LG전자 BS팀

『 2일차 』 : 오후

◈ 훈련과정명 : OS 기본

◈ 훈 련 기 간 : 2023.05.30-2[illegible]3.06.02

목차

『6과목』

7-8교시 : Synchronization Tools

Syllabus

학습목표

- 이 워크샵에서는 크리티컬 섹션 문제 설명 및 경쟁 조건 설명을 할 수 있다.
- 메모리 배리어, 비교 및 스왑 작업, 원자 변수를 사용하여 임계 영역 문제에 대한 하드웨어 솔루션 설명을 할 수 있다.
- 중요한 섹션 문제를 해결하기 위해 뮤텍스 잠금, 세마포어, 모니터 및 조건 변수를 사용할 수 있는 방법을 보여줄 수 있다.
- 경합이 낮음, 중간, 높음 시나리오에서 중요한 섹션 문제를 해결하는 도구를 평가할 수 있다.

눈높이 체크

- 배경
- 크리티컬 섹션 문제를 알고 계신가요?
- 피터슨의 솔루션을 알고 계신가요?
- 동기화를 위한 하드웨어 지원을 알고 계신가요?
- 뮤텍스 잠금을 알고 계신가요?
- 세마포어를 알고 계신가요?
- 모니터를 알고 계신가요?
- 생동감을 알고 계신가요?
- 평가

1. 배경

기본 개념

- 프로세스는 동시에 실행할 수 있다.
- 언제든지 중단될 수 있으며 부분적으로 실행이 완료됨
- 공유 데이터에 대한 동시 액세스로 인해 데이터 불일치가 발생할 수 있음
- 데이터 일관성을 유지하려면 협력 프로세스의 질서 있는 실행을 보장하는 메커니즘이 필요.

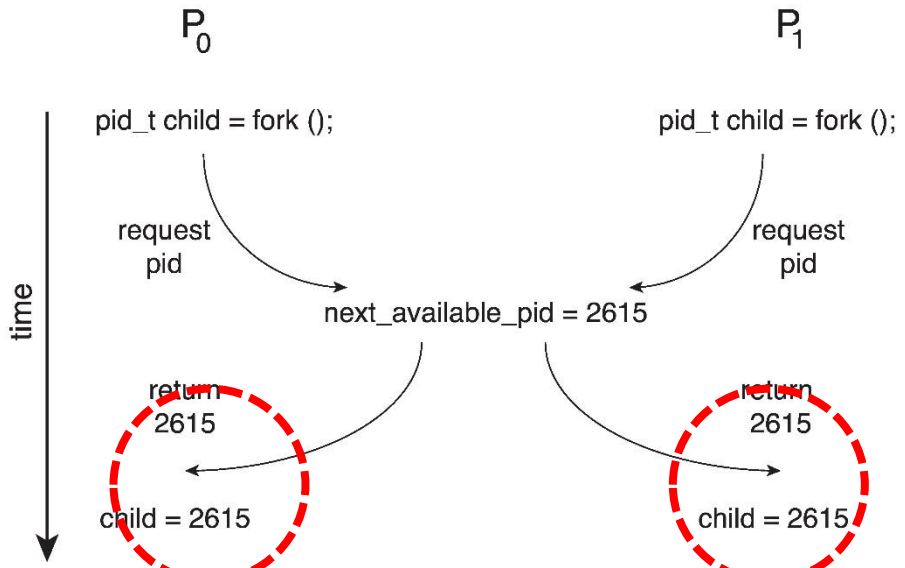
공유 자원과 임계구역

한정된 자원을 가지고 프로세스가 공동으로 작업할 때 발생할 수 있는 문제가 있다.

1. 배경

경쟁 조건 Race Condition

- 프로세스 Po 및 P1은 fork() 시스템 호출을 사용하여 자식 프로세스를 생성.
- 다음으로 사용 가능한 프로세스 식별자(pid)를 나타내는 커널 변수 next_available_pid의 경쟁 조건



- Po과 P1이 변수 next_available_pid에 접근하는 것을 막는 메커니즘이 없다면 같은 pid가 두 개의 서로 다른 프로세스에 할당될 수 있다

2. Critical-Section Problem

임계 영역 문제

- n 프로세스 {p0, p1, … pn-1}의 시스템을 고려.
- 각 프로세스에는 코드의 중요한 섹션 critical section 세그먼트가 있다.
  - 프로세스는 공통 변수 변경, 테이블 업데이트, 파일 작성 등일 수 있다.
  - 하나의 프로세스가 크리티컬 섹션에 있을 때 다른 프로세스는 크리티컬 섹션에 있을 수 없다.
- 크리티컬 섹션 문제 Critical section problem 는 이를 해결하기 위한 프로토콜을 설계하는 것이다.
- 각 프로세스는 시작 섹션에서 임계 섹션에 들어갈 수 있는 권한을 요청해야 하며 임계 섹션 다음에 종료 섹션, 나머지 섹션이 있을 수 있다.

2. Critical-Section Problem

임계 영역 문제

- 프로세스 Pi의 일반적인 구조

```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

2. Critical-Section Problem

임계 영역 관련 요구사항

- 크리티컬 섹션 문제 해결을 위한 요구 사항

1. 상호 배제 Mutual Exclusion - 프로세스 Pi가 임계 영역에서 실행 중인 경우 다른 프로세스는 임계 영역에서 실행될 수 없다.
2. 진행률 Progress(deadlock 회피)- 임계구역에서 실행 중인 프로세스가 없고 임계구역에 진입하려는 프로세스가 있는 경우 다음에 임계구역에 진입할 프로세스의 선택을 무한정 연기할 수 없다.
3. Bounded Waiting (기아 회피) - 프로세스가 크리티컬 섹션에 들어가도록 요청한 후 해당 요청이 승인되기 전에 다른 프로세스가 크리티컬 섹션에 들어갈 수 있도록 허용되는 횟수에 대한 바인딩이 존재해야 함.

- 각 프로세스가 0이 아닌 속도로 실행된다고 가정
- n개 프로세스의 상대 속도 relative speed 에 대한 가정 없음

2. Critical-Section Problem

Software Solution 1

- Two process 솔루션
- 로드 load 및 저장 store 기계 언어 명령이 원자적이라고 가정. 즉 중단할 수 없다.
- 두 프로세스는 하나의 변수를 공유.
  int turn;
- 변수 turn은 크리티컬 섹션에 들어갈 차례를 나타낸다.
- 처음에 turn 값은 i로 설정.

2. Critical-Section Problem

Algorithm for Process Pi

```
while (true){

        while (turn = = j);


        /* critical section */

        turn = j;

        /* remainder section */

}
```

2. Critical-Section Problem

소프트웨어 솔루션의 정확성

- 상호 배제가 유지됨

P_i enters critical section only if:

