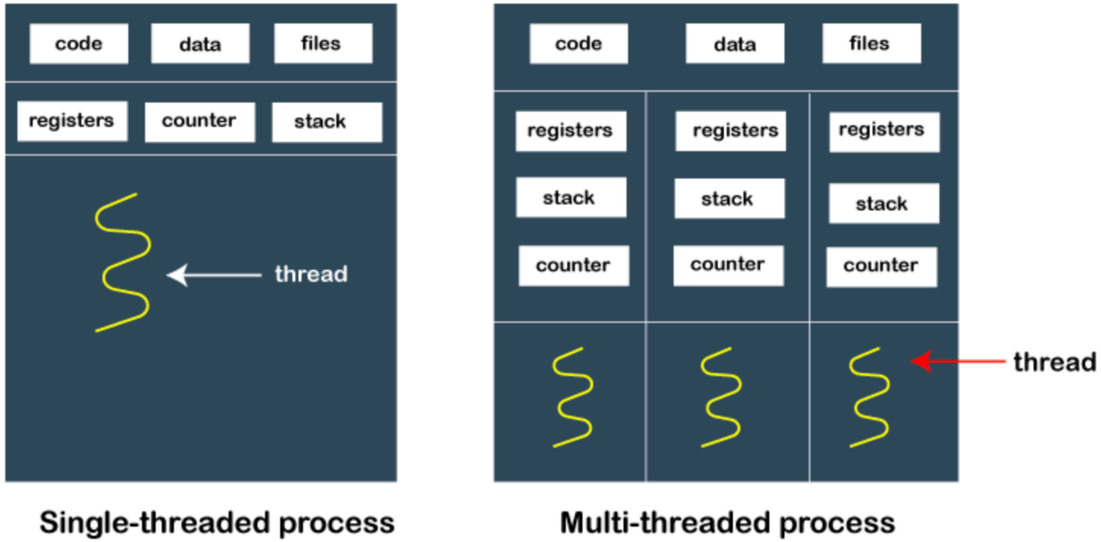
코딩 인터뷰 완전 분석(10_지식기반문제_개념_문제풀이)

CRACKING THE CODING INTERVIEW(CTCI)

23.08.21 전애지

목차

스레드



- 프로세스 내에서 실행되는 흐름의 단위
- 하나의 프로세스에는 하나 이상의 스레드가 존재
- 프로세스 내에서 병렬적으로 동작 → 여러 작업을 동시에 처리
- 프로세스 내에서 stack만 따로 할당 받고, code / data /heap 영역은 공유
- 두 개의 스레드가 번갈아가며 실행
- 통신은 메모리를 공유

자바의 스레드

자바에서 스레드 구현하는 방법

```
public class Main extends Thread {
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Thread 클래스 상속

자바의 스레드

자바에서 스레드 구현하는 방법

```
public class Main extends Thread {
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Thread 클래스 상속

1. main thread 생성
2. main 메서드에서 thread의 start( ) 호출
3. 새로운 thread 생성
4. run( ) 호출 → 작업 수행
5. 두 개의 스레드가 번갈아가며 실행

동기화와 락

동기화된 메서드

```
public class MyClass extends Thread {
  private String name;
  private MyObject myObj;

  public MyClass(MyObject obj, String n) {
    name = n;
    myObj = obj;
  }
  public void run() {
    myObj.foo(name);
  }
}

public class MyObject {
  public synchronized void foo(String name) {
    try {
      System.out.println("Thread " + name + ".foo(): starting");
      Thread.sleep(3000);
      System.out.println("Thread " + name + ".foo(): ending");
    } catch (InterruptedException exc) {
        System.out.println("Thread " + name + ": interrupted.");
    }
  }
}
```

동기화와 락

동기화된 메서드

```
public class MyClass extends Thread {
  private String name;
  private MyObject myObj;

  public MyClass(MyObject obj, String n) {
    name = n;
    myObj = obj;
  }
  public void run() {
    myObj.foo(name);
  }
}

public class MyObject {
  public synchronized void foo(String name) {
    try {
      System.out.println("Thread " + name + ".foo(): starting");
      Thread.sleep(3000);
      System.out.println("Thread " + name + ".foo(): ending");
    } catch (InterruptedException exc) {
        System.out.println("Thread " + name + ": interrupted.");
    }
  }
}
```

두 개의 MyClass 스레드 인스턴스가 foo를 동시에 호출할 수 있을까?

