 종료된 경우
    run 메서드가 정상적으로 종료되거나 발생한 예외를 처리하지 않은 경우

스레드 동기화와 동시성

Java Daemon Thread

- Java는 User Thread와 Daemon Thread 두 가지 스레드를 지원
- 일반적으로 데몬 스레드는 우선 순위가 낮으며 JVM 종료를 막지 않음
  즉 데몬 스레드가 실행되고 있어도 유저 스레드가 모두 작업을 마쳤다면 프로세스는 종료됨
  - 예외로 데몬 스레드에서 `Thread.join()`을 호출하면 프로세스 종료가 차단될 수 있음
- 따라서 I/O 작업 등에 사용하는 것을 권장하지 않음
- 데몬 스레드는 GC, 메모리 해제, 캐시에서 원하지 않는 항목 제거 등과 같은 작업에 적합함
  참고로 대부분의 JVM 스레드는 데몬 스레드

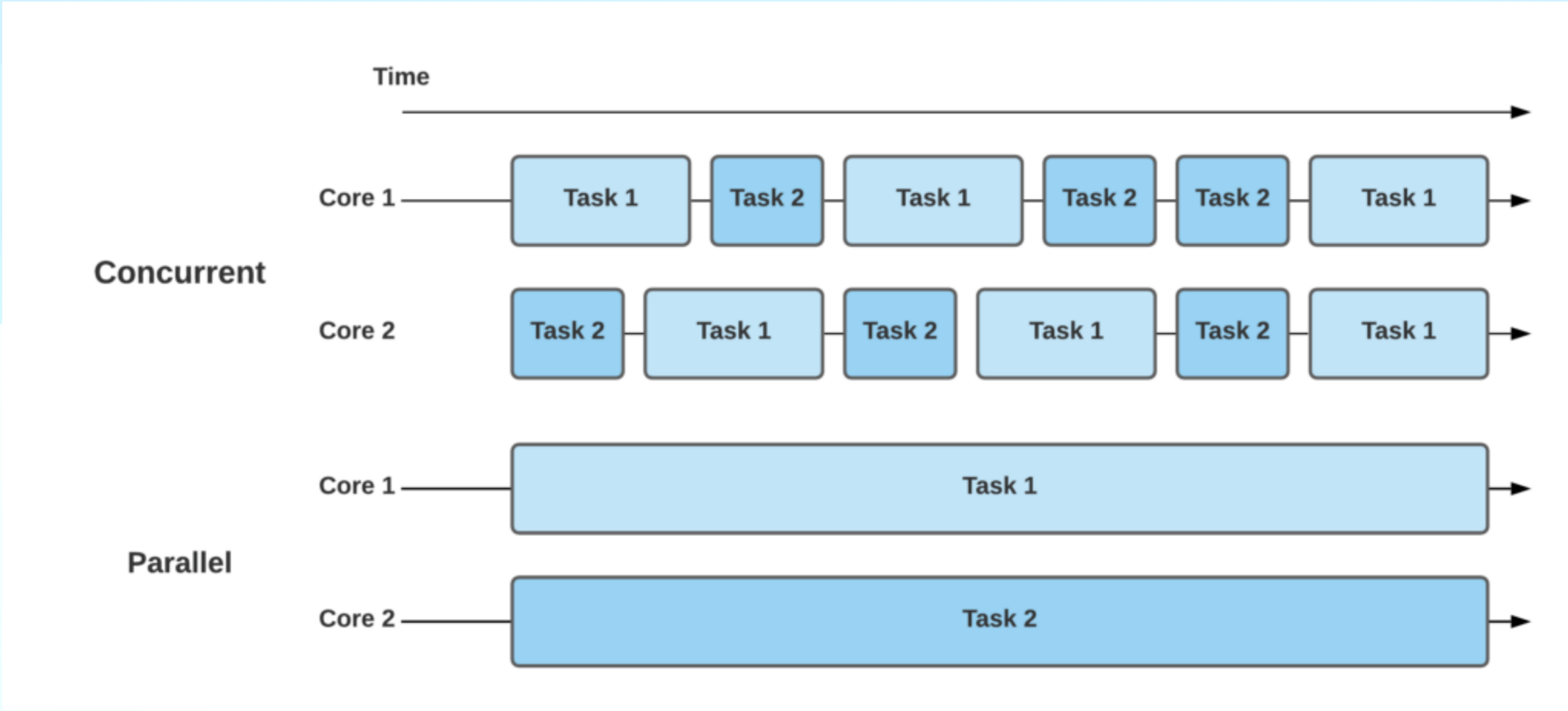
스레드 동기화와 동시성

* Concurrency vs Parallelism

- Concurrency
  - 실제 여러 작업이 겹치는 시점에 실행하는 것 (이전 작업 종료 전 새 작업 시작 가능)
    그러나 실제로 동시에 실행되지 않음
    CPU는 작업 당 시간을 조정, 적절한 컨텍스트 스위칭 처리
  - 동시성 처리에 주요 목표는 CPU의 유휴 시간을 줄이고 최대한 활용하는 것
    처리 작업이 블로킹 되면 점유하고 있던 리소스를 다른 프로세스/스레드가 할당 받음
- Parallelism
  - 동시에 독립적인 작업을 실행하는 것
    즉, 분산 시스템이나 멀티 코어처럼 완전히 격리된 컴퓨터에서 동시에 실행 가능

스레드 동기화와 동시성

* Concurrency vs Parallelism



https://www.baeldung.com/cs/concurrency-vs-parallelism

스레드 동기화와 동시성

Mutex (Mutual Exclusion)

- 상호 배제란 레이스 컨디션을 방지하기 위해서 정의된 동시성 제어 속성 (바이너리 세마포어)
- 한 스레드가 크리티컬 섹션에 액세스하는 동안 다른 스레드는 엑세스할 수 없다는 것을 의미
  공유 리소스(메모리)에 엑세스 시간의 차이를 의미함
  - 여기서 공유 리소스는 멀티 스레드 환경에서 동시에 쓰기 작업을 하려는 데이터 객체를 의미
    이 경우 데이터 불일치가 발생하기 때문에 허용되지 않음
- 뮤텍스는 크리티컬 섹션에 첫번째 프로세스(또는 스레드)가 액세스하면 작업을 완료할 때까지
  다른 프로세스의 액세스를 차단

스레드 동기화와 동시성

Semaphore

- 멀티 스레드가 공유 리소스에 액세스하는 것을 제어하고
  동시성 처리에서 크리티컬 섹션 문제를 해결하기 위한 변수, 추상 데이터 타입
- 즉, 세마포어는 `Synchronization` 원시 타입
- 카운팅 세마포어
  - 임의의 자원 카운트를 허용하는 세마포어

스레드 동기화와 동시성

Java Thread Synchronization

- 결국 스레드 동기화란 멀티 스레드 환경에서 공유된 자원에 같은 타이밍에 액세스할 때
  실행 순서를 제어하는 프로세스, 메커니즘, 기능 등을 포괄적으로 의미함
- 동시성 문제로 인해 데이터의 일관성이 깨지는 상황이 발생 (일관성 오류)
  이러한 것을 메모리 모델, 모니터 등 다양한 기술을 통해 Synchronization 메커니즘으로 극복
- 각 객체/클래스 별로 모니터를 소유하며, 특정 시점에 단 하나의 스레드만 액세스 가능하도록 락을 유지함
  락이 해제될 때까지 다른 스레드는 블락된 상태로 대기
  - 스레드는 모니터에 락을 여러번 걸 수 있음
  - synchronized 블록은 객체 참조를 확인, 모니터 락을 획득/소유하는 것이 성공한 후에 작업을 진행
    실행이 종료되면 락 해제는 자동으로 수행됨
  - synchronized 메서드는 호출될 때 자동으로 락 획득/소유하며 이또한 성공하기 전까지 실행되지 않음
  - synchronized 스태틱 메서드는 클래스 레벨에서 모니터를 획득
    모든 인스턴스 중 하나의 스레드만 획득 가능

