Process Synchronization

태그	OS
주차	5주차

목차

1. 프로세스 동기화(Synchronization)

- 프로세스의 실행 순서를 제어하고 동시에 접근할 수 없는 하나의 프로세스만 접근하게 하여 데이터의 일관성을 유지하는 과정

✅ **경쟁 상태(Race Condition)**

- 여러 프로세스들이 동시에 자원에 접근하는 상황에서 명령어 실행 순서에 따라 결과값이 달라질 수 있는 상황
- 자원의 일관성이 깨질 수 있음

✅ **임계 구역(Critical Section)**

- 공유 자원에 접근하는 코드 중 동시에 실행되면 문제가 발생하는 코드 구역
- 임계 구역 문제를 해결하기 위해 OS는 다음 조건을 만족해야 함
  1. **상호 배제(Mutual Exclusion)**
     - 임계 구역에는 단 하나의 프로세스만 진입 가능
  2. **유한 대기(Bounded Waiting)**
     - 모든 프로세스는 유한 시간 내에 임계 구역에 진입해야 함
  3. **진행(Progress)**
     - 임계 구역에 어떤 프로세스도 진입하지 않았다면, 진압하는 프로세스의 진행을 방해해선 안됨

1-1. 뮤텍스 락(Mutex Lock)

- 상호 배제를 위한 동기화 매커니즘

1) 구성

- `lock`: 임계 구역 진입 가능 여부를 표시하는 공유 자원
- `acquire()`: lock을 획득하는 함수
- `release()`: lock을 반환하는 함수

2) 동작

1. 공유 자원인 lock이 false일 때에만 임계 구역에 진입이 가능

2. 한 프로세스라도 임계구역에 진입하면 lock = true가 되어 다른 프로세스가 임계구역에 진입하지 못함
3. 임계구역의 코드가 종료되면 lock = false로 하여 다른 프로세스가 진입 가능하도록 함

3) 단점

- lock이 걸린 동안에는 임계 구역이 열릴 때 까지 임계 구역을 반복적으로 확인하는 **바쁜 대기(busy waiting)** 발생

4) 예시

```
global bool lock = false;

acquire() {
  while (lock == true); //busy waiting
  lock = true;
}

release() {
  lock = false;
}
```

```
acquire();
//임계 구역
release();
```

5) C언어 구현

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

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
volatile int sum = 0;

void *producer(void *arg) {
  pthread_mutex_lock(&lock);   // 뮤텍스 락 잠금
  for(int i = 0; i < 100000; i++) {
      sum++;                      // 임계 구역
```

```
    }
  pthread_mutex_unlock(&lock); // 뮤텍스 락 열림

  pthread_exit(NULL);
}

void *consumer(void *arg) {
  pthread_mutex_lock(&lock);   // 뮤텍스 락 잠금
  for(int i = 0; i < 100000; i++) {
      sum--;                    // 임계 구역
  }
  pthread_mutex_unlock(&lock); // 뮤텍스 락 열림

  pthread_exit(NULL);

}

int main() {
  pthread_t producer_tid, consumer_tid;

  pthread_create(&producer_tid, NULL, producer, NULL);
  pthread_create(&consumer_tid, NULL, consumer, NULL);

  pthread_join(producer_tid, NULL);
  pthread_join(consumer_tid, NULL);

