

독하게 시작하는 Java

Part 3 – 상

취업목적! 차별화된 경쟁력을 갖추려는 입문자를 위한 안내

넌넌한 개발자 최호성 (cx8537@naver.com)

YouTube: 넌넌한 개발자 TV

문서 개정이력

[illegible]

1. 시작에 앞서

제대로 달리는 학습순서

Spring framework으로 넘어가기 위한 준비

(Java reflection + Custom annotation, Maven+Gradle, WAS, Design pattern)

웹 인프라 기술에 대한 이해와 DB

(HTML+CSS+JS, HTTP, SSL, SQL+RDBMS, JDBC, ORM)

시스템 활용 프로그램 작성

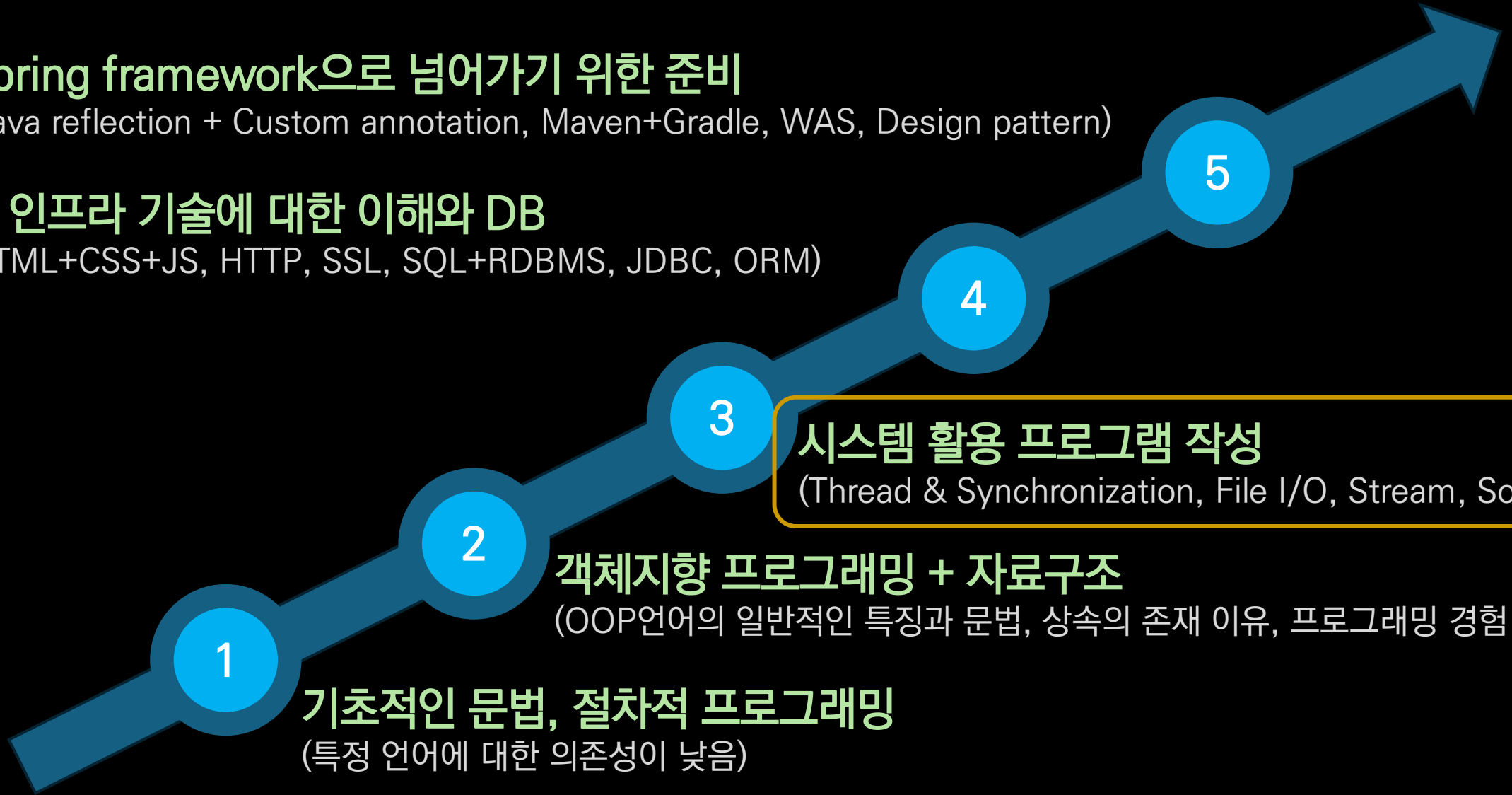
(Thread & Synchronization, File I/O, Stream, Socket)

객체지향 프로그래밍 + 자료구조

(OOP언어의 일반적인 특징과 문법, 상속의 존재 이유, 프로그래밍 경험 쌓기)

기초적인 문법, 절차적 프로그래밍

(특정 언어에 대한 의존성이 낮음)



알고 있다고 가정하는 것

- 독하게 시작하는 Java – Part 1

- 절차적 프로그래밍
- 프로그래밍 언어 기초

- 독하게 시작하는 Java – Part 2

- OOP 프로그래밍
- JVM 구조 및 GC 작동원리에 대한 이해

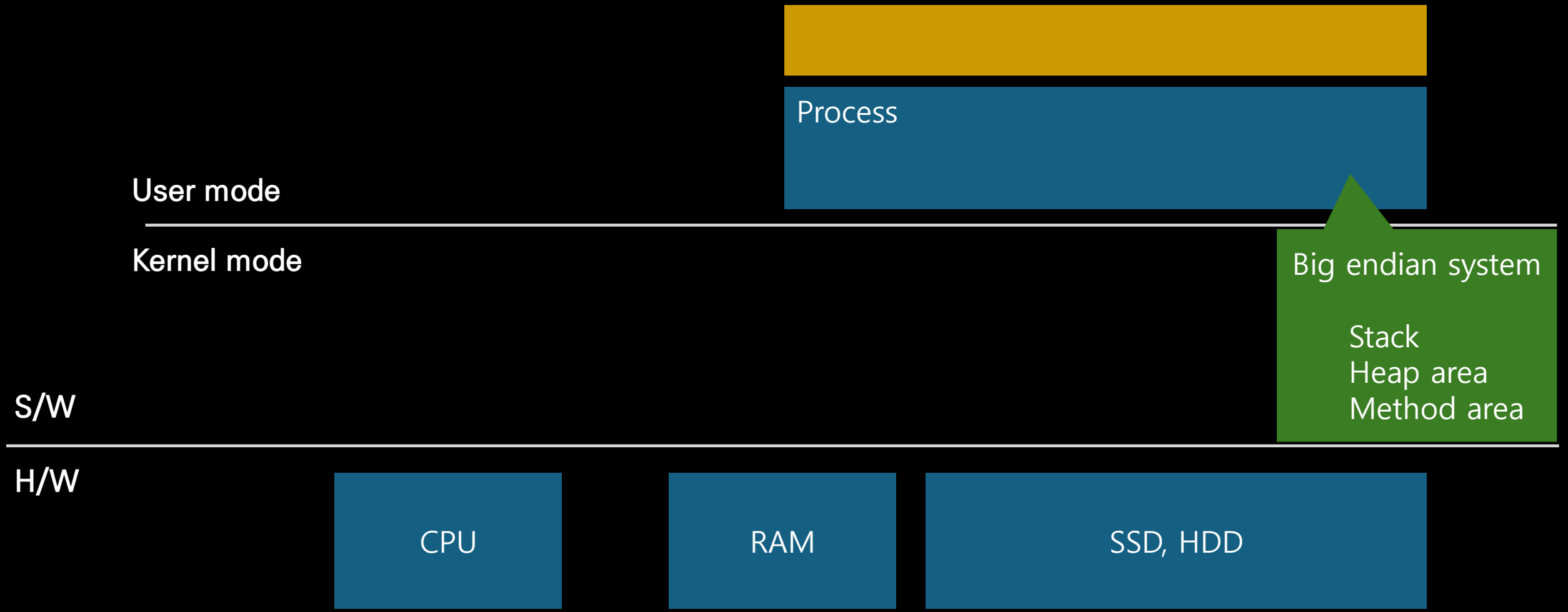
학습목표

- 멀티스레드 환경에 대한 기술적 이해
- 멀티스레드 응용 프로그램을 개발 할 수 있는 실질적인 개발능력 확보
 - 스레드 동기화
- 제네릭
- 컬렉션 프레임워크

시작에 앞서 드리는 말씀

- C/C++는 OS와 컴퓨터 구조에 대해 아는 것이 중요
 - JVM은 사용자 모드 응용 프로그램
 - 가상 메모리를 사용하는 일반적인 C++ 기반 응용 프로그램
- Java에서는 JVM을 아는 것이 중요
 - JVM은 H/W + OS + 기타
 - JVM의 H/W적 작동 특성은 실제 하드웨어와 유사한 측면이 있으나 사실 상 무관하다고 가정하는 것이 바람직함

User mode process JVM



2. 멀티스레드 기본 이론

OS와 Process

- OS는 Process 단위로 가상 메모리 공간을 제공하며 한 프로세스는 최소 1개의 Thread를 가지고 있음
 - Thread는 Process의 가상 메모리 공간을 사용
- OS는 Process 단위로 각종 접근 권한을 통제
 - 보통 접근은 File에 대한 접근을 의미
- JVM은 사용자 모드 응용 프로그램 (Process)

Thread

- Windows OS 환경에서 **Platform thread**는 CPU core를 사용하는 주체이며 실행의 최소 단위로 개별화된 흐름을 가짐
 - 특정 스레드가 CPU core 하나를 사용해 연산을 이어가는 구조
- 보통 한 컴퓨터에서 수 천개의 스레드가 동시에 작동
- 우선 순위에 따라 OS가 CPU를 사용할 스레드를 결정 (Thread scheduling)
- **Context switching** 따른 오버헤드가 있음

성능

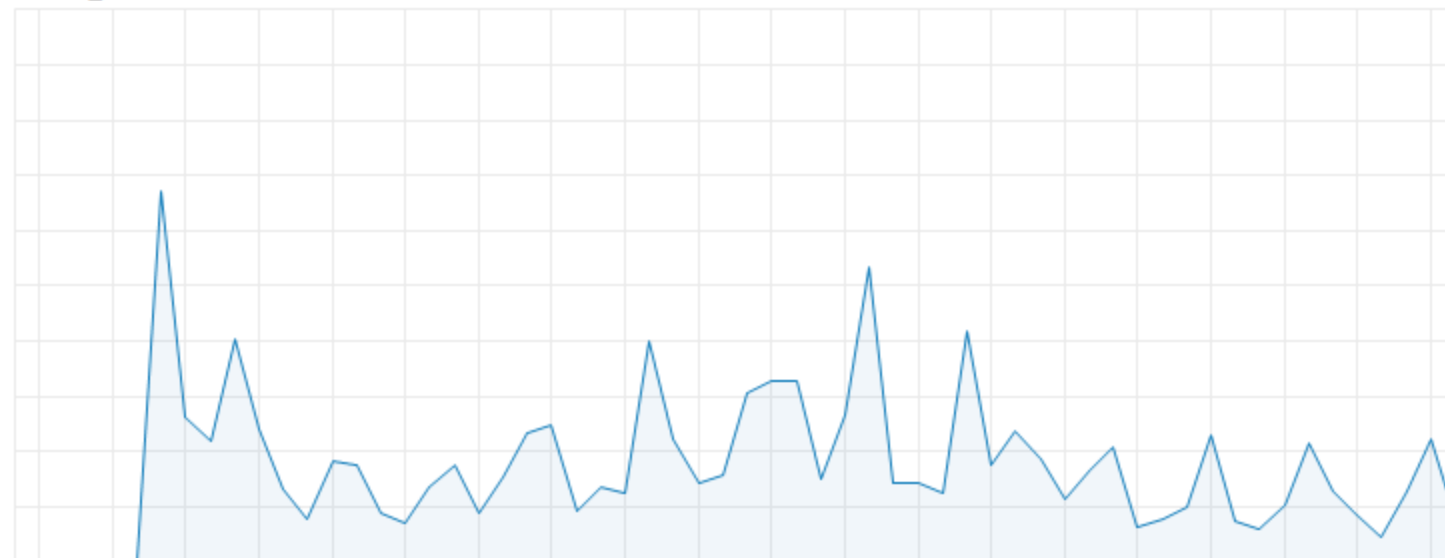
새 작업 실행

**CPU**
8% 4.58GHz**메모리**
12.3/63.9GB (19%)**디스크 0(D:)**
HDD
0%**디스크 1(C:)**
SSD
1%**디스크 2(H:)**
SSD
0%**디스크 3(F:)**
SSD
0%**디스크 4(E:)**
HDD
0%**디스크 5(L:)**
HDD
0%**CPU**

% 이용률

Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz

100%

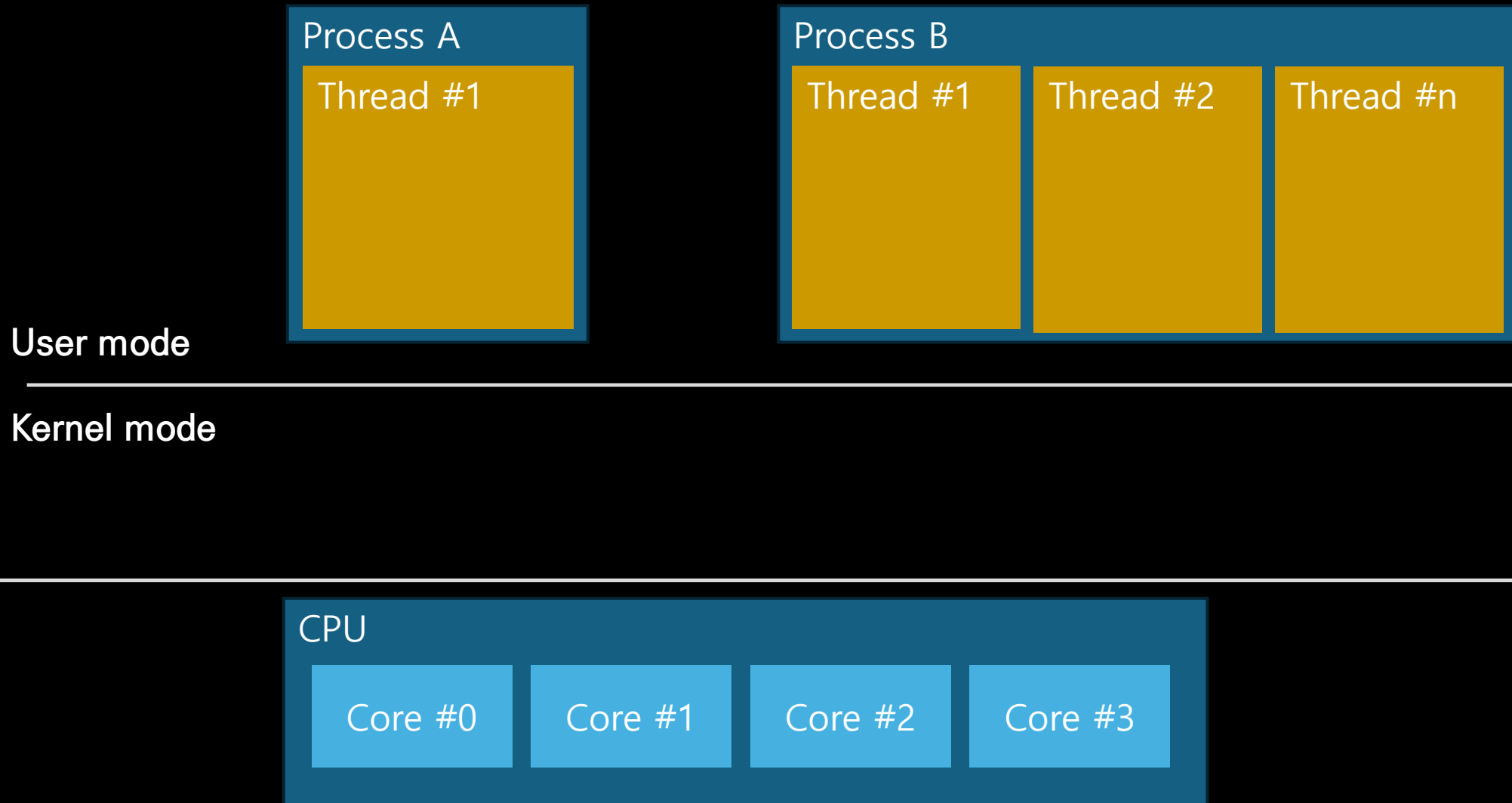


60초

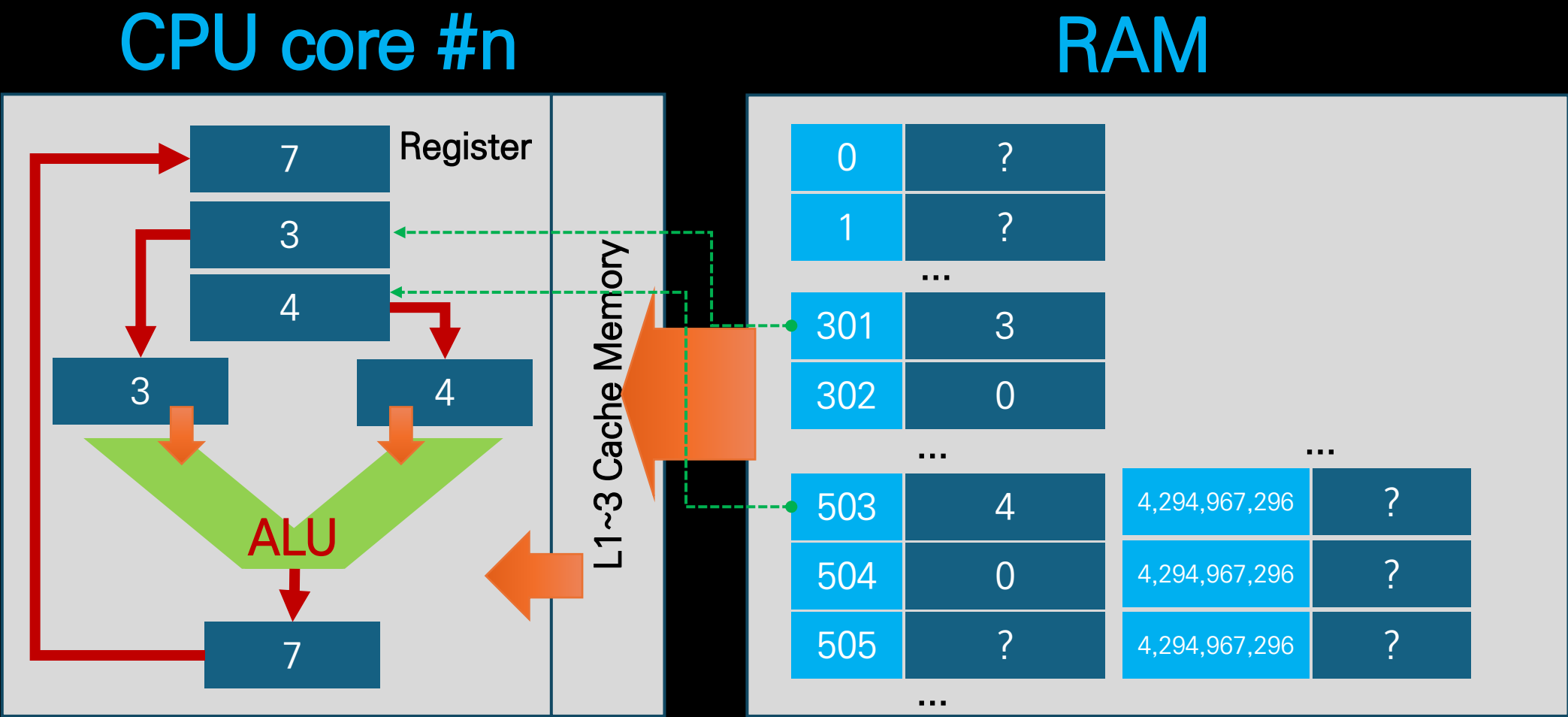
1

이용률	속도	기본 속도:	3.60GHz
8%	4.58GHz	소켓:	1
프로세스	스레드	코어:	8
358	5590	논리 프로세서:	8
작동 시간	핸들	가상화:	사용
0:03:32:23	175618	L1 캐시:	512KB
		L2 캐시:	2.0MB
		L3 캐시:	12.0MB

