Effective Java

Intro : Basic Lock

Item 79 : 과도한 동기화는 피하라

Item 80 : 스레드보다는 실행자, 태스크, 스트림을 애용하라

Item 81 : wait와 notify보다는 동시성 유틸리티를 애용하라

Intro

Basic Lock

Mutex

Wikipedia

In computer science, a **lock** or **mutex** (from mutual exclusion) is a **synchronization primitive** that prevents state from being modified or accessed by multiple threads of execution at once.

Locks enforce mutual exclusion concurrency control policies,

and with a **variety of possible methods**

there exist multiple unique implementations for different applications.

Mutex

Wikipedia

Language support [edit]

See also: Barrier (computer science)

Programming languages vary in their support for synchronization:

- Ada provides protected objects that have visible protected subprograms or entries[7] as well as rendezvous.[8]
- The ISO/IEC C standard provides a standard mutual exclusion (locks) API since C11. The current ISO/IEC C++ standard supports threading facilities since C++11. The OpenMP standard is supported by some compilers, and allows critical sections to be specified using pragmas. The POSIX pthread API provides lock support.[9] Visual C++ provides the `synchronize` attribute of methods to be synchronized, but this is specific to COM objects in the Windows architecture and Visual C++ compiler.[10] C and C++ can easily access any native operating system locking features.
- C# provides the `lock` keyword on a thread to ensure its exclusive access to a resource.
- VB.NET provides a `SyncLock` keyword like C#'s `lock` keyword.
- Java provides the keyword `synchronized` to lock code blocks, methods or objects[11] and libraries featuring concurrency-safe data structures.
- Objective-C provides the keyword `@synchronized`[12] to put locks on blocks of code and also provides the classes NSLock,[13] NSRecursiveLock,[14] and NSConditionLock[15] along with the NSLocking protocol[16] for locking as well.
- PHP provides a file-based locking [17] as well as a `Mutex` class in the `pthreads` extension. [18]
- Python provides a low-level mutex mechanism with a `Lock` class from the `threading` module.[19]

Mutex using POSIX API

```
void *run(void *mutex) {
    ...
    pthread_mutex_lock(&mutex);       /* locks the database   */
    ...                               /* handles the request  */
    pthread_mutex_unlock(&mutex);     /* unlocks the database */
}

int main() {
    pthread_mutex_t mutex;
    pthread_t thread_id[10];

    pthread_mutex_init(&mutex, NULL);  /* creates the mutex      */

    for (i = 0; i < 10; i++)           /* loop to create threads */
        pthread_create(&thread_id[i], NULL, rtn, &mutex);

    //pthread_join(thread_id[i], NULL) /* waits end of session   */

    pthread_mutex_destroy(&mutex);     /* destroys the mutex     */
}
```

Implement Mutex ...

Peterson's Solution

Bakery Algorithm(Lamport)

Test-And-Set

Compare-And-Swap

https://www.ibm.com/docs/ko/aix/7.3?topic=programming-using-mutexes

Semaphore

Wikipedia

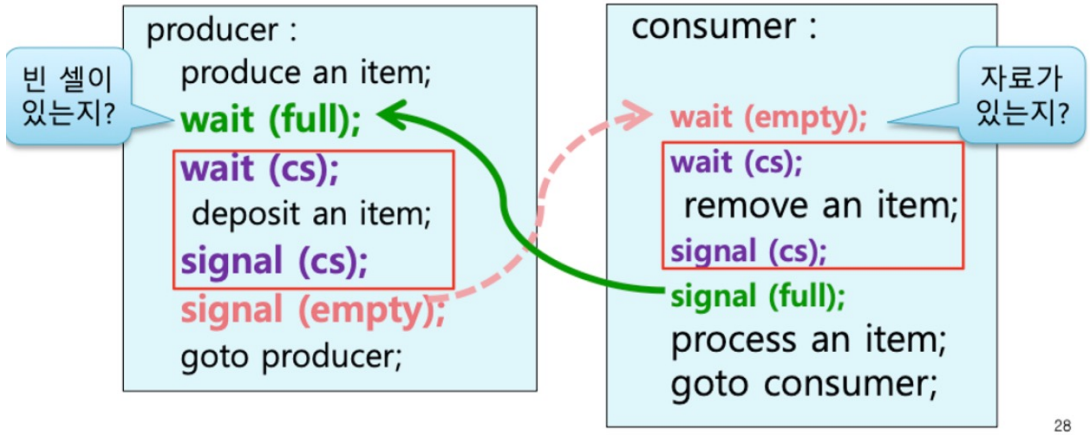
In computer science, a semaphore is a variable or abstract data type used to control access to a **common resource by multiple threads** and **avoid critical section problems** in a concurrent system such as a multitasking operating system.

Semaphores are a type of synchronization primitive.

Semaphore

```
typedef struct {
    int value = n;
    struct process *list;
} semaphore;

wait(semaphore *S) { // P operation
    S -> value--;
    if (S -> value < 0) {
        (S -> list).push(this.thread);
        block();
    }
}

signal(semaphore *S) { // V operation
    S -> value++;
    if (S -> value <= 0) {
        process* P = (S -> list).pop();
        wakeup(P);
    }
}
```