스레드 동기화와 동시성

Monitor

- 다음을 충족시키는 스레드의 동기화 메커니즘
  - mutex : 락을 통해 특정 시점에 단 하나의 스레드만 메서드를 실행할 수 있음
  - cooperation : wait-set을 사용해 특정 조건까지 대기하도록 하는 기능
- 이 동기화 메커니즘(기능)을 모니터라고 부르는 이유?
  스레드가 일부 리소스에 엑세스하는 것을 모니터링하기 때문
- 동시성 프로그래밍을 위해 제공하는 주요 특징
  - 한 번에 단 하나의 스레드만 크리티컬 섹션에 상호 배타적으로 액세스 가능
  - 특정 조건을 대기하는 동안 모니터에서 실행되는 스레드는 블락될 수 있음
  - 한 스레드는 대기 조건이 충족되면 다른 스레드들에게 알릴 수 있음

스레드 동기화와 동시성

Java Monitor Lock (Intrinsic Lock)

- Java에서 동시성 처리를 위한 동기화는 모니터 락 또는 고유 락(intrinsic lock)을 통해 구현되며 synchronized 키워드를 사용해서 크리티컬 섹션을 표현함
  - 크리티컬 섹션은 서로 다른 스레드가 동일한 데이터에 엑세스 하는 영역 (코드의 일부)
- 이 고유 락은 생호 배제 액세스를 적용하며 가시성을 위해 발생(호출) 순서를 정의(설정)함
- 모든 객체는 이 고유 락을 갖고 있으며 락을 획득 후 액세스하고 작업을 완료 후에 락을 해제해야 함
  락을 획득하면 고유 락을 `소유` 했다고 판단되며 이후
  다른 스레드에서 해당 인스턴스에 락을 획득하지 못하고 대기해야 함
- 락을 직접 반환하지 않아도 중간에 예외가 발생하면 락이 해제됨
- `static` 필드에 걸리는 락은 모든 인스턴스 예외 없이 단 하나의 인스턴스만 크리티컬 섹션에 엑세스 가능

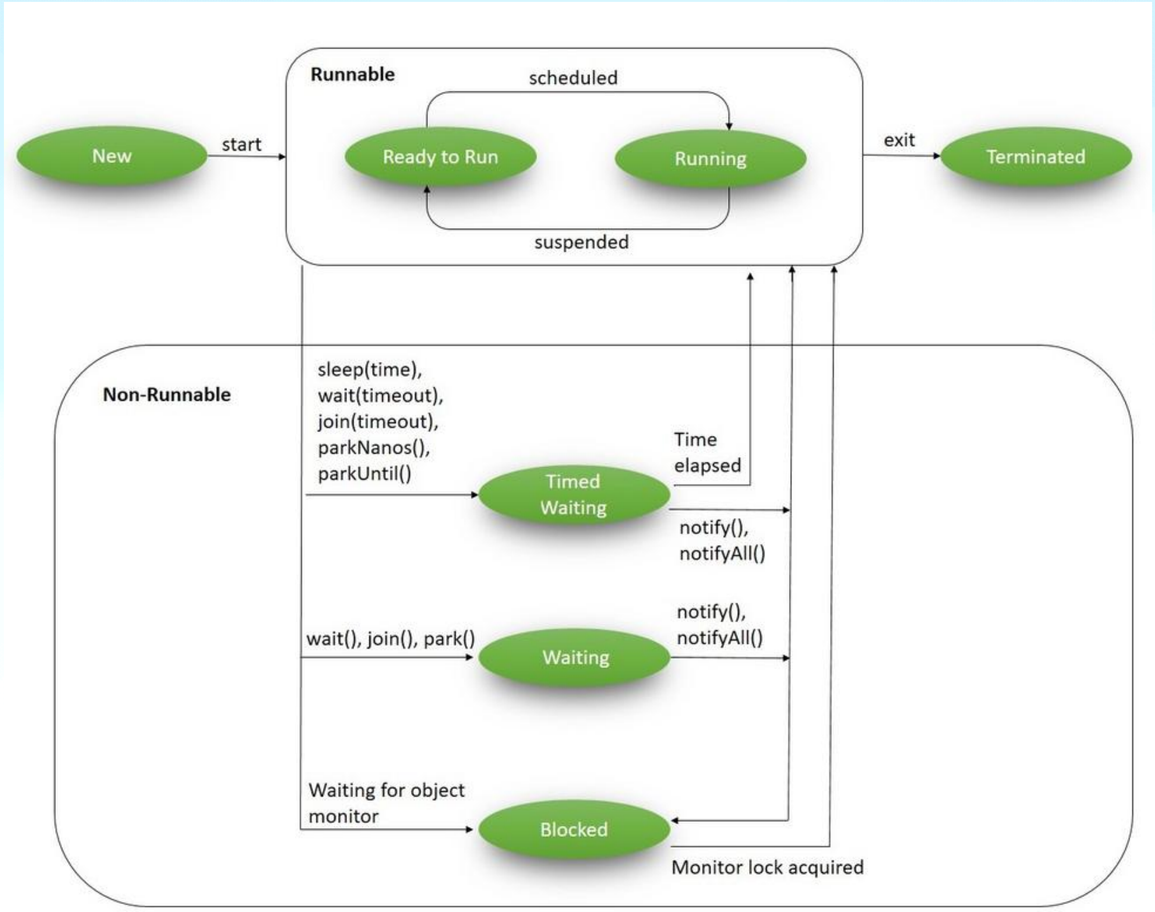
스레드 동기화와 동시성

Java Monitor

- synchronized 메서드 또는 블록
  synchronized 메서드는 간단히 사용할 수 있고, synchronized 블록은 성능 측면에서 효율적
- Java에서는 모니터가 객체(인스턴스)와 클래스 별로 존재해서 인스턴스와 스태틱이 모두 다름
  synchronized 스태틱 메서드 같은 경우는 모든 인스턴스를 통틀어 특정 시점에 하나의 스레드만 실행 가능
- 스레드 모니터 동기화 단계
  - 진입 > 획득 > 소유(유지) > 해제 > 퇴장
- wait 메서드는 다른 스레드가 이 모니터에 들어가 notify를 호출할 때까지
  모니터를 해제, 슬립 모드로 전환하도록 호출한 스레드에게 명령
  - notify 메서드는 특정 객체에서 wait 메서드를 호출한 첫번째 스레드를 깨움

스레드 동기화와 동시성

Java Thread Life Cycle



https://www.baeldung.com/java-thread-lifecycle

스레드 동기화와 동시성

Java Thread Life Cycle State

- NEW
  시작되지 않은 상태
- RUNNABLE
  JVM에서 실행 중인 상태
  하지만 프로세서와 같은 OS의 리소스 할당을 기다리는 상태일 수도 있음
- BLOCKED (Inactive)
  모니터 락을 기다리며 차단된 상태
  - `Object.wait()` 메서드를 호출한 후 `synchronized` 블록, 메서드에 진입/재진입을 위해서 모니터 락 획득을 기다림
    - `Object.wait()`
      - `notify()` 또는 `notifyAll()` 메서드나 인터럽트(중단)되어 깨어날 때까지 대기
- WAITING (Inactive)
  다른 스레드의 특정 작업이 수행되는 것을 무작정 기다리는 상태
  다음 중 하나가 호출되어 해당 상태가 적용됨
  - `Object.wait()` (no time)
    - `Object.notify()` 또는 `Object.notifyAll()`가 호출될 때까지 대기
  - `Thread.join()` (no time)
    - 현재 스레드가 지정된 스레드가 종료될 때까지 대기
  - `LockSupport.park()`
    - 퍼밋이 유효하지 않은 경우 해당 스레드를 비활성화하는 명령