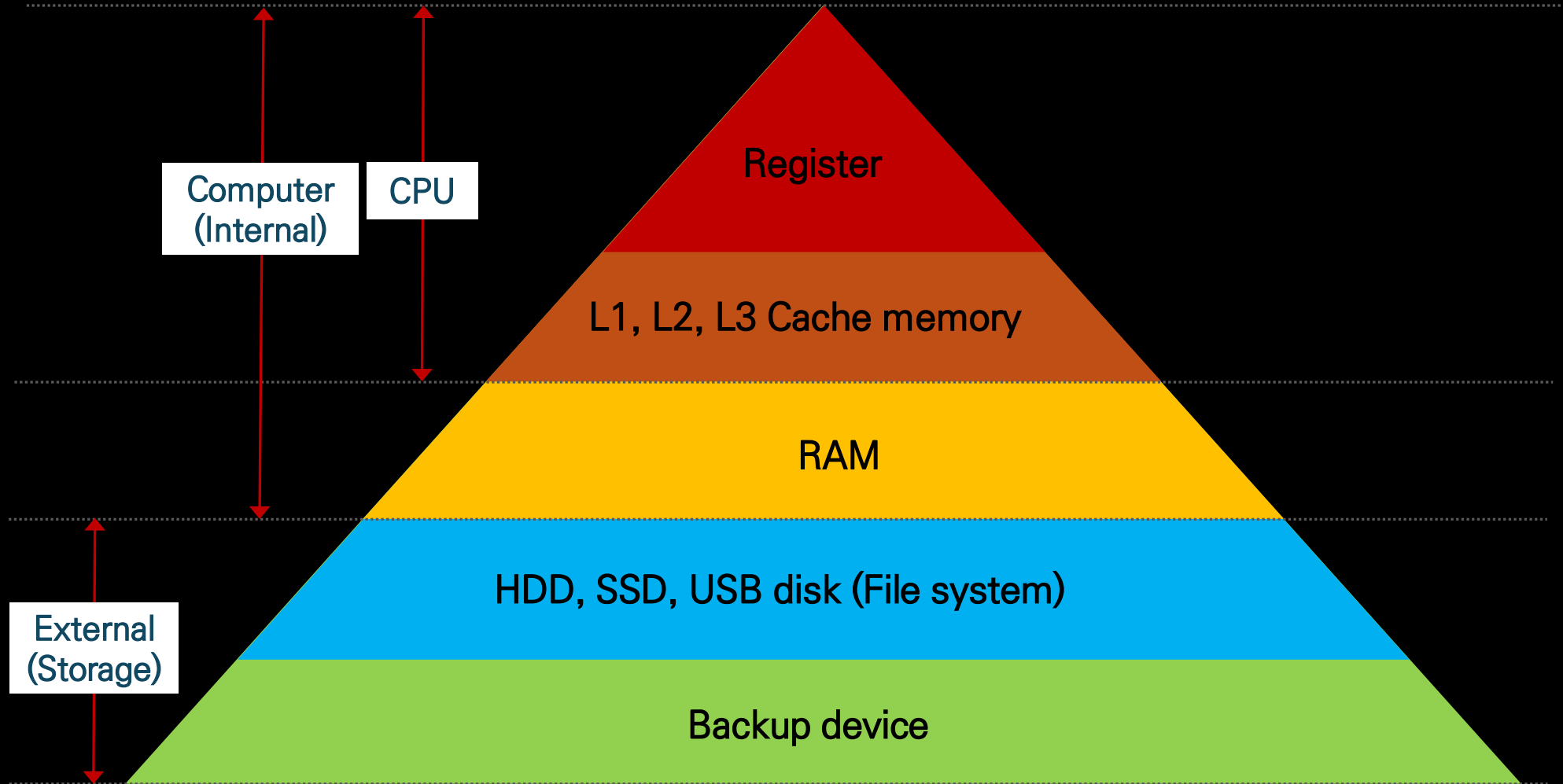
Process와 (Platform) Thread 구조



컴퓨터 구조에 대한 상식



메모리 구조



멀티스레딩이 필요한 이유

속도 빠른 core 1개 vs 조금 느린 core n개

- CPU는 매우 빠른 연산능력을 가진 반면 주변기기는 이를 따르지 못할 만큼 매우 느림
 - 단순 입/출력 대기에도 CPU time을 소모 할 수 있음
- CPU의 연산 속도는 클럭속도에 따라 결정되지만 한계가 있으며 대신 Core 개수를 여러 개로 늘려 조합하는 형태로 발전
- 멀티태스킹 환경에서 CPU 사용 효율을 극대화 할 수 있음

멀티스레딩 vs 멀티프로세싱

싱글 프로세스 + 멀티스레딩

- 한 프로세스에 모든 스레드가 종속되며 같은 가상 메모리 공간을 공유
 - IPC 불필요
- 프로세스에 할당된 접근 권한을 모든 스레드가 공유
- 한 스레드 오류 시 프로세스가 종료 될 수 있음

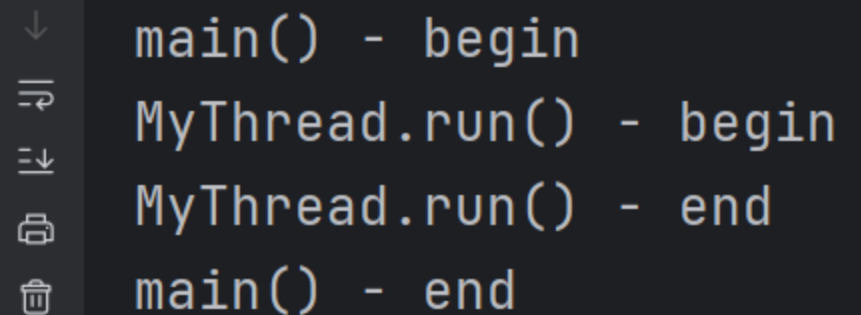
멀티 프로세스 + 싱글 스레딩

- 각각의 프로세스는 개별 가상 메모리 공간 및 접근 권한을 별도로 가짐
- 각 프로세스 간 통신 시 IPC 기술을 적용
- 오류는 각 프로세스 수준에서 통제 될 수 있음

02_threadSample

```
class MyThread implements Runnable {  
    public void run() {  
        System.out.println("MyThread.run() - begin");  
        System.out.println("MyThread.run() - end");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("main() - begin");  
        Runnable myThread = new MyThread();  
        Thread thread = new Thread(myThread);  
        thread.start();  
  
        try{ Thread.sleep(500); } catch (Exception e) {}  
        System.out.println("main() - end");  
    }  
}
```



A terminal window with a dark background and light gray text. On the left side, there is a vertical toolbar with five icons: a downward arrow, a double arrow pointing right, a double arrow pointing left, a printer icon, and a trash can icon. The output text is as follows:

```
main() - begin  
MyThread.run() - begin  
MyThread.run() - end  
main() - end
```

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("main() - begin");  
        Thread thread = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("MyThread.run() - begin");  
                System.out.println("MyThread.run() - end");  
            }  
        });  
        thread.start();  
  
        try{ Thread.sleep(500); } catch (Exception e) {}  
        System.out.println("main() - end");  
    }  
}
```

두 스레드(연산흐름)의 공존 및 동기화

Main.main()

```
Thread thread = new Thread(new Runnable() {});
```

```
try{ Thread.sleep(500); }
```

```
System.out.println("main() - end");
```

run()

```
thread.start();
```

```
@Override  
public void run() {  
    System.out.println("MyThread.run() - begin");  
    System.out.println("MyThread.run() - end");  
}
```

스레드 속성

- ID와 이름
- 우선순위
 - 높음, 보통, 낮음
- 상태
 - 생성, 실행, 대기, 종료
- 관계
 - 그룹

스레드 우선 순위

높을 수록 더 자주 CPU 사용시간이 늘어남

- Java가 지원하는 스레드 우선순위
 - Thread.MAX_PRIORITY
 - Thread.NORM_PRIORITY
 - Thread.MIN_PRIORITY
- 스케줄링 정책은 JVM이 결정

스레드 상태의 의미

NEW, TERMINATED

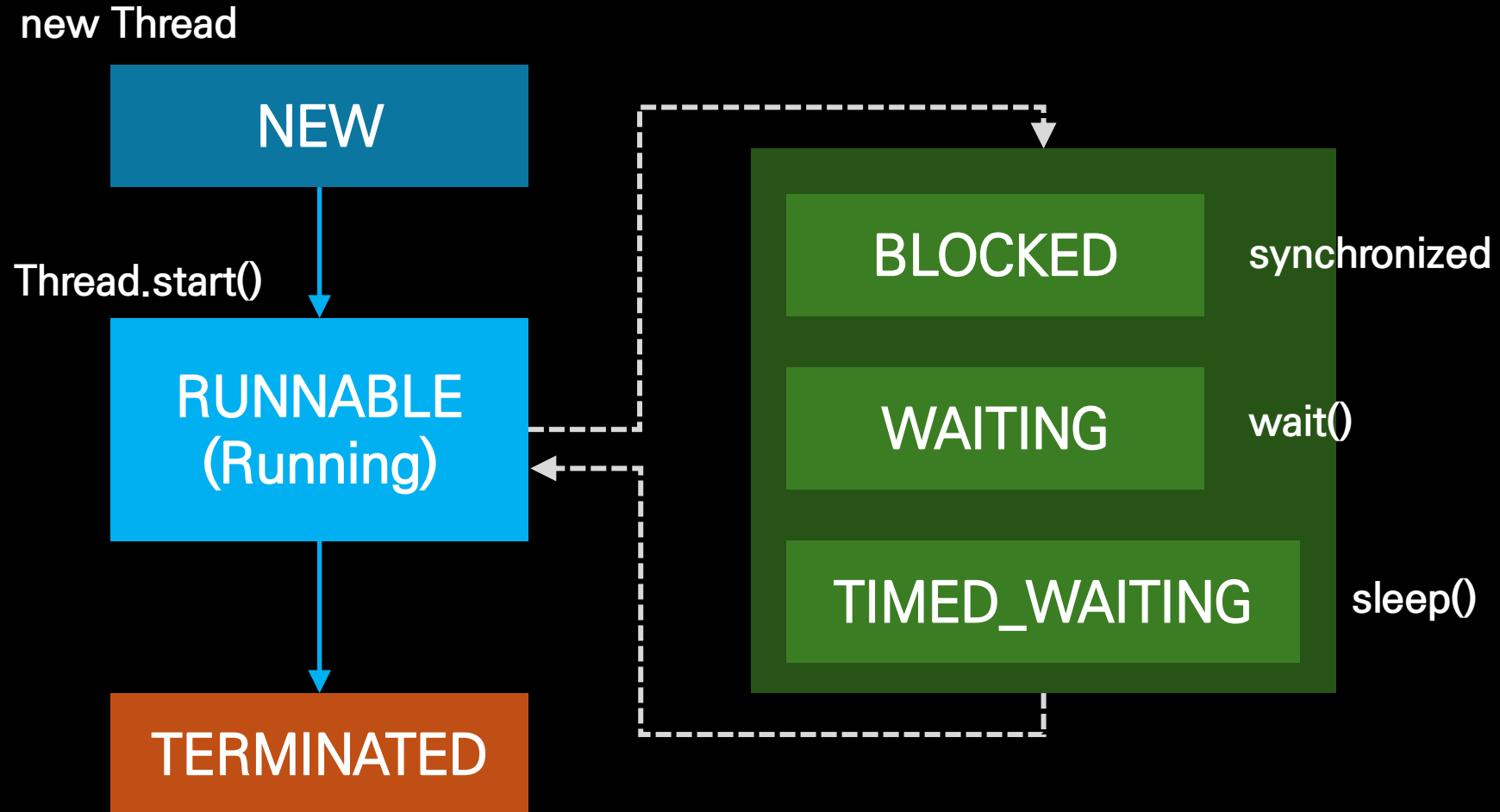
- 스레드 객체가 생성된 상태
 - NEW
- 스레드 작업이 끝난 상태로 `run()` 메서드가 반환한 상태
 - TERMINATED
 - 소멸 대상 인스턴스가 되며 구체적인 시점은 개발자가 알 수 없음

RUNNABLE

- OS 스케줄러가 CPU 타임을 배정 할 수 있는 상태 (RUNNABLE)
- BLOCKED
 - lock 획득을 위한 대기
- WAITING
 - `wait()`, `join()`
- TIME_WAITING
 - `sleep(ms)`, `wait/join(ms)`,

스레드 상태와 생명 주기

스레드가 연산을 시작하면 RUNNABLE 상태



스레드 ID와 이름 붙이기 – 02_threadIdName

```
class MyThread extends Thread {  
    @Override  
    public void run() {  
        System.out.println(getName() + " - begin");  
        System.out.println("Thread ID: " + threadId());  
        System.out.println(getName() + " - end");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Thread mainThread = Thread.currentThread();  
        mainThread.setName("Main thread");  
        System.out.println(mainThread.getName() + " - begin");  
        System.out.println("Thread ID: " + mainThread.threadId());  
        Thread thread = new MyThread();  
        thread.setName("Worker thread");  
        thread.start();  
  
        try{ Thread.sleep(500); } catch (Exception e) {}  
        System.out.println(mainThread.getName() + " - end");  
    }  
}
```



Main thread - begin



Thread ID: 1



Worker thread - begin



Thread ID: 22

Worker thread - end

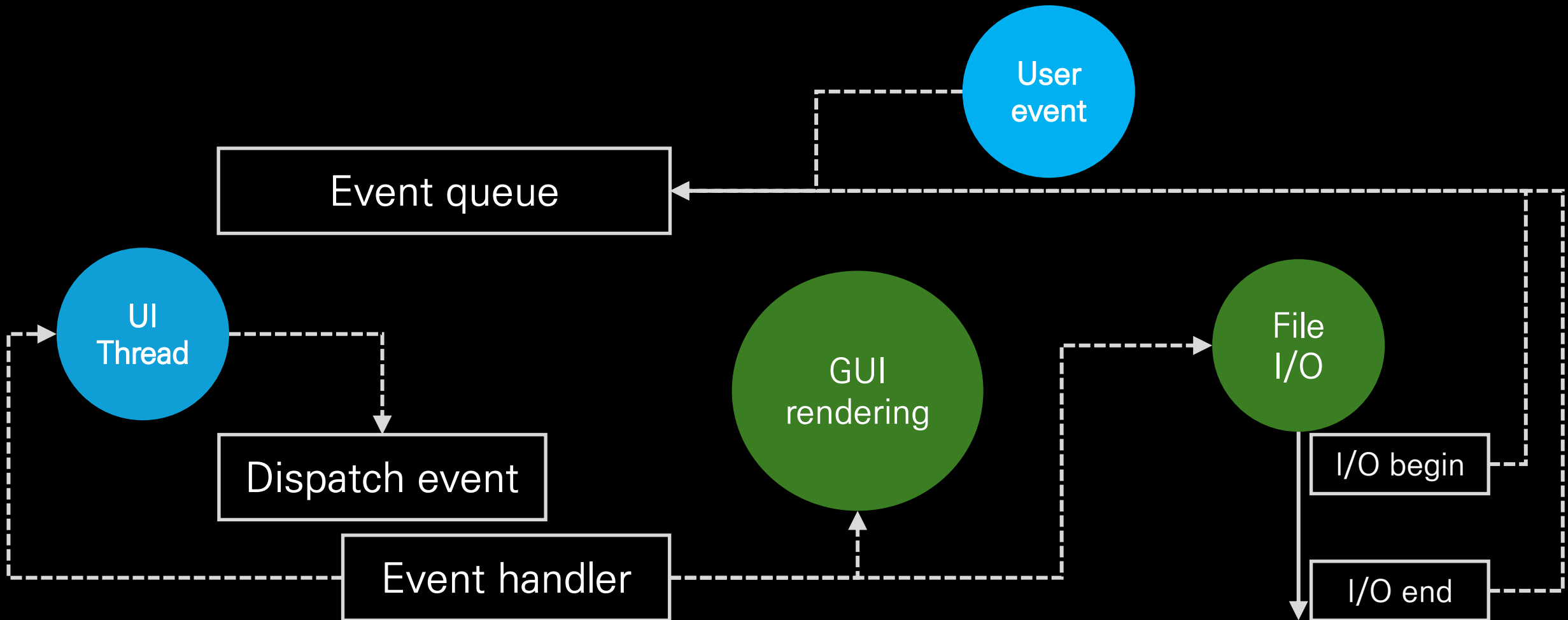
Main thread - end

파일 처리와 UI 분리

- CLI 환경에서 사용자 입력은 이벤트
- 이벤트에 대한 처리를 반복하는 Event-loop 방식은 전형적인 구조
- 파일 처리는 매우 느린 처리이며 완료 시점을 특정하지 못할 경우가 많음
- 느린 입/출력 처리(네트워크 포함)를 별도 스레드로 분리해 Event-loop가 멈추지 않아야 함

파일 처리와 UI 분리

모든 처리코드가 동시에 실행되어야 하는 상황



02_uiAndThread

```
import java.util.Scanner;

class MyThreadForIo extends Thread {
    @Override
    public void run() {
        System.out.println("* File I/O - start *, ID: " + threadId());
        for (int i = 10; i <= 100; i += 10) {
            System.out.printf("TID: %d - %d%%\n", threadId(), i);
            try{ Thread.sleep(1000); } catch (Exception e) {}
        }
        System.out.println("* File I/O - complite *");
    }
}
```

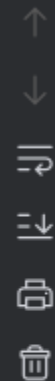
```
public class Main {  
    public static int printMenu() {  
        System.out.println("[1]File\t[2]View\t[3]Edit\t[0]Exit");  
        Scanner scanner = new Scanner(System.in);  
        return scanner.nextInt();  
    }  
  
    public static void main(String[] args) {  
        int input = 0;  
        while ((input = printMenu()) != 0) {  
            if(input == 1) {  
                Thread thread = new MyThreadForIo();  
                thread.start();  
            }  
        }  
    }  
}
```


데몬(Daemon) 스레드

`Thread.setDaemon(true);`

- 메인 스레드 종료 시 강제 종료되는 스레드
- `Thread.start()` 메서드를 호출 전에 설정