```
turn = i
```

- 그리고 `turn` 는 동시에 0과 1이 될 수 없다
- 진행 요건은 어떨까?
- Bounded-waiting 요구 사항은 어떨까?

2. Critical-Section Problem

자바 스레드에서의 경쟁 상태

- 경쟁 상태 (Race Condition)
- 경쟁 상태 (Race Condition)란 여러 개의 프로세스나 스레드가 공유 데이터에 동시에 접근하여 값을 변경할 때, 그 실행 결과가 특정한 접근 순서에 따라 달라지는 것을 말한다.
- 이러한 경쟁 상태를 막기 위해서, 우리는 오직 한번에 한 프로세스만 공유 데이터에 접근하게 만들어야 한다. 이를 위해 우리는 서로 다른 프로세스들을 동기화 (synchronization) 해줘야 한다.

2. Critical-Section Problem

자바 스레드에서의 경쟁 상태

파일	소스코드
실습환경	

소스코드	k8s@DESKTOP-RoEQ2U6:~$ cd java_workspaces/
	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ cat > RaceCondition1.java
	public class RaceCondition1 {
	  public static void main(String[] args) throws InterruptedException {
	  RunnableOne run1 = new RunnableOne();
	  RunnableOne run2 = new RunnableOne();
	  Thread t1 = new Thread(run1);
	  Thread t2 = new Thread(run2);
	  t1.start(); t2.start();
	  t1.join(); t2.join();
	  System.out.println("Result: " + run1.count + ", " + run2.count);
	  }
	}
	
	class RunnableOne implements Runnable {
	  int count = 0;
	
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  count++;
	  }
	  }
	}
	
	추천
	@Override
	public synchronized void run() {
	  for (int i = 0; i < 10000; i++) {
	  count++;
	  }
	}

결과값1	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac RaceCondition1.java k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java RaceCondition1 Result: 10000, 10000
비고	• RunnableOne 객체 run1, run2는 서로 다른 메모리 공간에 할당되어 별도의 count 변수를 갖는다. 즉, count 변수는 두 객체의 공유 데이터가 아니므로 실행 결과가 항상 일관성 있게 나온다.

2. Critical-Section Problem

자바 스레드에서의 경쟁 상태

파일	소스코드
실습환경	

소스코드	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ cat > RaceCondition2.java
	public class RaceCondition2 {
	  public static void main(String[] args) throws InterruptedException {
	  RunnableTwo run1 = new RunnableTwo();
	  RunnableTwo run2 = new RunnableTwo();
	  Thread t1 = new Thread(run1);
	  Thread t2 = new Thread(run2);
	  t1.start(); t2.start();
	  t1.join(); t2.join();
	  System.out.println("Result: " + RunnableTwo.count); // 클래스 멤버 변수
	  }
	}
	
	class RunnableTwo implements Runnable {
	  static int count = 0; // 모든 객체가 공유하는 데이터
	
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  count++;
	  }
	  }
	}

결과값1	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac RaceCondition2.java k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java RaceCondition2 Result: 20000
비고	• 반면에, static 키워드를 붙여서 count 변수를 모든 객체가 공유하는 데이터로 만들면, 실행할 때마다 결과가 다르게 나온다

```
@Override
public void run() {
  synchronized (RunnableTwo.class)
{
    for (int i = 0; i < 10000; i++) {
      count++;
    }
  }
}
```

2. Critical-Section Problem

자바 스레드에서의 경쟁 상태

파일	소스코드
실습환경	
소스코드	
결과값1	
비고	

또는 count 변수를 AtomicInteger으로 선언하여 원자적인 연산을 보장할 수도 있습니다.

java
Copy code

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

class RunnableTwo implements Runnable {
   static AtomicInteger count = new AtomicInteger(0);

   @Override
   public void run() {
      for (int i = 0; i < 10000; i++) {
         count.incrementAndGet();
      }
   }
}
```

3. Peterson's Solution

Two process solution

- Dekker's Algorithm / Eisenberg and McGuire's algorithm
- Peterson's Solution
- 로드 `load` 및 저장 `store` 기계 언어 명령이 원자적이라고 가정. 즉 중단할 수 없다.
- 두 프로세스는 두 가지 변수를 공유.

```
int turn;
boolean flag[2]
```

- 변수 turn은 크리티컬 섹션에 들어갈 차례를 나타낸다.
- 플래그 `flag` 배열은 프로세스가 크리티컬 섹션에 진입할 준비가 되었는지 여부를 나타내는 데 사용.

`flag[i] = true` implies that process P_i is ready!

3. Peterson's Solution

Algorithm for Process Pi

```
while (true){

        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

3. Peterson's Solution

Peterson 솔루션의 정확성

- 세 가지 CS 요구 사항이 충족됨을 증명할 수 있다.

1. 상호 배제가 유지.

P_i enters CS only if:

either `flag[j] = false` or `turn = i`

2. 진행 조건 충족
3. 제한된 대기 Bounded-waiting 요구 사항이 충족됨

3. Peterson's Solution

Peterson의 솔루션과 현대 아키텍처

- 알고리즘을 시연하는 데 유용하지만 Peterson의 솔루션은 최신 아키텍처에서 작동한다고 보장되지 않는다.
  - 성능을 향상시키기 위해 프로세서 그리고 / 또는 컴파일러는 종속성이 없는 작업을 재정렬할 수 있다.
- 작동하지 않는 이유를 이해하면 경쟁 조건을 더 잘 이해하는 데 유용.
- 단일 스레드single-threaded 의 경우 결과가 항상 동일하므로 괜찮다.
- 다중 스레드 multithreaded 의 경우 재정렬로 인해 일관성이 없거나 예기치 않은 결과가 발생할 수 있다

3. Peterson's Solution

Modern Architecture Example

- 두 스레드가 데이터를 공유.

```
boolean flag = false;
   int x = 0;
```

- 스레드 1수행.

```
while (!flag)
      ;
   print x
```

- 스레드 2수행.

```
x = 100;
   flag = true
```

- 예상되는 출력은 무엇일까?

100

3. Peterson's Solution

Modern Architecture Example

- 그러나 변수 flag와 x는 서로 독립적이므로 명령어는 다음과 같다.

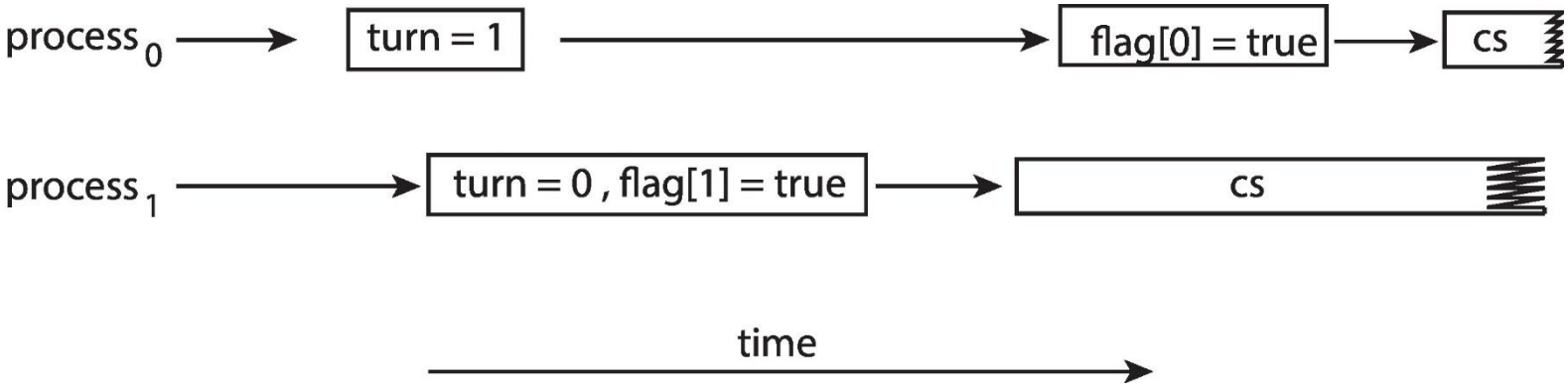
```
flag = true;
  x = 100;
```

- 스레드 2의 경우 재정렬될 수 있음
- 이 경우 출력이 0이 될 수 있다

3. Peterson's Solution

Peterson의 솔루션 재검토

- Peterson의 솔루션에서 명령어 재정렬의 효과



- 이렇게 하면 두 프로세스가 동시에 임계 영역에 있을 수 있다
- Peterson의 솔루션이 최신 컴퓨터 아키텍처에서 올바르게 작동하도록 하려면 Memory Barrier를 사용해야 함.

3. Peterson's Solution

Memory Barrier

- 메모리 모델 Memory model 은 컴퓨터 아키텍처가 응용 프로그램에 대해 보장하는 메모리.
- 메모리 모델은 다음 중 하나일 수 있다.
  - 강력하게 정렬됨 Strongly ordered - 한 프로세서의 메모리 수정이 다른 모든 프로세서에 즉시 표시.
  - 약하게 주문됨 Weakly ordered - 한 프로세서의 메모리 수정이 다른 모든 프로세서에 즉시 표시되지 않을 수 있다.
- 메모리 배리어 memory barrier 는 메모리의 모든 변경 사항을 다른 모든 프로세서에 전파(표시)하도록 강제하는 명령.

3. Peterson's Solution

Memory Barrier Instructions

- 메모리 장벽 명령이 수행되면 시스템은 후속 로드 또는 저장 작업이 수행되기 전에 모든 로드 및 저장이 완료되도록 함.
- 따라서 명령이 재정렬되더라도 메모리 배리어는 저장 작업이 메모리에서 완료되고 향후 로드 또는 저장 작업이 수행되기 전에 다른 프로세서에서 볼 수 있도록 함.

3. Peterson's Solution

Memory Barrier Example

- 스레드 1이 100을 출력하도록 다음 지침에 메모리 장벽을 추가할 수 있다.
- 스레드 1 수행