스레드 동기화와 동시성

Java Thread Life Cycle State

- TIMED_WAITING (Inactive)
  지정 시간까지만 다른 스레드의 특정 작업 수행을 기다리는 상태
  다음 중 하나로 호출될 때 해당 상태가 적용됨
  - `Thread.sleep()`
    - 시스템 시간을 기준으로 수면 상태로 변경되지만 모니터의 소유권은 `유지함`
  - `Object.wait()` (with time)
    - `Object.notify()` 또는 `Object.notifyAll()`가 호출될 때까지 대기
  - `Thread.join()` (with time)
    - 현재 스레드에서 지정된 스레드가 종료될 때까지 대기
  - `LockSupport.parkNanos()`
    - 퍼밋이 유효하지 않은 경우 주어진 시간만큼 스레드를 비활성화
  - `LockSupport.parkUntil()`
    - 퍼밋이 유효하지 않은 경우 지정한 시간(데드라인)까지 스레드를 비활성화
- TERMINATED (Inactive)
  스레드가 종료된 상태 (실행을 완료했거나 비정상적으로 종료된 상태)

스레드 동기화와 동시성

* Object 클래스의 wait 메서드

- 현재 스레드를 알림(notify)이나 중단(interrupted), 또는 특정 시간이 경과할 때까지 대기모드로 만듦
  현재 스레드가 이 객체에 모니터 락을 소유하고 있어야 함
- 현재 스레드를 대기 셋에 넣고나서 해당 객체에 대한 동기화를 못하게 함
  해당 객체에 대한 락만 해제하고, 현재 스레드가 동기화할 수 있는 다른 객체는 그대로 락이 걸림
- 휴면 상태로 접어든 스레드가 활성화되는 경우
  - 다른 스레드가 notify 메서드를 호출한 경우 (대기 셋 안에 있는 스레드 중 임의로 선정, 이때 타겟이 된 경우)
  - 다른 스레드가 notifyAll 메서드를 호출한 경우
  - 다른 스레드의 의해 해당 스레드가 인터럽트된 경우
  - 지정된 시간이 경과한 경우
- 깨어난 스레드는 대기 셋에서 제거되고 다시 활성화 상태가 되고
  해당 객체에 동기화 권한을 획득하기 위해 다른 스레드와 경쟁하고 권한을 획득한다면 wait 메서드 호출 전과 같은 상태가 됨
- 간헐적이지만 알림, 중단, 시간 초과 없이 스레드가 활성화될 수 있음 (Spurious Wakeup)
- 대기 상태가 되기 직전이나 대기 상태일 때 다른 스레드에 의해 인터럽트가 발생하면 InterruptedException이 발생함
  하지만 이 예외는 스레드에서 해당 객체의 락을 소유한 상태로 복구될 때까지 던져지지 않음
  따라서 상태가 복구되면 해당 예외가 발생하고 이때 스레드는 대기 상태에서 벗어남

스레드 동기화와 동시성

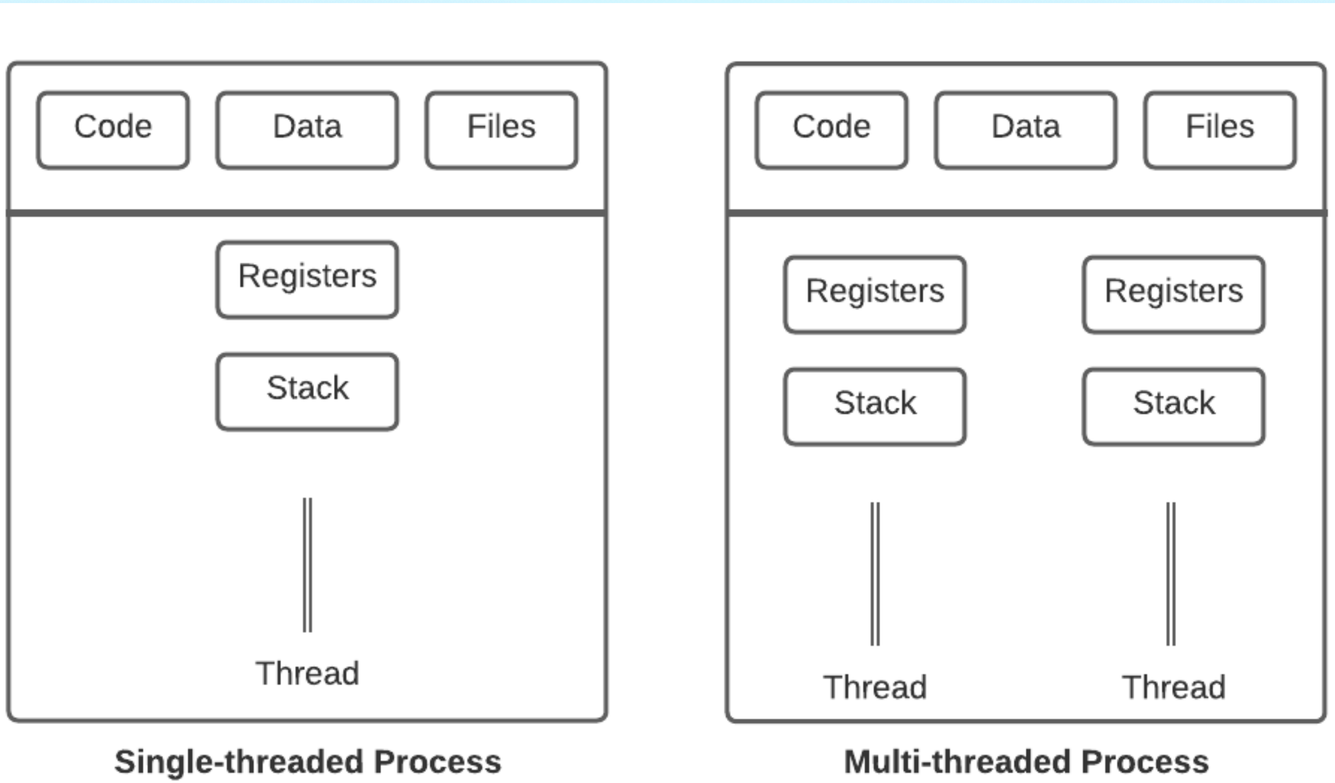
* Object 클래스의 notify 메서드

- 객체의 모니터에서 대기 상태의 스레드 중 하나(임의 선택)를 깨워 활성화 상태가 되도록 선택
- 깨어난 스레드는 다른 스레드와 해당 객체에 대한 모니터 획득을 위해 경쟁함
  하지만 락 획득 측면에서 다른 스레드보다 특권을 가지고 있지는 않음
- 스레드가 해당 객체의 모니터의 락을 획득하는 조건
  - 해당 객체의 `synchronized` 인스턴스 메서드를 실행하는 경우
  - 해당 객체의 `synchronized` 구문의 바디를 실행하는 경우
  - `Class` 타입의 객체에 대해 해당 클래스의 `synchronized` 스태틱 메서드를 실행하는 경우