```
public static void main(String[] args) {  
    int input = 0;  
    while ((input = printMenu()) != 0) {  
        if(input == 1) {  
            Thread thread = new MyThreadForIo();  
  
            thread.setDaemon(true);  
            thread.start();  
        }  
    }  
}
```



TID: 24 - 30%

0TID: 22 - 40%

TID: 25 - 30%

TID: 23 - 40%

종료 코드 0(으)로 완료된 프로세스

3. 스레드 제어 및 기초적 동기화

sleep()

- 스레드를 일정 시간 동안 Suspend(TIME_WAIT 상태) 시켰다가 시간이 지나면 자동으로 Resume 되어 RUNNABLE 상태로 전환
- 보통 설정한 시간 보다 더 많은 시간이 흐르며 정확성이 떨어짐
- 우연에 맞기는 코드를 만드는 주범

우연에 맡긴 종료 방법 – 03_sleepAndExit

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("main() - begin");  
        Thread thread = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("MyThread.run() - begin");  
                try{ Thread.sleep(500); } catch (Exception e) {}  
                System.out.println("MyThread.run() - end");  
            }  
        });  
        thread.start();  
  
        try{ Thread.sleep(500); } catch (Exception e) {}  
        System.out.println("main() - end");  
    }  
}
```



main() - begin



MyThread.run() - begin



MyThread.run() - end



main() - end



main() - begin



MyThread.run() - begin



main() - end



MyThread.run() - end

exit 플래그 활용

```
public class Main {
    public static boolean exitFlag = false;

    public static void main(String[] args) {
        System.out.println("main() - begin");
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("MyThread.run() - begin");
                while(!exitFlag) {
                    try{ Thread.sleep(1); } catch (Exception e) {}
                }
                System.out.println("MyThread.run() - end");
            }
        });
        thread.start();
    }
}
```

```
        System.out.println("exitFlag = true;");  
        exitFlag = true;  
        try{ Thread.sleep(0); } catch (Exception e) {}  
        System.out.println("main() - end");  
    }  
}
```

↓
≡
≡↓
🖨
🗑

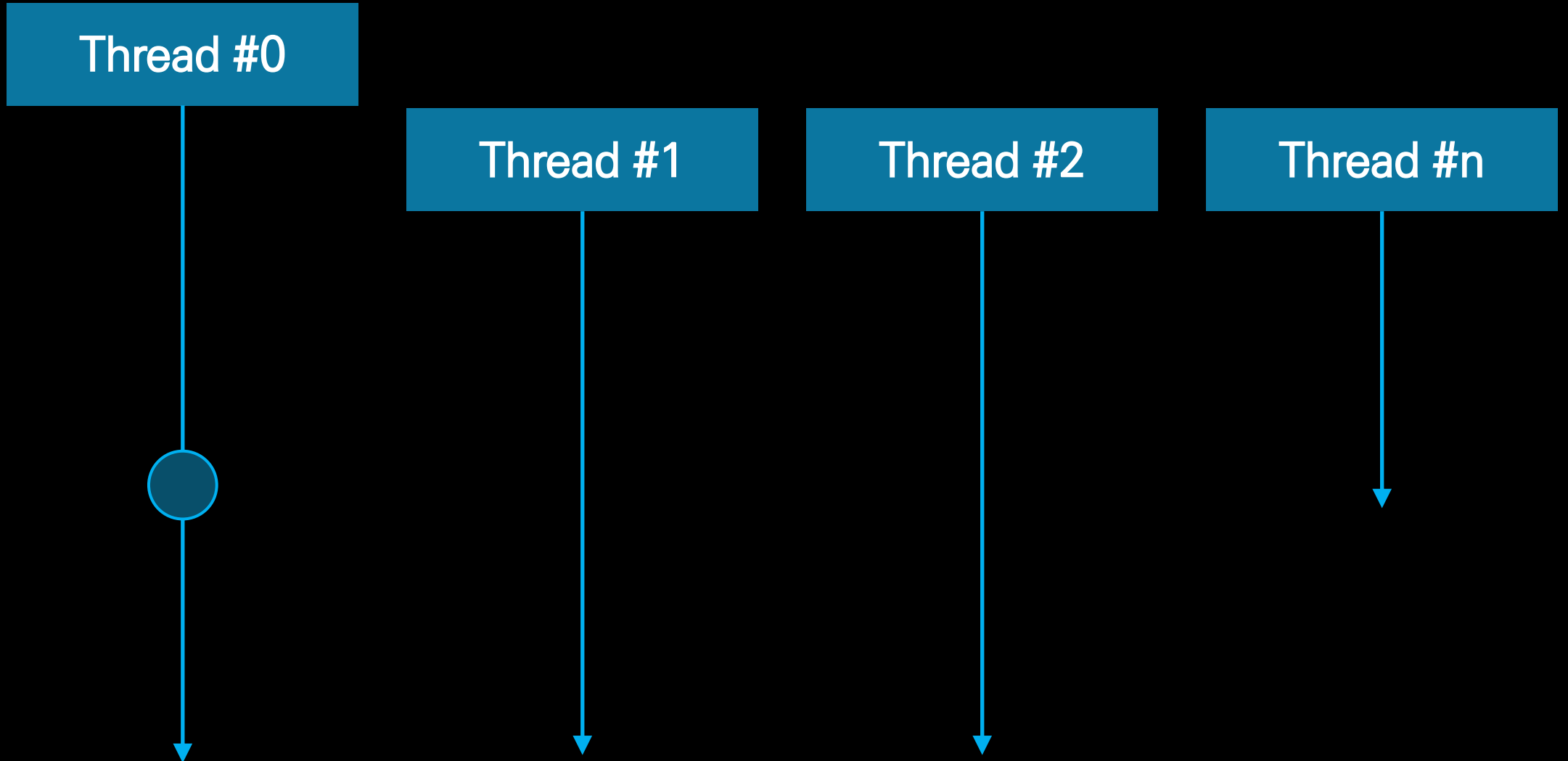
```
main() - begin  
exitFlag = true;  
MyThread.run() - begin  
MyThread.run() - end  
main() - end
```


인터럽트를 이용한 스레드 종료

```
class MyThread extends Thread {  
    private boolean exit = false;  
    @Override  
    public void run() {  
        System.out.println("*** Begin ***");  
        while (!exit) {  
            try {  
                sleep(1000);  
            } catch (InterruptedException e) {  
                System.out.println("* [InterruptedException] *");  
                break;  
            }  
        }  
  
        System.out.println("*** End ***");  
    }  
}
```

```
public class Main {  
    public static int printMenu() {  
        ...  
    }  
  
    public static void main(String[] args) {  
        int input = 0;  
        Thread thread = null;  
        while ((input = printMenu()) != 0) {  
            if(input == 1) {  
                if(thread == null) {  
                    thread = new MyThread();  
                    thread.start();  
                }  
            }  
        }  
        if(thread != null) {  
            thread.interrupt();  
        }  
    }  
}
```

스레드 강제 종료 및 동기화



스레드 종료 시점 동기화 – 03_joinSample

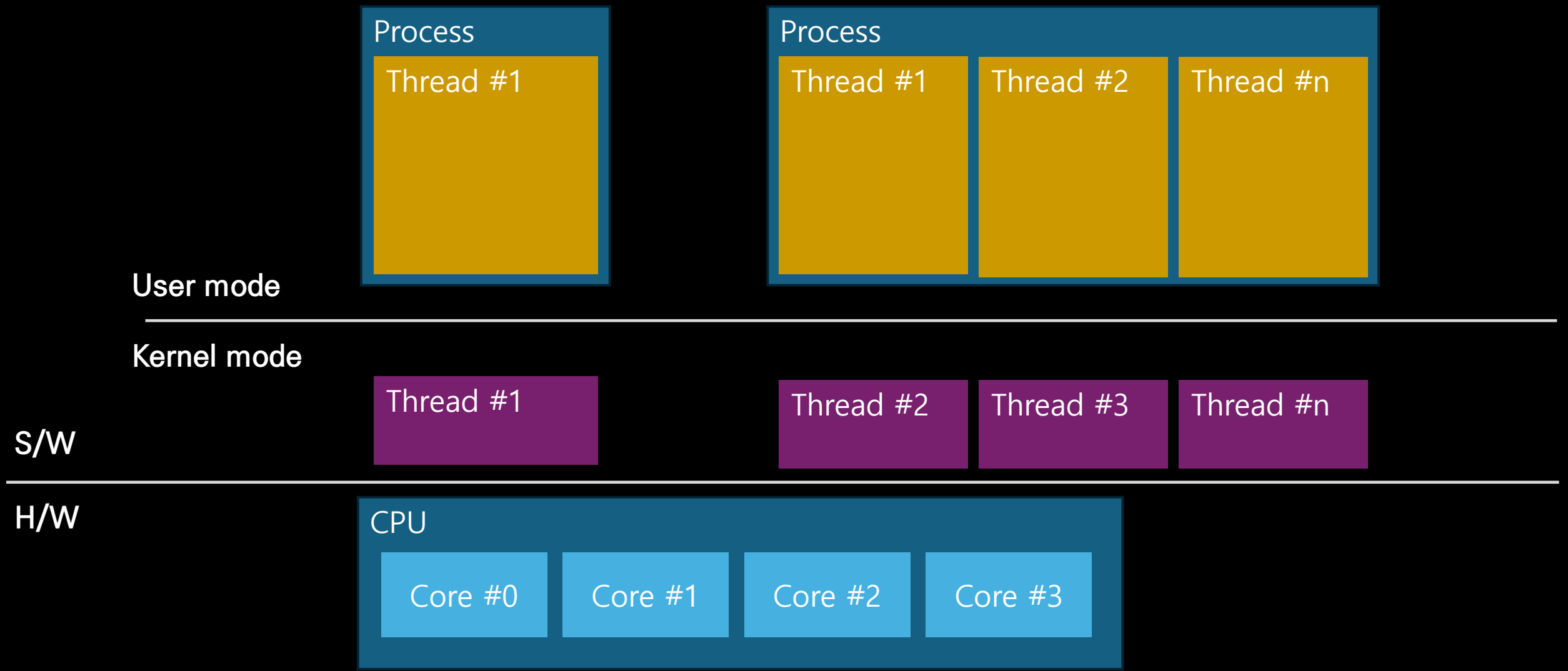
```
class MyThread implements Runnable {
    @Override
    public void run() {
        System.out.println("MyThread.run() - begin");
        for(int i = 0; i < 10; ++i) {
            try {
                sleep(1);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        System.out.println("MyThread.run() - end");
    }
}
```

```
public class Main {  
    public static void main(String[] args) throws InterruptedException {  
        Thread[] threads = new Thread[3];  
        for(int i = 0; i < 3; ++i) {  
            threads[i] = new Thread(new MyThread(), "TestThread");  
            threads[i].start();  
        }  
  
        for(int i = 0; i < 3; ++i)  
            threads[i].join();  
        System.out.println("main() - end");  
    }  
}
```

↓	MyThread.run() - begin
⇌	MyThread.run() - begin
⇌	MyThread.run() - begin
🖨	MyThread.run() - end
🗑	MyThread.run() - end
	MyThread.run() - end
	main() - end

4. JVM과 스레드

플랫폼 스레드와 커널 스레드



스위칭에 드는 비용

현재 상태 백업 및 이전 상태 복원이 스위칭

- 어떤 연산 흐름의 ‘현재 상태’라는 것은 CPU core 레지스터 값의 스냅샷
- 다른 흐름으로 전환(Context switching)한다는 것은 CPU core의 레지스터 값을 본래의 것으로 복원하는 것
 - OS의 TCB(Thread Control Block)에 담기는 정보를 이용해 복원

JVM 구성요소 – PC 레지스터

Class loader

Loading
(Bootstrap/Extension/Application class loader)

Linking
Verify → Prepare → Resolve

Initialization

Runtime data area

Method area
(Runtime constant pool)

Heap area

Stack area

PC register

Native method stack

Execution engine

Interpreter

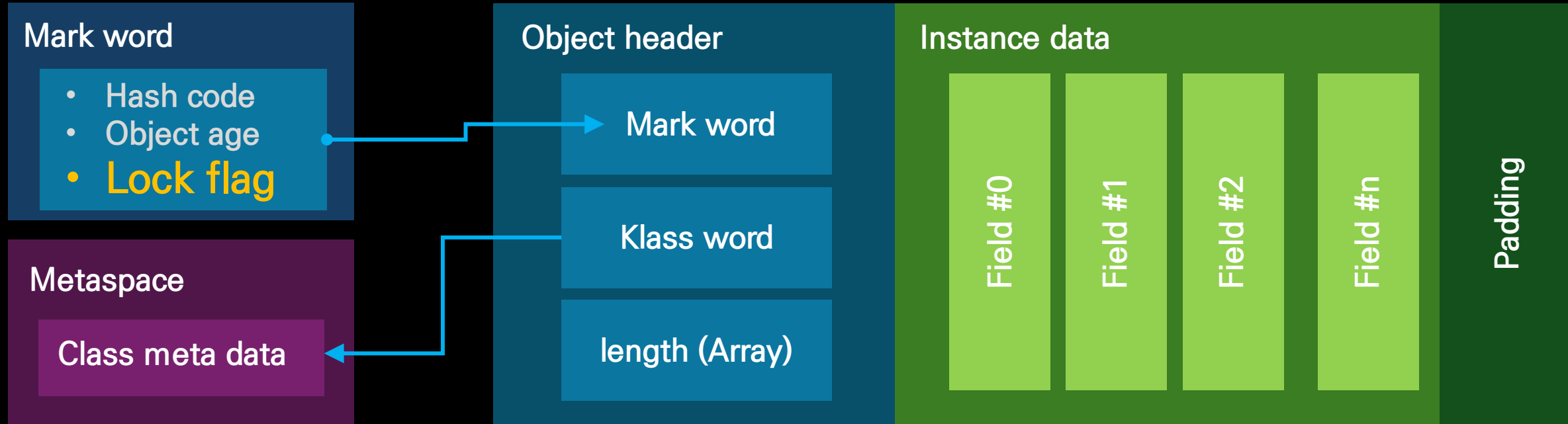
JIT compiler

Garbage collector

JNI,
Native method
interface

Native method
library

객체 메모리 레이아웃과 해시코드



- Hash code는 `Object.hashCode()` 함수가 호출되는 시점에 계산
- 나이는 GC에서 살아남은 횟수
- **Lock flag**는 객체를 중심으로 멀티스레드 환경에서 경쟁조건이 발생하는 문제를 해결하기 위한 것

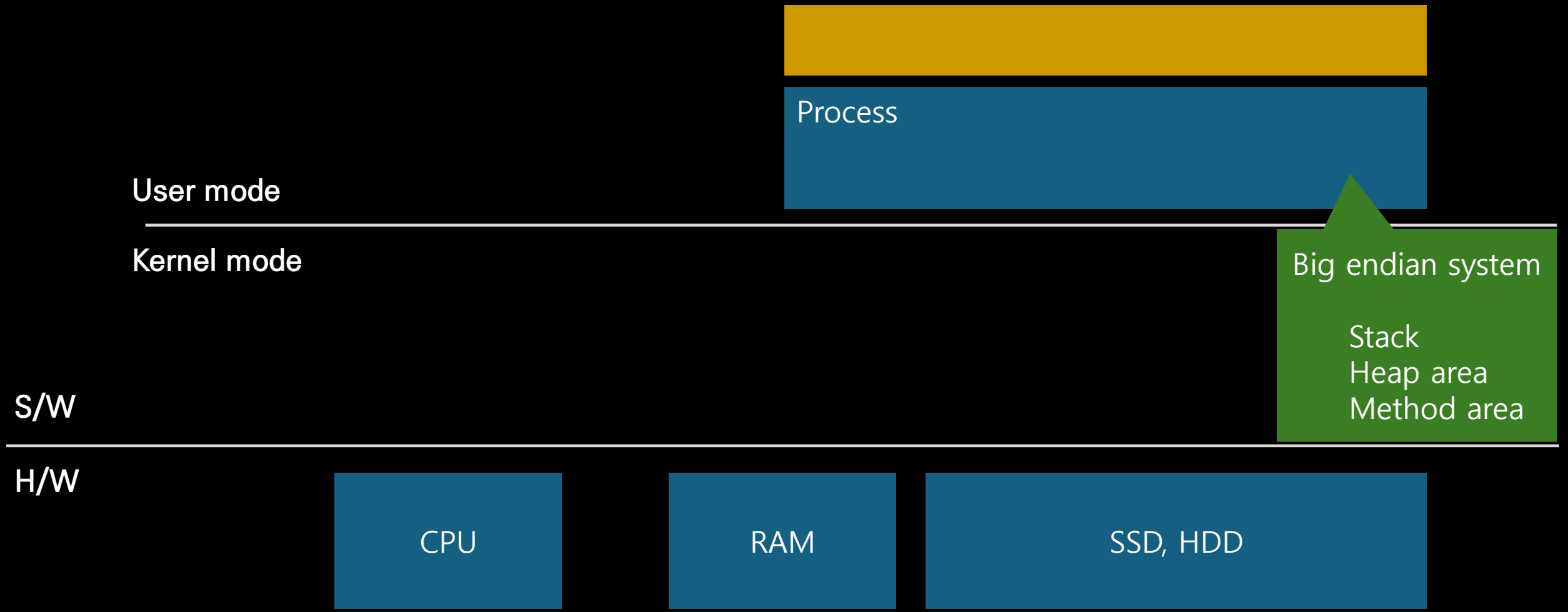
하스팟 VM 객체 Lock flag

Lock flag	상태	Mark word 저장 정보
00	Lightweight locking	Lock 레코드 (스핀락 동기화)
01	Unlock	객체의 Hash code 및 나이
01	Biased locking	스레드 ID, 타임스탬프, 객체 나이
10	Heavyweight locking	- (뮤텍스로 동기화)
11	GC mark	- (GC가 객체 이동 중)