```
while (!flag)
      memory_barrier();
  print x
```

- 스레드 2 수행
-

```
 x = 100;
memory_barrier();
flag = true
```

- 스레드 1의 경우 플래그 값이 x 값보다 먼저 로드됨을 보장.
- 스레드 2의 경우 x에 대한 할당이 할당 플래그 `flag` 이전에 발생하는지 확인.

3. Peterson's Solution

원자 변수 atomic variable

- 일반적으로 compare_and_swap()과 같은 하드웨어 명령어는, 원자적 변수 (Atomic Variable) 같은 소프트웨어 도구를 만드는 데 사용된다.

- 원자적 변수 (Atomic Variable)는 정수, boolean과 같은 기본 데이터 타입에 대한 원자적 연산을 제공하며, 공유 데이터에 대한 race condition이 발생할 수 있는 상황에서 상호 배제를 보장해준다.

3. Peterson's Solution

Java implementation of Peterson's solution

파일	소스코드
실습환경	

소스코드	s@DESKTOP-R0EQ2U6:~/java_workspaces$ cat > Peterson1.java
	public class Peterson1 {
	  static int count = 0; // shared data
	  static int turn = 0;
	  static boolean[] flag = new boolean[2];
	
	  public static void main(String[] args) throws InterruptedException {
	  Thread t1 = new Thread(new Producer());
	  Thread t2 = new Thread(new Consumer());
	  t1.start(); t2.start();
	  t1.join(); t2.join();
	  System.out.println(Peterson1.count);
	  }
	
	  static class Producer implements Runnable{
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  // entry section
	  flag[0] = true;
	  turn = 1;
	  while(flag[1] && turn == 1)
	  ;
	
	  // critical section
	  count++;
	
	  // exit section
	  flag[0] = false;
	
	  // remainder section
	  }
	  }
	  }

3. Peterson's Solution

Java implementation of Peterson's solution

파일	소스코드
실습환경	

소스코드	static class Consumer implements Runnable{
	@Override
	public void run() {
	for (int i = 0; i < 10000; i++) {
	// entry section
	flag[1] = true;
	turn = 0;
	while(flag[0] && turn == 0)
	;
	
	// critical section
	count--;
	
	// exit section
	flag[1] = false;
	
	// remainder section
	}
	}
	}
	}

결과값1	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac Peterson1.java k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java Peterson1 0 k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java Peterson1 0 k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java Peterson1 0 k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java Peterson1 0
비고	• 이 코드를 실제로 실행해보면, 매번 결과가 다르게 나온다.

3. Peterson's Solution

Java implementation of Peterson's solution

파일	소스코드
실습환경	

소스코드	import java.util.concurrent.atomic.AtomicBoolean;
	
	public class Peterson2 {
	  static int count = 0;
	
	  static int turn = 0;
	  static AtomicBoolean[] flag;
	
	  static{
	  flag = new AtomicBoolean[2];
	  for (int i = 0; i < flag.length; i++) {
	  flag[i] = new AtomicBoolean();
	  }
	  }
	
	  public static void main(String[] args) throws InterruptedException {
	  Thread t1 = new Thread(new Producer());
	  Thread t2 = new Thread(new Consumer());
	  t1.start(); t2.start();
	  t1.join(); t2.join();
	  System.out.println(Peterson2.count);
	  }

결과값1	
비고	

3. Peterson's Solution

Java implementation of Peterson's solution

파일	소스코드
실습환경	

소스코드	static class Producer implements Runnable {
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  // entry section
	  flag[0].set(true);
	  turn = 1;
	  while(flag[1].get() && turn == 1)
	  ;
	
	  // critical section
	  count++;
	
	  // exit section
	  flag[0].set(false);
	
	  // remainder section
	  }
	  }
	}

결과값1	
비고	

3. Peterson's Solution

Java implementation of Peterson's solution

파일	소스코드
실습환경	

소스코드	static class Consumer implements Runnable {
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  // entry section
	  flag[1].set(true);
	  turn = 0;
	  while(flag[0].get() && turn == 0)
	  ;
	
	  // critical section
	  count--;
	
	  // exit section
	  flag[1].set(false);
	
	  // remainder section
	  }
	  }
	  }
	}

결과값1	} k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac Peterson2.java k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java Peterson2 0 k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java Peterson2 0 k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java Peterson2 0 k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java Peterson2 0
비고	• 이렇게 하면, 매번 실행 결과가 0으로 동일하게 나온다! • 왜냐하면 AtomicBoolean 타입의 flag 배열 원소는, entry section에서 set, get 함수로 값을 변경할 때 원자성이 보장되어, 중간에 스레드 간의 컨텍스트 스위치가 발생하지 않기 때문이다. • 따라서 생산자와 소비자 스레드의 공유 데이터인 count 변수는 실행할 때마다 동일한 결과가 나온다.

4. Synchronization Hardware

하드웨어 기반 솔루션

- 많은 시스템이 크리티컬 섹션 코드 구현을 위한 하드웨어 지원을 제공.
- 단일 프로세서 Uniprocessors – 인터럽트를 비활성화할 수 있음
  - 현재 실행 중인 코드는 선점 없이 실행.
  - 일반적으로 다중 프로세서 시스템에서 너무 비효율적.
  - 이를 사용하는 운영 체제는 광범위하게 확장할 수 없다.

- 다음 세 가지 형태의 하드웨어 지원을 살펴보겠다.

1. 하드웨어 Instructions
2. 원자 변수

4. Synchronization Hardware

Hardware Instructions

- 단어의 내용을 테스트 및 수정 test-and-modify 하거나 두 단어의 내용을 원자적으로(중단 없이) 교환할 수 있는 특수 하드웨어 명령.
- 테스트 및 설정 지침 Test-and-Set instruction
- 비교 및 교환 명령Compare-and-Swap instruction

4. Synchronization Hardware

The test_and_set Instruction

- **Definition**

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv:
}
```

- **속성**

- 원자적으로 실행됨
- 전달된 매개변수의 원래 값을 반환합니다.
- 전달된 매개변수의 새 값을 true로 설정

4. Synchronization Hardware

Solution Using test_and_set()

- `false` 로 초기화된 공유 부울 변수 잠금 `lock`

Solution:

```
do {
  while (test_and_set(&lock))
    ; /* do nothing */

      /* critical section */

  lock = false;
      /* remainder section */
} while (true);
```

- 크리티컬 섹션 문제를 해결할까?

4. Synchronization Hardware

compare_and_swap 명령어

- **Definition**

```
int compare_and_swap(int *value, int expected, int new_value)
  {
   int temp = *value;
    if (*value == expected)
        *value = new_value;
     return temp;
 }
```

- **속성**

- 원자적으로 실행됨
- 전달된 매개변수 값의 원래 값을 반환.
- *value == expected가 true인 경우에만 변수 값을 전달된 매개변수 new_value의 값으로 설정. 즉, 스왑은 이 조건에서만 발생.

4. Synchronization Hardware

compare_and_swap을 사용한 솔루션

- 공유 정수 잠금이 0으로 초기화.

```
int lock = 0
```

- Definition

```
while (true){
        while (compare_and_swap(&lock, 0, 1) != 0)
               ; /* do nothing */

          /* critical section */

          lock = 0;

          /* remainder section */
      }
```

- 크리티컬 섹션 문제를 해결할까?