[2] 멀티 스레드 환경에서 발생하는 스레드 동기화 문제

멀티 스레드 환경에서 발생하는 스레드 동기화 문제

Multi Thread



https://www.baeldung.com/cs/process-vs-thread

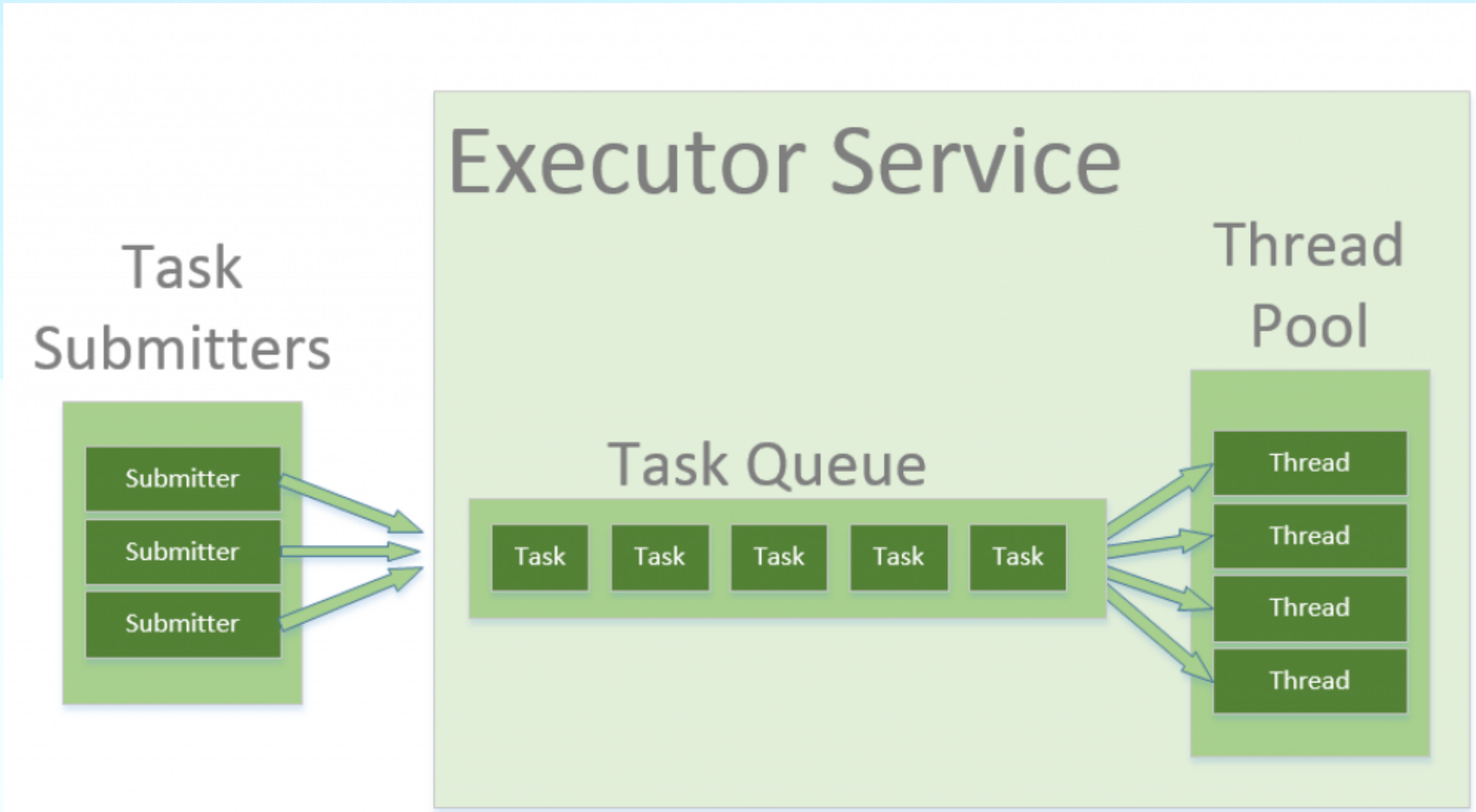
멀티 스레드 환경에서 발생하는 스레드 동기화 문제

Thread Pool

- Java 스레드는 시스템(OS) 스레드와 매핑됨
  따라서 별다른 제어 없이 무분별하게 생성해서 사용하면 리소스 낭비가 심해짐
- OS는 병렬 처리를 위해 스레드 간 컨텍스트 스위칭도 수행
- 단순하게 본다면 스레드가 많은 경우 작업을 나눠서 수행하기 때문에 소요되는 시간이 줄어듦
  하지만 실제로 많은 스레드를 사용한다고 해서 무조건 빠르지만은 않음
- 결론적으로 멀티 스레드를 활용해 병렬 작업을 하는 것이 효율적이기 때문에
  스레드를 관리해주는 스레드 풀 패턴을 활용
- 스레드 풀 사용시엔 병렬 처리 형태로 동시에 수행될 코드를 작성한 후 제출하는 방식으로 사용
- 스레드 풀을 활용하면 스레드 수와 생명 주기를 쉽게 제어할 수 있으며
  실행할 작업들의 스케줄링이 가능하며 큐에 보관할 수도 있음