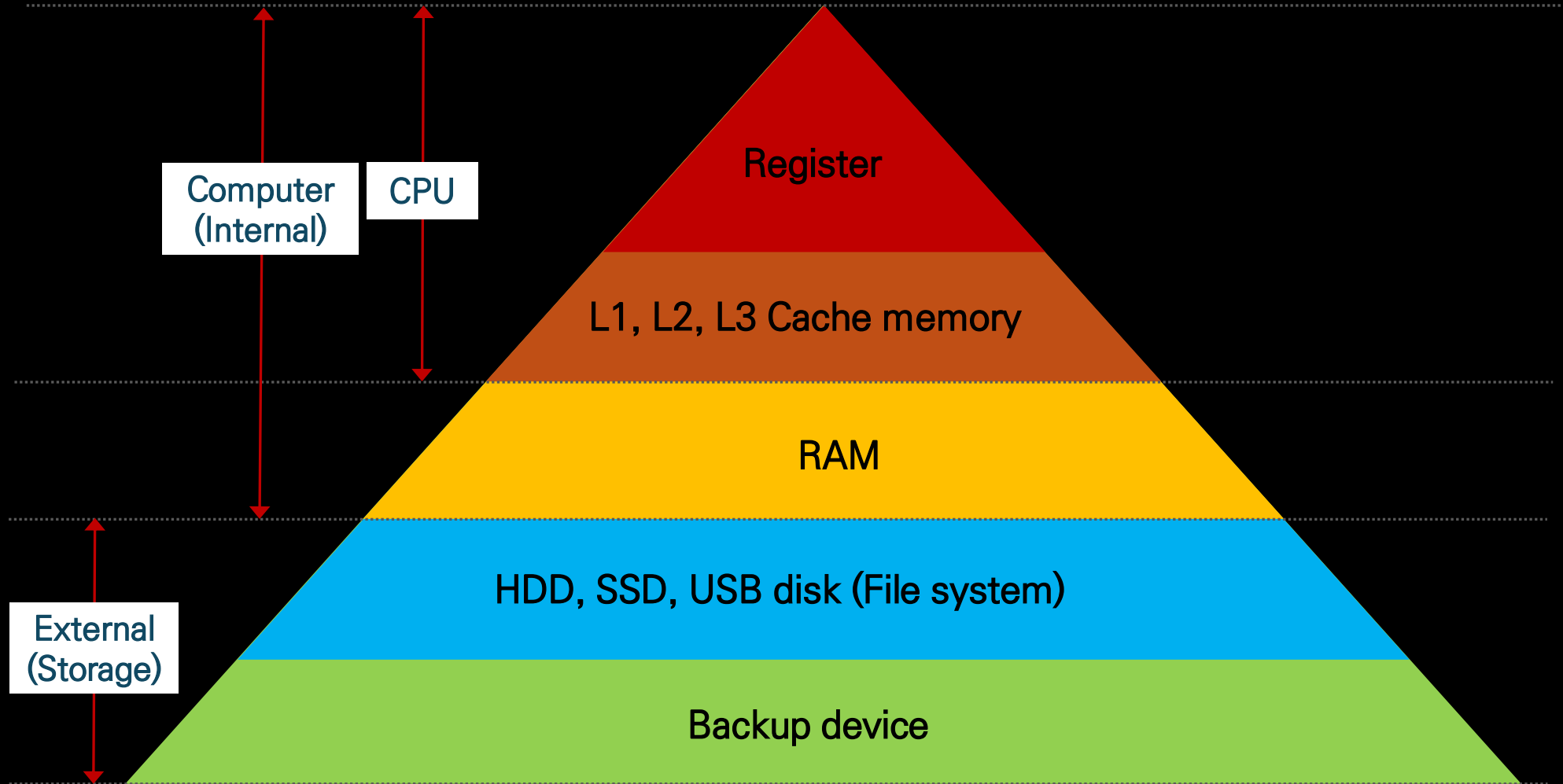
JVM의 객체 수준 Lock

- Java에서 모든 함수와 자료는 class에 속함
- 정적 멤버의 경우 인스턴스 없이 별도로 존재하며 동기화가 필요할 경우 임계 영역 통제를 위한 인스턴스가 필요함 (Monitor lock)

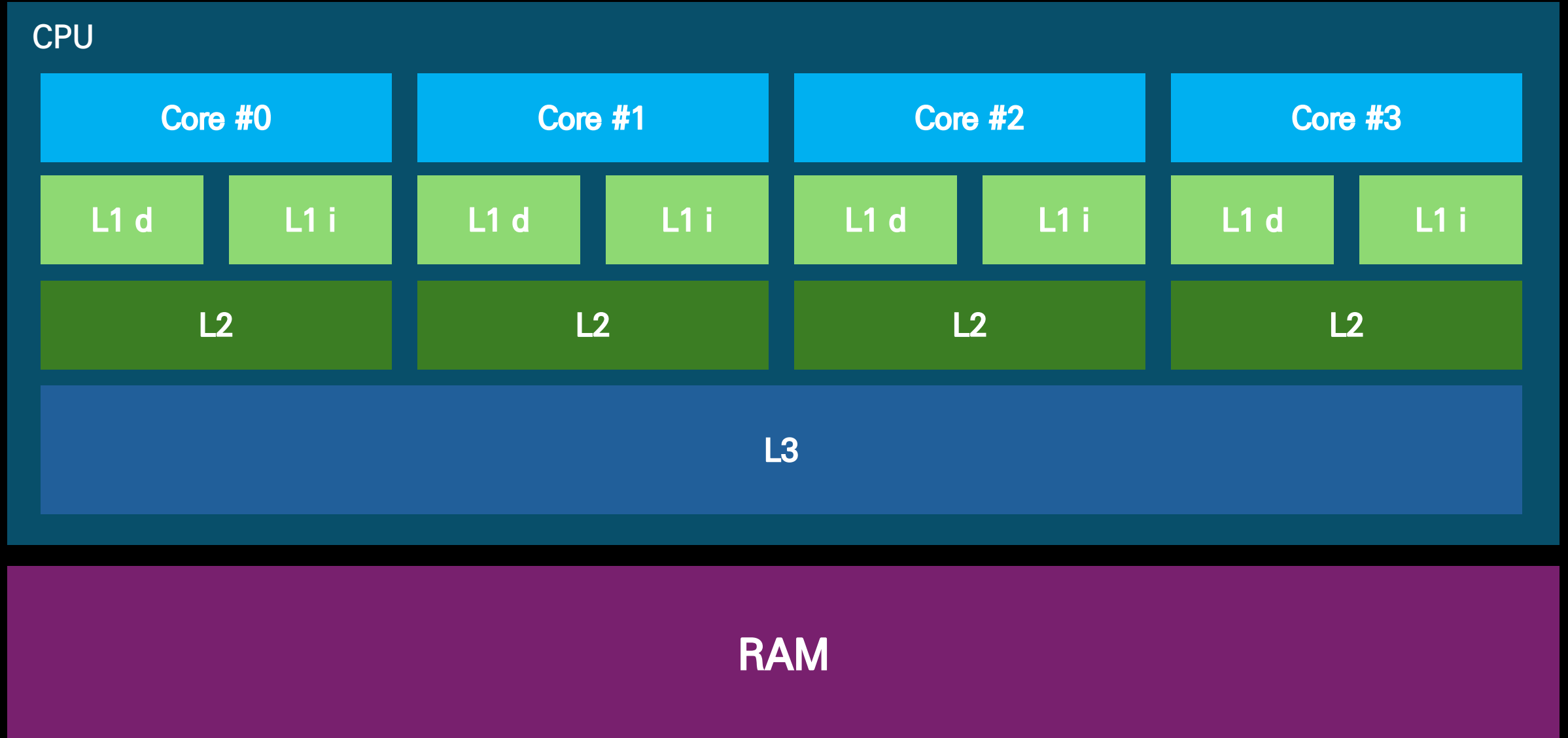
User mode process JVM



메모리 구조



캐시 일관성 이슈



JVM과 컴퓨터 구조

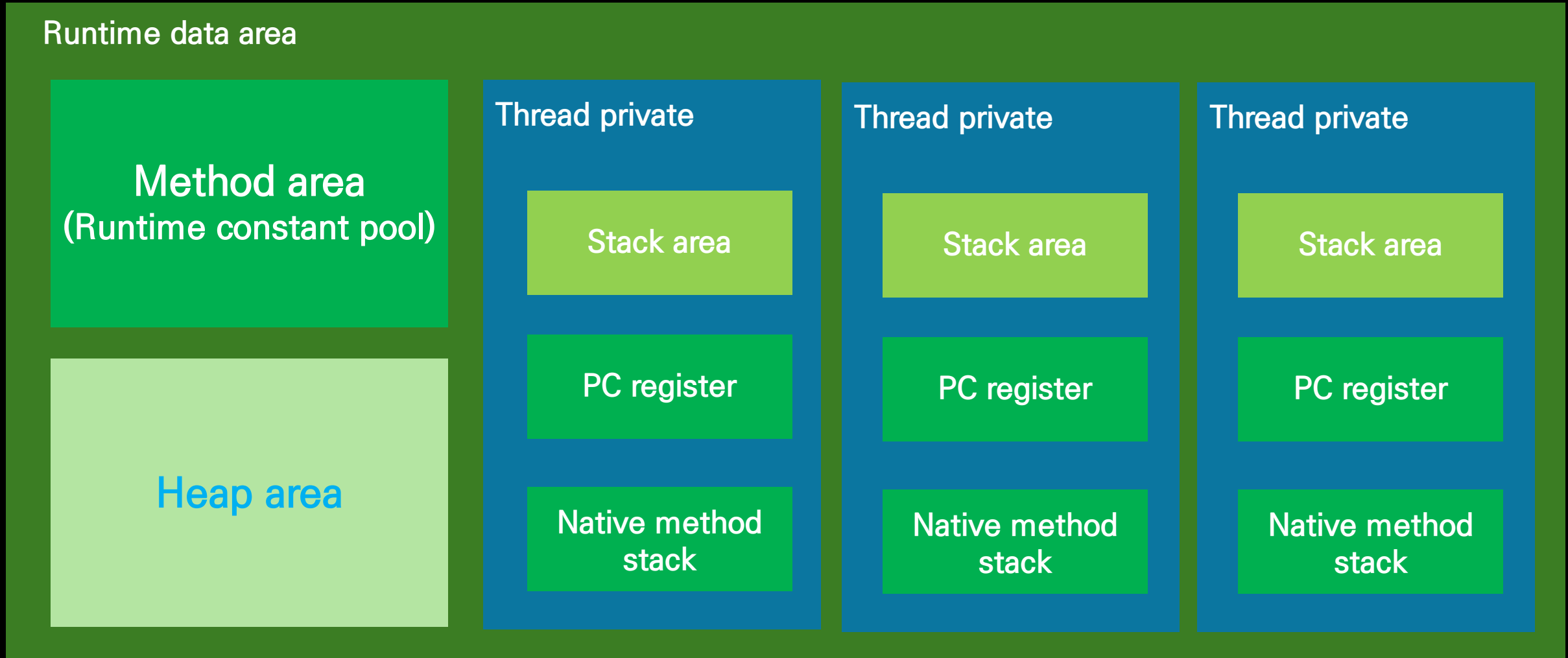
JVM은 H/W + OS 같은 역할을 모두 수행

- C/C++ 개발자 관점의 JVM은 사용자 모드 응용 프로그램
- CPU 캐시 등 실제 H/W 수준을 통제하는 대부분의 코드는 C/C++ 기반 코드이며 Java에서는 불가능
 - CPU 캐시 메모리 통제 API는 Native method로 구현
 - Java 응용 프로그램이 특정 환경에 종속되지 않기 위한 것

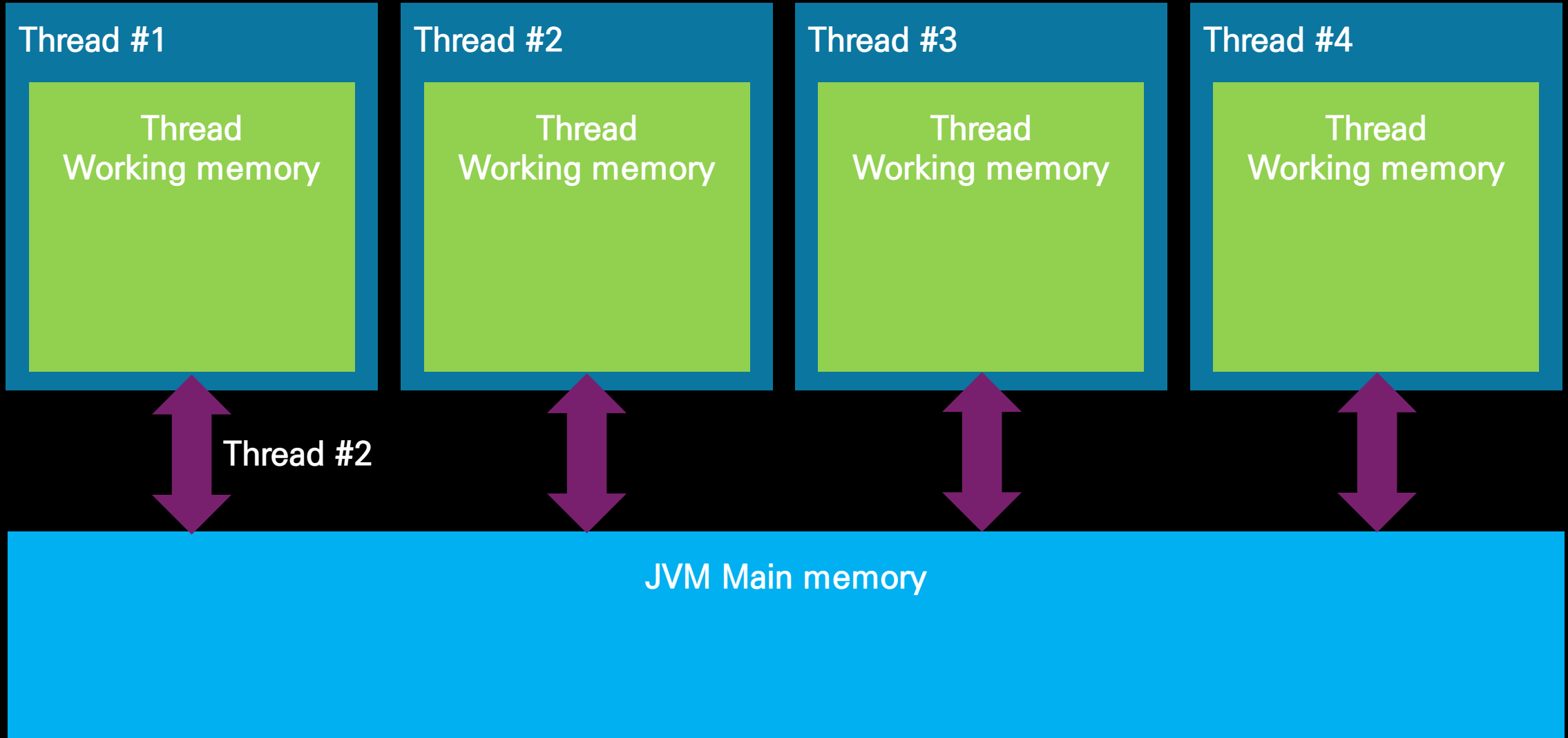
JVM 메인 메모리와 작업 메모리

- Java 메모리 모델의 핵심 목표는 변수에 접근(읽기, 쓰기) 규칙을 정하는 것
 - 메인 메모리와 작업 메모리로 구분
 - 지역변수와 매개변수는 제외 (Stack 사용)
- 모든 변수는 JVM 메인 메모리에 저장된다고 규정
- 작업 메모리는 스레드가 사용하는 변수의 사본이 저장되며 스레드 내부에 연산은 작업 메모리에만 반영
 - 스레드마다 독립적인 작업 메모리가 존재하며 접근 불가
 - 스레드는 JVM 메인 메모리 직접 접근 불가

JVM runtime data area



JVM 메인 메모리와 작업 메모리



JVM 메인 메모리와 작업 메모리

- 읽기(Read)는 메인 메모리에서 변수의 값을 읽어 작업 메모리로 전송하는 것
- 적재(Load)는 메인 메모리가 전송한 값을 작업 메모리(사본)에 저장하는 것
- 저장(Store)은 작업 메모리 변수의 값을 메인 메모리로 전송하는 것
- 쓰기(Write)는 작업 메모리가 보내준 값을 메인 메모리 변수에 반영(저장)하는 것

JVM 메인 메모리와 작업 메모리

작업 메모리와 메인 메모리 동기화 이슈

- 작업 메모리의 변화는 메인 메모리에 즉시 반영되지 않고 일정 시간 지연됨 (일괄 처리에 따른 성능 향상)
- 알려진 일반 변수의 동기화 시점
 - 명시적 동기화(synchronized, volatile)
 - Thread.start(), join() 호출
 - Lock, Atomic 클래스 사용
 - 클래스 로딩 과정에서 정적 변수 초기화 시
 - 기타 JVM이 정한 최적화, 동기화 기준 충족 시

작업 메모리 동기화 – 스레드 시작, 종료

- 새로운 스레드가 시작될 때 부모 스레드의 작업 메모리에 저장된 변수 값을 메인 메모리로 동기화
 - 만일 기존에 이미 실행 중인 스레드가 있을 경우 이 시점에 동기화된 값을 확인 할 수 있음
- 새로 시작된 스레드는 동기화가 완료된 메인 메모리에서 변수 값을 로딩

작업 메모리 동기화 이슈 – 04_workMemoryError

```
static boolean exitFlag = false;

public static void main(String[] args) throws InterruptedException{
    System.out.println("[main] begin");

    MyThread myThread = new MyThread();
    Thread t1 = new Thread(myThread, "TestThread");
    t1.start();

    Thread.sleep(100);
    exitFlag = true;
    Thread.sleep(2000);

    System.out.println("[main] end, exitFlag: " + exitFlag);
}
```

```
public static class MyThread implements Runnable {  
    @Override  
    public void run(){  
        System.out.println("MyThread.run() - begin");  
        int counter = 0;  
        while(!exitFlag) {  
            ++counter;  
        }  
  
        System.out.println("MyThread.run() - end, exitFlag: " + exitFlag);  
        System.out.println("MyThread.run() - end, counter: " + counter);  
    }  
}
```



[main] begin



MyThread.run() - begin



[main] end, exitFlag: true



volatile이 갖는 의미

- 멀티스레드 환경에서 여러 스레드가 접근하는 변수(메모리)에 대해 가시성(Visibility)을 제공
 - 스레드 코드가 해당 변수에서 값을 읽을 때마다 메인 메모리와 동기화 (variable modifier)
- C/C++에서는 컴파일러 최적화 규칙을 적용하지 않기 위해 사용
 - Java에서 성능향상을 위한 명령어 재정렬 최적화를 적용하지 않음 (최적화 방지)
 - 눈에 보이는 코드와 실제 실행 흐름은 다를 수 있음!