4. Synchronization Hardware

비교 및 교환을 통한 경계 대기

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

4. Synchronization Hardware

원자 변수 atomic variable

- 일반적으로 비교 및 교환과 같은 명령은 다른 동기화 도구의 빌딩 블록으로 사용.
- 한 가지 도구는 정수 및 부울과 같은 기본 데이터 유형에 대한 원자 atomic (무중단 uninterruptible) 업데이트를 제공하는 원자 변수 atomic variable 다.
- 예를 들어:
  - 시퀀스 sequence 를 원자 변수로 설정
  - increment()가 원자 변수 시퀀스에 대한 연산이 되도록 하시오.
  - 명령:

```
increment(&sequence);
```

- 시퀀스 sequence 가 중단 없이 증가하도록 함

4. Synchronization Hardware

원자 변수 atomic variable

- increment() 함수는 다음과 같이 구현할 수 있다.

```
void increment(atomic_int *v)
{
       int temp;
       do {
          temp = *v;
       }
       while (temp != (compare_and_swap(v,temp,temp+1));
}
```

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	

소스코드	k8s@DESKTOP-R0EQ2U6:~/java_workspaces$ cat > SynchExample1.java
	public class SynchExample1 {
	  static class Counter {
	  public static int count = 0;
	  public static void increment(){
	  count++;
	  }
	  }
	
	  static class MyRunnable implements Runnable {
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  Counter.increment();
	  }
	  }
	  }
	
	  public static void main(String[] args) throws InterruptedException {
	  Thread[] threads = new Thread[5];
	
	  for (int i = 0; i < threads.length; i++) {
	  threads[i] = new Thread(new MyRunnable());
	  threads[i].start();
	  }
	  for (Thread thread : threads) {
	  thread.join();
	  }
	
	  System.out.println("counter = " + Counter.count);
	  }
	}

결과값1	
비고	

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	
소스코드	
결과값1	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac SynchExample1.java k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java SynchExample1 counter = 40920
비고	• 50000 근처에도 가지 못하는 심각한 동기화 문제가 발생하는 걸 볼 수 있다.

Counter.increment() 메서드는 count 변수를 1씩 증가시킵니다. 그러나 이 코드에는 경쟁 상태(Race Condition) 문제가 있습니다.

여러 스레드가 동시에 Counter.increment() 메서드를 호출하여 count 변수를 증가시키는 작업을 수행할 때, 서로의 작업 결과가 덮어씌워질 수 있습니다. 이는 예상치 못한 결과를 초래할 수 있습니다.

이러한 경쟁 상태를 해결하기 위해서는 동기화 메커니즘을 사용해야 합니다. 가장 간단한 방법은 increment() 메서드에 synchronized 키워드를 사용하여 동기화하는 것입니다. 이렇게 하면 한 번에 하나의 스레드만 increment() 메서드에 접근하도록 합니다.

4. Synchronization Hardware

Java Synchronization Example

```
public static synchronized void increment(){
   count++;
}
++++++++++++++++++++++++++++++++++++++++++++
import java.util.concurrent.atomic.AtomicInteger;

public static AtomicInteger count = new AtomicInteger(0);

public static void increment(){
   count.incrementAndGet();
}
++++++++++++++++++++++++++++++++++++++++++++
import java.util.concurrent.locks.ReentrantLock;

static class Counter {
   private static int count = 0;
   private static final ReentrantLock lock = new ReentrantLock();

   public static void increment(){
      lock.lock();
      try {
         count++;
      } finally {
         lock.unlock();
      }
   }
}
```

Java Synchronization Example

파일	소스코드
실습환경	

소스코드	k8s@DESKTOP-R0EQ2U6:~/java_workspaces$ cat > SynchExample2.java
	public class SynchExample2 {
	  static class Counter {
	  public static int count = 0;
	  synchronized public static void increment(){ // 이 부분만 달라짐.
	  count++;
	  }
	  }
	
	  static class MyRunnable implements Runnable {
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  Counter.increment();
	  }
	  }
	  }
	
	  public static void main(String[] args) throws InterruptedException {
	  Thread[] threads = new Thread[5];
	
	  for (int i = 0; i < threads.length; i++) {
	  threads[i] = new Thread(new MyRunnable()); // run 메서드 호출
	  threads[i].start();
	  }
	  for (Thread thread : threads) {
	  thread.join();
	  }
	
	  System.out.println("counter = " + Counter.count);
	  }
	}

결과값1	
비고	

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	
소스코드	
결과값1	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac SynchExample2.java k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java SynchExample2 counter = 50000
비고	• 예제 1의 문제를 해결하기 위해서, count라는 공유 변수의 값을 변경하는 increment()라는 메서드에 synchronized 키워드를 붙일 수 있다. 그러면, increment()의 내용 전체는 임계 영역이 되며, 모니터락을 획득한 스레드만 진입할 수 있게 된다. 그리고 함수가 종료되면, this 객체는 모니터락을 반납한다. 이 코드를 실행하면, 항상 50000이라는 값이 나온다. 동기화 성공! • 그런데, 위의 예시는 간단해서 그렇지 synchronized 키워드가 붙은 increment() 메서드의 내용이 많아지면, 공유 데이터와 관련된 실질적인 임계 영역에만 synchronized 키워드를 붙이는 게 더 좋다. 왜냐하면 그 외의 나머지 부분들은 remainder section이므로, critical section에만 synchronized 키워드를 붙여야 멀티 스레딩의 장점을 살릴 수 있기 때문이다.

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	

소스코드	k8s@DESKTOP-R0EQ2U6:~/java_workspaces$ cat > SynchExample3.java
	public class SynchExample3 {
	  static class Counter {
	  private static final Object object = new Object();
	  public static int count = 0;
	  public static void increment(){
	  // 모니터락을 획득할 객체 지정
	  synchronized (object) {
	  count++;
	  }
	  }
	  }
	
	  static class MyRunnable implements Runnable {
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  Counter.increment();
	  }
	  }
	  }
	
	  public static void main(String[] args) throws InterruptedException {
	  Thread[] threads = new Thread[5];
	
	  for (int i = 0; i < threads.length; i++) {
	  threads[i] = new Thread(new MyRunnable()); // run 메서드 호출
	  threads[i].start();
	  }
	  for (Thread thread : threads) {
	  thread.join();
	  }
	
	  System.out.println("counter = " + Counter.count);
	  }
	}

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	
소스코드	
결과값1	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac SynchExample3.java k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java SynchExample3 counter = 5
비고	• 항상 50000이라는 결과가 나온다.

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	

소스코드	k8s@DESKTOP-R0EQ2U6:~/java_workspaces$ cat > SynchExample4.java
	public class SynchExample4 {
	  static class Counter {
	  // 5개의 counter 객체가 공유하는 데이터 (static 필수)
	  public static int count = 0;
	  public void increment(){
	  // 자신의 객체 인스턴스가 모니터락을 획득하도록
	  // 그런데, 각 counter 객체가 자신의 모니터를 하나씩 갖고 있으므로
	  // 5개의 스레드가 동기화 되는 게 아니라, 혼자서만 동기화 됨.
	  synchronized (this) {
	  Counter.count++;
	  }
	  }
	  }
	
	  static class MyRunnable implements Runnable {
	  Counter counter;
	  public MyRunnable(Counter counter) {
	  this.counter = counter;
	  }
	
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  // static 메서드가 아니므로
	  // counter 객체를 생성하여 호출해야 함.
	  counter.increment();
	  }
	  }
	  }

결과값1	
비고	

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	

소스코드	public static void main(String[] args) throws InterruptedException {
	Thread[] threads = new Thread[5];
	
	for (int i = 0; i < threads.length; i++) {
	// MyRunnable 클래스마다 다른 counter 객체를 가짐.
	// 그 counter 객체들은 count 변수를 공유함.
	threads[i] = new Thread(new MyRunnable(new Counter()));
	threads[i].start();
	}
	for (Thread thread : threads) {
	thread.join();
	}
	
	System.out.println("counter = " + Counter.count);
	}
	}

결과값1	kk8s@DESKTOP-R0EQ2U6:~/java_workspaces$ javac SynchExample4.java k8s@DESKTOP-R0EQ2U6:~/java_workspaces$ java SynchExample4 counter = 49099
비고	• 위의 코드를 실행하면, counter 객체가 5개 만들어지고 각 객체는 서로 다른 모니터를 갖고 있으므로, 결국 하나의 모니터로 5개의 스레드가 동기화 되는 게 아니라, 각 스레드가 혼자서만 동기화 되는 문제가 발생한다. 그래서 count 값은 실행할 때마다 다른 결과가 나온다.