멀티 스레드 환경에서 발생하는 스레드 동기화 문제

Thread Pool



https://www.baeldung.com/thread-pool-java-and-guava

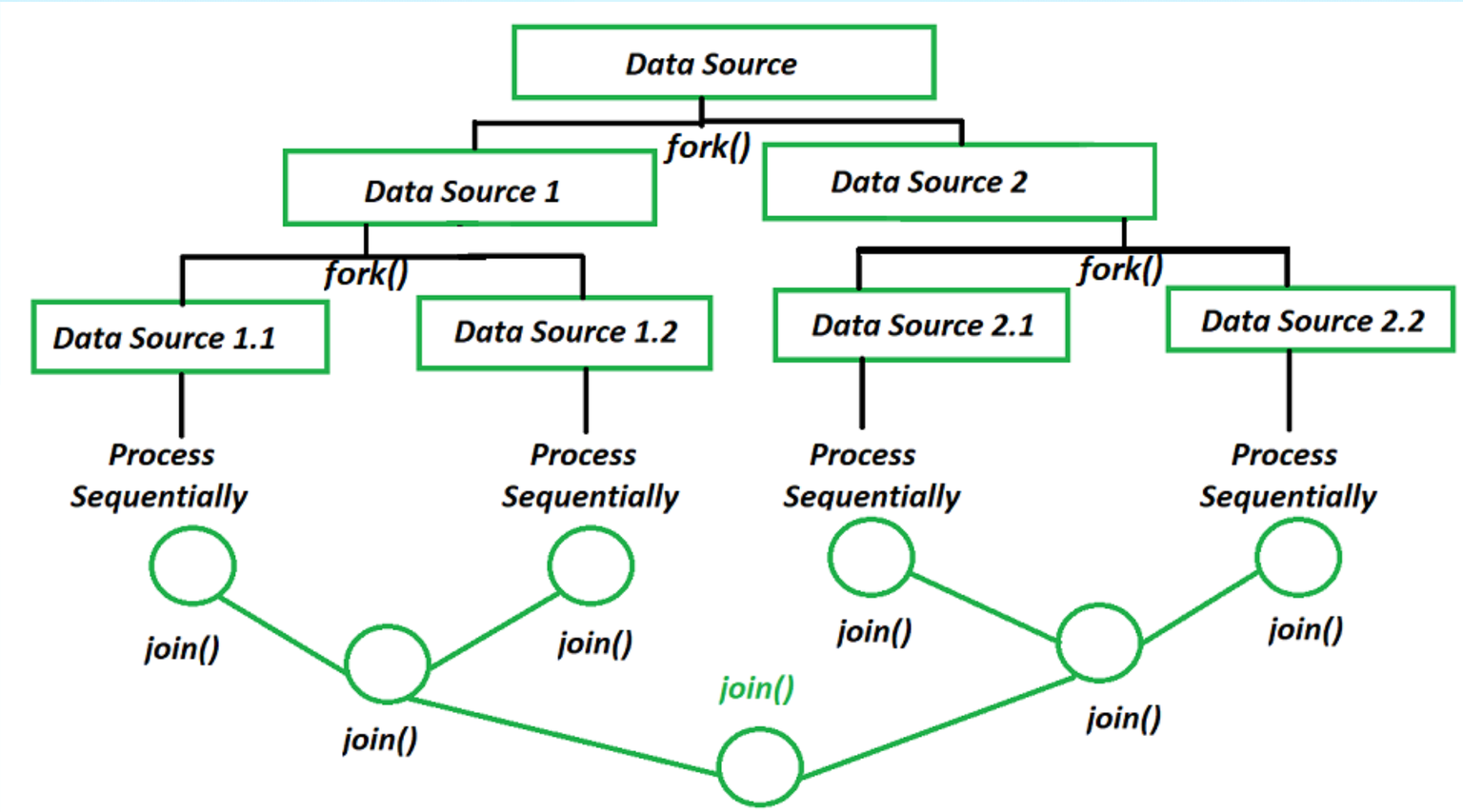
멀티 스레드 환경에서 발생하는 스레드 동기화 문제

Thread Pool Usage - java.util.concurrent.Executors

- newCachedThreadPool
  - 기존 스레드를 재활용하여 필요할 때만 스레드를 생성하는 스레드 풀 생성
- newFixedThreadPool
  - 무제한 크기의 작업 큐에서 가져온 작업을 고정된 스레드 수만큼만 사용하며 처리하는 스레드 풀 생성
- newScheduledThreadPool
  - 나중에 실행할 작업과 주기적으로 실행할 작업을 스케줄링(예약)할 수 있는 스레드 풀 생성
- newSingleThreadExecutor
  - 무제한 크기의 작업 큐에서 가져온 작업을 단일 워커 스레드로 처리하는 Executor를 생성
- newSingleThreadScheduleExecutor
  - 나중에 실행할 작업과 주기적으로 실행할 작업을 스케줄링(예약)할 수 있는 단일 스레드 Executor를 생성
- newWorkStealingPool
  - 사용 가능한 프로세서 수를 병렬 처리에 필요한 수준으로 사용하는 스레드 풀 생성
    할당량이 끝난 스레드는 다른 스레드의 작업을 뺏어와 수행함

멀티 스레드 환경에서 발생하는 스레드 동기화 문제

Thread Pool Usage - ForkJoinPool computation Model



https://www.geeksforgeeks.org/difference-between-fork-join-framework-and-executorservice-in-java/

멀티 스레드 환경에서 발생하는 스레드 동기화 문제

Thread Pool Usage - ForkJoin Framework

- JDK 7 버전에 도입된 프레임워크로 분할 정복 방식으로 병렬 처리 속도를 개선
  `fork` 작업으로 적정 수준으로 데이터 소스의 양을 나눠 하위 작업들로 분할
  분할된 작업을 병렬 처리로 수행
  `join` 작업으로 분할되었던 작업들을 재귀적으로 병합
- ForkJoinPool
  해당 프레임워크는 스레드 풀로 ForkJoinPool을 사용함
  ExecutorService 구현체로 스레드 관리, 스레드 풀 상태/성능 등 정보 확인 가능
  워커 스레드는 한 번에 하나의 작업만 처리 가능하지만 추가로 스레드를 생성하지 않음
- 스레드 별로 작업을 저장할 수 있는 Deque로 작업을 관리
  워커 스레드는 맨앞에서 작업을 꺼내지만
  다른 스레드의 작업을 가져올 땐 맨뒤에서 꺼내거나 전체 작업 큐에서 꺼내옴
- ForkJoinTask<V>
  ForkJoinPool 내에서 실행되는 작업의 기본 타입
  RecursiveAction 또는 RecursiveTask<V> 중 하나를 택해 상속받아야 함
- 사용 시 주의 사항
  가능한 적은 수의 스레드 풀을 사용할 것
  별도의 커스터마이징이 필요 없다면 커먼 스레드 풀을 사용할 것
  ForkJoinTask를 하위 작업으로 분할하기 위한 합리적인 임계값을 사용할 것
  ForkJoinTask에서 블락킹 상황을 피할 것

멀티 스레드 환경에서 발생하는 스레드 동기화 문제

Thread Pool Usage - Parallel Streams

- Stream API는 데이터 소스를 감싸 대량의 데이터의 핸들링을 더욱 편리하게 해주는 프레임워크
- 패러럴 스트림은 ForkJoin 프레임워크와 커먼 스레드 풀을 사용
  커먼 풀의 스레드 수는 `(프로세서 코어 수 - 1)` 이지만 직접 지정할 수 있음
  - `-D java.util.concurrent.ForkJoinPool.common.parallelism=<pool size>`
  - 커먼 풀을 사용한다는 것은 똑같이 커먼 풀을 사용하는 모든 곳에 영향을 미칠 수 있기 때문에 주의할 것
- Custom Thread Pool을 사용할 수 있으나 오라클에서는 커먼 풀을 사용하는 것을 권장함

멀티 스레드 환경에서 발생하는 스레드 동기화 문제

동기화 문제

- Race Condition
  다른 이벤트 순서 등에 따라 시스템 상태가 바뀌는 현상
  여러 스레드가 공유 데이터에 접근할 때 발생, 데이터 일관성 에러를 야기함
- Deadlock
  획득한 리소스의 락을 걸고 다른 프로세스/스레드의 리소스를 무한히 기다리면서 아무것도 진행 못하는 상황
- LiveLock
  실제적으로 비활성 상태는 아니나 서로의 상태가 계속 엇갈려 프로세스 자체는 비활성 상태인 상황
  데드락은 상태를 변경하지 않는데 반해 라이브락은 리소스의 상태는 계속 변경하는 상황
- Starvation
  그리디 스레드가 장기간 공유 리소스를 점유하여 장기간 리소스에 대한 액세스 권한을 얻지 못해 진행을 못하는 상황
  데드락, 라이브락 및 기타 이유로 해당 상태가 발생함
- Priority Inversion
  우선 순위가 높은 작업이 낮은 작업으로 대체되어 공유 리소스를 기다리는 상황
- False Sharing
  분산 시스템에서 잘못된 캐싱 메커니즘에 의해 발생하는 성능 저하 상황
  공유되는 데이터를 코어 캐시에 의해 잘못 읽거나 필요 없는 잘못된 참조로 인한 성능 저하

[3] Java에서 스레드 동기화를 위해 제공하는 기능

Java에서 스레드 동기화를 위해 제공하는 기능

종류

- synchronized
- volatile
- Atomic Variables
- Semaphore
  TimedSemaphore
- ReentrantLock
  ReentrantReadWriteLock
  StampedLock
- ConcurrentHashMap
- ThreadLocal
- CountDownLatch
- CyclicBarrier

Java에서 스레드 동기화를 위해 제공하는 기능

synchronized

- 멀티 스레드 환경에서 공유 리소스에 대한 레이스 컨디션이 발생할 때
  동기화를 위해 지원하는 메커니즘(기술)
- synchronized 키워드는 다음 3곳에 사용 가능
  인스턴스 메서드
  스태틱 메서드
  코드 블록
- A 클래스의 특정 synchronized 인스턴스 메서드 호출 시 락을 획득한 스레드를 제외하고
  해당 인스턴스에 접근하려는 모든 스레드는 모든 synchronized 인스턴스 메서드에 액세스하지 못하고 대기
- A 클래스의 synchronized 스태틱 메서드 호출 시 락을 획득한 스레드를 제외, 인스턴스 불문
  모든 스레드에서 모든 synchronized 스태틱 메서드에 엑세스하지 못하고 대기
- 따라서 락을 획득하지 못한 스레드는 블록, 서스펜드 상태가 됨
  이로 인해 스레드 컨텍스트 스위칭 등으로 리소스가 많이 들어 성능 이슈가 존재하기 때문에 주의할 것

Java에서 스레드 동기화를 위해 제공하는 기능

* synchronized - 주의 사항

- 외부에서 락을 걸지 않을 객체들에게만 적용할 것
- String 리터럴은 풀링 되기 때문에 가급적 적용하지 말 것
  적용해야 한다면 new 키워드와 함께 사용할 것
- Boolean 리터럴 또한 가급적 적용하지 말 것
  이 또한 인스턴스 공유 때문
- 박싱된 원시타입 또한 적용하지 말 것 (클래스를 생성해서 적용할 것)
  JVM이 바이트로 표현할 수 있는 값을 캐싱하여 공유하고 있기 때문
- this 키워드로 Synchronized를 적용하지 말 것 (락 객체를 따로 구현해 활용)
  JVM은 클래스가 this로 Synchronized를 적용할 때 고유 락을 사용함