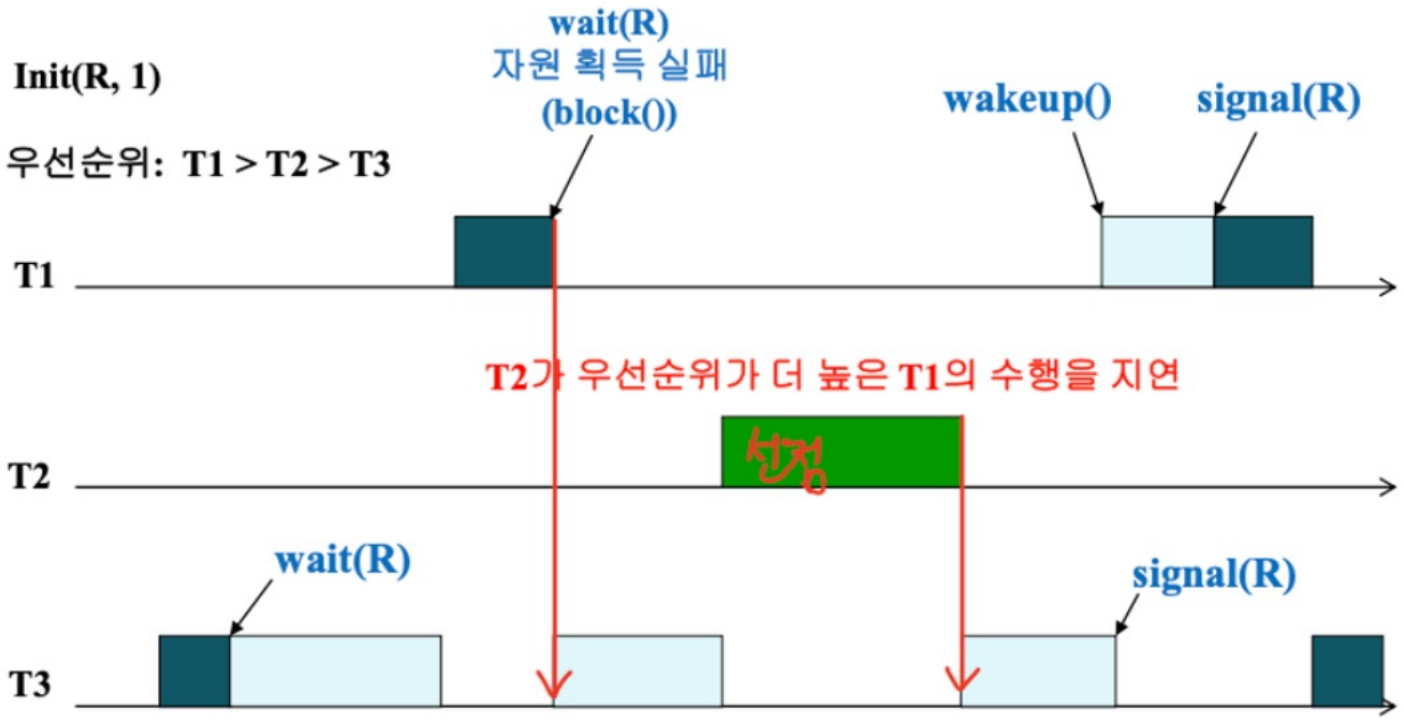
n > 1 : multiple resource
n = 1 : mutex (binary semaphore)
n = 0 : serialization



Mutex vs. binary Semaphore

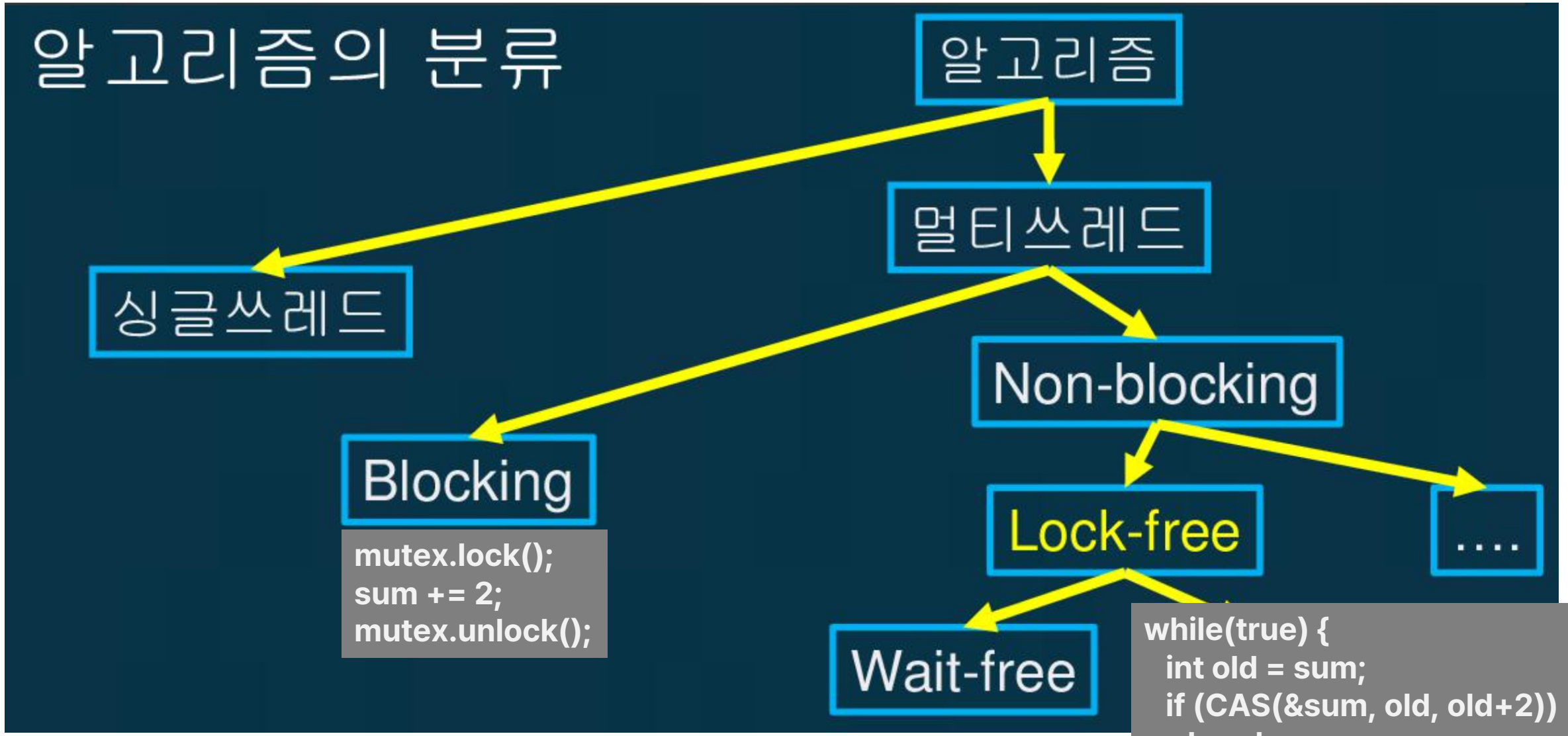
1. **Priority inversion**
2. Premature task termination
3. Termination deadlock
4. Recursion deadlock
5. Accidental release

Priority Inversion Problem



Simple solution: 우선순위 상속 (T3에게 T1의 우선순위를 임시 부여)