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	

소스코드	k8s@DESKTOP-R0EQ2U6:~/java_workspaces$ cat > SynchExample5.java
	public class SynchExample5 {
	  static class Counter {
	  public static int count = 0;
	  public void increment(){
	  // counter 객체의 모니터락을 5개의 스레드가 공유함.
	  synchronized (this) {
	  Counter.count++;
	  }
	  }
	  }
	
	  static class MyRunnable implements Runnable {
	  Counter counter;
	  public MyRunnable(Counter counter) {
	  this.counter = counter;
	  }
	
	  @Override
	  public void run() {
	  for (int i = 0; i < 10000; i++) {
	  counter.increment();
	  }
	  }
	  }

결과값1	
비고	

4. Synchronization Hardware

Java Synchronization Example

파일	소스코드
실습환경	

소스코드	public static void main(String[] args) throws InterruptedException {
	Thread[] threads = new Thread[5]; // 스레드는 5개
	Counter counter = new Counter(); // counter 객체는 하나
	
	for (int i = 0; i < threads.length; i++) {
	threads[i] = new Thread(new MyRunnable(counter));
	threads[i].start();
	}
	for (Thread thread : threads) {
	thread.join();
	}
	
	System.out.println("counter = " + Counter.count);
	}
	}

결과값1	k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac SynchExample5.java k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java SynchExample5 counter = 50000
비고	• 항상 50000이라는 결과가 나온다

5. Mutex Locks

Simplest is mutex(mutual exclusion) lock

- 이전 솔루션은 복잡하고 일반적으로 애플리케이션 프로그래머가 액세스할 수 없다.
- OS 설계자는 중요한 섹션 문제를 해결하기 위한 소프트웨어 도구를 구축.
- 가장 간단한 것은 뮤텍스 잠금mutex lock.
  - 잠금을 사용할 수 있는지 여부를 나타내는 부울 변수
- 크리티컬 섹션 보호
  First acquire() a lock
  Then release() the lock
- acquire() and release() 에 대한 호출은 원자적이어야 .
  - 일반적으로 비교 및 스왑과 같은 하드웨어 원자 명령어를 통해 구현.
- 그러나이 솔루션에는 바쁜 대기busy waiting가 필요
  - 따라서 이 잠금 장치를 스핀 잠금 장치spinlock라고 합니다.

5. Mutex Locks

Mutex Locks 사용한 CS 문제 해결 방법

```
while (true) {
        acquire lock
            critical section
        release lock
remainder section
}
```

Mutex Locks 사용한 CS 문제 해결 방법

k8s@DESKTOP-R0EQ2U6:~/c_workspaces$ cat > MutexLocks.c

```
#include <stdio.h>
#include <pthread.h>

int sum = 0;  //a shared variable

pthread_mutex_t mutex;

void *counter(void *param)
{
	int k;
	for(k = 0; k < 10000; k++){
		//entry section
		pthread_mutex_lock(&mutex);

		//critical section
		sum++;

		//exit section
		pthread_mutex_unlock(&mutex);

	}
	pthread_exit(0);
}

int main()
{
	pthread_t tid1, tid2;
	pthread_mutex_init(&mutex, NULL);
	pthread_create(&tid1, NULL, counter, NULL);
	pthread_create(&tid2, NULL, counter, NULL);
	pthread_join(tid1, NULL);
	pthread_join(tid2, NULL);

	printf("SUM = %d\n", sum);
}
```

5. Mutex Locks

Mutex Locks 사용한 CS 문제 해결 방법

```
k8s@DESKTOP-R0EQ2U6:~/c_workspaces$ gcc -pthread MutexLocks.c
k8s@DESKTOP-R0EQ2U6:~/c_workspaces$ ./a.out
SUM = 20000
k8s@DESKTOP-R0EQ2U6:~/c_workspaces$
```

6. Semaphore

Semaphore

- 프로세스가 활동을 동기화할 수 있는 보다 정교한 방법(Mutex 잠금보다)을 제공하는 동기화 도구.
- Semaphore S– 정수 변수
- 두 개의 불가분(원자적) 작업을 통해서만 액세스할 수 있다.

- wait() and signal()

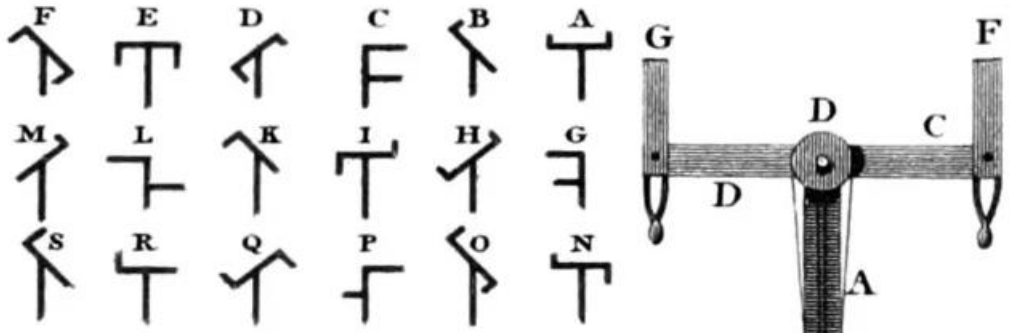
Originally called P() and V()

- wait() 작업의 정의

```
wait(S) {
   while (S <= 0)
     ; // busy wait
   S--;
}
```

- signal() 연산의 정의

```
signal(S) {
   S++;
}
```



P() and V() are introduced by Edsger Dijkstra

- Proberen(to test) and Verhogen(to increment)

출처 : https://namu.wiki/w/%EC%84%B8%EB%A7%88%ED%8F%AC%EC%96%B4%20%ED%86%B5%EC%8B%A0

6. Semaphore

Semaphore

- 세마포 계산 Counting semaphore - 정수 값은 무제한 도메인에 걸쳐 있을 수 있다.
- 이진 세마포 Binary semaphore – 정수 값의 범위는 0과 1 사이여야 한다.

- mutex lock과 동일

- 카운팅 세마포어 S를 바이너리 세마포어로 구현할 수 있다.
- 세마포어를 사용하면 다양한 동기화 문제를 해결할 수 있다.

6. Semaphore

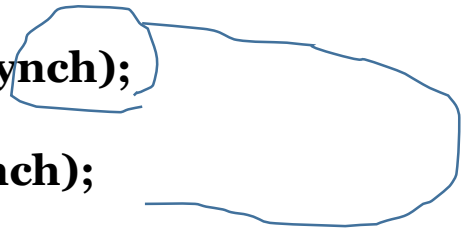
세마포어 사용 예

- CS 문제에 대한 해결책
- 1로 초기화된 세마포어 "뮤텍스" 생성1

```
wait(mutex);
  CS
signal(mutex);
```

- 두 개의 명령문 S1과 S2가 있고 S1이 S2보다 먼저 발생해야 한다는 요구 사항이 있는 P1과 P2를 고려하십시오.
- 0으로 초기화된 세마포어 "동기화" 생성

```
P1:
  S1;
  signal(synch);
P2:
  wait(synch);
  S2;
```



시그널을 줘야 P2 실행

6. Semaphore

Semaphore Implementation

- 두 프로세스가 동시에 같은 세마포어에서 wait() 및 signal()을 실행할 수 없도록 보장해야 함.
- 따라서 구현은 대기 wait 및 신호 signal 코드가 임계 영역에 배치되는 임계 영역 문제가 된다.
- 이제 중요한 섹션 구현에서 바쁜 대기 busy waiting 를 할 수 있다.
  - 그러나 구현 코드는 짧다.
  - 크리티컬 섹션이 거의 점유되지 않은 경우 조금 바쁜 대기
- 응용 프로그램은 중요한 섹션에서 많은 시간을 소비할 수 있으므로 이는 좋은 솔루션이 아니다.

6. Semaphore

바쁜 대기 없이 세마포어 구현

- 각 세마포어에는 연결된 대기 큐가 있다.
- 대기 대기열의 각 항목에는 두 개의 데이터 항목이 있다.
  - 값(정수 유형)
  - 목록의 다음 레코드에 대한 포인터
- 두 작업:
  - 블록 block - 작업을 호출하는 프로세스를 적절한 대기 큐에 배치
  - wakeup – 대기 큐에서 프로세스 중 하나를 제거하고 준비 큐에 넣는다.