04_volatileFlag

```
static volatile boolean exitFlag = false;

public static void main(String[] args) throws InterruptedException{
    System.out.println("[main] begin");

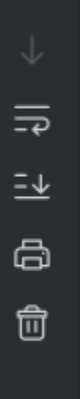
    MyThread myThread = new MyThread();
    Thread t1 = new Thread(myThread, "TestThread");
    t1.start();

    Thread.sleep(100);
    exitFlag = true;
    Thread.sleep(2000);

    System.out.println("[main] end, exitFlag: " + exitFlag);
}
```

```
public static class MyThread implements Runnable {
    @Override
    public void run(){
        System.out.println("MyThread.run() - begin");
        int counter = 0;
        while(!exitFlag) {
            ++counter;
        }

        System.out.println("MyThread.run() - end, exitFlag: " + exitFlag);
        System.out.println("MyThread.run() - end, counter: " + counter);
    }
}
```



```
[main] begin
MyThread.run() - begin
MyThread.run() - end, exitFlag: true
MyThread.run() - end, counter: 317054774
[main] end, exitFlag: true
```

경쟁 조건 (Race condition)

- 경쟁 조건은 한 대상에 대해 여러 스레드가 동시에 접근하면서 발생하는 동기화 이슈
- 논리적 오류에 해당하지만 컴파일 타임 오류로 확인 하기 어려울 수 있음 (개발자 역량으로 대응)
- C/C++ 같은 네이티브 코드도 예외가 아니며 Java의 경우 C/C++의 포인터를 이용한 참조 경우와 유사

04_raceConditionCounter

```
public class Main {
    static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        System.out.println("main - begin");

        Thread[] threads = new Thread[5];
        for(int i = 0; i < 5; ++i) {
            threads[i] = new Thread(new MyThread(), "TestThread" + i);
            threads[i].start();
        }

        for(int i = 0; i < 5; ++i)
            threads[i].join();

        System.out.println("main - end, counter: " + counter);
    }
}
```

```
public static class MyThread implements Runnable {  
    @Override  
    public void run() {  
        for(int i = 0; i < 1000000; ++i) {  
            ++counter;  
        }  
    }  
}
```



main - begin

main - end, counter: 173556



main - begin

main - end, counter: 118313

C++의 느린 참조 이슈

```
4 volatile int g_counter = 0;
5 void threadCounter() {
6     volatile int* pCounter = &g_counter;
7     for (int i = 0; i < 100000; ++i)
8         ++(*pCounter);
9 }
10
11 int main() {
12     std::thread t1(threadCounter);
13     std::thread t2(threadCounter);
14     t1.join();
15     t2.join();
16     std::cout << "Counter: " << g_counter << "\n";
17 }
```

Microsoft Visual Studio 디버그

Counter: 122843

F:\독하게 시작하는 Java - Part 3\C++\counterThread\x64\Debug\counterThread.exe(프로세스 39368)이(가) 0 코드(0x0)와 함께 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요 ...|

기계어 수준에서 발생하는 차이

고급어의 한 행은 여러 기계어로 구성

```
    for (int i = 0; i < 100000; ++i)
    {
        00007FF7EDDB1020  mov             eax,186A0h
        00007FF7EDDB1025  nop
                        word ptr [rax+rax]
                        ++(*pCounter);
        00007FF7EDDB1030  mov             ecx,dword ptr [g_counter (07FF7EDDB570Ch)]
        00007FF7EDDB1036  inc             ecx
        00007FF7EDDB1038  mov             dword ptr [g_counter (07FF7EDDB570Ch)],ecx
        00007FF7EDDB103E  sub             rax,1
        00007FF7EDDB1042  jne             threadCounter+10h (07FF7EDDB1030h)
    }
```


빌드 모드 변경에 따른 결과 차이

컴파일러 최적화에 따른 결과 보정

```
00007FF7BFB3101F  int          3
--- F:\독하게 시작하는 Java - Part 3\C++\counterThread\counterThread\
    int* pCounter = &g_counter;
00007FF7BFB31020  add          dword ptr [g_counter (07FF7BFB3570Ch)]
    for (int i = 0; i < 100000; ++i)
        ++(*pCounter);
}
00007FF7BFB3102A  ret
```

Microsoft Visual Studio 디버그

Counter: 200000

F:\독하게 시작하는 Java - Part 3\C++\counterThread\x64\Release\counterThread.exe(프로세스 38500)이(가) 0 코드(0x0)와 함께 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요 ...|

최적화 회피에 따른 근접

volatile + Release build

```
--- F:\독하게 시작하는 Java - Part 3\C++\counterThread\counterThread\
    volatile int* pCounter = &g_counter;
    for (int i = 0; i < 100000; ++i)
00007FF729271020  mov     eax,186A0h
00007FF729271025  nop
    ++(*pCounter);
00007FF729271030  mov     ecx,dword ptr [g_counter (07FF72927570Ch)]
00007FF729271036  inc     ecx
00007FF729271038  mov     dword ptr [g_counter (07FF72927570Ch)],ecx
00007FF72927103E  sub     rax,1
00007FF729271042  jne     threadCounter02+10h (07FF729271030h)
```

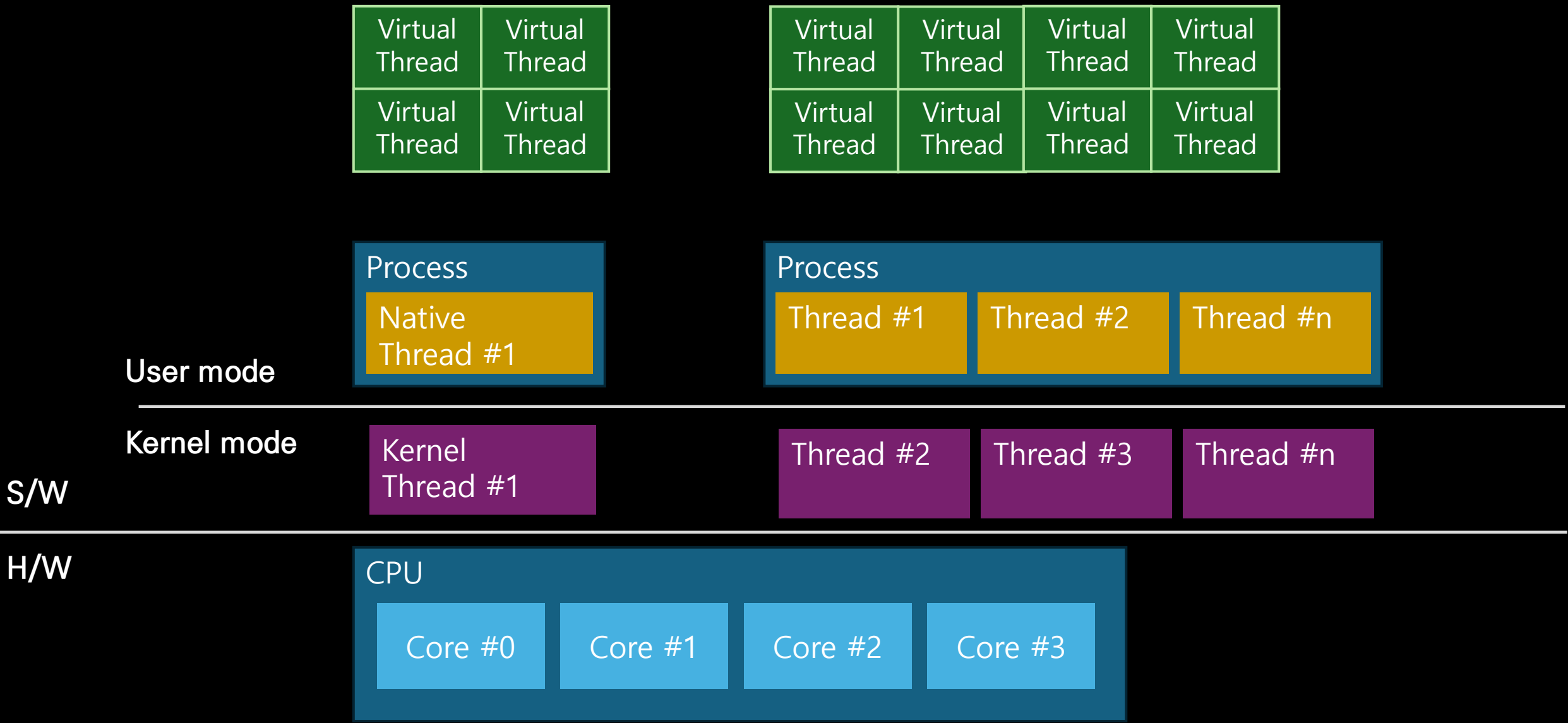
Microsoft Visual Studio 디버그

Counter: 199591

가상 스레드 개념

- 가상 스레드는 실제로 한 (커널)스레드에서 분리된 여러 단위 코드 실행하는 개념
 - CPU core 1개가 여러 스레드 코드를 번갈아 가며 실행하더라도 동시에 실행되는 것처럼 보이는 것과 같은 원리
 - 커널 스레드는 통상 전용 Native API를 통해 생성되며 OS가 직접 제어
- 마치 각각이 개별 스레드로 보이지만 실제로는 한 스레드로 실행되며 이를 기반으로 스위칭 오버헤드를 줄일 수 있음

플랫폼 스레드와 커널 스레드

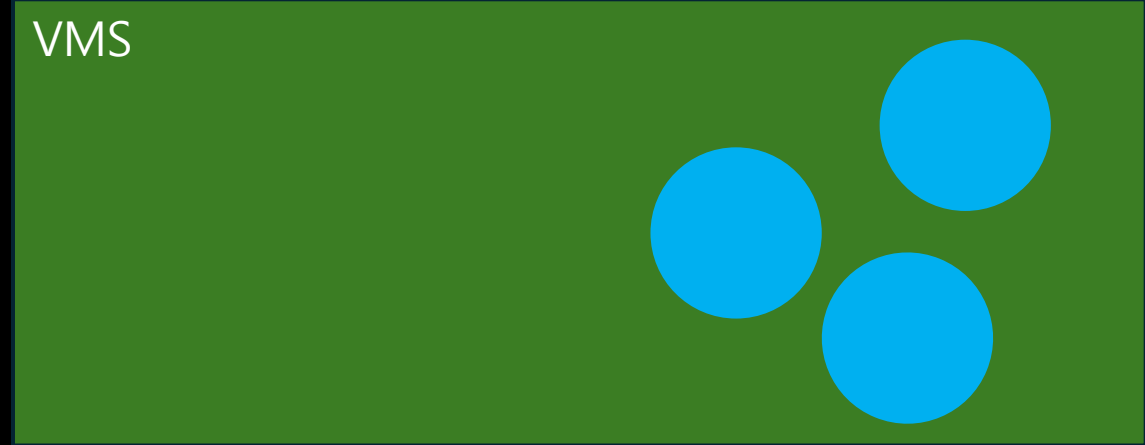
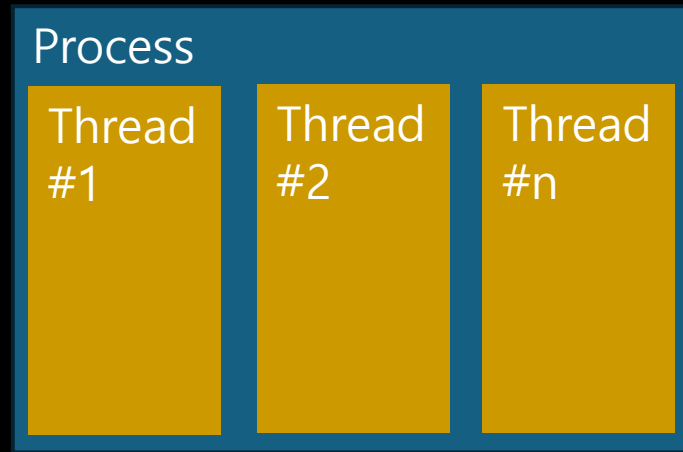


5. 스레드 동기화

Windows OS 수준 커널 객체 종류

- C/C++ 코드에서 커널 객체는 포인터 혹은 카운터로 사용되는 메모리로 생각 할 수 있음
 - Event
 - Mutex
 - Semaphore
- 커널 객체는 생성할 수 있는 개수가 제한적이며 Spin lock에 비해 무거운 것으로 볼 수 있음
 - Spin lock은 객체가 아니며 CPU를 계속 사용하는 반복문

커널 객체



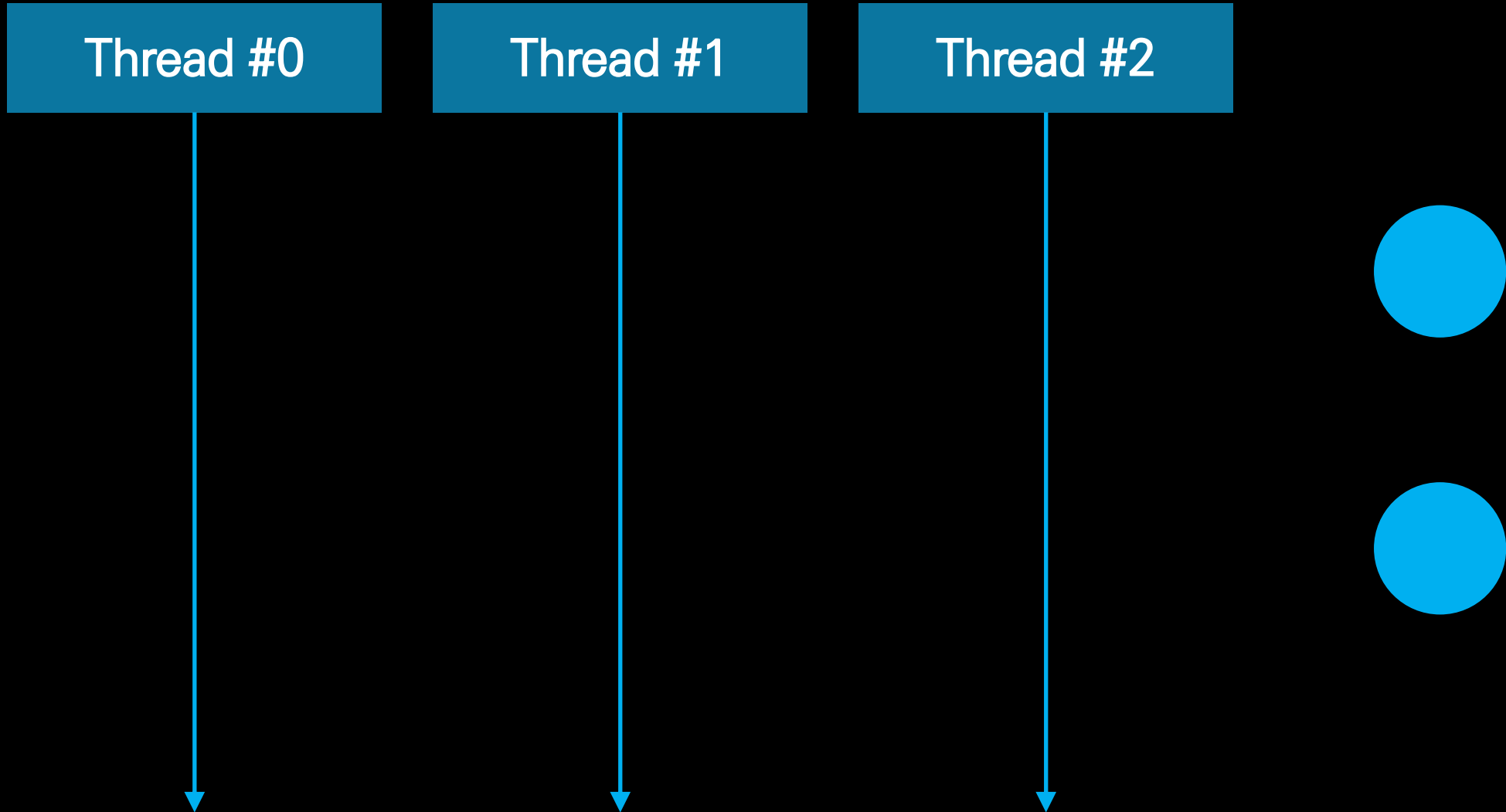
User mode

Kernel mode

S/W

H/W

커널 객체를 이용한 스레드 동기화

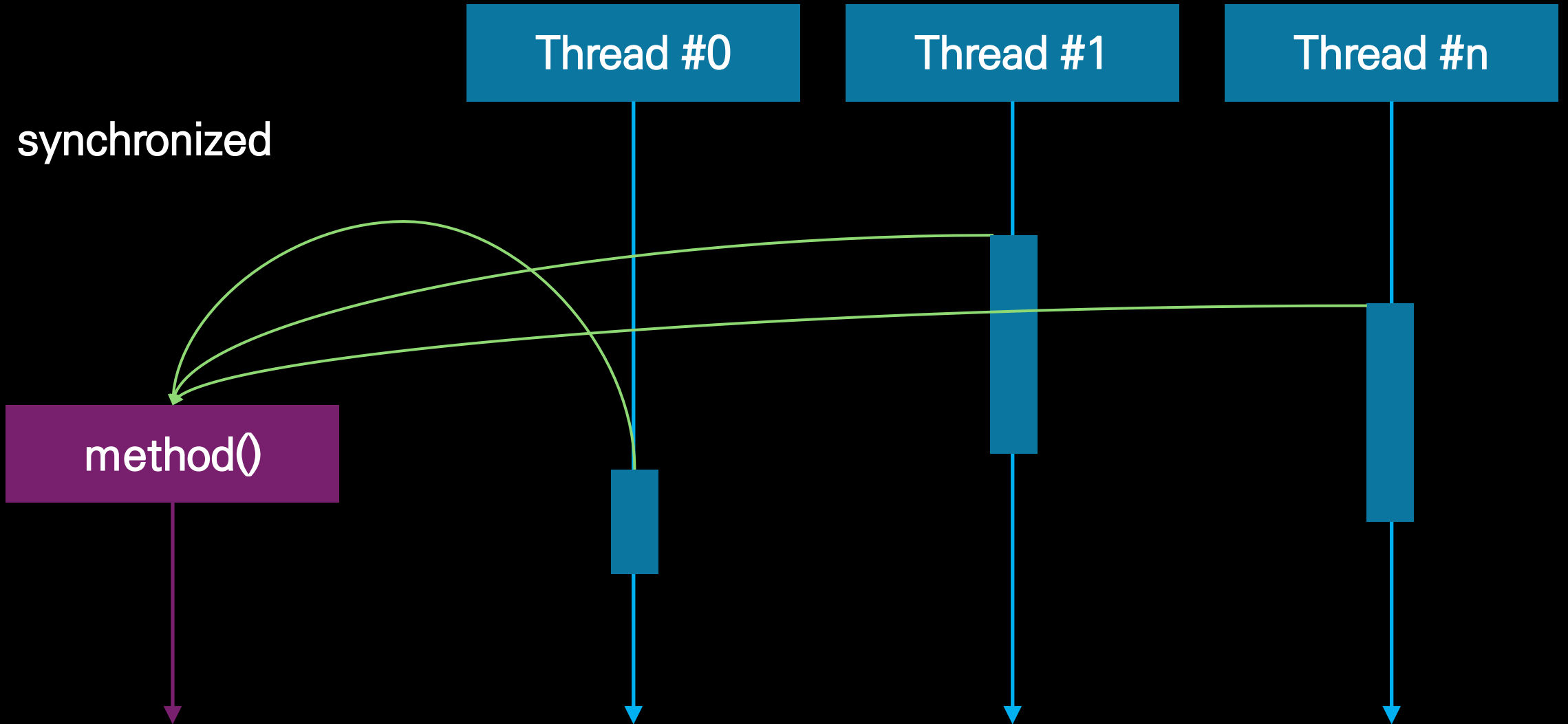


동기화 메서드

`synchronized void testFunc()`

- Critical section(임계 영역) 기반 동기화 기법으로 메서드 코드 전체를 임계 영역으로 설정
- 메서드를 여러 스레드에서 호출하더라도 동시 실행이 허용되지 않음
 - 특정 스레드가 동기화 메서드를 호출하고 코드가 실행되는 동안 다른 스레드에서 동기화 메서드를 호출할 경우 BLOCKED 상태로 전환

스레드 실행 시점 동기화



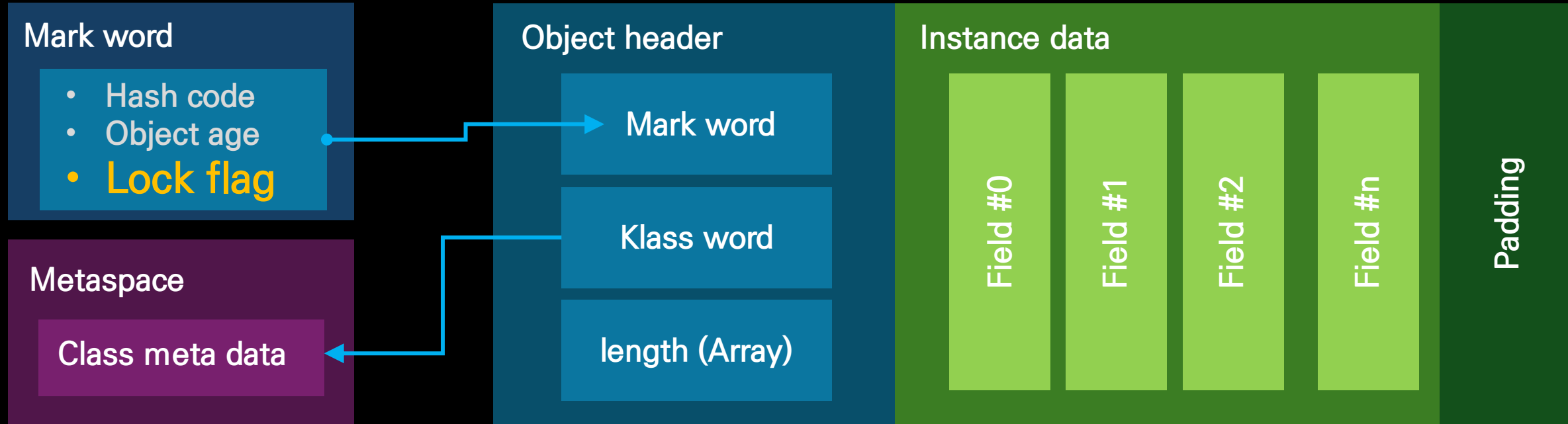
모니터 락 동작 순서

- ① (호출자) 스레드가 synchronized 메서드 (혹은 코드 블록) 호출 시 **모니터 락을 요청**
- ② 타 스레드가 모니터 락을 **이미 점유하고 있다면** 모니터 락을 요청한 스레드는 **Runnable 상태에서 Blocked 상태로 전환**
- ③ 기존 모니터 락 점유 스레드가 구간을 벗어나 모니터 락을 해제하면 **Blocked 상태로 대기 중인 스레드 중 하나가 락을 얻고 동기화 구간에 진입**
(코드 구간에 대한 원자성 보장)