Java에서 스레드 동기화를 위해 제공하는 기능

volatile

- 캐싱과 재정렬(reordering) 같은 최적화로 인해 동시 컨텍스트에서 문제가 발생할 수 있는데
  이런 문제를 메모리 순서 제어로 해결하기 위해 지원하는 기능, 메커니즘
- 메모리 가시성, 재정렬이 부족해서 발생하는 문제가 있음
  - 현대에는 기술적으로 성능 최적화를 위해 쓰기 작업의 결괏값을 버퍼에 두고 나중에 메인 메모리에 적용
    즉 어떤 값을 업데이트할 때 다른 스레드가 무엇을 볼지에 대한 보장이 없음
  - 재정렬로 인해 실제 처리된 순서가 아닌 다른 순서로 쓰기 값을 볼 수 있음
- 이런 문제가 발생하는 이유
  - 프로세서는 처리 순서가 아닌 다른 순서로 버퍼를 플러시 할 수 있음
  - 프로세서는 순차적이지 않은 기술을 적용할 수 있음
  - JIT 컴파일러는 최적화를 수행하기 위해 재정렬을 함
- 그렇다고 해서 volatile 키워드가 스레드를 동기화 하는 해결책은 아님

Java에서 스레드 동기화를 위해 제공하는 기능

Atomic Variables

- Java는 스레드 동기화 문제를 해결하기 위해 원자적 연산(CAS)을 처리하는 Atomic Variables를 지원함
  AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference 등과 같은 클래스를 제공함
- 원자적 연산 (Atomic Operation)
  CAS(Compare And Swap) 같은 알고리즘은 로우 레벨의 인스트럭션을 활용함
- 변경 값을 업데이트 하기전에 기존 값이 그 전에 가져갔던 값과 동일한지를 확인 후 처리
  만약 다르다면 아무것도 처리하지 않음
- 중요한 건 스레드가 서스펜드 상태가 되지 않아 성능적으로 덜 영향을 받음
  synchronized 키워드를 통한 동기화는 액세스 권한이 없는 스레드는 블록, 서스펜드 상태가 됨

Java에서 스레드 동기화를 위해 제공하는 기능

Semaphore

- 카운팅 세마포어, 공유되는 리소스에 동시성 처리를 제한, 지원하기 위한 클래스
  스레드 수 제한 가능
- 세마포어는 각 스레드가 원하는 요소를 얻기 전에 허가를 얻어 해당 요소를 사용할 수 있음을 보장함
- 상황에 따라 바이너리 세마포어(뮤텍스)로도 사용 가능
  이때 바이너리 세마포어는 다른 락 구현과 달리 스레드에 의해 락이 해제될 수 있음
- fairness 파라미터를 false로 주면 락 획득 순서를 보장하지 않음 (새치기 가능)
  true로 설정하면 FIFO 방식으로 호출이 처리된 순서대로 락 획득 요청을 시도

Java에서 스레드 동기화를 위해 제공하는 기능

TimedSemaphore

- 아파치 재단에서 제공하는 라이브러리로 지정 시간 프레임에 대한 권한을 제공하는 세마포어
- 앞선 세마포어 클래스와 거의 유사하나 release 메서드를 제공하지 않음
  지정된 시간 프레임이 끝나면 모든 퍼밋이 자동으로 해제됨
- 지정된 모든 퍼밋이 점유되고 있다면 액세스를 시도하는 스레드는 블록됨
- 프로세스의 부하를 인위적으로 제한할 때 사용되기도 함
  예를 들어 백그라운드에서 DB 쿼리를 발생시키는 경우 너무 많은 리소스를 사용하지 않아야 함
  이런 경우 타임드 세마포어로 특정 시간내에 지정된 쿼리만 실행하도록 활용할 수 있음

Java에서 스레드 동기화를 위해 제공하는 기능

ReentrantLock

- 암묵적인 모니터 락과 유사하지만 확장된 기능이 있는 재진입 뮤텍스 락 (가장 기본적인 락 클래스)
- ReentrantLock은 마지막으로 락 획득에 성공한 스레드가 권한 소유
- 생성자 파라미터로 fairness값을 받는데 이 값이 true라면 스레드의 대기 순서대로 엑세스 권한이 부여됨
  반대로 값이 false라면 엑세스 권한 순서를 보장하지 않음
- 락의 소유권을 획득하는 순서가 공정해도 스레드 스케줄링이 공정하다는 보장은 없음

Java에서 스레드 동기화를 위해 제공하는 기능

ReentrantReadWriteLock

- 다수의 스레드에서 읽기 작업과 하나의 스레드에서 쓰기 작업을 할 때 유용한 재진입 락 뮤텍스
  읽기 작업이 많고 읽기 작업 사이의 데이터 일관성이 문제가 되지 않는 경우 적합
  - 반대로 읽기, 쓰기 작업이 균등하거나, 데이터 일관성이 중요한 경우에는 ReentrantLock이 적합함
- 락 획득 순서에 대해 Non-fair 모드와 Fair 모드가 존재
  일반적인 경우라면 Non-fair가 더 빠름
- Reentrancy 락의 경우 쓰기, 읽기 스레드 모두 재진입할 수 있음
  하지만 Non-reentrant 리더는 쓰기 작업 스레드의 락이 풀려야 재진입 가능
  또한 읽기 락이 걸려 있는 경우 쓰기 작업은 락을 획득할 수 있지만 반대는 불가능함
- 쓰기 작업 락에서 읽기 작업 락으로 다운 그레이드가 가능하지만 반대는 불가능함
- 쓰기 작업 락에 대해서만 컨디션(Condition) 객체를 제공, 스레드 동기화 관리
  반대로 읽기 작업 락은 컨디션 객체를 제공하지 않음
- 잠금 상태에 대한 모니터링 제공하는 메서드 제공
- 직렬화-역직렬화를 거친 후 락은 잠금 해제 상태

Java에서 스레드 동기화를 위해 제공하는 기능

* Condition

- 컨디션 인터페이스는 Object의 모니터 메서드(wait, notify 등)를 `락` 구현을 통해 개별 객체로 나눠 객체당 여러 wait-set을 갖기 위한 수단으로 사용되는 인터페이스
- 다시 말해서 스레드의 상태(실행, 중지, 대기 등)를 위한 메커니즘의 일부이며 wait-set을 통해 동시성 제어를 가능하게 함
- synchronized 블록과 메서드를 대신해 락 클래스를 사용하는 경우 컨디션 인터페이스는 모니터 메서드를 대신함

Java에서 스레드 동기화를 위해 제공하는 기능

StampedLock

- Java에서 지원하는 동시성 처리를 위한 락 중 하나로 `ReentrantReadWriteLock`의 향상된 버전
  총 3가지 모드 제공 (Writing, Reading, Optimistic Reading)
  - Writing
    쓰기 작업 시 다른 스레드에서 엑세스하지 못하는 선점(독점) 락이며
    이때 반환되는 `stamp`는 락 해제 시 사용됨
    쓰기 락인 경우 모든 읽기 작업을 수행할 수 없음
  - Reading
    읽기 작업을 위한 비선점 락이며 동시에 다른 읽기 작업들이 수행될 수 있음
    이때 반환되는 `stamp`는 락 해제 시 사용됨
  - Optimistic Reading
    Reading에 비해 가벼운 락이며 Writing 락이 아닌 경우에만 사용 가능
    이때 반환되는 `stamp`는 이후 유효성 검사에 사용됨
    성능 향상을 위한 모드로 중간에 데이터 쓰기 작업이 일어나는 경우 주의가 필요함
- 위 3가지 모드는 조건부로 모드를 변환할 수 있음
- 재진입이 불가능하기 때문에 락을 다시 거는 메서드 등에서 호출하는 것을 지양할 것
- 직렬화-역직렬화를 거친 후 락은 잠금 해제 상태
- 소유권 개념이 없기 때문에 다른 스레드에서 락을 해제하거나 변환할 수 있음