- **Waiting queue**

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

6. Semaphore

바쁜 대기 없이 세마포어 구현

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->list;
    block();  // 또는 sleep();
  }
}

signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
  }
}
```

Problems with Semaphores

- 세마포어 연산의 잘못된 사용:

  `signal(mutex)` …. `wait(mutex)`

  `wait(mutex)` … `wait(mutex)`

  Omitting of `wait (mutex)` and/or `signal (mutex)`

- 이것들과 다른 것들은 세마포어와 다른 동기화 도구가 잘못 사용될 때 발생할 수 있는 일의 예.

6. Semaphore

EX with Semaphores

k8s@DESKTOP-R0EQ2U6:~/c_workspaces$ cat > SemaphoresEx1.c

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int sum = 0;  //a shared variable

sem_t sem;

void *counter(void *param)
{
	int k;
	for(k = 0; k < 10000; k++){
		//entry section
		sem_wait(&sem);

		//critical section
		sum++;

		//exit section
		sem_post(&sem);

	}
	pthread_exit(0);
}
```

6. Semaphore

EX with Semaphores

```
int main()
{
	pthread_t tid1, tid2;
	sem_init(&sem, 0, 1);
	pthread_create(&tid1, NULL, counter, NULL);
	pthread_create(&tid2, NULL, counter, NULL);
	pthread_join(tid1, NULL);
	pthread_join(tid2, NULL);

	printf("SUM = %d\n", sum);
}
```

```
k8s@DESKTOP-R0EQ2U6:~/c_workspaces$ gcc -pthread SemaphoresEx1.c
k8s@DESKTOP-R0EQ2U6:~/c_workspaces$ ./a.out
SUM = 20000
k8s@DESKTOP-R0EQ2U6:~/c_workspaces$
```

7. Monitors

Monitors?

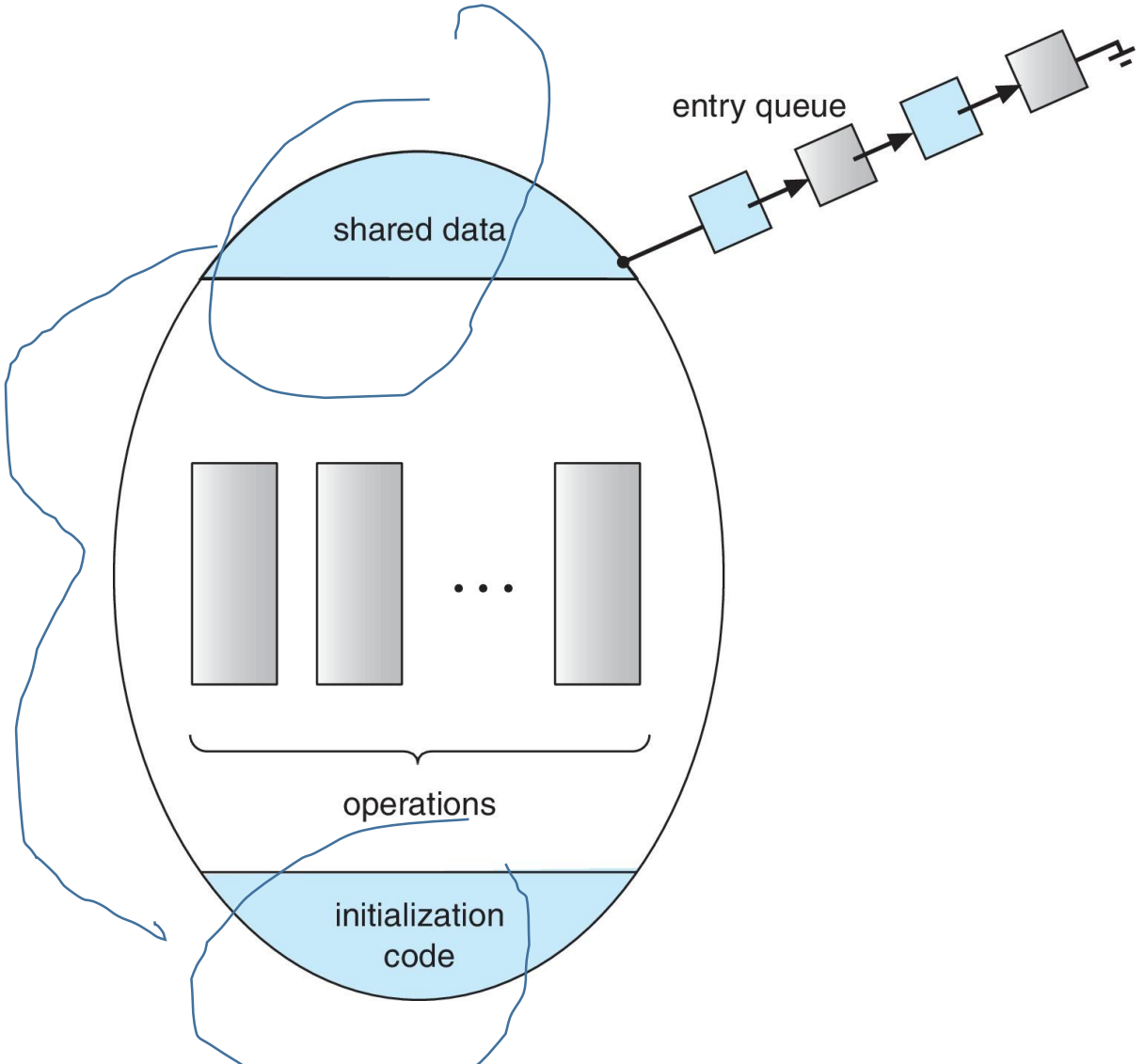
- 프로세스 동기화를 위한 편리하고 효과적인 메커니즘을 제공하는 높은 수준의 추상화
- 추상 데이터 유형, 프로시저 내의 코드로만 액세스할 수 있는 내부 변수
- 한 번에 하나의 프로세스만 모니터 내에서 활성화될 수 있다.
- 모니터의 의사 코드 구문:

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }

        procedure P2 (…) { …. }

        procedure Pn (…) {……}

  initialization code (…) { … }
}
```

7. Monitors

모니터의 개략도

Monitors



7. Monitors

세마포어를 사용하여 구현 모니터링

- **Variables**