모니터 락(Monitor lock)

- **모니터 락**(혹은 모니터)는 Java에서 사용되는 객체 **인스턴스 단위 락**을 의미
- **synchronized** 기반 동기화를 시도할 경우 해당 인스턴스의 모니터 락을 획득해야 실행 가능
- **모니터 락을 획득 할 수 있는 스레드는 오직 한 스레드만 허용**

객체 메모리 레이아웃과 해시코드



- Hash code는 `Object.hashCode()` 함수가 호출되는 시점에 계산
- 나이는 GC에서 살아남은 횟수
- Lock flag는 객체를 중심으로 멀티스레드 환경에서 경쟁조건이 발생하는 문제를 해결하기 위한 것

05_synchronizedCounter

```
static int counter = 0;
public synchronized static void incCounter() {
    ++counter;
}

public static void main(String[] args) throws InterruptedException {
    System.out.println("main - begin");

    Thread[] threads = new Thread[3];
    for(int i = 0; i < 3; ++i) {
        threads[i] = new Thread(new MyThread(), "TestThread" + i);
        threads[i].start();
    }

    for(int i = 0; i < 3; ++i)
        threads[i].join();

    System.out.println("main - end, counter: " + counter);
}
```

```
public static class MyThread implements Runnable {  
    @Override  
    public void run() {  
        for(int i = 0; i < 1000000; ++i) {  
            //++counter;  
            incCounter();  
        }  
    }  
}
```



main - begin



main - end, counter: 300000



05_counterObject

synchronized가 적용되는 것은 속한 클래스 인스턴스

```
class MyCounter {  
    private int counter = 0;  
    public int getCounter() {  
        return counter;  
    }  
  
    public void incCounter() {  
        ++counter;  
    }  
    public void synchoIncCounter() {  
        synchronized (this) {  
            ++counter;  
        }  
    }  
}
```



```
class MyThread extends Thread {  
    public final MyCounter counter;  
    MyThread(MyCounter cnt) {  
        counter = cnt;  
    }  
  
    synchronized void incInMyThread() {  
        counter.incCounter();  
    }  
  
    static synchronized void incStatic(MyThread thread) {  
        thread.counter.incCounter();  
    }  
}
```

```
@Override
public void run() {
    for(int i = 0; i < 1000000; ++i) {
        incInMyThread();
        //incStatic(this);
        //counter.synchoIncCounter();
    }
}
}
```

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("main - begin");
        MyCounter cnt = new MyCounter();

        Thread[] threads = new Thread[3];
        for(int i = 0; i < 3; ++i) {
            threads[i] = new MyThread(cnt);
            threads[i].start();
        }
    }
}
```

동기화를 하지 않을 경우

```
@Override
public void run() {
    for(int i = 0; i < 1000000; ++i) {
        counter.incCounter();
        //incInMyThread();
        //incStaticInMyThread(this);
        //counter.synchoIncCounter();
    }
}
```

```
↑ main - begin
↓ main - end, counter: 131044
_
```

각 스레드 일반 메서드 동기화

일반 메서드는 각 인스턴스 Lock 별도 사용

```
class MyThread extends Thread {  
    ...  
    synchronized void incInMyThread() {  
        counter.incCounter();  
    }  
}
```

```
↓  
⇌  
⇓  
main - begin  
main - end, counter: 184253
```

스레드 static 메서드 동기화

static 메서드는 모든 인스턴스에 적용

```
static synchronized void incStaticInMyThread(MyThread thread) {  
    thread.counter.incCounter();  
}
```

```
↑ main - begin  
↓ main - end, counter: 300000  
=
```

동시 접근 대상 인스턴스를 이용한 동기화

스레드가 동시접근(공유)하는 인스턴스의 Lock 활용

```
class MyCounter {  
    ...  
    public void synchoIncCounter() {  
        synchronized (this) {  
            ++counter;  
        }  
    }  
}
```

```
↑ main - begin  
↓ main - end, counter: 300000  
⇒
```

Non-blocking 동기화

- 블로킹 동기화 기법 적용 시 스레드 상태가 일시 정지되었다가 다시 실행 상태로 돌아와야 하는 오버헤드를 감수해야 하는 문제가 있음
- 경쟁 조건에서 위험을 일부 감수하고 작업을 진행(Spin lock)하면 스레드 상태 전환 없이 빠른 처리가 가능하며 결과적으로 Non-blocking 동기화가 가능
 - Lock free 구조
 - CPU 수준에서 두 개 이상의 단계를 한 단계처럼 원자성을 보장해 처리

05_atomicCounter

```
import java.util.concurrent.atomic.AtomicInteger;

public class Main {
    static AtomicInteger counter = new AtomicInteger(0);
    // public synchronized static void incCounter() {
    //     ++counter;
    // }
```



```
public static class MyThread implements Runnable {  
    @Override  
    public void run() {  
        for(int i = 0; i < 1000000; ++i) {  
            counter.incrementAndGet();  
            //Main.incCounter();  
        }  
    }  
}
```



main - begin

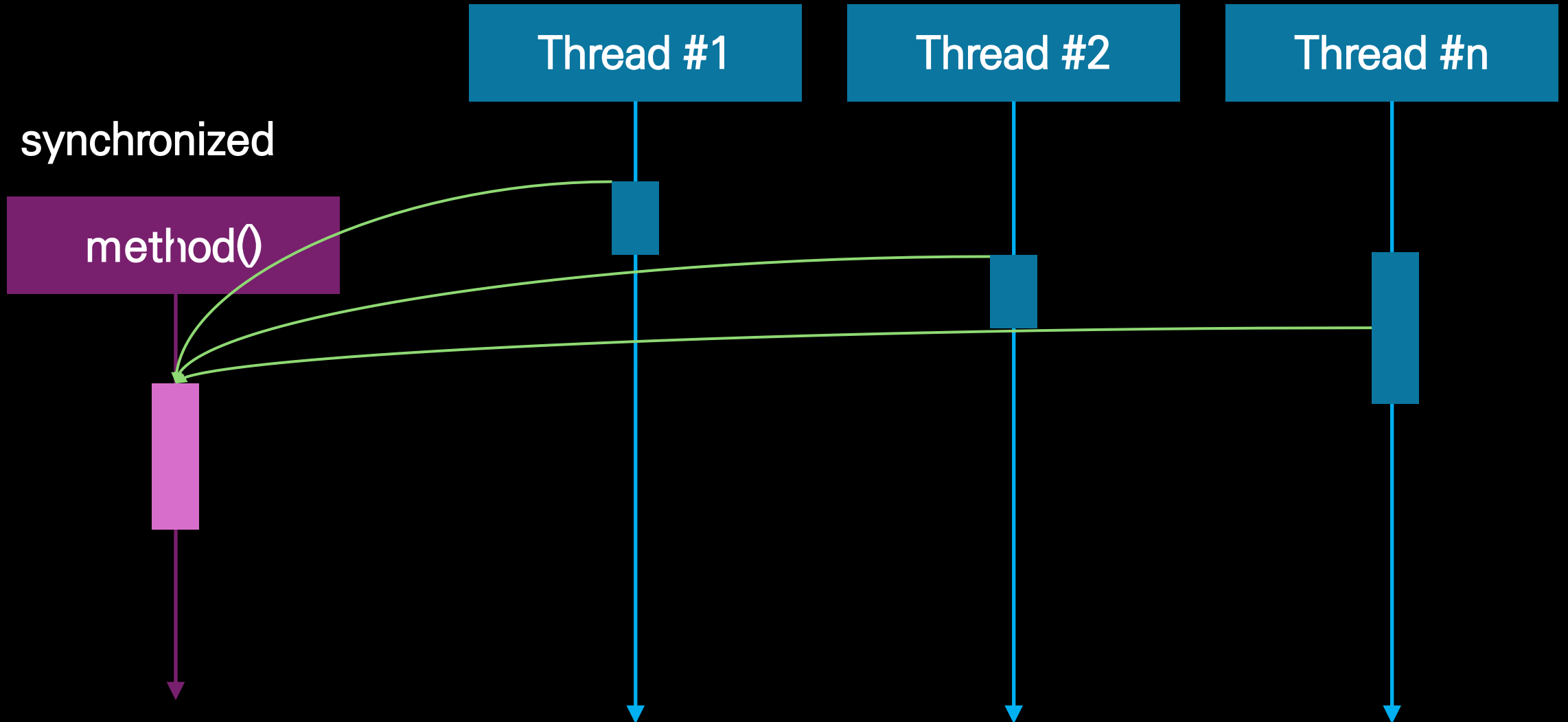


main - end, counter: 300000

ReentrantLock

- 함수 바디 전체가 아니라 특정 부분에 대해서만 동기화를 시도하기 위한 방법
- synchronized 방식이 인스턴스 Lock flag를 이용하는 방식과 달리 별도 객체를 이용해 동기화 하는 방법
- 임계 영역에 속한 코드가 많아질 경우 발생 할 수 있는 성능저하 문제를 해결하기 위해 사용
 - 임계 영역에 속한 코드는 '무조건' 최소화

ReentrantLock



05_reentrantLockSample

```
public class Main {  
    private static final ReentrantLock lock = new ReentrantLock();  
    private static int counter01 = 0;  
    private static int counter02 = 0;  
  
    public static synchronized void increment() {  
        ++counter01;  
    }  
    public static void incCounter() {  
        lock.lock();  
        try { ++counter02; }  
        finally {  
            lock.unlock();  
        }  
    }  
}
```

```
public static void main(String[] args)
                                throws InterruptedException {
    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 30000; i++)
                increment();
        }
    });

    Thread t2 = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 30000; i++)
                incCounter();
        }
    });
}
```

```
t1.start();  
t2.start();  
t1.join();  
t2.join();
```

```
System.out.println("counter01: " + counter01);  
System.out.println("counter02: " + counter02);
```

```
↑ counter01: 30000  
↓ counter02: 30000  
⇌
```

05_timeoutLock

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

class TestThread extends Thread{
    private static final ReentrantLock lock = new ReentrantLock();

    @Override
    public void run() {
```

```
for(int i = 0; i < 5; ++i)
    try {
        if(lock.tryLock(1000, TimeUnit.MILLISECONDS)) {
            System.out.println(currentThread().getName()
                               + ": Lock 획득 성공");

            sleep(1000);
            lock.unlock();
        }
        else
            System.out.println("\t" + currentThread().getName()
                               + ": Lock 획득 실패");
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
```



```

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new TestThread(), "T1");
        Thread t2 = new Thread(new TestThread(), "T2");

        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
}

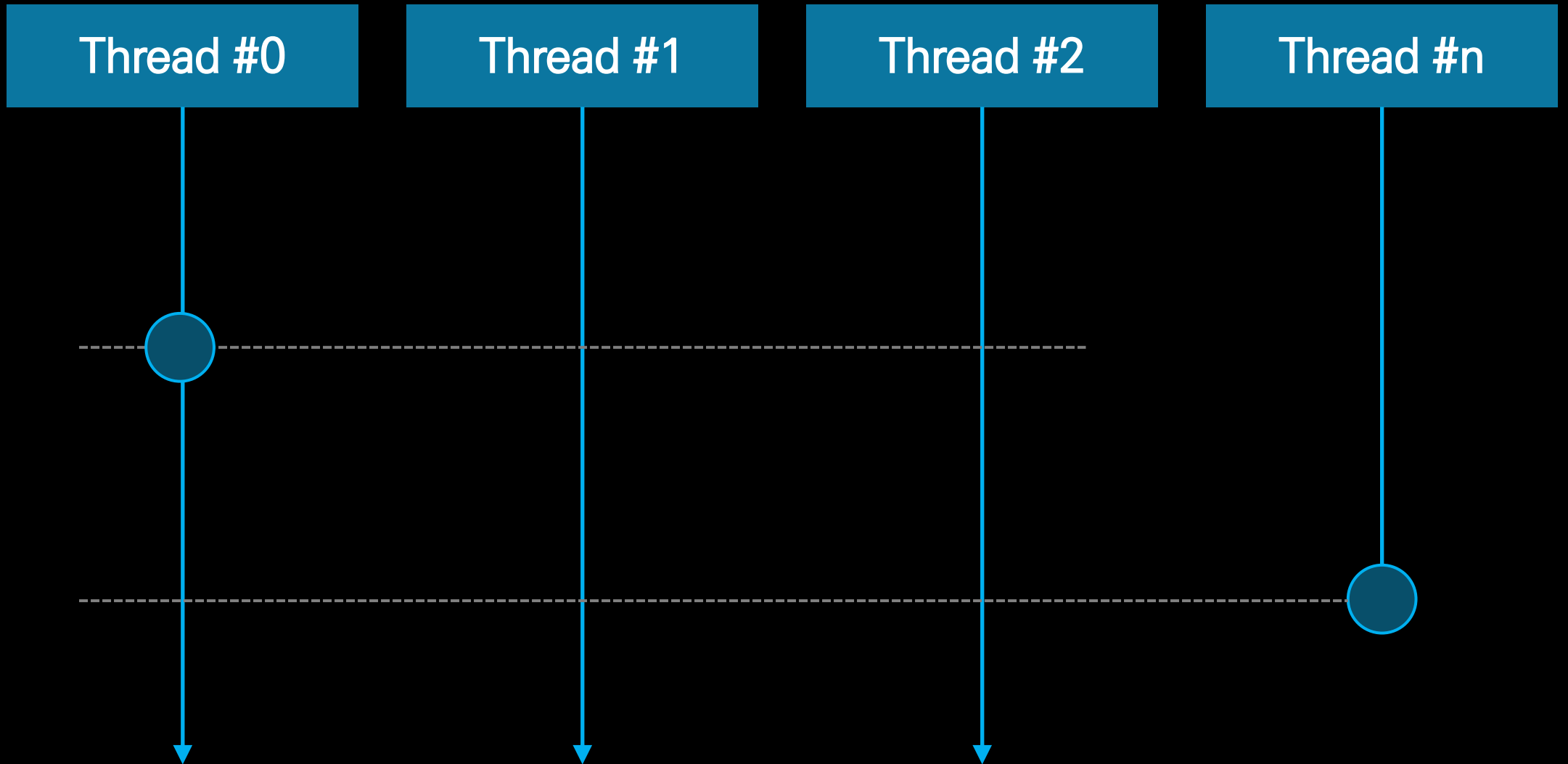
```

Thread	Lock	획득	성공
T1	Lock	획득	성공
T2	Lock	획득	성공
T2	Lock	획득	성공
T1	Lock	획득	실패
T2	Lock	획득	성공
T1	Lock	획득	실패
T2	Lock	획득	성공
T1	Lock	획득	실패
T1	Lock	획득	성공
T2	Lock	획득	성공

대기와 알림

- 멀티스레드 환경에서 신호를 주고 받는 방식으로 흐름을 동기화하는 구조
 - 대기 후 알림
 - synchronized 기반 Thread.wait(), notify()
 - LockSupport.park(), unpark()
- 개별 스레드의 실행 흐름은 스케줄링에 따라 달라질 수 있으므로 반드시 순서를 맞추야 하는 경우에 유용

스레드 실행 시점 동기화



05_waitAndNotify

```
class MyMonitorLock {
    public void consume() {
        System.out.println("MyMonitorLock.consume() - begin");
        synchronized (this) {
            try {
                System.out.println("MyMonitorLock.consume() - before wait()");

                wait();

                System.out.println("MyMonitorLock.consume() - after wait()");
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        System.out.println("MyMonitorLock.consume() - end");
    }
}
```

```
public void produce() {
    System.out.println("MyMonitorLock.produce() - begin");
    synchronized (this) {
        System.out.println("MyMonitorLock.produce() - notify()");
        notify();
    }
    System.out.println("MyMonitorLock.produce() - end");
}
```

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        MyMonitorLock myLock = new MyMonitorLock();

        Thread consumer = new Thread(new Runnable() {
            @Override
            public void run() {
                myLock.consume();
            }
        });
    }
}
```

```
Thread producer = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        myLock.produce();  
    }  
});
```

```
consumer.start();  
sleep(100);  
producer.start();
```

```
consumer.join();  
producer.join();
```

```
}
```

↓	MyMonitorLock.consume() - begin
↺	MyMonitorLock.consume() - before wait()
↻	MyMonitorLock.produce() - begin
🖨	MyMonitorLock.produce() - notify()
🗑	MyMonitorLock.consume() - after wait()
	MyMonitorLock.produce() - end
	MyMonitorLock.consume() - end

Deadlock – 최악의 논리 오류

```
public void produce() {  
    System.out.println("MyMonitorLock.produce() - begin");  
    synchronized (this) {  
        try {  
            System.out.println("MyMonitorLock.produce() - before wait()");  
            wait();  
            System.out.println("MyMonitorLock.produce() - after wait()");  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
    }  
    System.out.println("MyMonitorLock.produce() - end");  
}
```



MyMonitorLock.consume() - begin

MyMonitorLock.consume() - before wait()

MyMonitorLock.produce() - begin

MyMonitorLock.produce() - before wait()

LockSupport 클래스

`java.util.concurrent.locks.LockSupport;`

- 스레드 제어를 위한 유틸리티 클래스로 매우 가볍고 유연한 것이 특징
- `wait()`, `notify()`가 `synchronized` 기반으로 작동하는 것과 달리 `Lock`이 필요 없음
- Spin lock 구현 시 불필요한 CPU 사용을 줄일 수 있고 성능도 향상시킬 수 있음

LockSupport 클래스 주요 메서드

- **LockSupport.park()**
 - 호출자 스레드를 WAITING 상태로 전환
- **LockSupport.unpark()**
 - 호출자 스레드를 RUNNABLE 상태로 전환
- **LockSupport.parkNanos()**
 - 나노초 동안 TIMED_WAIT 상태로 전환 후 복귀
- **LockSupport.parkUntil()**
 - 매개변수로 전달된 밀리초까지 TIMED_WAIT

05_parkUnpark

```
Thread consumer = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("consumer - begin");  
        System.out.println("\t*consumer - park()");  
        LockSupport.park();  
        System.out.println("consumer - end");  
    }  
});
```

```
Thread producer = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("producer - begin");  
        System.out.println("\t*producer - park()");  
        LockSupport.park();  
        System.out.println("producer - end");  
    }  
});
```

```
consumer.start();
producer.start();

sleep(100);
System.out.println("*LockSupport.unpark()");
LockSupport.unpark(consumer);
LockSupport.unpark(producer);