동기화와 락

동기화된 메서드

```
public class MyClass extends Thread {
  private String name;
  private MyObject myObj;

  public MyClass(MyObject obj, String n) {
    name = n;
    myObj = obj;
  }
  public void run() {
    myObj.foo(name);
  }
}

public class MyObject {
  public synchronized void foo(String name) {
    try {
      System.out.println("Thread " + name + ".foo(): starting");
      Thread.sleep(3000);
      System.out.println("Thread " + name + ".foo(): ending");
    } catch (InterruptedException exc) {
        System.out.println("Thread " + name + ": interrupted.");
    }
  }
}
```

main 메서드 구성

```
/* 서로 다른 객체인 경우 동시에 MyObject.foo() 호출이 가능하다. */
MyObject obj1 = new MyObject();
MyObject obj2 = new MyObject();
MyClass thread1 = new MyClass(obj1, "1");
MyClass thread2 = new MyClass(obj2, "2");
thread1.start();
thread2.start();
```

동기화와 락

동기화된 메서드

```
public class MyClass extends Thread {
  private String name;
  private MyObject myObj;

  public MyClass(MyObject obj, String n) {
    name = n;
    myObj = obj;
  }
  public void run() {
    myObj.foo(name);
  }
}

public class MyObject {
  public synchronized void foo(String name) {
    try {
      System.out.println("Thread " + name + ".foo(): starting");
      Thread.sleep(3000);
      System.out.println("Thread " + name + ".foo(): ending");
    } catch (InterruptedException exc) {
        System.out.println("Thread " + name + ": interrupted.");
    }
  }
}
```

main 메서드 구성

```
/* 서로 다른 객체인 경우 동시에 MyObject.foo() 호출이 가능하다. */
MyObject obj1 = new MyObject();
MyObject obj2 = new MyObject();
MyClass thread1 = new MyClass(obj1, "1");
MyClass thread2 = new MyClass(obj2, "2");
thread1.start();
thread2.start();
```

```
/* 같은 obj를 가리키고 있는 경우에는 하나만 foo를 호출할 수 있고,
*다른 하나는 기다리고 있어야 한다. */
MyObject obj = new MyObject();
MyClass thread1 = new MyClass(obj, "1");
MyClass thread2 = new MyClass (obj, "2");
thread1.start();
thread2.start();
```