Lock-Free Algorithm

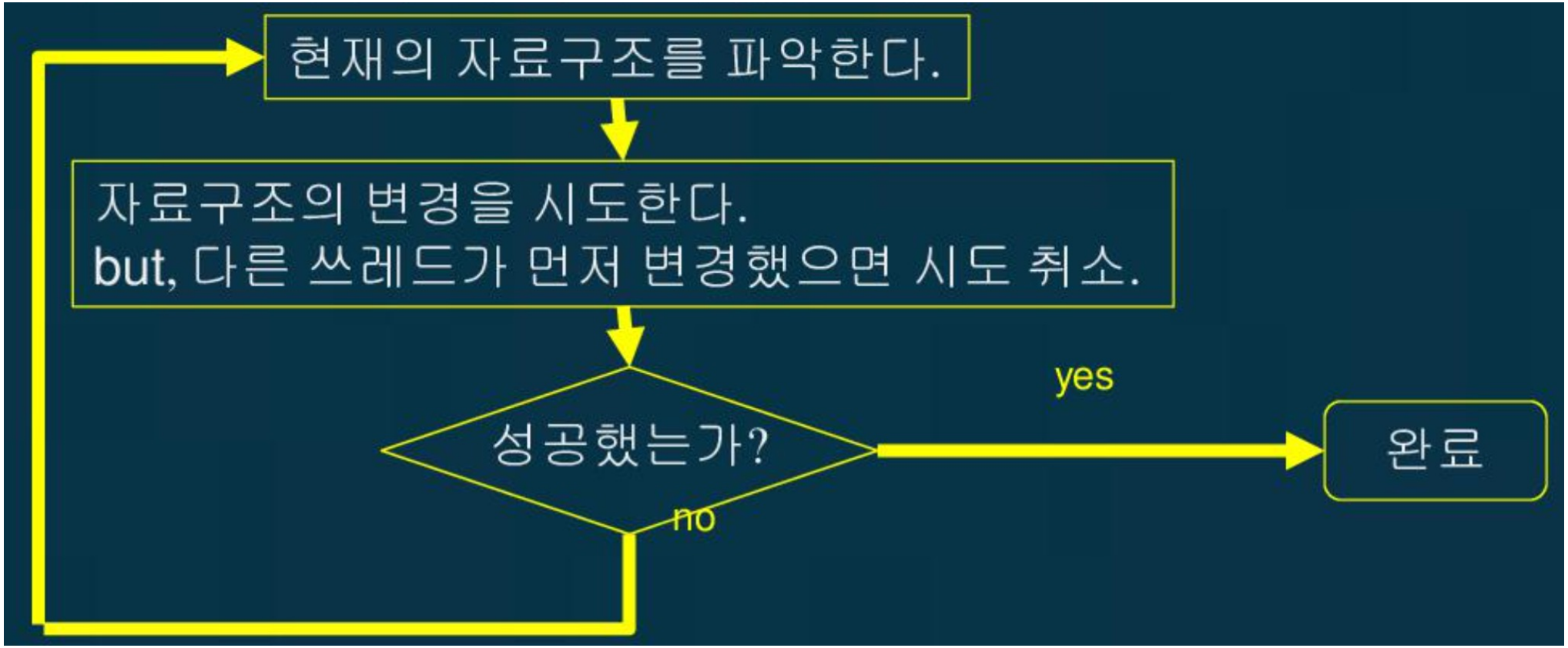


http://ndcreplay.nexon.com/NDC2014/sessions/NDC2014_0048.html

made by Seungtaek

Lock-Free Algorithm

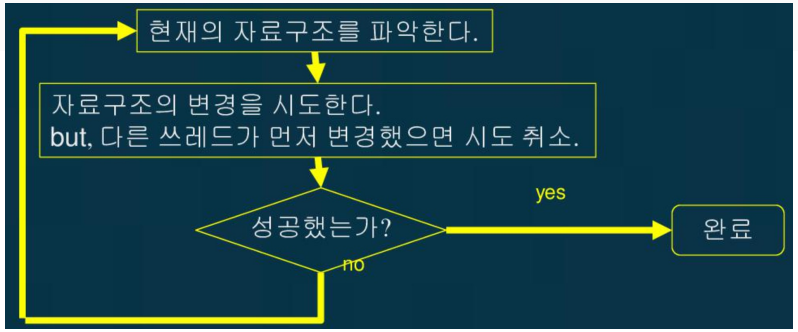
- 여러 쓰레드에서 동시 호출했을 때에도, 적어도 한 개의 호출이 완료되는 알고리즘
- No using Lock, Non-Blocking Algorithm
- Wait-Free Algorithm : 모든 스레드가 한정된 시간 내에 진행할 수 있는 것을 보장



Lock-Free Algorithm

- 여러 쓰레드에서 동시 호출했을 때에도, 적어도 한 개의 호출이 완료되는 알고리즘
- No using Lock, Non-Blocking Algorithm

```
void push(T val) {
    do {
        Node* nNode = new Node(val);
        Node* top = st.top();
        nNode->next = top;
    while(!CAS(st.top(), top, nNode))
}
```



Compare And Swap(Set)

```
bool CAS(&value, expected, new) {
   if (value != expected) return false;
   value = new;
   return true;
}
```

Lock-Free Algorithm

Lock-Free 알고리즘의 장단점

- 장점
  - 성능!!
  - 높은 부하에도 안정적!!
- 단점
  - 생산성 : 복잡한 알고리즘
  - 신뢰성 : 정확성을 증명하는 것은 어렵다.
  - 확장성 : 새로운 메소드를 추가하는 것이 매우 어렵다.
  - 메모리 : 메모리 재사용이 어렵다. ABA문제

Monitor

Operating System Concepts, 10th. Ch 6.7

Semaphore, Mutex 락을 이용하면 임계구역 문제를 해결할 수 있다.
하지만 잘못 사용하면 다양한 유형의 오류가 너무나도 쉽게 발생할 수 있다.

```
signal(mutex);
    ... criticial section...
wait(mutex);
```