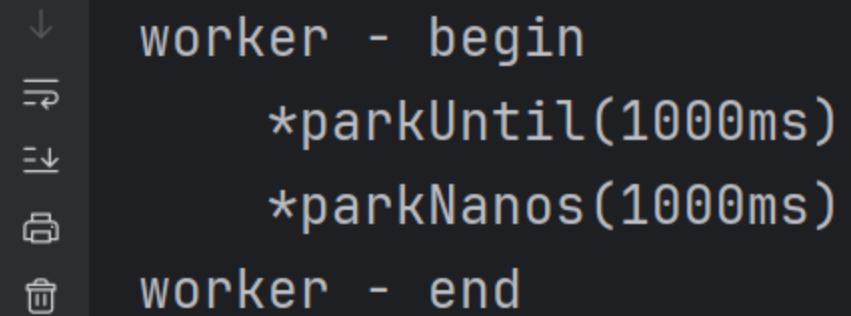
consumer.join();
producer.join();
```

↓	consumer - begin
↵	*consumer - park()
⇓	producer - begin
🖨	*producer - park()
🗑	*LockSupport.unpark()
	producer - end
	consumer - end

05_parkUntil

```
Thread worker = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("worker - begin");
        System.out.println("\t*parkUntil(1000ms)");
        //sleep(1000);
        long wakeUpTime = System.currentTimeMillis() + 1000;
        LockSupport.parkUntil(wakeUpTime);

        System.out.println("\t*parkNanos(1000ms)");
        LockSupport.parkNanos(1000 * 1000 * 1000);
        System.out.println("worker - end");
    }
});
```



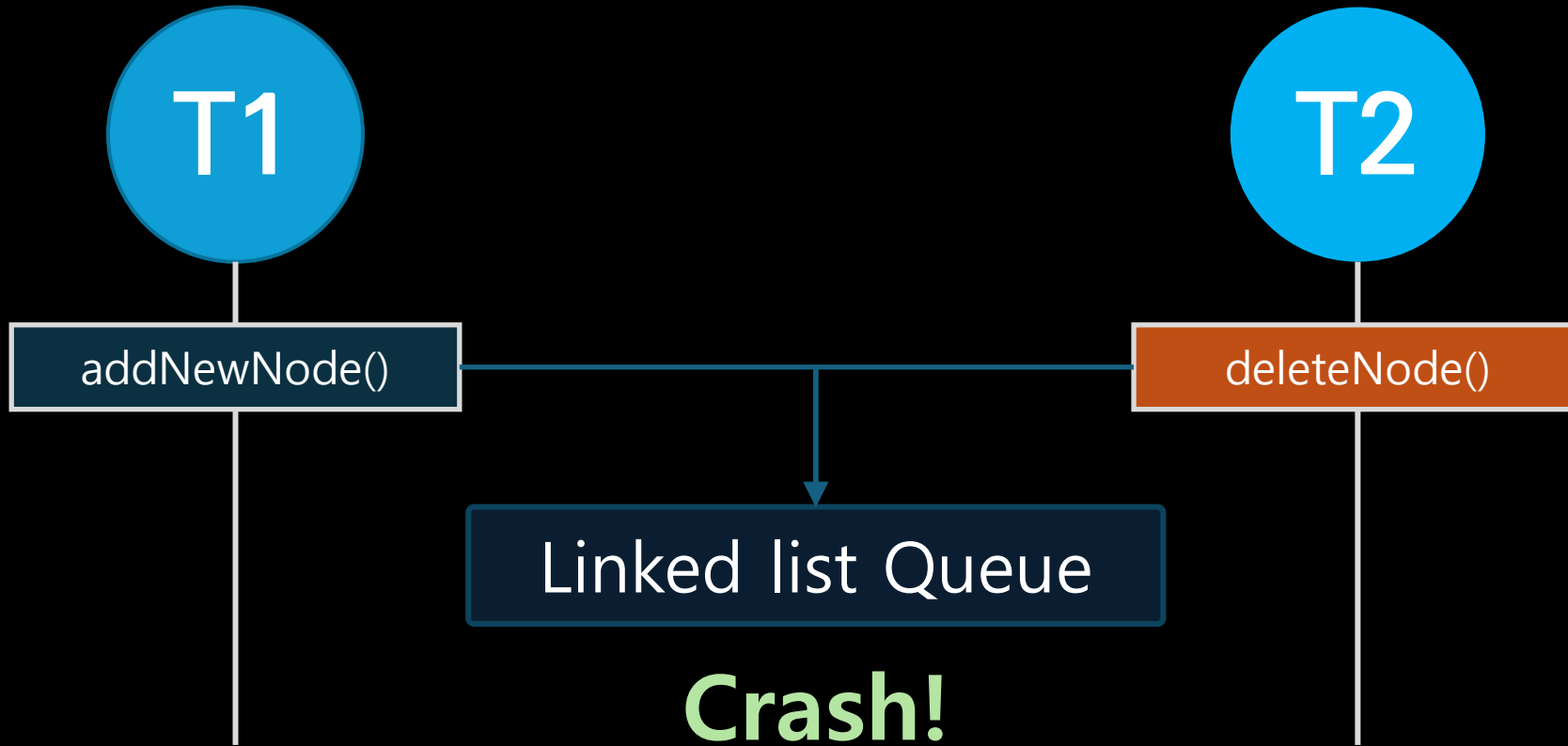
```
↓
⇅
⇅
🖨
🗑
worker - begin
    *parkUntil(1000ms)
    *parkNanos(1000ms)
worker - end
```

6. 연결 리스트와 동기화

Non-blocking 동기화

- 블로킹 동기화 기법 적용 시 스레드 상태가 일시 정지되었다가 다시 실행 상태로 돌아와야 하는 오버헤드를 감수해야 하는 문제가 있음
- 경쟁 조건에서 위험을 일부 감수하고 작업을 진행(Spin lock)하면 스레드 상태 전환 없이 빠른 처리가 가능하며 결과적으로 Non-blocking 동기화가 가능
 - Lock free 구조
 - CPU 수준에서 두 개 이상의 단계를 한 단계처럼 원자성을 보장해 처리

임계영역 기반 동기화

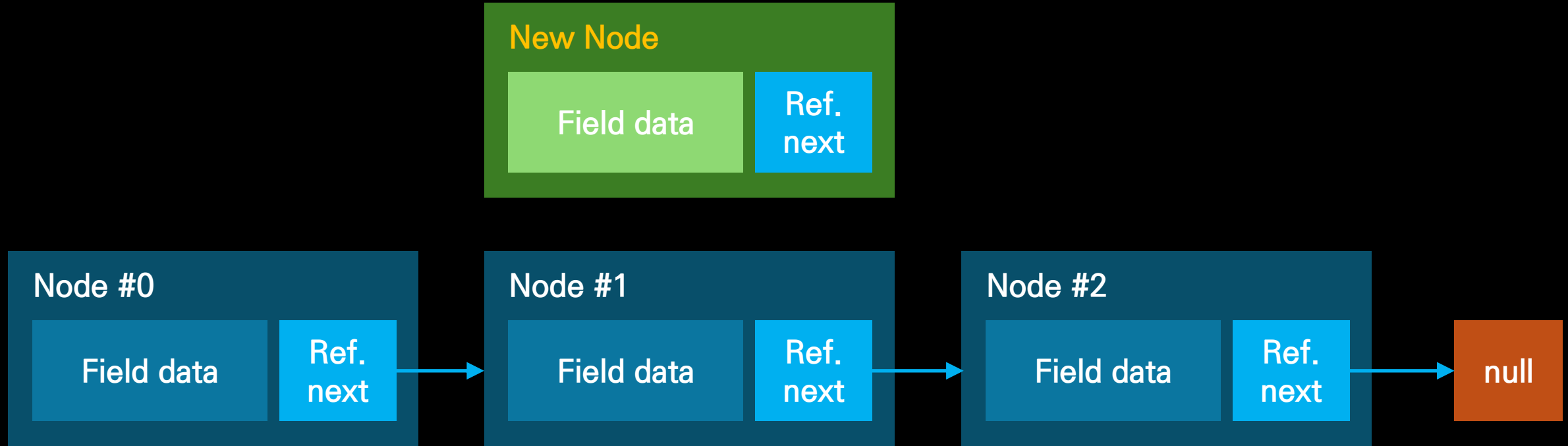


연결 리스트

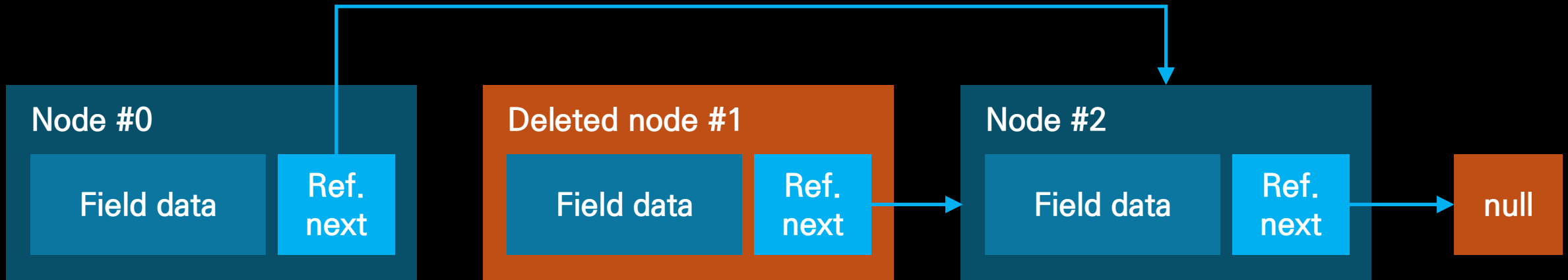


- 배열과 유사하게 단위 데이터(Node)가 논리적으로 연결된 선형 구조
- 배열과 달리 next 참조의 값을 수정하면 쉽게 순서를 변경하거나 새 Node를 추가/삭제 할 수 있음

새 노드 추가 하기



노드 삭제



06_linkedListSample

```
class UserData {  
    UserData(String name) {  
        this.name = name;  
    }  
    String name;  
    UserData prev;  
    UserData next;  
}
```

```
class MyList {  
    protected int counter = 0;  
    protected UserData head = new UserData("DummyHead");  
    protected UserData tail = new UserData("DummyTail");  
    MyList() {  
        head.next = tail;  
        tail.prev = head;  
    }  
  
    public int size() {  
        return counter;  
    }  
}
```

```
public boolean appendNode(String name) {  
    UserData newUser = new UserData(name);  
    newUser.prev = tail.prev;  
    newUser.next = tail;  
    tail.prev.next = newUser;  
    tail.prev = newUser;  
  
    ++counter;  
    return true;  
}
```

```
public boolean isEmpty() {  
    if(head.next == tail)  
        return true;  
  
    return false;  
}
```

```
public void printAll() {  
    System.out.println("-----");  
    System.out.println("Counter: " + counter);  
    UserData tmp = head.next;  
    while(tmp != tail) {  
        System.out.println(tmp.name);  
        tmp = tmp.next;  
    }  
    System.out.println("-----");  
}
```

```
public UserData removeAtHead() {  
    if(isEmpty())  
        return null;  
  
    UserData node = head.next;  
    node.next.prev = head;  
    head.next = node.next;  
  
    --counter;  
    return node;  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        MyList db = new MyList();  
  
        db.appendNode("tester01");  
        db.appendNode("tester02");  
        db.appendNode("tester03");  
        db.printAll();  
    }  
}
```



```
-----  
Counter: 3  
tester01  
tester02  
tester03  
-----
```

06_listThreadAndError

```
private static MyList db = new MyList();
public static void main(String[] args) throws InterruptedException {
    Thread consumer = new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("Consumer - begin");
            UserData node;
            while (true) {
                node = db.removeAtHead();
                if (node == null)
                    try { sleep(1); }
                    catch (InterruptedException e) {
                        System.out.println("Consumer - interrupted");
                        break;
                    }
            }
            System.out.println("Consumer - end");
        }
    });
}
```




```
Thread producer = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("producer - begin");  
        for(int i = 0; i < 100000000; ++i)  
            db.appendNode("tester" + i);  
        System.out.println("producer - end");  
    }  
});
```

```
producer.start();
producer.join();
System.out.println("List size: " + db.size());


consumer.start();
sleep(5000);
consumer.interrupt();

db.printAll();
}
```



```
producer - begin
producer - end
List size: 100000000
Consumer - begin
-----
Consumer - interrupted
Consumer - end
Counter: 0
-----
```

```
producer.start();  
//producer.join();  
sleep(1000);  
System.out.println("List size: " + db.size());  
  
consumer.start();  
sleep(5000);  
consumer.interrupt();  
  
db.printAll();  
}
```




```
↓  
⇐  
⇨  
⇩  
🖨  
🗑  
  
producer - begin  
List size: 4515027  
Consumer - begin  
producer - end  
  
-----  
Consumer - interrupted  
Consumer - end  
Counter: 352296  
-----
```

06_synchronizedList

```
public synchronized boolean appendNode(String name) {  
    UserData newUser = new UserData(name);  
    newUser.prev = tail.prev;  
    newUser.next = tail;  
    tail.prev.next = newUser;  
    tail.prev = newUser;  
  
    ++counter;  
    return true;  
}
```

```
public synchronized UserData removeAtHead() {  
    if(isEmpty())  
        return null;  
  
    UserData node = head.next;  
    node.next.prev = head;  
    head.next = node.next;  
  
    --counter;  
    return node;  
}
```



```
↓  
⇐  
⇒  
🖨  
🗑  
  
producer - begin  
Consumer - begin  
producer - end  
-----  
Consumer - interrupted  
Consumer - end  
Counter: 0  
-----
```

06_lockedList

```
class MyList {  
    protected ReentrantLock lock = new ReentrantLock();  
    protected int counter = 0;  
    protected UserData head = new UserData("DummyHead");  
    protected UserData tail = new UserData("DummyTail");  
    MyList() {  
        head.next = tail;  
        tail.prev = head;  
    }  
}
```

```
public boolean appendNode(String name) {  
    UserData newUser = new UserData(name);  
  
    lock.lock();  
    newUser.prev = tail.prev;  
    newUser.next = tail;  
    tail.prev.next = newUser;  
    tail.prev = newUser;  
  
    ++counter;  
    lock.unlock();  
  
    return true;  
}
```

```
public synchronized UserData removeAtHead() {  
    if(isEmpty())  
        return null;  
  
    lock.lock();  
    UserData node = head.next;  
    node.next.prev = head;  
    head.next = node.next;  
    --counter;  
    lock.unlock();  
  
    return node;  
}
```



producer - begin

Consumer - begin

producer - end

Consumer - interrupted

Consumer - end

Counter: 0

CAS

Compare And Swap, Compare And Set

- Lock-free 구조를 구현하기 위한 핵심원리
- 별도의 동기화 코드(Lock) 없이 원자적 연산 가능
 - 변수 값 확인, 비교, 교환 등 여러 연산을 하나의 CPU 연산으로 처리
 - H/W 수준에서 동기화 보장
- CPU의 Atomic instruction 사용
 - `cmpxchg` (Compare and Exchange)

CAS 구현 원리

```
public final int incrementAndGet() {  
    return U.getAndAddInt(this, VALUE, 1) + 1;  
}  
  
...  
  
@IntrinsicCandidate  
public final int getAndAddInt(Object o, long offset, int delta) {  
    int v;  
    do {  
        v = getIntVolatile(o, offset);  
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));  
    return v;  
}
```

```
@IntrinsicCandidate
```

```
public native int      getIntVolatile(Object o, long offset);
```

```
@IntrinsicCandidate
```

```
public final boolean weakCompareAndSetInt(Object o, long offset,  
                                           int expected,  
                                           int x) {
```

```
    return compareAndSetInt(o, offset, expected, x);
```

```
}
```

```
@IntrinsicCandidate
```

```
public final native boolean compareAndSetInt(Object o, long offset,  
                                              int expected,  
                                              int x);
```

Spin lock 구현

- Lock 을 획득할 때까지 반복문을 수행해 CPU 타임을 계속 사용함으로써 스레드가 Runnable 상태를 유지해 동기화
- 스레드 스위칭에 따른 오버헤드를 제거 하는 방식으로 향상된 성능을 얻는 방식
 - 임계 영역 코드가 짧고 경쟁하는 스레드 개수가 적을 수록 유리
 - 구현하기에 따라 오히려 성능이 저하 될 수 있음
- CAS(Compare-And-Swap)로 구현

06_spinLockSample

CAS 원리가 적용된 SpinLock 클래스 Java 코드가 필요해

```
class SpinLock {  
    private final AtomicReference<Thread> owner = new AtomicReference<>();  
  
    public void lock() {  
        Thread currentThread = Thread.currentThread();  
        // CAS를 이용하여 락 획득 (반복적으로 시도)  
        while (!owner.compareAndSet(null, currentThread)) {  
            // 계속 시도 (바쁜 대기, Spin)  
        }  
    }  
}
```

```
public void unlock() {  
    Thread currentThread = Thread.currentThread();  
    // 현재 스레드만 락을 해제할 수 있음  
    owner.compareAndSet(currentThread, null);  
}
```

```
public boolean isLocked() {  
    return owner.get() != null;  
}
```

```
}
```

```
class SpinLockBool {
    private final AtomicBoolean owner = new AtomicBoolean(false);

    public void lock() {
        while (!owner.compareAndSet(false, true))
            LockSupport.parkNanos(1);
    }

    public void unlock() {
        owner.set(false);
    }
}
```

```
class UserData {
    UserData(String name) {
        this.name = name;
    }
    String name;
    UserData next;
}
```

```
class MyList {  
    protected SpinLock lock = new SpinLock();  
    //protected SpinLockBool lock = new SpinLockBool();  
    //protected ReentrantLock lock = new ReentrantLock();  
    protected AtomicInteger counter = new AtomicInteger();  
    protected UserData head = new UserData("DummyHead");  
  
    public int size() {  
        return counter.get();  
    }  
}
```



```
public int getCount() {  
    int count = 0;  
    UserData tmp = head.next;  
    while(tmp != null) {  
        ++count;  
        tmp = tmp.next;  
    }  
    return count;  
}
```

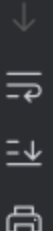
```
public boolean appendNode(String name) {  
    UserData newUser = new UserData(name);  
    //lock.lock();  
    newUser.next = head.next;  
    head.next = newUser;  
    counter.incrementAndGet();  
    //lock.unlock();  
  
    return true;  
}  
}
```

```
class TestThread extends Thread {  
    TestThread(MyList db) {  
        list = db;  
    }  
  
    private final MyList list;  
    @Override  
    public void run() {  
        for (int i = 0; i < 10000000; i++)  
            list.appendNode("TEST" + i);  
    }  
}
```

```
public static void main(String[] args) throws InterruptedException {
    MyList db = new MyList();
    int threadCount = 2;
    TestThread[] threads = new TestThread[threadCount];

    long beginTime = System.currentTimeMillis();
    for (int i = 0; i < threadCount; i++)
        threads[i] = new TestThread(db);
    for (int i = 0; i < threadCount; i++)
        threads[i].start();
    for (int i = 0; i < threadCount; i++)
        threads[i].join();
    long endTime = System.currentTimeMillis();

    System.out.println("Duration: " + (endTime - beginTime) + " ms");
    System.out.println("size: " + db.size());
    System.out.println("getCount(): " + db.getCount());
}
```



```
Duration: 352 ms
size: 2000000
getCount(): 1444719
```

동기화 방식에 따른 List 접근 속도 비교

SpinLock