  printf("sum: %d\n", sum);
  return 0;
}
```

1-2. 세마포어(Semaphore)

- 프로세스의 임계 구역 진입 여부 제어
- 뮤텍스와 달리 여러 프로세스가 임계 구역에 들어가도록 할 수 있음

1) 구성

- `S`: 진입 가능한 프로세스의 개수를 나타내는 전역변수

- `wait()`: 임계 구역에 진입하면서 `S`를 1 감소하는 함수 (acquire과 유사)
- `signal()`: 임계 구역에서 나오면서 `S`를 1 증가하는 함수 (release와 유사)

2) 동작

1. 프로세스가 `wait()`함수를 호출할 때 마다 임계 구역에 진입할 수 있는 프로세스의 수인 `S`를 확인하고, `S`가 0이라면 `S`가 증가할 때 까지 반복하여 확인
2. `S`가 0보다 크다면 `S`를 1 감소시키고 임계구역에 진입
3. 임계구역의 코드가 실행된 후 signal 함수를 호출해 `S`를 1 증가

3) 단점

- `wait()` 함수가 뮤텍스와 같이 바쁜 대기 상태이므로 CPU 성능을 저하시킬 수 있음

4) 해결

- `wait()` 함수에서 임계구역에 진입하지 못하면 프로세스를 대기상태로 만들고, signal 함수가 호출될 때 대기상태의 프로세스를 준비상태로 만들어 줌
- 대기상태의 프로세스는 CPU 자원을 사용하지 않으므로 작업 사이클의 낭비를 막을 수 있음

5) 예시

```
global int S = 2;

wait() {
  S--;
  if (S < 0) { // 현재 프로세스의 PCB를 대기 큐에 삽입
    sleep();   // 현재 프로세스를 대기 상태로 전환
  }
}

signal() {
  S++;
  if(S <= 0) { // 대기 큐에 있는 p 프로세스의 PCB 제거
    wakeup(p); // p 프로세스를 준비 상태로 전
  }
}
```