```
semaphore mutex
mutex = 1
```

- 각 절차 P는 다음으로 대체

```
wait(mutex);
        …
          body of P;
        …
signal(mutex);
```

- 모니터 내 상호 배제 보장

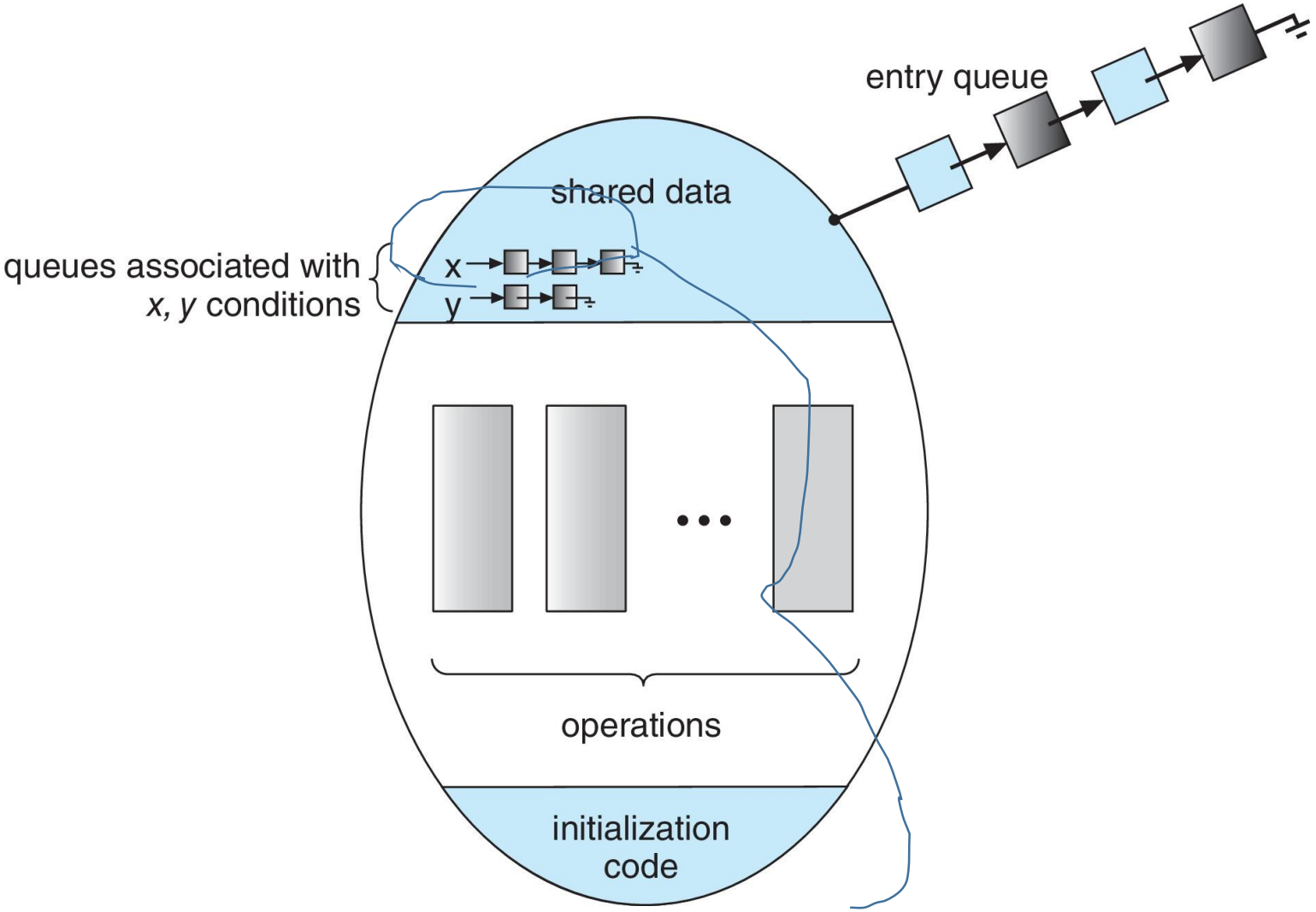
7. Monitors

조건 변수

- 조건 x, y;
- 조건 변수에는 두 가지 작업이 허용.
- x.wait() – 작업을 호출하는 프로세스는 x.signal()까지 일시 중지된다.
- x.signal() – x.wait()를 호출한 프로세스(있는 경우) 중 하나를 재개한다.
- 변수에 x.wait()가 없으면 변수에 영향을 미치지 않는다.

7. Monitors

조건 변수로 모니터링



7. Monitors

조건변수 사용 예시

- 두 개의 명령문 S1과 S2를 실행해야 하는 P1과 P2와 S1이 S2보다 먼저 발생해야 한다는 요구 사항을 고려.

- P1 및 P2에서 각각 호출되는 두 개의 프로시저 F1 및 F2로 모니터를 작성하십시오.
- 하나의 조건 변수 "x"가 0으로 초기화됨
- 하나의 부울 변수 "완료“

```
F1:
        S1;
        done = true;
        x.signal();
F2:
        if done = false
           x.wait()
        S2;
```

7. Monitors

세마포어를 사용하여 구현 모니터링

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting
          inside the monitor
```

- 각 함수 P는 다음으로 대체

```
wait(mutex);
      …
body of P;
      …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- 모니터 내 상호 배제 보장

7. Monitors

구현 – 조건 변수

- 각 조건 변수 x에 대해 다음이 있다.

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- x.wait() 작업은 다음과 같이 구현할 수 있다.

```
x_count++;
if (next_count > 0)
     signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

7. Monitors

구현 – 조건 변수

- x.signal() 작업은 다음과 같이 구현할 수 있다.

```
if (x_count > 0) {
      next_count++;
      signal(x_sem);
      wait(next);
      next_count--;
}
```

7. Monitors

모니터 내에서 프로세스 재개

- 여러 프로세스가 조건 변수 x에 대기 중이고 x.signal()이 실행되면 어떤 프로세스를 재개해야 할까?
- FCFS는 종종 적절하지 않음
- 다음 형식의 조건부 대기 conditional-wait 구성을 사용.

```
x.wait(c)
```

- 장소 :
  - c는 정수(우선순위 번호라고 함)
  - 가장 낮은 번호(가장 높은 우선 순위)의 프로세스가 다음에 예약.

7. Monitors

단일 리소스 할당

- 프로세스가 리소스를 사용할 최대 시간을 지정하는 우선 순위 번호를 사용하여 경쟁 프로세스 간에 단일 리소스를 할당.

```
R.acquire(t);
    ...
 access the resurce;
    ...

R.release;
```

- 여기서 R은 ResourceAllocator 유형의 인스턴스.

7. Monitors

단일 리소스 할당

- 프로세스가 리소스를 사용할 최대 시간을 지정하는 우선 순위 번호를 사용하여 경쟁 프로세스 간에 단일 리소스를 할당.
- 시간이 가장 짧은 프로세스에게 자원을 먼저 할당
- R이 ResourceAllocator 유형의 인스턴스라고 가정 ResourceAllocator에 대한 액세스는 다음을 통해 수행.

```
R.acquire(t);
    ...
 access the resurce;
    ...

R.release;
```

- 여기서 t는 프로세스가 리소스를 사용하기 위해 계획하는 최대 시간.

7. Monitors

단일 리소스를 할당하는 모니터

```
monitor ResourceAllocator
{
                    boolean busy;
                    condition x;
                    void acquire(int time) {
                        if (busy)
                           x.wait(time);
                        busy = true;
                    }
                    void release() {
                        busy = false;
                        x.signal();
                    }
  initialization code() {
                     busy = false;
                    }
}
```

7. Monitors

단일 리소스를 할당하는 모니터

- 용법:

```
acquire
  ...
release
```

모니터 작업의 잘못된 사용

```
release()   …   acquire()
acquire()   …   acquire())
```

- capture() 및/또는 release() 생략

7. Monitors

- 앞 선 Java Synchronization Example에서 설명이 되어 있음

8. Liveness

단일 리소스를 할당하는 모니터

- 프로세스는 뮤텍스 잠금 또는 세마포어와 같은 동기화 도구를 획득하려고 시도하는 동안 무기한 대기해야 할 수 있다.
- 무기한 대기는 이 장의 시작 부분에서 논의된 진행 및 제한된 대기 기준을 위반.
- 활성도는 프로세스가 진행되도록 하기 위해 시스템이 충족해야 하는 일련의 속성을 의미.
- 무기한 대기는 활동성 실패의 한 예.

8. Liveness

단일 리소스를 할당하는 모니터

- 교착 상태 – 두 개 이상의 프로세스가 대기 중인 프로세스 중 하나만으로 인해 발생할 수 있는 이벤트를 무한정 기다리고 있다.
- S와 Q를 1로 초기화된 두 개의 세마포어라고 하자.

P_0

```
wait(S);
wait(Q);
...
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
...
signal(Q);
signal(S);
```

- Po이 wait(S) 및 P1 wait(Q)를 실행하는지 고려하십시오. Po이 wait(Q)를 실행하면 P1이 signal(Q)을 실행할 때까지 기다려야 한다.
- 그러나 P1은 Po이 신호(S)를 실행할 때까지 대기.
- 이러한 signal() 작업은 실행되지 않으므로 Po과 P1은 교착 상태.