Java에서 스레드 동기화를 위해 제공하는 기능

* Semaphore vs ReentrantLock

- Mechanism
  - 바이너리 세마포어는 일종의 시그널 메커니즘
    리엔트런트락은 락 기반 메커니즘
- Ownership
  - 바이너리 세마포어는 소유자인 스레드가 없음
  - 리엔트런트락은 마지막 락을 획득한 스레드가 소유자
- Nature
  - 바이너리 세마포어는 재진입이 불가능, 크리티컬 섹션에 다시 엑세스할 수 없고 그 외는 데드락이 발생
  - 리엔트런트락은 동일 스레드가 여러 번 진입 가능 (예를 들어 특정 스레드에서 메서드 재귀 호출하는 형태 등)
- Flexibility
  - 바이너리 세마포어는 락, 데드락 복구, 사용자 지정 구현 등 더 높은 수준이 동기화 메커니즘
  - 리엔트런트락은 고정된 락 메커니즘만을 제공하는 저수준 동기화
- Modification
  - 바이너리 세마포어는 모든 프로세스에서 사용 가능한 대기, 시그널 등을 지원
  - 리엔트런트락은 리소스에 락/언락을 한 동일 스레드에만 변경할 수 있음
- Deadlock Recovery
  - 바이너리 세마포어는 비소유 해제 메커니즘을 제공, 모든 스레드는 데드락 복구에 대한 허가를 해제할 수 있음
  - 리엔트런트락은 데드락 복구가 어려움, 진입한 소유자 스레드가 슬립 또는 무한 대기모드라면 데드락 발생

Java에서 스레드 동기화를 위해 제공하는 기능

ConcurrentHashMap

- HashTable을 보완한 객체이며 HashMap과는 다르게 key/value에 null을 허용하지 않음
  HashTable은 synchronized로 컬렉션 전체의 락을 걸기 때문에 성능적인 이슈가 있음
- HashMap과의 주요 차이점은 읽기/쓰기 작업에 대한 동시성 처리 지원
  읽기 작업은 키에 대해 블록을 걸지 않음
  쓰기 작업은 블록되어 맵 엔트리 레벨에서 다른 쓰기 작업을 차단함 (해당 항목만 잠금)
- HashTable과 동일한 API(메서드)를 지원
  - get 메서드는 synchronized 메서드가 아니며 따라서 put, remove 작업과 동시에 처리될 수 있음
    따라서 동시 작업 직전 마지막 최신 값을 기준으로 처리됨
  - put 메서드는 로직 중간에 synchronized 블록을 사용해 구현되어 있음

Java에서 스레드 동기화를 위해 제공하는 기능

ThreadLocal

- 스레드로컬 변수를 제공하는 클래스
  스레드 별로 고유하게 보관되며 각 스레드는 복사본 변수를 갖게됨
- 스레드로컬 인스턴스는 객체의 상태와 스레드를 연결하려는 클래스의 프라이빗 스태틱 필드로 선언됨
- 각 스레드가 살아 있고, 스레드로컬 인스턴스에 액세스가 가능하면
  스레드로컬에서 복사본에 대한 참조가 유지됨
- 스레드로컬을 스레드 풀과 함께 사용할 때는 주의할 것
  - 일반적으로 스레드 풀은 스레드를 제거하지 않는데
    스레드로컬에 저장된 값은 명시적으로 지우지 않으면 스레드 생명주기에 의존해서 그대로 보존됨
    이 상태로 스레드가 스레드 풀에 반환된 후 다른 요청에 의해 해당 스레드가

Java에서 스레드 동기화를 위해 제공하는 기능

CountDownLatch

- 다른 스레드에서 수행 중인 작업이 완료될 때까지 여러 스레드를 대기하는 동기화 하는 기술(도구)
- 주어진 값으로 초기화되며 카운트가 0에 도달할 때까지 await 메서드는 계속 차단됨
  하지만 일회성 처리 (재설정이 필요하면 CyclicBarrier 활용)
- 주요 메서드
  - await 메서드는 래치 카운트가 0이 될 때까지 현재 스레드 차단
  - countDown 메서드는 래치 카운트를 하나 감소

Java에서 스레드 동기화를 위해 제공하는 기능

CyclicBarrier

- 여러 스레드가 각각 특정 배리어 지점에 도달하는 것을 대기, 동기화 하는 기능, 메커니즘
  CountDownLatch와 다르게 카운트를 재설정할 수 있음
- 멀티 스레딩 환경에서 스레드가 서로를 기다리는 환경에서 유용하며
  대기 중인 스레드가 재사용될 수 있기 때문에 도달하는 배리어를 순환(cyclic)이라고 함
- interruption, failure, timeout 등으로 배리어를 먼저 벗어나게 되면 대기 중인
  다른 모든 스레드도 비정상적으로 종료됨 (BrokenBarrierException, InterruptedException)

Java에서 스레드 동기화를 위해 제공하는 기능

* CountDownLatch vs CyclicBarrier

- 동작
  - CountDownLatch
    래치가 0이 될 때까지 다른 스레드가 래치를 카운트다운 하는 동안 스레드가 대기하는 구조
  - CyclicBarrier
    모든 스레드가 배리어에 도착할 때까지 스레드 그룹이 함께 대기하는 재사용 가능한 구조
- CountDownLatch 는 Tasks 기반, CyclicBarrier는 Threads 기반 동작
  - CountDownLatch (태스크 수를 유지)
    하나 이상의 스레드가 여러 작업이 완료될 때까지 대기하도록 허용
    동일 스레드에서 래치를 여러 번 줄일 수도 있음
  - CyclicBarrier (스레드 수를 유지)
    여러 스레드가 서로 대기하도록 허용
    대기 중인 스레드 자체가 배리어이며 한 스레드에서 await를 여러 번 호출해도 무의미 함
- 재사용성
  - CountDownLatch
    카운트 재설정이 안됨
  - CyclicBarrier
    배리어에 도달하면 값이 재설정 됨

[4] JDK 19에 추가된 가상 스레드

JDK 19에 추가된 가상 스레드

Virtual Thread

- 가상 스레드는 처리량이 많은 앱을 작성, 유지, 관찰하는 경량 스레드
  - 기존의 Java 스레드는 OS의 스레드와 매핑되는 리소스가 많이 드는 스레드
    따라서 OS 스펙에 영향을 많이 받았으며 컨텍스트 스위칭 등은 성능 부하를 가져옴
- 간단하게 요청별로 스레드가 생성되는 형태로 구현된 서버앱에서 하드웨어를 거의 최적의 상태로 활용, 확장
- 기존의 Java 스레드 API를 최대한 수정하지 않고 사용할 수 있음
- 기존 JDK 도구를 통해 가상 스레드의 디버깅, 프로파일링을 쉽게 할 수 있음
- 하지만 기존의 스레드를 가상 스레드로 대체하기 위한 것은 아님

실습

아하! 모먼트

- [내가] 코드 리뷰 문화 도입과 개선을 위해 시도한 방법

[내가] 코드 리뷰 문화 도입과 개선을 위해 시도한 방법

코드 리뷰

- 코드 리뷰란 무엇인가?
- 필요한 이유는 무엇이고 왜 해야 할까?
- 필요 없는 이유는 무엇일까? (뭐가 불편할까?)