```
wait();
//임계 구역
signal();
```

✅ Mutex와 Semaphore는 보통 프로그래밍 언어에서 지원되는 경우가 많아 직접 구현할 일이 많지 않음!

1-3. 모니터(Monitor)

1) 개요

- 공유자원과 인터페이스를 하나로 묶어 세마포어의 불편함을 해소한 동기화 도구
  - 세마포어는 임계구역 앞뒤로 `wait()`과 `signal()` 함수를 직접 호출해야 하므로 불편
- 하나의 프로세스 내 다른 스레드 간의 동기화에 사용
- 라이브러리 혹은 프레임워크가 제공 (하이레벨 언어로 제공)

2) 구성 요소

- **Mutex Lock**: critical section에서 mutual exclusion을 보장하는 장치
  - mutex lock을 획득해야 critical section에 진입 가능
  - mutex lock을 획득하지 못한 스레드는 큐에 들어가 대기(waiting) 상태로 전환
- **Conditaion Variable**
  - waiting queue를 가짐
  - 조건이 충족되길 기다리는 스레드들이 대기상태로 머무는 곳
  - 주요 동작(operation)
    - wait: 스레드가 자기 자신을 대기열에 넣고 대기상태로 전환 (Java의 `wait()`)
    - signal: 대기열에서 대기중인 스레드를 하나 깨움 (Java의 `notify()`)
    - broadcast: 대기열에서 대기중인 스레드를 모두 깨움 (Java의 `notifyAll()`)
- **2개의 큐** (배타동기 + 조건동기)
  - Entry Queue: critical section에 진입을 기다리는 큐

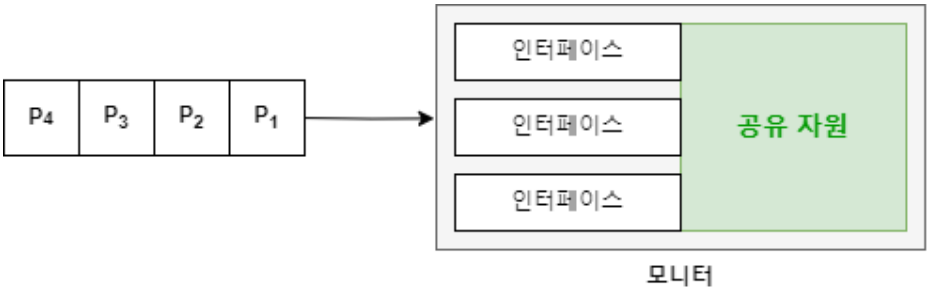
- Waiting Queue: 조건이 충족되길 기다리는 큐

3) 발생 가능한 문제

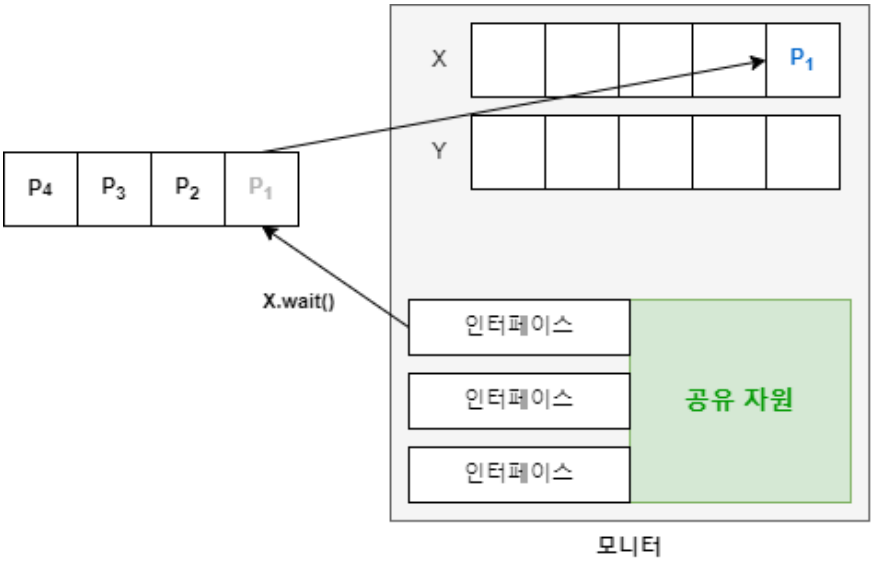
- **Bounded Producer**: 버퍼가 가득 차서 producer가 이를 계속 확인해야 하는 문제 ⇒ *오버플로우 방지해야 함*
- **Consumer Problem**: 버퍼가 비어서 consumer가 이를 계속 확인해야 하는 문제 ⇒ *언더플로우 방지해야 함*

4) 동작

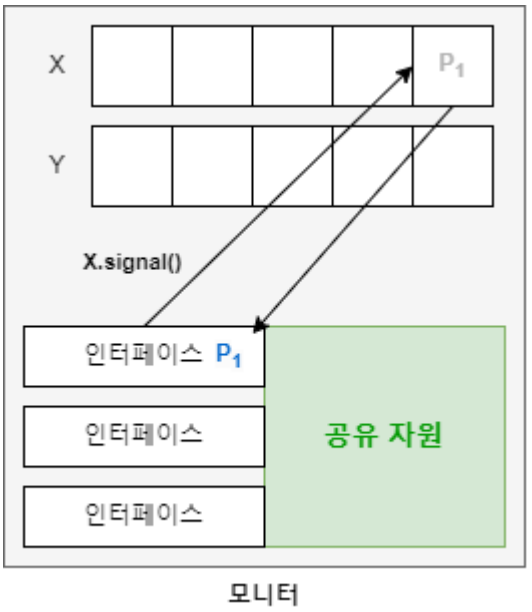
- 공유자원에 접근하고자 하는 프로세스는 특정 인터페이스를 위한 큐에 쌓이며, 하나의 인터페이스를 통해서만 공유자원에 접근할 수 있게 함으로써 상호배제 동기화가 가능



- 실행 순서를 제어하기 위해 모니터 내부의 조건 변수 사용
  - 모니터 내부에는 조건 변수에 대한 큐 존재
  - 공유자원과 인터페이스 조건 변수를 캡슐화하여 추상화된 데이터 형(Abstract Data Type, ADT)을 형성
- 모니터는 `wait()` 함수를 통해 프로세스를 대기상태로 전환하여 조건 변수 큐에 삽입



- 조건 변수 상태에서 대기중인 프로세스는 `signal()` 함수를 통해 실행 상태로 전환되어 인터페이스를 통해 공유 자원에 접근할 수 있음



☝🏻 **자바에서는 모든 객체가 모니터를 가지고 있음!**

임계 구역은 `synchronized` 키워드를 통해 설정할 수 있으며, 모니터를 가지는 스레드는 해당 객체에 락을 걸어 다른 스레드가 해당 객체에 접근할 수 없게 할 수 있다.

1-4. 자바에서의 Monitor

1) wait()

- synchronized 블록 내에서만 호출 가능
- 객체에 대한 lock을 release (정확히는, 제어권을 넘겨주는 것)
- 만약 wait()을 호출하는 스레드가 lock을 소유하고 있지 않다면 에러 발생
- 모니터 락을 쥐고 있는 스레드가 (모니터락 객체에 대해) wait() 메소드를 호출하면, lock을 release하고 다른 스레드가 해당 lock을 취함

2) notify(), notifyAll()

- `notify()`: 모니터락 객체에 `wait()`를 건 모든 스레드 중 임의로 하나를 깨움
- `notifyAll()`: `wait()` 걸린 모든 스레드를 깨움

+) Monitor Java 예제