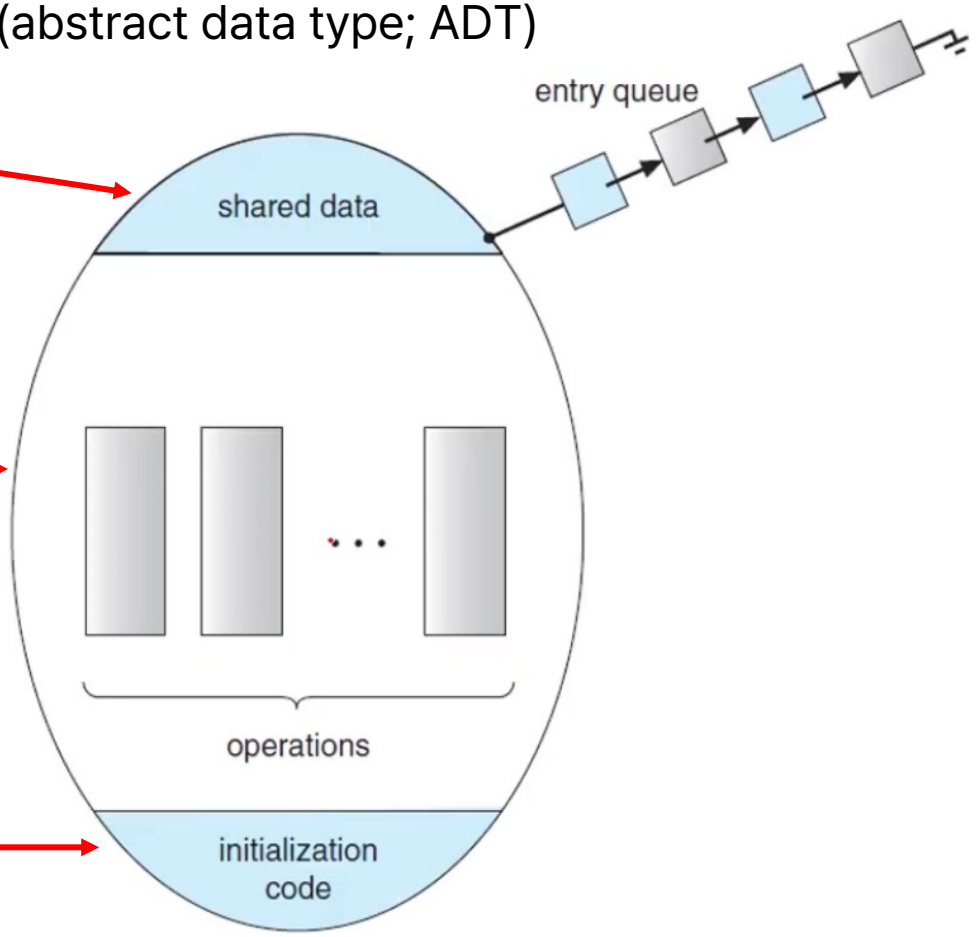
```
wait(mutex);
    ... criticial section...
wait(mutex);
```

이러한 오류를 처리하기 위한 한 가지 전략은
간단한 동기화 도구를 통합하여 고급 언어 구조물을 제공하는 것이다.

Monitor

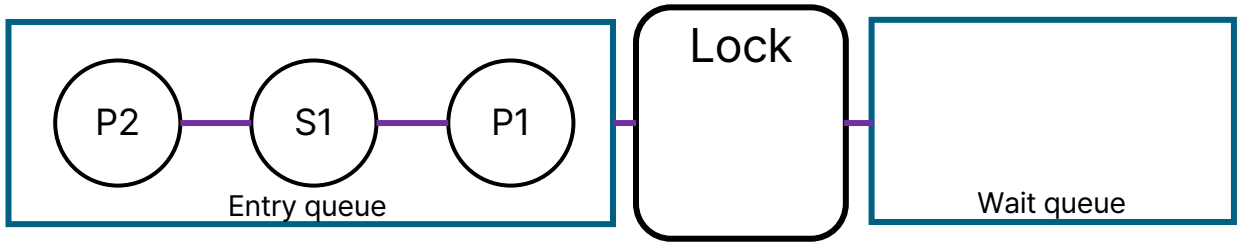
```
monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        . . .
    }

    function P2 ( . . . ) {
        . . .
    }
        .
        .
        .
        .
    function Pn ( . . . ) {
        . . .
    }

    initialization_code ( . . . ) {
        . . .
    }
}
```

Monitor data type
(abstract data type; ADT)

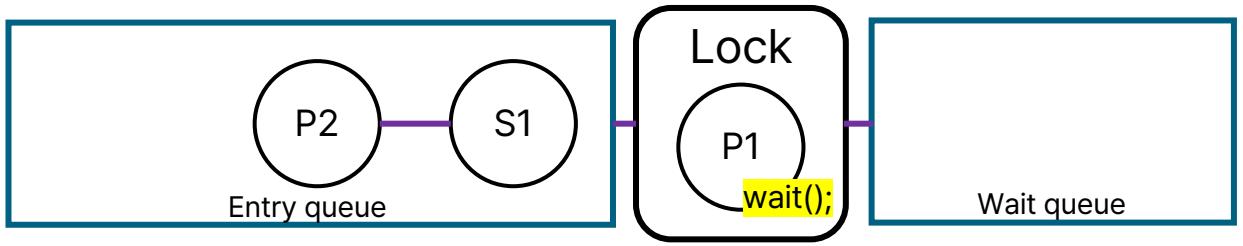


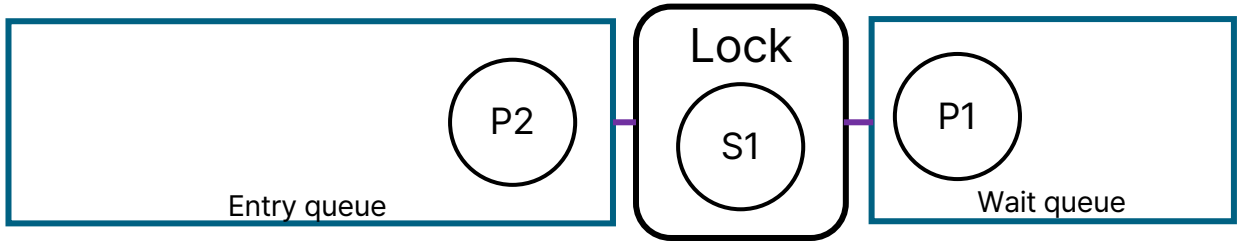
Java Monitors

```
public synchronized void insert(E item) {
  while (BUFFER.isFull()) wait();

  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  count++;

  notify();
}

public synchronized E remove() {
  while(BUFFER.isEmpty()) wait();

  E item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;

  notify();
  return item;
}
```