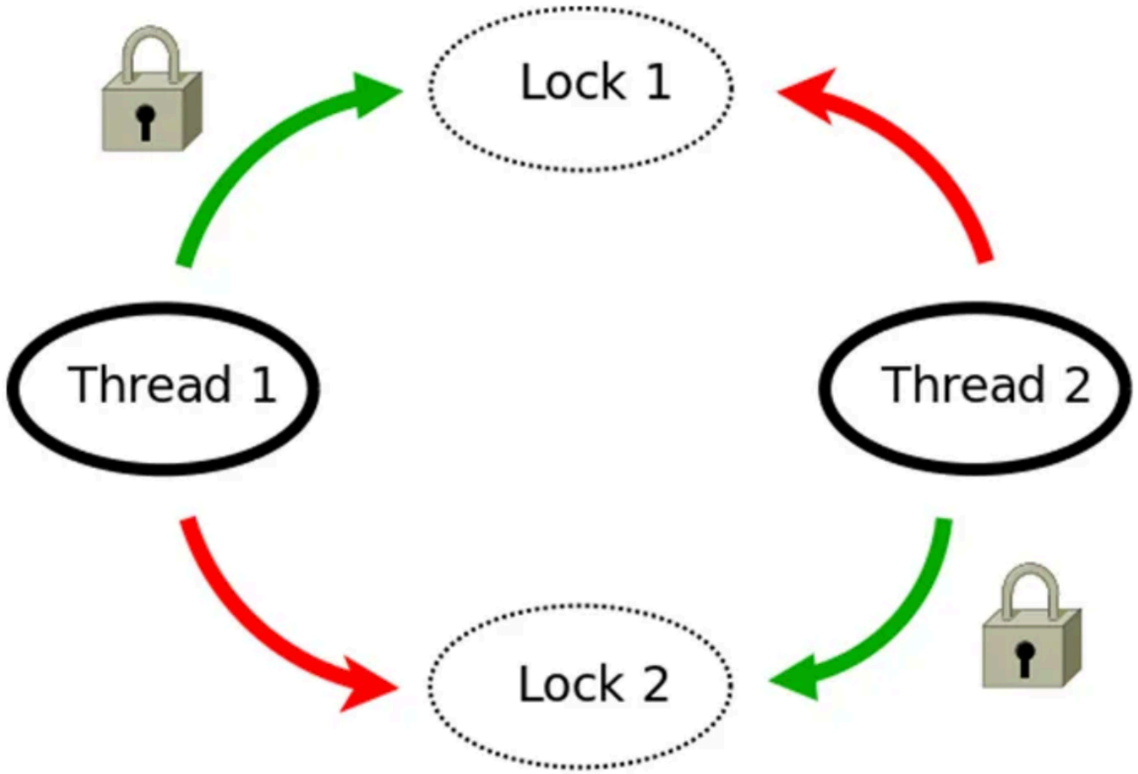
동기화와 락

동기화된 블록

```
public class MyClass extends Thread {
  ...
  public void run() {
    myObj.foo(name);
  }
}

public class MyObject {
  public void foo(String name) {
    synchronized (this) {
      ...
    }
  }
}
```

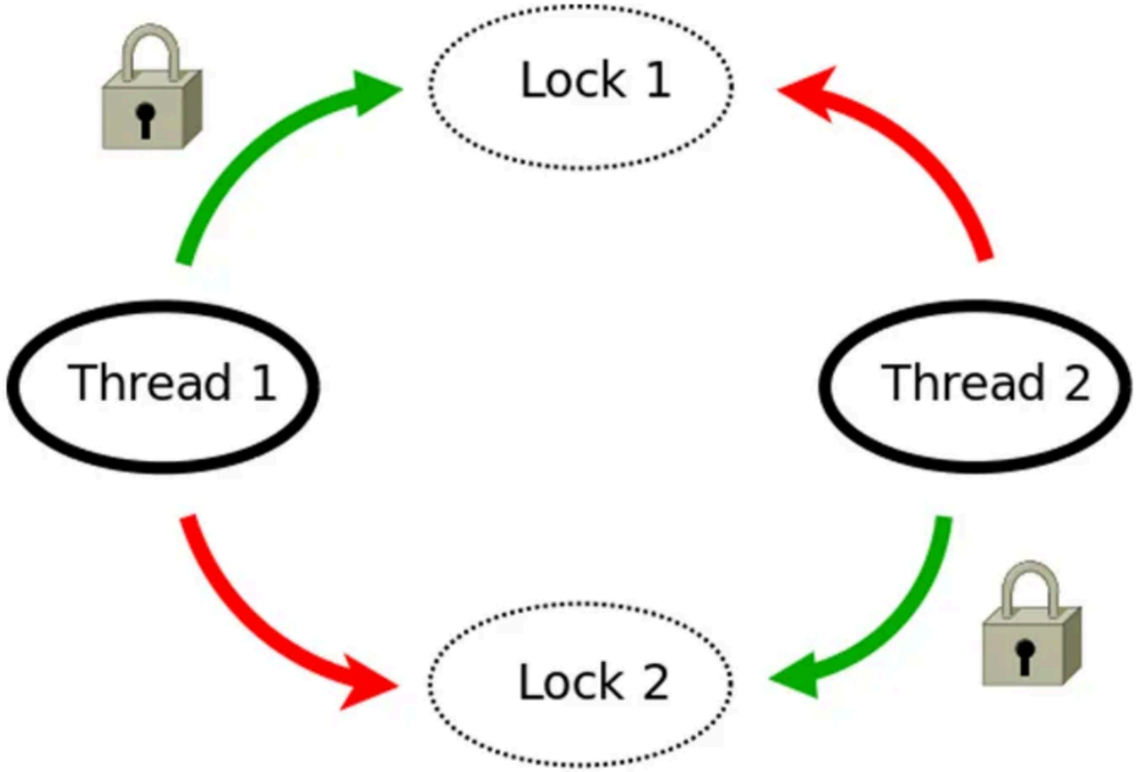
동기화와 락

락(lock)