```
public class Main {

  public static void main(String[] args) throws Exception {
    BankAccount bankAccount = new BankAccount();
    Producer producer = new Producer(bankAccount, 10000);
    Consumer consumer = new Consumer(bankAccount, 10000);

    Thread t1 = new Thread(producer);
    Thread t2 = new Thread(consumer);

    t1.start()
    t2.start();

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

    System.out.println("--------------");
    System.out.println("잔액: " + bankAccount.money);
  }

}
```

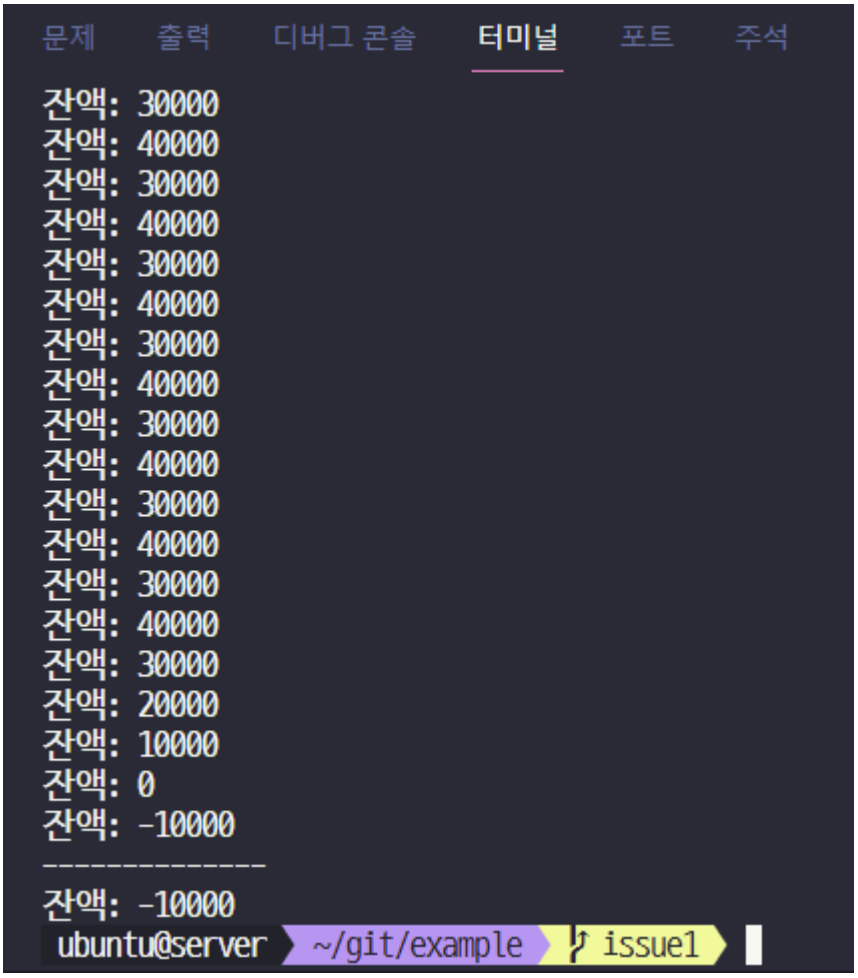
```
public class Consumer implem

  BankAccount bankAccount;
  int amount;