[내가] 코드 리뷰 문화 도입과 개선을 위해 시도한 방법

코드 리뷰 문화 장단점

- 장점
  - Shift Left (버그, 장애, 문제점 등을 조기에 발견할 수 있어 조치가 빨라짐)
  - 집단 지성을 통해 다양한 해결책을 교류, 프로덕션 퀄리티 향상
  - 팀원들의 실력 상향 평준화
  - 팀내 작업에 대한 히스토리를 파악 가능하며 기록으로 남길 수 있음
- 단점
  - 개발 생산성 저하
  - 장점에 대한 무지
  - 코드 리뷰 없이도 큰 문제 없이 업무 수행
  - 코드 리뷰 도입 실패 경험

[내가] 코드 리뷰 문화 도입과 개선을 위해 시도한 방법

코드 리뷰 문화 현실

- 코드 리뷰를 하고 있는 회사는 얼마나 될까?
- 팀을 이끌어가는 시니어분들 중 코드 리뷰를 제대로 경험하신 분은?
- 신입, 주니어분들 중 코드 리뷰를 제대로 받은 적은 있으신 분은?
- 프로페셔널한 사람들이 얼마나 될까?

[내가] 코드 리뷰 문화 도입과 개선을 위해 시도한 방법

코드 리뷰 문화에 대해 냉정하게 생각해보기

- 하다가 안하면 어떤 점이 불편한가?
  - 내 실력에 대한 불신과 걱정
  - 팀원 간 소통의 부재
  - 설계, 코드 리뷰로 인한 장점들이 사라짐
  - 개인적으로 성장의 느낌을 받지 못함
- 안하다가 하면 어떤 점이 불편한가?
  - ?

[내가] 코드 리뷰 문화 도입과 개선을 위해 시도한 방법

코드 리뷰 문화 도입을 위한 노력

- 코드 리뷰 문화를 도입하기 위해 팀원들 설득하기
  - 도입해야 하는 이유를 정리해 전파하기
  - 도입으로 인한 단점을 개선할 방법 구상하기
- 코드 리뷰를 이끌 리더 구하기
- 팀원들의 실력에 대한 치부를 이겨낼 방법 찾아보기
- 프로토타입을 만들어 보여주기
- 기존 업무 프로세스와의 격차 줄이기

[내가] 코드 리뷰 문화 도입과 개선을 위해 시도한 방법

코드 리뷰 문화 개선을 위한 노력

- 모듈, 패키지, 도메인 등 특정 기준으로 나눠 담당자를 정하기
  - 오너십을 갖게하고 리뷰어, 어사이니를 정해보자
- 코드 리뷰를 위한 시간을 확보하기
- PR 템플릿 만들기
- 린터, 포맷터 등을 최대한 활용하기
- 클린코드, 프로그래밍 패러다임, 테스트 코드, 아키텍처 등을 깊게 파기
- 레이블, 태그 등을 적절히 활용하기

[내가] 코드 리뷰 문화 도입과 개선을 위해 시도한 방법

코드 리뷰 문화 돌아보기

- 모두가 같은 목적으로 같은 방향을 바라보고 있는가?
- 코드 리뷰 말고 다른 방법은 없을까?

참고 및 출처

- https://en.wikipedia.org/wiki/Computer_program
- https://en.wikipedia.org/wiki/Process_(computing)
- https://en.wikipedia.org/wiki/Computer_multitasking
- https://en.wikipedia.org/wiki/Thread_(computing)
- https://www.baeldung.com/cs/process-vs-thread
- https://www.geeksforgeeks.org/difference-between-process-and-thread/
- https://www.baeldung.com/linux/process-vs-thread
- https://en.wikipedia.org/wiki/File:Concepts-_Program_vs._Process_vs._Thread.jpg
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html
- https://www.geeksforgeeks.org/java-threads/
- https://www.baeldung.com/cs/concurrency-vs-parallelism
- https://www.baeldung.com/java-daemon-thread

- https://www.geeksforgeeks.org/daemon-thread-java/
- https://en.wikipedia.org/wiki/Mutual_exclusion
- https://en.wikipedia.org/wiki/Semaphore_(programming)
- https://www.baeldung.com/cs/what-is-mutex
- https://www.baeldung.com/cs/semaphore
- https://www.baeldung.com/java-mutex
- https://en.wikipedia.org/wiki/Lock_(computer_science)
- https://www.baeldung.com/cs/monitor
- https://en.wikipedia.org/wiki/Synchronization_(computer_science)
- https://docs.oracle.com/javase/specs/jls/se17/html/jls-17.html
- https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html
- https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

- https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.State.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#wait(long)
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html#join(long)
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/LockSupport.html#park()
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html#sleep(long)
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/LockSupport.html#parkNanos(java.lang.Object,long)
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/LockSupport.html#parkUntil(java.lang.Object,long)
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#notify()
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#notifyAll()
- https://www.baeldung.com/java-runtime-halt-vs-system-exit
- https://www.baeldung.com/java-thread-lifecycle

- https://www.baeldung.com/java-start-thread
- https://www.baeldung.com/java-synchronization-bad-practices
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html#newCachedThreadPool()
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html#newFixedThreadPool(int)
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html#newScheduledThreadPool(int)
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html#newSingleThreadExecutor()
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html#newSingleThreadScheduledExecutor()
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html#newWorkStealingPool()
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ForkJoinPool.html
- https://www.baeldung.com/java-executor-service-tutorial

- https://www.baeldung.com/java-fork-join
- https://www.baeldung.com/java-work-stealing
- https://www.geeksforgeeks.org/difference-between-fork-join-framework-and-executorservice-in-java/
- https://www.geeksforgeeks.org/forkjoinpool-class-in-java-with-examples/
- https://www.baeldung.com/java-threadpooltaskexecutor-core-vs-max-poolsize
- https://www.baeldung.com/java-when-to-use-parallel-stream
- https://www.baeldung.com/java-8-parallel-streams-custom-threadpool
- https://www.baeldung.com/cs/race-conditions
- https://en.wikipedia.org/wiki/Race_condition
- https://en.wikipedia.org/wiki/Deadlock
- https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html
- https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

- https://www.baeldung.com/java-deadlock-livelock
- https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html
- https://en.wikipedia.org/wiki/Starvation_(computer_science)
- https://www.baeldung.com/cs/deadlock-livelock-starvation
- https://en.wikipedia.org/wiki/Priority_inversion
- https://en.wikipedia.org/wiki/False_sharing
- https://www.baeldung.com/java-false-sharing-contended
- https://www.baeldung.com/cs/aba-concurrency
- https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html
- https://www.baeldung.com/java-synchronized
- https://www.baeldung.com/java-synchronization-bad-practices
- https://www.baeldung.com/java-volatile

- https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/atomic/package-summary.html
- https://www.baeldung.com/java-atomic-variables
- https://www.baeldung.com/java-volatile-vs-atomic
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ConcurrentHashMap.html
- https://www.baeldung.com/java-concurrent-map
- https://www.baeldung.com/java-semaphore
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Semaphore.html
- https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/concurrent/TimedSemaphore.html
- https://www.baeldung.com/java-concurrent-locks
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/ReentrantReadWriteLock.html

- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/locks/Condition.html
- https://www.baeldung.com/java-binary-semaphore-vs-reentrant-lock
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ThreadLocal.html
- https://www.baeldung.com/java-threadlocal
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/CountDownLatch.html
- https://www.baeldung.com/java-countdown-latch
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/CyclicBarrier.html
- https://www.baeldung.com/java-cyclic-barrier
- https://www.baeldung.com/java-cyclicbarrier-countdownlatch
- https://openjdk.org/jeps/444
- https://docs.oracle.com/en/java/javase/20/core/virtual-threads.html
- https://www.javacodegeeks.com/2023/01/java-19-virtual-threads.html

- https://www.javacodegeeks.com/2023/03/intro-to-java-virtual-threads.html
- https://blogs.oracle.com/javamagazine/post/java-loom-virtual-threads-platform-threads