동기화와 락

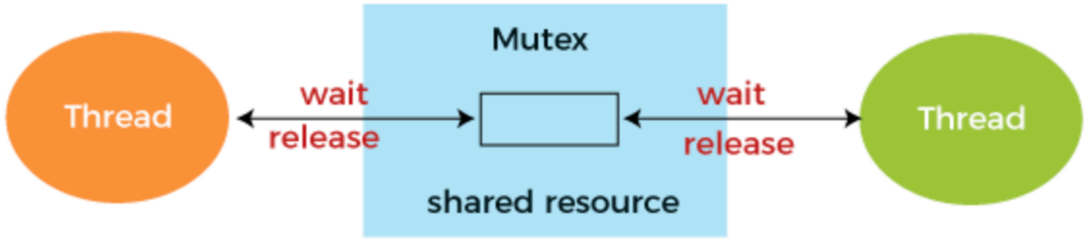
락(lock)




```
public class LockedATM {
  private Lock lock;
  private int balance = 100;

  public LockedATM() {
    lock = new ReentrantLock();
  }

  public int withdraw(int value) {
    lock.lock();
    int temp = balance;
    try {
      Thread.sleep(100);
      temp = temp - value;
      Thread.sleep(100);
      balance = temp;
    } catch (InterruptedException e) { }
    lock.unlock();
    return temp;
  }

  public int deposit (int value) {
    lock.lock();
    int temp = balance;
    try {
      Thread.sleep(100);
      temp = temp + value;
      Thread.sleep(300);
      balance = temp;
    } catch (InterruptedException e) { }
    lock.unlock();
    return temp;
  }
}
```

동기화와 락

뮤텍스와 세마포어



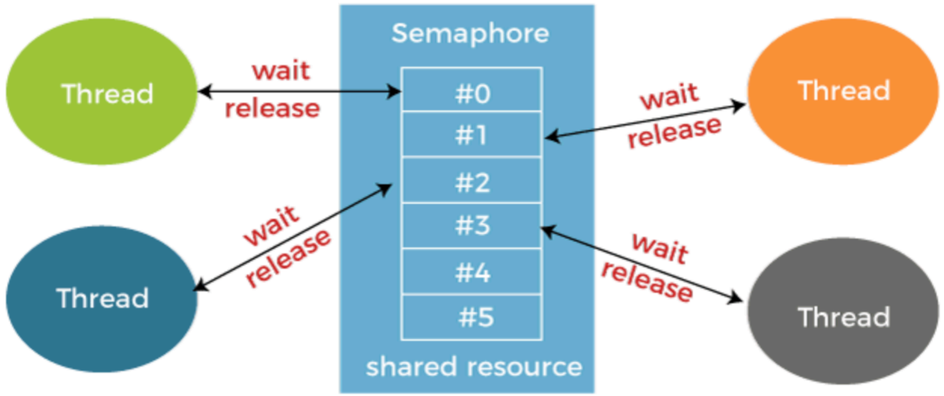

```
wait (mutex);
...
Critical Section
...
signal (mutex);
```

Mutex

- 뮤텍스 변수를 사용하여 상호 배제를 구현
- 락(Lock) 메커니즘, 락을 걸은 스레드만이 임계 영역을 나갈 때 락을 해제

동기화와 락

뮤텍스와 세마포어




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

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

Semaphore

- 카운팅 기반으로 동작
- 시그널(Signal) 메커니즘, 스레드나 프로세스 간에 신호를 주고 받음
- 0 이상의 정수 값을 가짐
- Wait (P)와 Signal (V) 연산

(15.5) 순서대로 호출

Foo 인스턴스 하나를 서로 다른 세 스레드에 전달한다. threadA는 first를 호출할 것이고, threadB는 second를 호출할 것이며, threadC는 third를 호출 할 것이다. first가 second보다 먼저 호출되고, second가 third보다 먼저 호 출되도록 보장하는 메커니즘을 설계하라.

```
public class Foo {
  public Foo() { ... }
  public void first() { ... }
  public void second() { ... }
  public void third() { ... }
}
```

(15.5) 순서대로 호출

Foo 인스턴스 하나를 서로 다른 세 스레드에 전달한다. threadA는 first를 호출할 것이고, threadB는 second를 호출할 것이며, threadC는 third를 호출 할 것이다. first가 second보다 먼저 호출되고, second가 third보다 먼저 호 출되도록 보장하는 메커니즘을 설계하라.

```
public class Foo {
  public Foo() { ... }
  public void first() { ... }
  public void second() { ... }
  public void third() { ... }
}
```

second( )를 호출하기 전에 first( )가 종료될 것이라는 걸 어떻게 확신할 수 있을까?