  public Consumer(BankAccoun
    this.bankAccount = bankA
    this.amount = amount;
  }

  @Override
  public void run() {
    try {
      for (int i = 0; i < 10
        Thread.sleep(1);
        bankAccount.withdraw
        System.out.println("
      }
    } catch (InterruptedExce
  }

}
```

```
public class Producer implem

  BankAccount bankAccount;
  int amount;

  public Producer(BankAccoun
    this.bankAccount = bankA
    this.amount = amount;
  }

  @Override
  public void run() {
    try {
      for (int i = 0; i < 10
        Thread.sleep(1);
        bankAccount.deposit(
        System.out.println("
      }
    } catch (InterruptedExce
  }

}
```

1. `synchronized` 키워드가 없으면 여러 프로세스가 동시에 money 변수를 변경하여 값이 이상해짐 (동기화가 되지 않아 최종 잔액이 0원이 아니게 됨)

```
public class BankAccount {

  int money = 0;

  public void deposit(int amount) {
    money += amount;
  }

  public void withdraw(int amount) {
    money -= amount;
    }
```

```
}
```

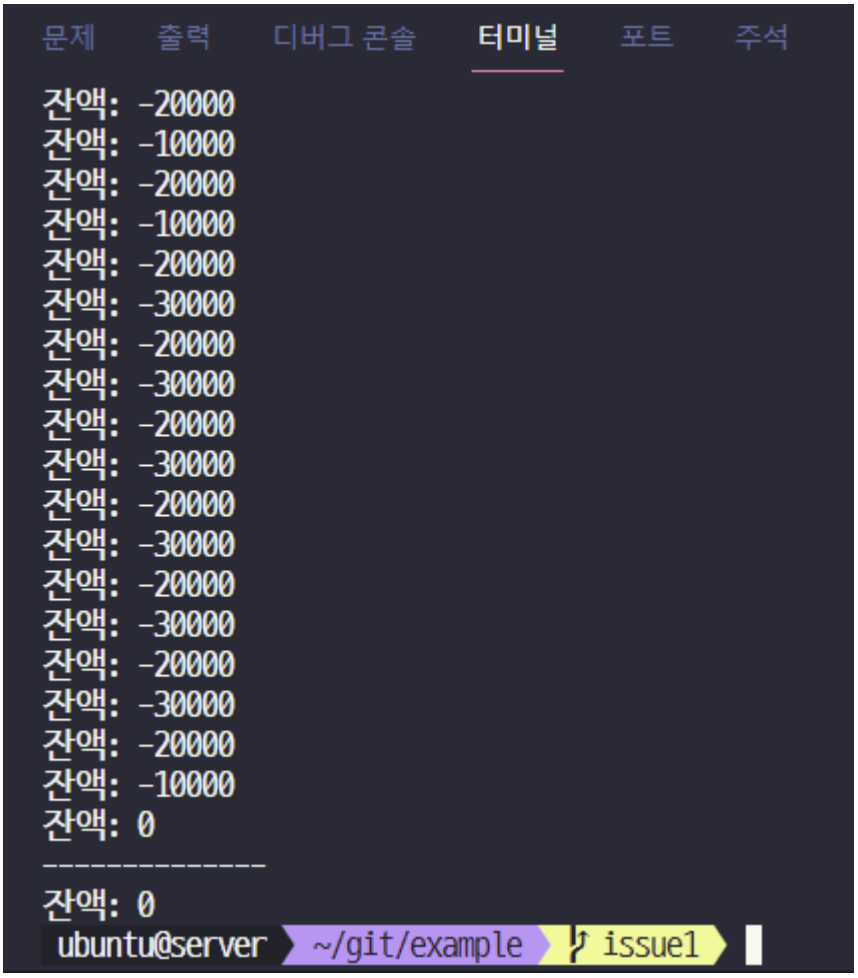


실행 결과

2. `synchronized` 키워드를 사용하더라도 중간 과정에 잔액이 마이너스가 됨