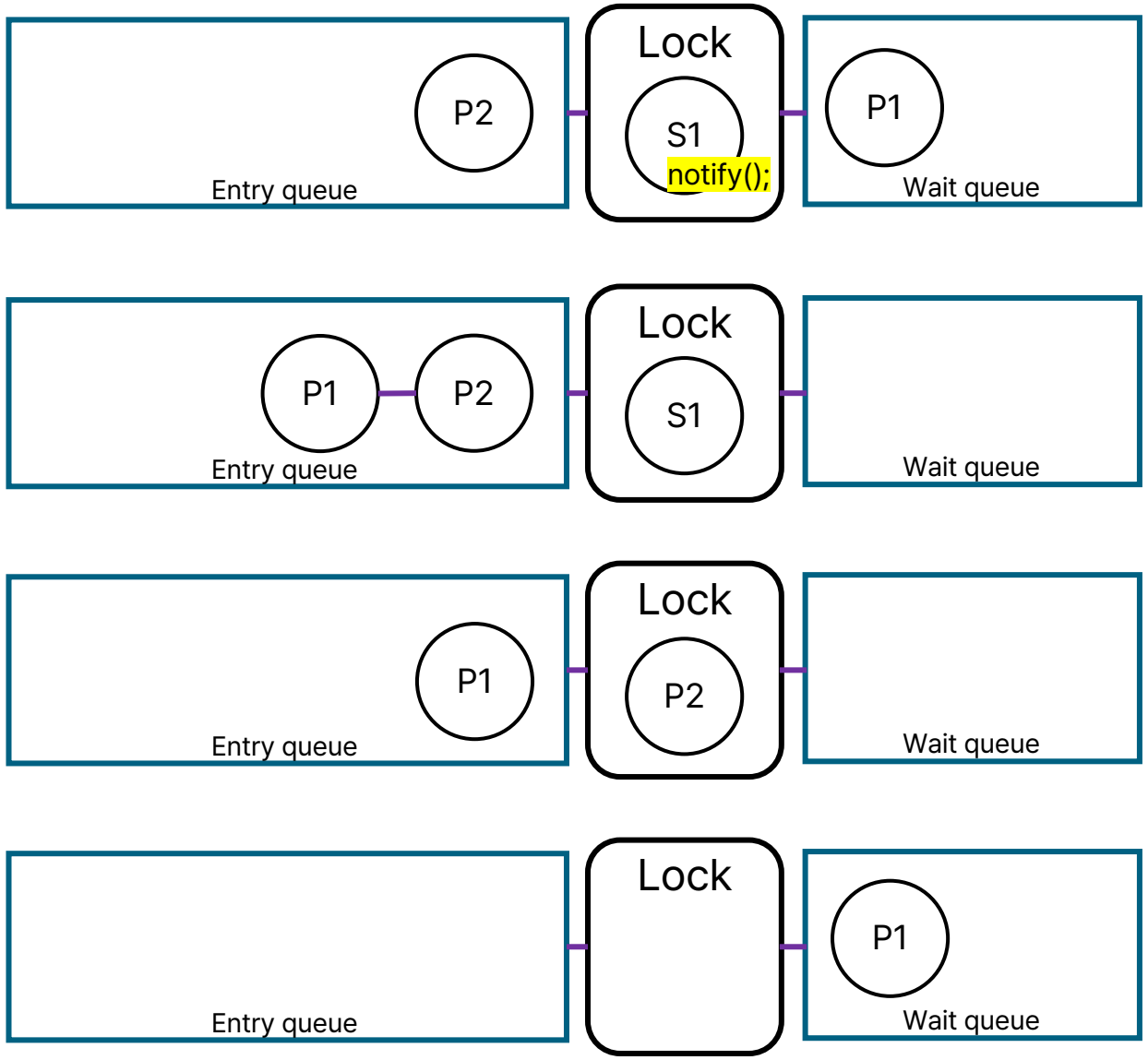
가정: BUFFER is Full



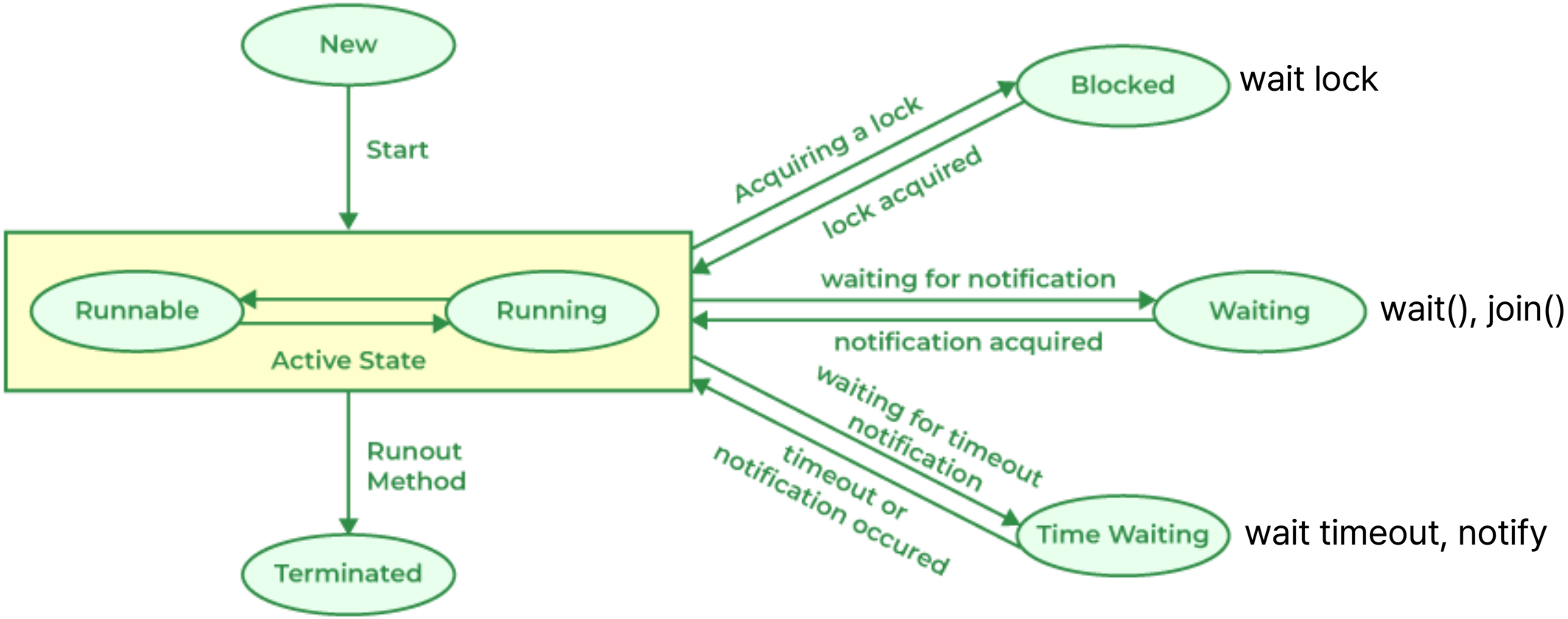


Java Monitors

```
public synchronized void insert(E item) {
  while (BUFFER.isFull()) wait();

  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  count++;

  notify();
}

public synchronized E remove() {
  while(BUFFER.isEmpty()) wait();

  E item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;

  notify();
  return item;
}
```



Java Threads



Yielding, Sleeping 주의할 점

Thread.yield() : CPU 포기하기

```
public synchronized void method1() {
  while (true) {
    Thread.yield();
  }
}
```

```
public synchronized void method1() {
  try {
      Thread.sleep(3000);
  } catch (InterruptedException e) {}
}
```

The thread does not lose ownership of any monitors.

| CPU는 양보하지만 Lock은 양보 못해

다음에 자바 스레드에 대해서 더 자세히 알아보자..

Item 79

과도한 동기화는 피하라

Problem 1

```
private void notifyElementAdded(E element) {
    synchronized (observers) {
        for (SetObserver<E> observer : observers) {
            observer.added(this, element);
        }
    }
}
```

ConcurrentModificationException

```
set.addObserver(new SetObserver<>() {
    @Override
    public void added(ObservableSet<Integer> set, Integer element) {
        System.out.println(element);
        if (element == 23)
            set.removeObserver(this);
    }
});
```

Problem 2

```
private void notifyElementAdded(E element) {
    synchronized (observers) {
        for (SetObserver<E> observer : observers) {
            observer.added(this, element);
        }
    }
}
```

```
public boolean removeObserver(SetObserver<E> observer) {
    synchronized (observers) {
        return observers.remove(observer);
    }
}
```

Dead Lock!

```
set.addObserver(new SetObserver<>() {
    @Override
    public void added(ObservableSet<Integer> set, Integer element) {
        System.out.println(element);
        if (element == 23) {
            ExecutorService exec = Executors.newSingleThreadExecutor();