(15.5) 순서대로 호출

```
public class FooBad {
  public int pauseTime = 1000;
  public ReentrantLock lock1, lock2;

  public FooBad() {
    try {
      lock1 = new ReentrantLock();
      lock2 = new ReentrantLock();
      lock1.lock();
      lock2.lock();
    } catch (...) { ... }
  }

public void first() {
  try {
    ...
    lock1.unlock(); // first()가 끝났다고 표시
  } catch (...) { ... }
}


public void second() {
  try {
    lock1.lock(); // first()가 끝날 때까지 대기
    lock1.unlock();
    ...
    lock2.unlock(); // second()가 끝났다고 표시
  } catch (...) { ... }
}

public void third() {
  try {
    lock2.lock(); // third()가 끝날 때까지 대기
    lock2.unlock();
    ...
  } catch (...) { ... }
  }
}
```

(15.5) 순서대로 호출

```
public class FooBad {
  public int pauseTime = 1000;
  public ReentrantLock lock1, lock2;

  public FooBad() {
    try {
      lock1 = new ReentrantLock();
      lock2 = new ReentrantLock();
      lock1.lock();
      lock2.lock();
    } catch (...) { ... }
  }

public void first() {
  try {
    ...
    lock1.unlock(); // first()가 끝났다고 표시
  } catch (...) { ... }
}


public void second() {
  try {
    lock1.lock(); // first()가 끝날 때까지 대기
    lock1.unlock();
    ...
    lock2.unlock(); // second()가 끝났다고 표시
  } catch (...) { ... }
}

public void third() {
  try {
    lock2.lock(); // third()가 끝날 때까지 대기
    lock2.unlock();
    ...
  } catch (...) { ... }
  }
}
```

⇒ 예외(exception) 발생!

```
public class Foo {
  public Semaphore sem1, sem2;

  public Foo() {
    try {
      sem1 = new Semaphore(1);
      sem2 = new Semaphore(1);

      sem1.acquire();
      sem2.acquire();
    } catch (...) { ... }
  }

  public void first() {
    try {
    ...
    sem1.release();
    } catch (...) { ... }
  }

  public void second() {
    try {
      sem1.acquire();
      sem1.release();
      ...
      sem2.release();
    } catch (...) { ... }
  }

  public void third() {
    try {
      sem2.acquire();
      sem2.release();
      ...
    } catch (...) { ... }
  }
}
```

(15.7) FizzBuzz

1부터 n까지 출 력하는 프로그램을 작성해야 한다. 3으로 나누어 떨어질 땐 “Fizz”를, 5로 나누어 떨어질 땐 “Buzz”를, 3과 5 둘 다로 나누어 떨어지면 "FizzBuzz"를 출력

(조건) 다중 스레드(multi-thread)를 이용, 4개의 스레드 사용

- 스레드 1: 3으로 나누어 떨어지는지 확인한 뒤 나누어 떨어지면 "Fizz" 를 출력
- 스레드 2: 5로 나누어 떨어지는지 확인한 뒤 나누 어 떨어지면 "Buzz"를 출력
- 스레드 3: 세 번째 스레드는 동시에 3과 5로 나 누어 떨어지는지 확인한 뒤 나누어 떨어지면 "FizzBuzz"를 출력
- 스레드 4: 그 외의 숫자를 출력

(15.7) FizzBuzz

단일 스레드

```
void fizzbuzz(int n) {
  for (int i = 1; i <= n; i++) {
    if (i % 3 0 && i % 5 == 0) {
      System.out.println("FizzBuzz");
    } else if (i % 3 == 0) {
      System.out.println("Fizz");
    } else if (i % 5 == 0) {
      System.out.println("Buzz");
    } else {
      System.out.println(i);
    }
  }
}
```

(15.7) FizzBuzz

다중 스레드

FizzBuzz Thread	Fizz Thread

if i div by 3 && 5
  print FizzBuzz
  increment i
repeat until i > n
	if i div by only 3
	  print Fizz
	  increment i
	repeat until i > n

Buzz Thread	Number Thread

if i div by only 5
  print Buzz
  increment i
repeat until i > n
	if i not div by 3 or 5
	  print i
	  increment i
	repeat until i > n

```
while (true) {
  if (current > max) {
    return;
  }
if (/* 나누어 떨어지는지 확인 */) {
  System.out.println(/* 출력 */);
  current++;
}
```

(15.7) FizzBuzz

다중 스레드

	FizzBuzz	Fizz	Buzz	Number
current % 3 == 0	true	true	false	false
current % 5 == 0	true	false	true	false
to print	FizzBuzz	Fizz	Buzz	current

(sol) https://qna.programmers.co.kr/code_runners

END