```
public class BankAccount {

  int money = 0;

  synchronized public void deposit(int amount) {
    money += amount;
  }

  synchronized public void withdraw(int amount) {
    money -= amount;
    }
```

```
}
```

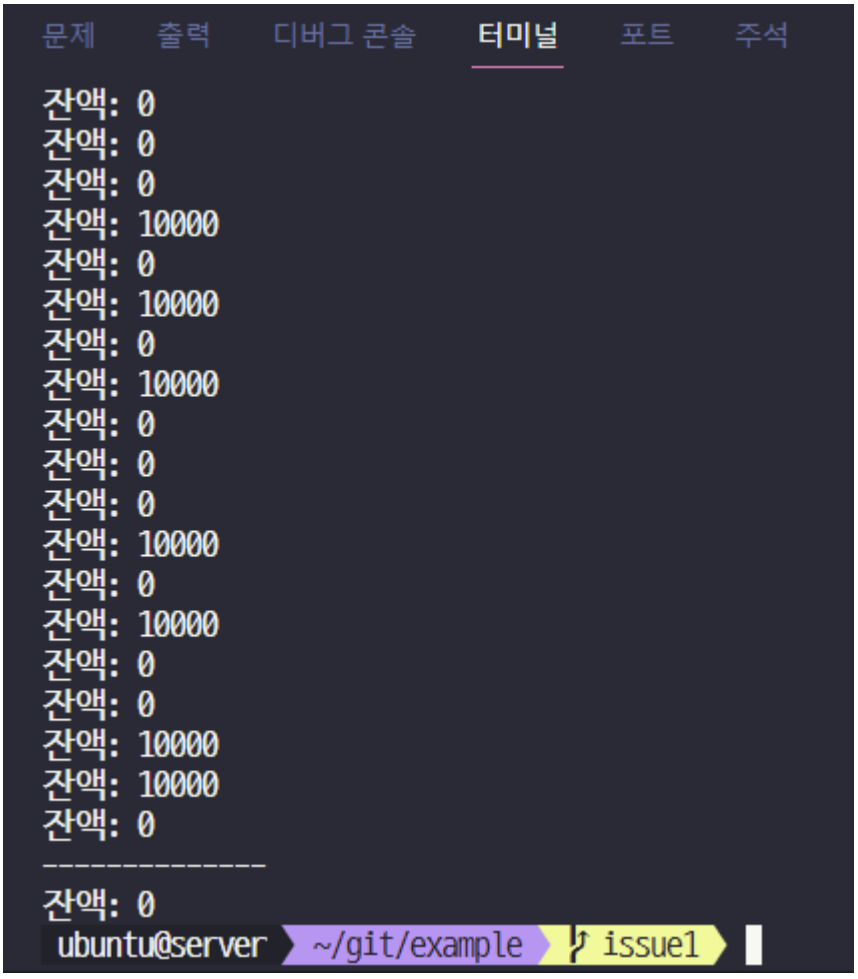


실행 결과

3. 실행 순서 제어를 위해 `wait()`, `notifyAll()` 사용

```
public class BankAccount {

  int money = 0;

  synchronized public void deposit(int amount) {
    money += amount;
    notifyAll();
  }

  synchronized public void withdraw(int amount) {
        if(money == 0){
```

```
                wait();
            }
        money -= amount;
        notifyAll();
        }

}
```



실행 결과