            try {
                Future<Boolean> submit = exec.submit(() -> set.removeObserver(this));
                Boolean result = submit.get();
            } catch (Exception e) {
                exec.shutdown();
            }
        }
    }
});
```

Blocking method

Solution

```
private void notifyElementAdded(E element) {
    List<SetObserver<E>> snapshot = null;
    synchronized (observers) {
        snapshot = new ArrayList<>(observers);
    }
    for (SetObserver<E> observer : snapshot) {
        observer.added(this, element);
    }
}
```

open call

```
set.addObserver(new SetObserver<>() {
    @Override
    public void added(ObservableSet<Integer> set, Integer element) {
        System.out.println(element);
        if (element == 23) {
            set.removeObserver(this);
        }
    }
});
```

CopyOnWriteArrayList

A thread-safe variant of ArrayList in which **all mutative operations(add, set, and so on)** are implemented by making a **fresh copy** of the underlying array.

```java
public boolean add(E e) {
    synchronized (lock) {
        Object[] es = getArray();
        int len = es.length;
        es = Arrays.copyOf(es, newLength: len + 1);
        es[len] = e;
        setArray(es);
        return true;
    }
}
```

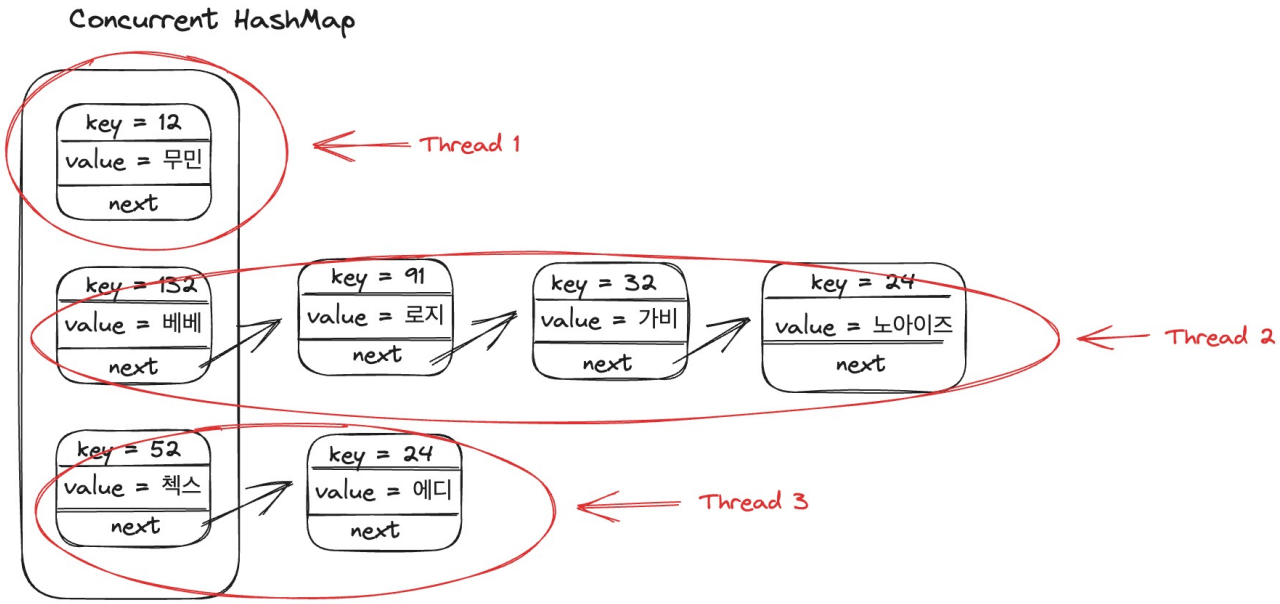
```java
public E remove(int index) {
    synchronized (lock) {
        Object[] es = getArray();
        int len = es.length;
        E oldValue = elementAt(es, index);
        int numMoved = len - index - 1;
        Object[] newElements;
        if (numMoved == 0)
            newElements = Arrays.copyOf(es, newLength: len - 1);
        else {
            newElements = new Object[len - 1];
            System.arraycopy(es, srcPos: 0, newElements, destPos: 0, index);
            System.arraycopy(es, srcPos: index + 1, newElements, index,
                    numMoved);
        }
        setArray(newElements);
        return oldValue;
    }
}
```