8. Liveness

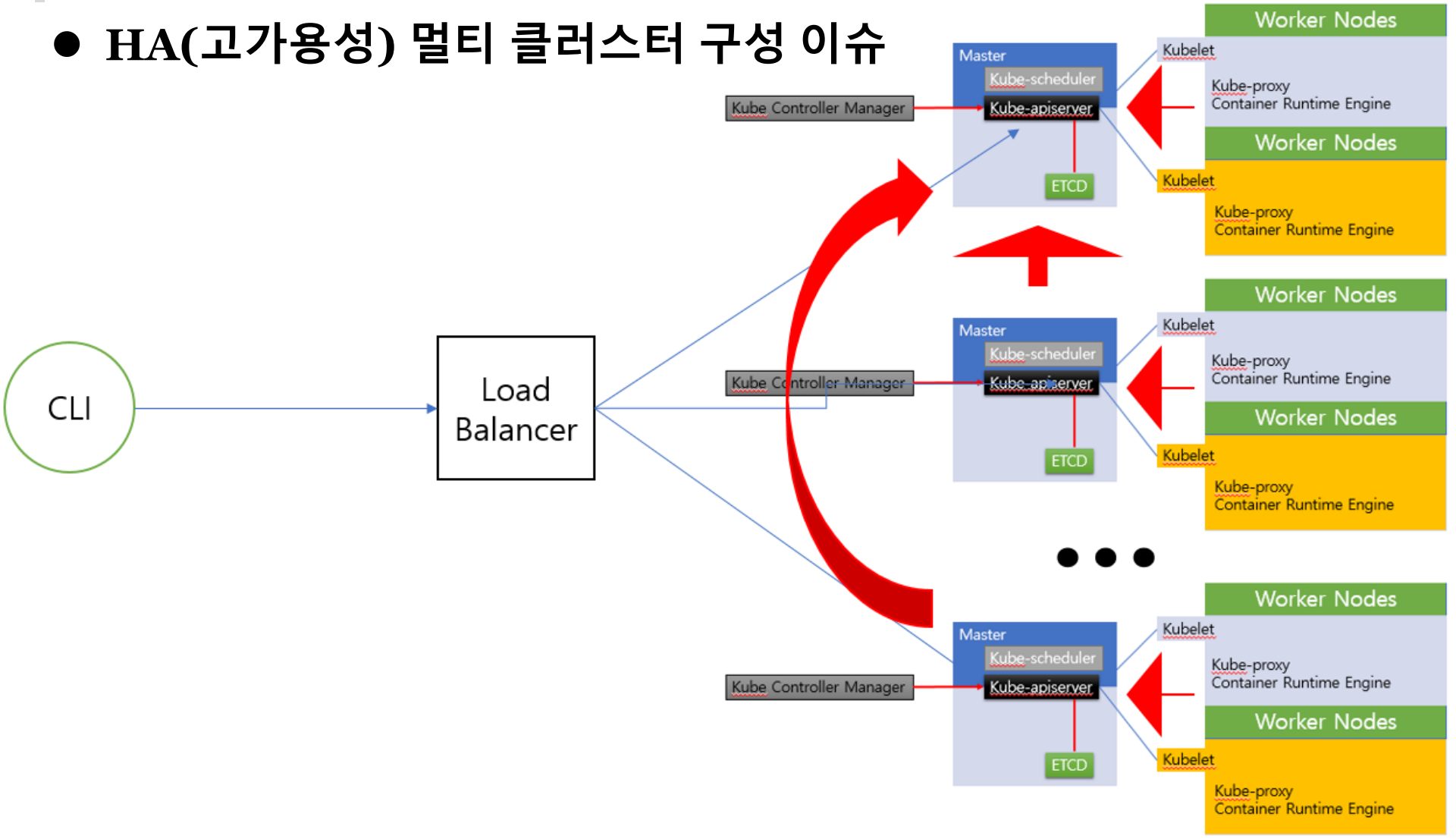
단일 리소스를 할당하는 모니터

- 다른 형태의 교착 상태:
- 기아 Starvation – 무기한 차단
  - 정지된 세마포 큐에서 프로세스를 제거할 수 없다.
- 우선 순위 역전 Priority Inversion – 우선 순위가 낮은 프로세스가 우선 순위가 높은 프로세스에 필요한 잠금을 보유하는 경우 스케줄링 문제
  - 우선 순위 상속 프로토콜priority-inheritance protocol을 통해 해결됨

1. Kubernetes scheduling

1. Kubernetes 클러스터

- HA(고가용성) 멀티 클러스터 구성 이슈



1. Kubernetes scheduling

2. Advanced scheduling

- Kubernetes의 가장 강력한 장점 중 하나는 강력하면서도 유연한 스케줄러. 스케줄러의 역할은 새로 생성 된 파드를 실행할 노드를 선택하는 것
- 노드에 파드 할당하기

- 특정한 노드(들) 집합에서만 동작하도록 파드를 제한할 수 있다. 이를 수행하는 방법에는 여러 가지가 있으며 권장되는 접근 방식은 모두 라벨 셀렉터를 사용하여 선택을 용이하게 한다. 보통 스케줄러가 자동으로 합리적인 배치(예: 자원이 부족한 노드에 파드를 배치하지 않도록 노드 간에 파드를 분배하는 등)를 수행하기에 이러한 제약 조건은 필요하지 않지만 간혹 파드가 배포할 노드를 제어해야 하는 경우가 있다. 예를 들어 SSD가 장착된 머신에 파드가 연결되도록 하거나 또는 동일한 가용성 영역(availability zone)에서 많은 것을 통신하는 두 개의 서로 다른 서비스의 파드를 같이 배치할 수 있다.

1. Kubernetes scheduling

2. Advanced scheduling

- 노드 셀렉터(nodeSelector)

- nodeSelector 는 가장 간단하고 권장되는 노드 선택 제약 조건의 형태이다. nodeSelector는 PodSpec의 필드이다. 이는 키-값 쌍의 매핑으로 지정한다. 파드가 노드에서 동작할 수 있으려면, 노드는 키-값의 쌍으로 표시되는 라벨을 각자 가지고 있어야 한다(이는 추가 라벨을 가지고 있을 수 있다). 일반적으로 하나의 키-값 쌍이 사용된다.
- 파드를 생성하면 쿠버네티스 마스터는 어떤 노드 위에서 실행할지 판단하여 스케줄링함. 쿠버네티스는 클러스터링 시스템이어서 사용자가 매번 노드를 선택할 필요 없이 쿠버네티스가 파드 스케줄링을 관리

1. Kubernetes scheduling

2. Advanced scheduling

- 그러나 명시적으로 노드를 선택해야 할 경우 있음

```
k8s@master1:~/fieldtrip-6$ kubectl get node  --show-labels
```

- 노드 확인

```
k8s@master1:~/fieldtrip-6$ kubectl describe node | grep kubernetes.io/hostname
```

- disktype 이라는 라벨 추가

• 마스터 노드에 disktype=ssd 라벨을 부여하고, 워커 노드에 disktype=hdd 라벨 부여

```
k8s@master1:~/fieldtrip-6$ kubectl label node master1.example.com disktype=ssd
k8s@master1:~/fieldtrip-6$  kubectl label node m1-worker1.example.com disktype=hdd1
k8s@master1:~/fieldtrip-6$ kubectl label node m1-worker2.example.com disktype=hdd2
```

- disktype 라벨 확인

```
k8s@master1:~/fieldtrip-6$ kubectl get node --show-labels | grep disktype
```

1. Kubernetes scheduling

2. Advanced scheduling

```
k8s@master1:~/fieldtrip-6$ nano node-selector.yaml
k8s@master1:~/fieldtrip-6$ kubectl create -f node-selector.yaml
k8s@master1:~/fieldtrip-6$ kubectl get pod node-selector -o wide
NAME          READY  STATUS   RESTARTS  AGE  IP         NODE                  NOMINATED NODE  READINESS GATES
node-selector  1/1    Running  0         8s   10.44.0.2  m1-worker1.example.com  <none>        <none>
```

```
apiVersion: v1
kind: Pod
metadata:
  name: node-selector
spec:
  containers:
  - name: nginx
    image: nginx
  # 특정 노드 라벨 선택
  nodeSelector:
    disktype: hdd1
```

1. Kubernetes scheduling

2. Advanced scheduling

- YAML 파일을 변경 후, 파드 생성 조회를 다시 해 본다. 잘 변경 되어 있음을 알 수 있다.

```
k8s@master1:~/fieldtrip-6$ nanio node-selector.yaml
k8s@master1:~/fieldtrip-6$ kubectl delete pod node-selector
k8s@master1:~/fieldtrip-6$ kubectl apply -f node-selector.yaml
k8s@master1:~/fieldtrip-6$ kubectl get pod node-selector -o wide
```

```
apiVersion: v1
kind: Pod
metadata:
 name: node-selector
spec:
 containers:
 - name: nginx
  image: nginx
 # 특정 노드 라벨 선택
 nodeSelector:
  disktype: hdd2
```

1. Kubernetes scheduling

2. Advanced scheduling

- Advanced scheduling은 nodeSelector외에도 특정 노드에 파드 스케줄링을 피하거나 감수하고 실행하는 Taint와 Toleration, 특정 노드나 파드와의 거리를 조절하는데 사용, 예를 들어 특정 파드끼리 서로 가까이 스케줄링되고 싶은 경우, Affinity(친밀함)을 사용하고 반대로 서로 멀리 스케줄링되고 싶은 경우, AntiAffinity(반친밀함)을 사용하는 Affinity와 AntiAffinity도 있다.