```
Duration: 561 ms  
size: 2000000  
getCount(): 2000000
```

```
Duration: 563 ms  
size: 2000000  
getCount(): 2000000
```

```
Duration: 424 ms  
size: 2000000  
getCount(): 2000000
```

ReentrantLock

```
Duration: 486 ms  
size: 2000000  
getCount(): 2000000
```

synchronized

```
Duration: 465 ms  
size: 2000000  
getCount(): 2000000
```

06_lockFreeQueue

```
class LockFreeUser {
    LockFreeUser(String name) {
        this.name = name;
        next = new AtomicReference<LockFreeUser>(null);
    }
    String name;
    AtomicReference<LockFreeUser> next;
}

class LockFreeList {
    protected LockFreeUser headNode;
    protected AtomicReference<LockFreeUser> tail;

    LockFreeList() {
        headNode = new LockFreeUser("DummyHead");
        tail = new AtomicReference<LockFreeUser>(headNode);
    }
}
```

```
public void push(String name) {
    LockFreeUser newUser = new LockFreeUser(name);
    while (true) {
        LockFreeUser last = tail.get(); //last = tail;
        LockFreeUser next = last.next.get(); //next = last.next;
        if (last == tail.get()) { //last == tail;
            if (next == null) { //last node?
                if (last.next.compareAndSet(null, newUser)) {
                    //last.next = newUser;
                    tail.compareAndSet(last, newUser); //tail = newUser;
                    return;
                }
            }
            else
                tail.compareAndSet(last, next);
        }
        LockSupport.parkNanos(1);
    }
}
```

```
class LockFreeTestThread extends Thread {  
    LockFreeTestThread(LockFreeList db) {  
        list = db;  
    }  
  
    private final LockFreeList list;  
    @Override  
    public void run() {  
        for (int i = 0; i < 10000000; i++)  
            list.push("TEST" + i);  
    }  
}
```

```
LockFreeList db = new LockFreeList();
int threadCount = 5;
LockFreeTestThread[] threads = new LockFreeTestThread[threadCount];

long beginTime = System.currentTimeMillis();
for (int i = 0; i < threadCount; i++)
    threads[i] = new LockFreeTestThread(db);
for (int i = 0; i < threadCount; i++)
    threads[i].start();
for (int i = 0; i < threadCount; i++)
    threads[i].join();
long endTime = System.currentTimeMillis();

System.out.println("Duration: " + (endTime - beginTime) + " ms");
System.out.println("getCount(): " + db.getCount());
```


7. 제네릭

가볍게 기초적인 문법만 살펴보는 제네릭!

제네릭 타입 활용하기

```
class Test<Type-parameter> {}
```

- 형안정성을 유지하면서 코드 재사용성을 극대화 할 수 있는 문법
- 런타임 타입 캐스팅이 필요 없는 간결한 코드
- 메서드, 클래스, 인터페이스 선언 시 타입 매개변수(Type parameter)를 기술하는 방식으로 활용
 - 기존 문법을 알고 있다는 가정하에 활용 할 수 있음
- 컴파일 타임에 타입 변수는 실제 타입으로 변환

제네릭 타입 활용하기

- 타입 변수를 특정 클래스의 파생 형식으로 제한 가능
 - `<T extends Parent>`
- 타입 변수는 반드시 클래스 형식 (int, double 불가)
- static 메서드는 허용되지 않음

클래스 자료를 관리해야 하는 경우 – 07_myData

```
class MyDataInt {  
    private int data;  
    public int get() { return data; }  
    public void set(int param) { data = param; }  
}
```

```
class MyDataString {  
    private String data;  
    public String get() { return data; }  
    public void set(String param) { data = param; }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        MyDataInt myInt = new MyDataInt();  
        myInt.set(5);  
        System.out.println(myInt.get());  
  
        MyDataString myString = new MyDataString();  
        myString.set("Hello");  
        System.out.println(myString.get());  
    }  
}
```



5



Hello



추상형식 적용 – 07_myObejct

```
class MyDataObj {  
    private Object data;  
    public Object get() {  
        return data;  
    }  
    public void set(Object param) {  
        data = param;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        MyDataObj myInt = new MyDataObj();  
        myInt.set(5);  
        int data = (int)myInt.get();  
        System.out.println(data);  
  
        MyDataObj myString = new MyDataObj();  
        myString.set("Hello");  
        String result = (String)myString.get();  
        System.out.println(result);  
        //double dbl = (double)myString.get();  
        //System.out.println(dbl);  
    }  
}  
  
class java.lang.String cannot be cast to class java.lang.Double  
(java.lang.String and java.lang.Double are in module java.base of  
loader 'bootstrap')  
    at Main.main(Main.java:22)
```

07_genericSample

```
class MyDataObj<T> {  
    private T data;  
    public T get() {  
        return data;  
    }  
  
    public void set(T param) {  
        data = param;  
    }  
}
```



```
public class Main {  
    public static void main(String[] args) {  
        MyDataObj<Integer> myInt = new MyDataObj<Integer>();  
        myInt.set(5);  
        int data = myInt.get();  
        System.out.println(data);  
  
        MyDataObj<String> myString = new MyDataObj<String>();  
        myString.set("Hello");  
        String result = myString.get();  
        System.out.println(result);  
    }  
}
```

↓	5
⇌	Hello
⇌	

타입 매개변수 이름 관례

```
class Test<K, V> {}
```

- E (Element)
- K (Key)
- N (Number)
- T (Type)
- V (Value)
- S, U, V (두 번째, 세 번째, 네 번째 형식)

제네릭 메서드

```
public static <T> T genMethod(T t) {}
```

- 클래스 전체가 아닌 특정 메서드에 대해서만 제네릭을 적용하기 위한 방법

07_genericMethod

```
class Test {  
    private String data;  
    public String get() { return data; }  
    public void set(String param) { data = param; }  
}
```

```
class TestEx extends Test {  
    public void printData() {  
        System.out.println(get());  
    }  
}
```

```
public class Main {  
    public static <T extends Test> void printData(T param) {  
        System.out.println(param.get());  
    }  
  
    public static void main(String[] args) {  
        TestEx testEx = new TestEx();  
        testEx.set("Hello");  
        printData(testEx);  
    }  
}
```

8. 컬렉션 프레임워크 선형

컬렉션 프레임워크 소개

java.util 패키지에서 제공되는 자료구조 클래스

- Collection 클래스 파생
- List
 - ArrayList, Vector, LinkedList
- Set
 - HashSet, TreeSet
- Map
 - HashMap, Hashtable, TreeMap, Properties

List 주요 메서드

- boolean add(E)
- void add(int index, E)
- E set(int index, E)
- void clear()
- E remove(int index)
- boolean remove(Object)
- boolean contains(Object)
- E get(int index)
- boolean isEmpty()
- int size()

08_arrayListSample

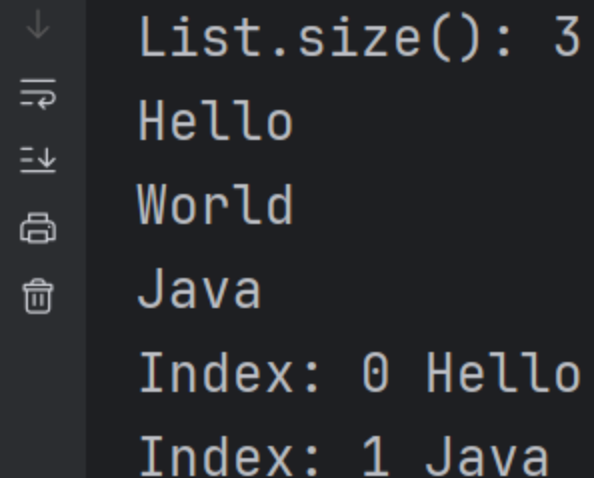
```
ArrayList<String> list = new ArrayList<String>();

list.add("Hello");
list.add("World");
list.add("Java");

System.out.println("List.size(): " + list.size());
for(String data : list)
    System.out.println(data);

list.remove(1);

for(int i = 0; i < list.size(); ++i)
    System.out.println("Index: " + i + " " + list.get(i));
```



```
List.size(): 3
Hello
World
Java
Index: 0 Hello
Index: 1 Java
```

08_listThreadError

ArrayList는 동시성을 고려하지 않음

```
public static void main(String[] args) throws InterruptedException {
    ArrayList<String> list = new ArrayList<String>();
    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("T1 - begin");
            for (int i = 0; i < 10000000; i++) {
                list.addLast("Test" + i);
            }
            System.out.println("T1 - end");
        }
    });
}
```

```
Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("T2 - begin");
        for (int i = 0; i < 10000000; i++) {
            list.add("Data" + i);
        }
        System.out.println("T2 - end");
    }
});

t1.start();
t2.start();
sleep(10);
t1.join();
t2.join();
System.out.println("ArrayList: " + list.size());
```



T1 - begin



T2 - begin



T1 - end



T2 - end

ArrayList: 1274822

동시성을 고려한 Vector – 08_vectorAndThread

```
Vector<String> list = new Vector<String>();
```

```
Thread t1 = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("T1 - begin");  
        for (int i = 0; i < 10000000; i++) {  
            list.addLast("Test" + i);  
        }  
        System.out.println("T1 - end");  
    }  
});
```

↓	T1 - begin
↵	T2 - begin
⇓	T1 - end
🖨	T2 - end
🗑	Vector: 20000000

LinkedList

요소 개수 업데이트가 있을 경에 사용

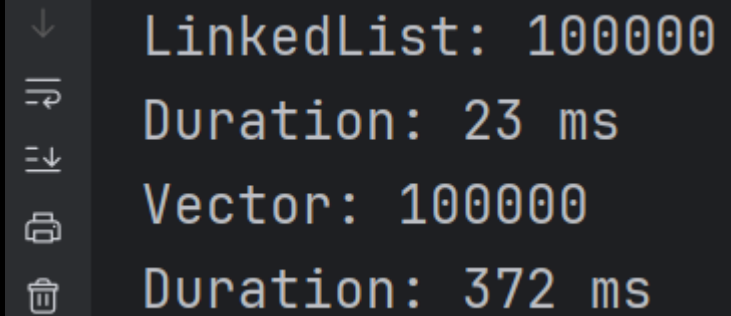
- 2중 연결 리스트 구조를 갖는 선형 자료구조
- 새로운 데이터를 추가/삭제할 경우 ArrayList, Vector 같은 배열 구조에 비해 압도적으로 성능이 우수
 - 배열 구조에서 요소의 개수를 추가하는 경우 메모리의 크기를 늘려야 하는데 이것이 불가능 할 경우 새 요소 및 기존 요소를 모두 수용할 수 있는 새로운 메모리를 할당해 추가

데이터 추가 성능비교 예제

```
LinkedList<String> linkedList = new LinkedList<String>();  
Vector<String> list = new Vector<String>();
```

```
long beginTime = System.currentTimeMillis();  
for (int i = 0; i < 1000000; i++)  
    linkedList.add(0, "Test" + i);  
long endTime = System.currentTimeMillis();  
System.out.println("LinkedList: " + linkedList.size());  
System.out.println("Duration: " + (endTime - beginTime) + " ms");
```

```
beginTime = System.currentTimeMillis();  
for (int i = 0; i < 1000000; i++)  
    list.add(0, "Test" + i);  
endTime = System.currentTimeMillis();  
System.out.println("Vector: " + list.size());  
System.out.println("Duration: " + (endTime - beginTime) + " ms");
```



A terminal window showing the output of the Java code. It displays the size of the LinkedList (1000000) and its duration (23 ms), followed by the size of the Vector (1000000) and its duration (372 ms). The terminal has a dark background with light gray text. On the left side of the terminal window, there is a vertical toolbar with icons for scrolling (up, down, search), copying, and deleting.

↓	LinkedList: 1000000
↕	Duration: 23 ms
↕	Vector: 1000000
🗑️	Duration: 372 ms

9. 컬렉션 프레임워크 비선형

Set

비선형 구조로 객체를 관리할 수 있는 방법

- 비선형 구조에 대한 인터페이스 클래스
- 리스트 구조와 달리 대상 객체가 연속적이지 않음
- 객체를 중복해서 저장 할 수 없음
- 요소를 인덱스로 식별 할 수 없음
- 요소 검색 속도가 매우 빠름

Set 객체 추가, 삭제, 검색, 접근

- `boolean add(E)`
- `void clear()`
- `boolean remove(Object)`
- `boolean contains(Object)`
- `boolean isEmpty()`
- `iterator<E> iterator()`
 - `hasNext()`, `next()`, `void remove()`
- `int size()`

HashSet

해시 테이블을 이용한 객체 관리

- Set 인터페이스의 대표적 구현 클래스
- 중복 요소를 허용하지 않으며 입력 순서를 유지하지 않는 비선형 구조
 - null요소 1개 사용
 - 내부적으로 HashMap 사용
- 요소가 되는 객체에 대해 hashCode()를 계산해 저장할 버킷을 선택하고 equals() 메서드를 이용해 중복여부 최종 결정

TreeSet

Red-Black tree 기반 비선형 구조

- 정렬된 순서로 요소를 저장하며 null 요소를 허용하지 않음
 - 요소 중복 허용하지 않음
- Comparator를 이용한 사용자 정의 정렬 지원
- 대량의 데이터를 다룰 때는 HashSet이 유리

09_hashSetSample

```
public static void main(String[] args) {  
    HashSet<String> set = new HashSet<>();  
    System.out.println(set.add("Hello"));  
    System.out.println(set.add("World"));  
    System.out.println(set.add("Java"));  
    System.out.println(set.add("Java") + "\n");  
    System.out.println("size():" + set.size());  
    System.out.println(set.contains("Java"));  
  
    //for (String s : set) System.out.println(s);  
    Iterator<String> it = set.iterator();  
    while(it.hasNext())  
        System.out.println(it.next());  
}
```

```
true  
true  
true  
false  
  
size():3  
true  
Java  
Hello  
World
```

09_hashSetSample

열거자에서만 삭제해도 컬렉션까지 함께 삭제

```
public static void main(String[] args) {  
    Set<String> set = new HashSet<String>();  
    System.out.println(set.add("Hello"));  
    System.out.println(set.add("World"));  
    System.out.println(set.add("Java"));  
  
    Iterator<String> it = set.iterator();  
    while(it.hasNext()) {  
        it.next();  
        it.remove();  
    }  
    System.out.println(set.size());  
}
```

true

true

true

0

Set과 동기화 문제

```
public static void main(String[] args) throws InterruptedException {
    Set<String> set = new HashSet<>();
    Set<String> syncSet = Collections.synchronizedSet(new HashSet<>());

    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 1000000; i++) {
                set.add("TEST" + i);
                syncSet.add("TEST" + i);
            }
        }
    });
}
```

```
Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            set.add("DATA" + i);
            syncSet.add("DATA" + i);
        }
    }
});

t1.start(); t2.start();
t1.join(); t2.join();

System.out.println("set.size(): " + set.size());
System.out.println("syncSet.size(): " + syncSet.size());
}
```

```
↓ set.size(): 195281
⇨ syncSet.size(): 200000
```

객체 중복여부 판정

- Set 컬렉션은 요소 객체가 중복되는 것을 허용하지 않음
 - 동등성, 동일성 문제 고려
- 내용이 같더라도 인스턴스가 다르다면 다른 것으로 판정
- 인스턴스가 다르더라도 내용이 같은 경우를 중복으로 처리하려면 별도로 비교 코드를 작성
 - hashCode()
 - equals()

09_setAndEquals

```
class UserData {
    UserData(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }
    String name;
    String phone;
    UserData next;

    @Override
    public boolean equals(Object obj) {
        if(obj == null)
            return false;
        UserData user = (UserData)obj;
        return name.equals(user.name) && phone.equals(user.phone);
    }
}
```

```
public class Main {  
    public static void main(String[] args) {  
        HashSet<UserData> set = new HashSet<>();  
        set.add(new UserData("Tester", "1234-1234"));  
        set.add(new UserData("Tester", "1234-1234"));  
        set.add(new UserData("Tester", "1234-1234"));  
  
        System.out.println("size(): " + set.size());  
    }  
}
```

```
size(): 3
```

```
@Override
public boolean equals(Object obj) {
    if(obj == null)
        return false;
    UserData user = (UserData)obj;
    return name.equals(user.name) && phone.equals(user.phone);
}
```

```
@Override
public int hashCode() {
    return (name + phone).hashCode();
}
```

...

```
size(): 1
```

Map

Key + value 구조로 객체를 관리

- Map은 인터페이스 클래스
- 키 + 값으로 구성되는 Map.Entry 객체를 비선형 구조로 관리
- 값은 중복이 허용되지만 키는 중복을 허용하지 않음
- 키, 값 각각으로 검색 가능
- HashMap, Hashtable(synchronized 메서드)

Map

- `V put(K, V)`
- `boolean contains(K)`
- `boolean containsValue(V)`
- `Set<Map.Entry<K, V>> entrySet()`
- `V get(K)`
- `boolean isEmpty()`
- `Set<K> keyset()`
- `int size()`

Map

- `Collection<V> values()`
- `void clear()`
- `V remove(K)`

09_hashMapSample

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

```
map.put("Tester1", "1111-1111");  
map.put("Tester2", "2222-2222");  
map.put("Tester3", "3333-3333");  
map.put("Tester1", "1111-1111");
```

```
System.out.println("size(): " + map.size());  
Set<String> keySet = map.keySet();  
Iterator<String> it = keySet.iterator();  
while(it.hasNext()) {  
    String key = it.next();  
    String value = map.get(key);  
    System.out.println(key + "\t" + value);  
}
```

↓	size(): 3
↺	Tester3 3333-3333
↻	Tester2 2222-2222
🗑️	Tester1 1111-1111

LinkedList 혼합 운영 – 09_listAndHash


```
public static void main(String[] args) {  
    Set<String> set = new TreeSet<>();  
    List<String> list = new LinkedList<>();  
    long beginTime, endTime;  
  
    beginTime = System.currentTimeMillis();  
    int i = 0;  
    for (i = 0; i < 100000000; i++)  
        list.add("Test" + i);  
    endTime = System.currentTimeMillis();  
    System.out.println("list.add(): " + (endTime - beginTime) + " ms");  
}
```



```
beginTime = System.currentTimeMillis();  
i = 0;  
for (i = 0; i < 100000000; i++)  
    set.add("Test" + i);  
endTime = System.currentTimeMillis();  
System.out.println("set.add(): " + (endTime - beginTime) + " ms");
```

```
beginTime = System.currentTimeMillis();  
list.contains("Test" + (i - 1));  
endTime = System.currentTimeMillis();  
System.out.println("list.contains(): " + (endTime - beginTime) + " ms");
```

```
beginTime = System.currentTimeMillis();  
set.contains("Test" + (i - 1));  
endTime = System.currentTimeMillis();  
System.out.println("set.contains(): " + (endTime - beginTime) + " ms");
```



```
list.add(): 2088 ms  
set.add(): 3157 ms  
list.contains(): 132 ms  
set.contains(): 0 ms
```

수업을 마치며...

- 스레드 및 동기화와 관련된 내용은 아무리 강조해도 지나치다 할 수 없을 만큼 중요함
 - 다수 복습 권장
 - 당장 다 알려 하지 말고 프로젝트를 통해 직접 실패를 경험하는 것이 중요
- 컬렉션은 잘 사용하는 것이 중요
- 여러 스레드를 관리하기 위한 스레드 풀에 대해 알아볼 것
 - ExecutorService, ForkJoinPool

감사합니다!