Lock splitting, Lock striping

Lock splitting
different locks for different purposes on the whole data structure

ex) read lock, write lock

Lock striping
Multiple locks for different parts of a data structure



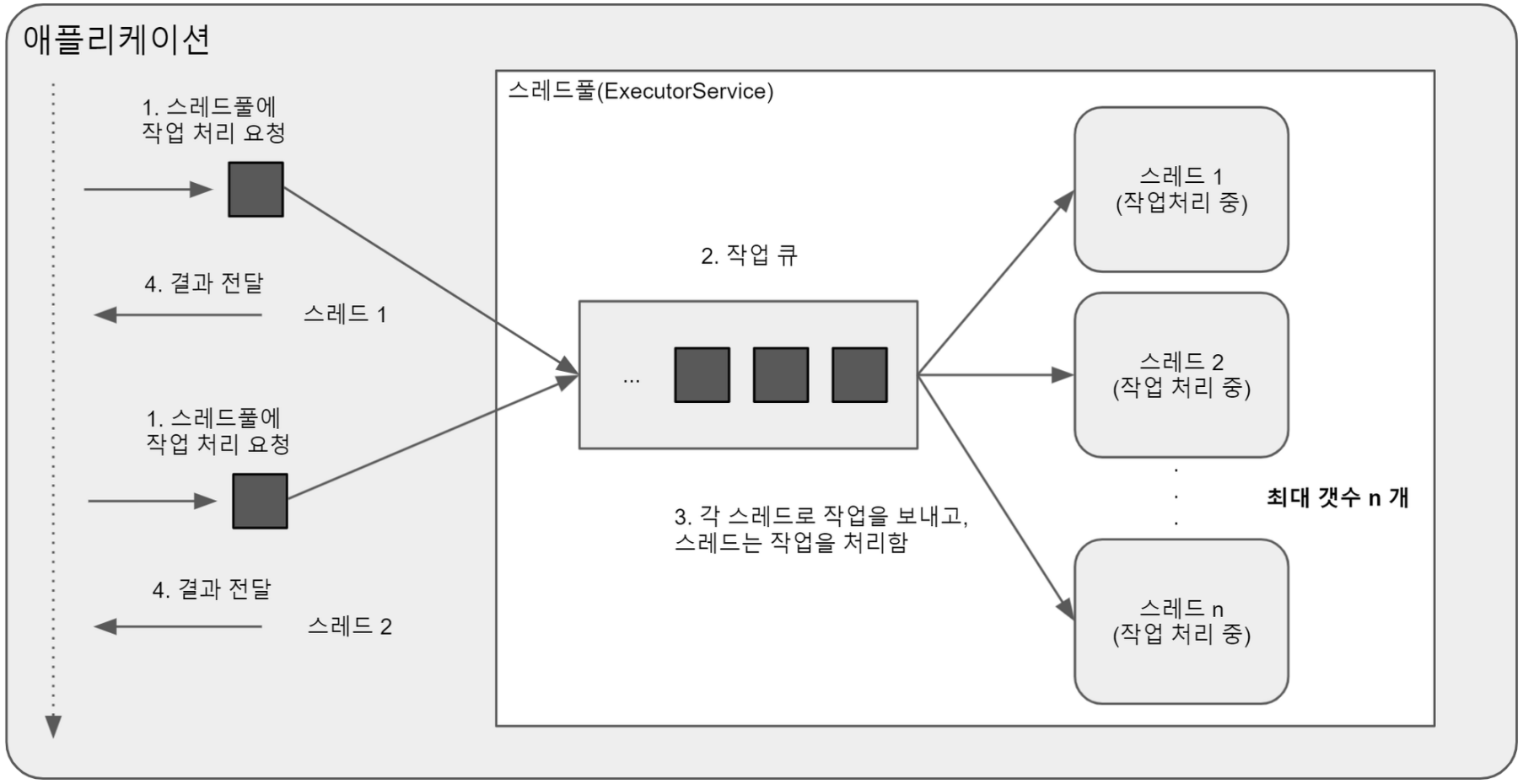
https://parkmuhyeun.github.io/woowacourse/2023-09-09-Concurrent-Hashmap/

결론
Critical section을 최소한으로 줄이자

Item 80

스레드보다는 실행자, 태스크, 스트림

ThreadPool


애플리케이션
1. 스레드풀에
작업 처리 요청
4. 결과 전달
스레드 1
1. 스레드풀에
작업 처리 요청
4. 결과 전달
스레드 2
스레드풀(ExecutorService)
2. 작업 큐
...
3. 각 스레드로 작업을 보내고,
스레드는 작업을 처리함
스레드 1
(작업처리 중)
스레드 2
(작업 처리 중)
최대 갯수 n 개
스레드 n
(작업 처리 중)

ExecutorService

메소드명	초기 스레드 수	코어 스레드 수	최대 스레드 수
newCachedThreadPool()	0	0	Integer.MAX_VALUE
newFixedThreadPool(int nThreads)	0	nThreads	nThreads

초기 스레드 수 : ExecutorService 객체가 생성될 때 기본적으로 생성되는 스레드 수

코어 스레드 수 : 사용되지 않는 스레드를 제거할 때 최소한 유지해야 할 스레드 수

최대 스레드 수 : 스레드풀에서 관리하는 최대 스레드 수

```
ExecutorService executorService1 = Executors.newCachedThreadPool();

ExecutorService executorService2 = Executors.newFixedThreadPool(
        Runtime.getRuntime().availableProcessors()
);
```

CPU 코어 수
ex) Apple M1 = 8코어

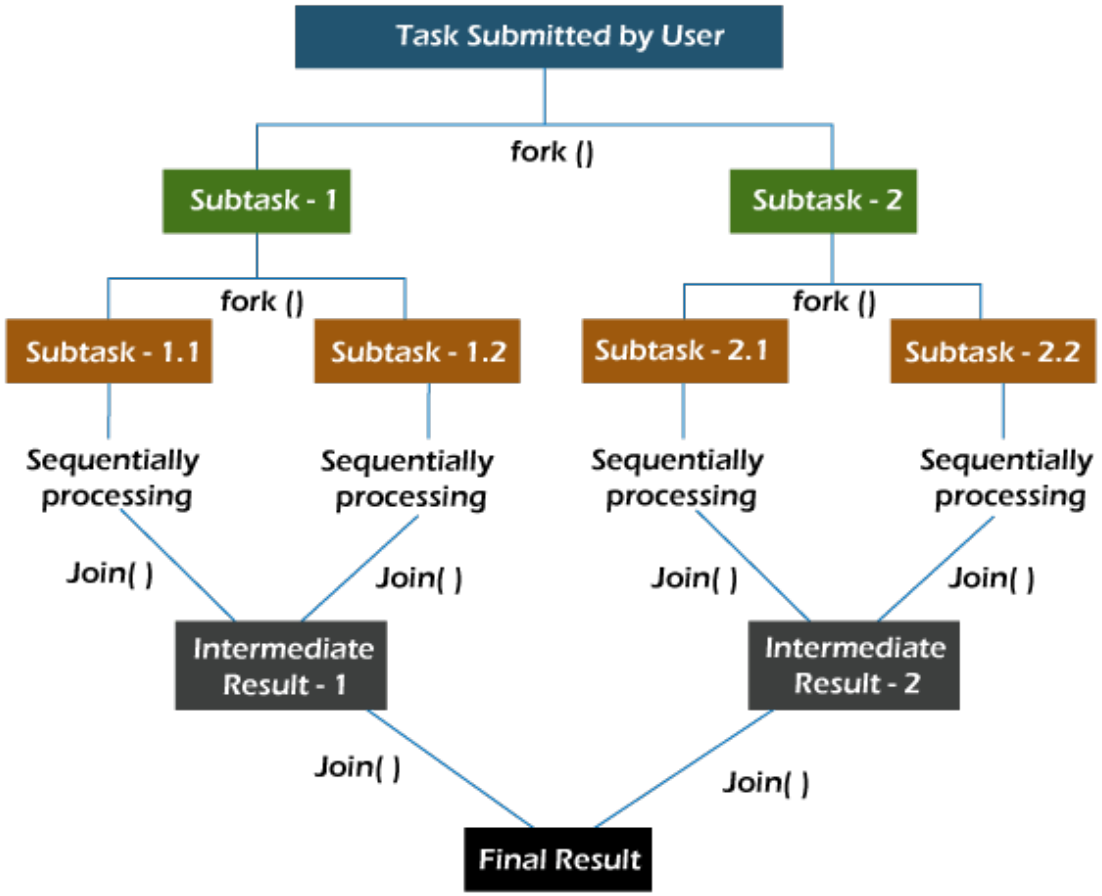
ExecutorService

```
ExecutorService threadPool = new ThreadPoolExecutor(
        3,    // 코어 스레드 개수
        100,  // 최대 스레드 개수
        120L, // 놀고 있는 시간
        TimeUnit.SECONDS, // 놀고 있는 시간 단위
        new SynchronousQueue<>() // 작업 큐
);
```

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor( corePoolSize: 0, Integer.MAX_VALUE,
            keepAliveTime: 60L, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>());
}
```

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
            keepAliveTime: 0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>());
}
```

ForkJoinPool



ParellelStream은 내부적으로 ForkJoinPool 사용

To be continue..

Item 81

wait와 notify보다는 동시성 유틸리티를 애용

java.util.concurrent

Executor

Executors

ScheduledThreadPoolExecutor

ThreadPoolExecutor

ExecutorService

ForkJoinPool

Concurrent collection

ConcurrentSkipListSet

ConcurrentHashMap

CopyOnWriteArraySet

SynchronousQueue

ConcurrentLinkedDeque

BlockingQueue

Synchronizer

Semahpore

CyclicBarrier

CountDownLatch

Exchanger

Phaser

ConcurrentHashMap

```
Map<String, String> map = new ConcurrentHashMap<>();

public synchronized void putIfAbsent(String k, String v) {
    if (!map.containsKey(k)) {
        map.put(k, v);
    }
}
```

```
public void putIfAbsent(String k, String v) {
    map.putIfAbsent(k, v);
}
```

```
public void putIfAbsent(String k, String v) {
    if (map.get(k) == null) {
        map.putIfAbsent(k, v);
    }
}
```

CountDownLatch

```
CountDownLatch cdl = new CountDownLatch(count);
cdl.countDown();      // decrements the count of the latch
cdl.await();          // wait until the latch has counted down to zero
long t = cdl.getCount(); // returns the current count
```

```
public long time(Executor executor, int concurrency, Runnable action) {
    CountDownLatch ready = new CountDownLatch(concurrency);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done = new CountDownLatch(concurrency);

    for (int i = 0; i < concurrency; i++) {
        executor.execute(() -> {
            // 타이머에게 준비가 됐음을 알린다.
            ready.countDown();
            try {
                // 모든 작업자 스레드가 준비될 때까지 기다린다.
                start.await();
                action.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                // 타이머에게 작업을 마쳤음을 알린다.
                done.countDown();
            }
        });
    }

    ready.await(); // 모든 작업자가 준비될 때까지 기다린다.
    long startNanos = System.nanoTime();
    start.countDown(); // 작업자들을 깨운다.
    done.await(); // 모든 작업자가 일을 끝마치기를 기다린다.
    return System.nanoTime() - startNanos;
}
```

currentTimeMillis vs. nanoTime

System.currentTimeMillis()

the granularity of the value depends on the underlying operating system

(1970.01.01. 자정 기준)

다른 시스템과의 통신(DB, DS..),
시스템 시간 필요 시

System.nanoTime()

This method can only be used to measure elapsed time

and is not related to any other notion of system or wall-clock time

시간 간격 측정

wait, notify

wait()의 표준 방식

```
synchronized(obj) {
    while(조건이 충족되지 않음)
        obj.wait();
  // do something
}
```

혹시 모를 상황에 대비..

notify()보다는 notifyAll()

혹시 모를 상황에 대비..