UI/UX 전문가 과정비트캠프

2021년 01월 25일 19회차

장 해 솔

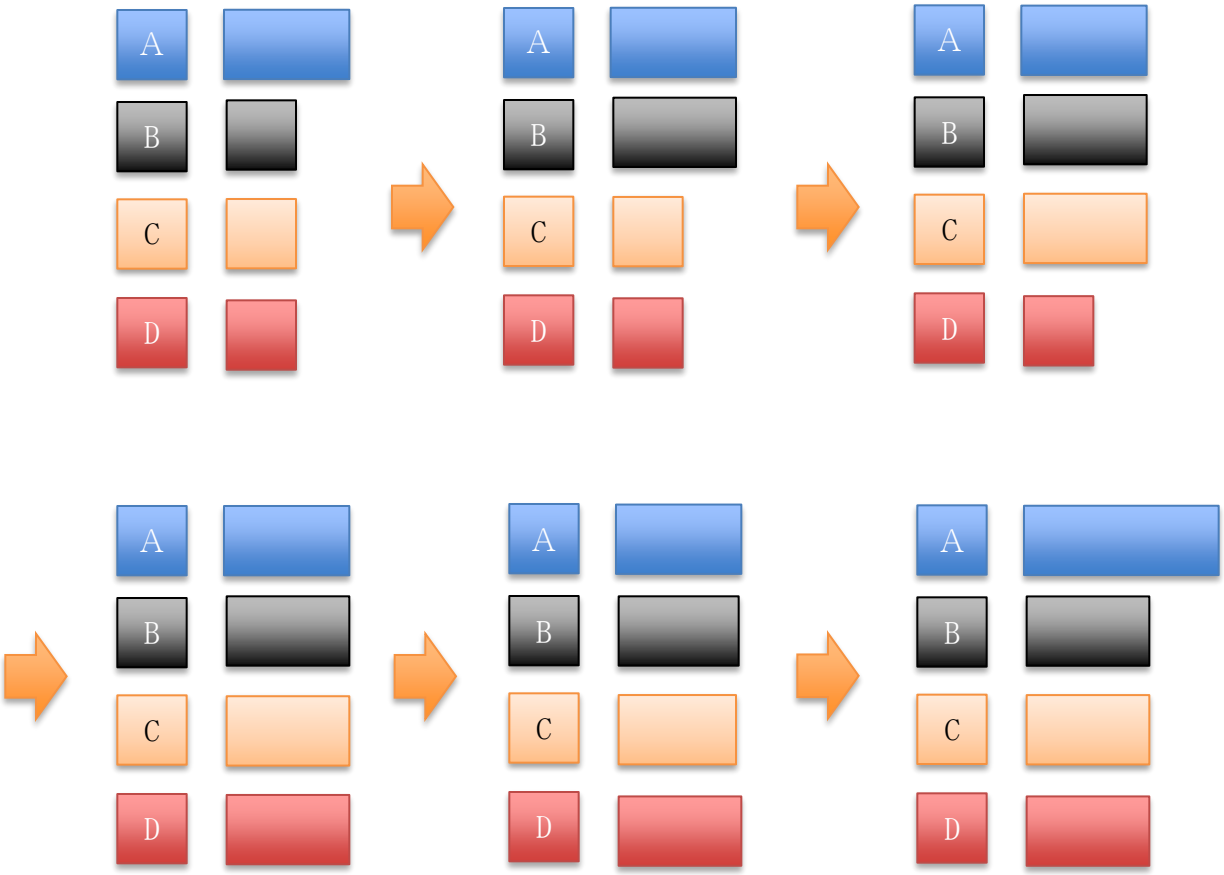
이메일 : wkdgothf@gmail.com

github : https://github.com/legossol

목차

내용

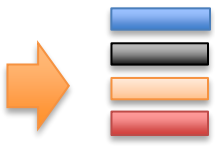
1.Context Switching



Context Switching은 위의 그림처럼 하나의 프로세스가 여러개의 작업이 있으면 이것을 하나 조금하고 다른거 하나 조금하고 하는 식으로 여러 작업을 돌아가며 일부분씩 진행하는 것을 말함.

(위 작업이 컴퓨터상에서는 엄청 빠르게 진행되니까 사람은 동시에 진행된다고 생각하게 됨)

그리고 작업들 돌아가는 갈래들이 쓰레드이다.



2.Critical Saction(임계구역)

각각의 프로세스(또는 쓰레드)는 "임계구역"이라고 불리는 code영역을 가지는데, 여기서 프로세스(또는 쓰레드)는 공통변수를 변경하거나, DB 테이블을 최신화하거나 파일작성 등의 작업이 가능하다.
문제는 이렇게 공유하는 영역에 두 개 이상의 프로세스 또는 스레드가 접근해서 작업을 하게되면 의도하지 않는 상황들이 발생할 수 있게 된다. 이는 굉장히 치명적이기 때문에, 이런 공통변수 구역을 "임계구역"이라고 하는 것이다.

이 문제를 해결하기 위해서는 몇 가지 조건이 존재하는데, 이는 아래와 같이 3가지이다.

1. 상호 배타(Mutual Exclusion)
2. 진행(Progress)
3. 유한 대기(Bounded waiting)

상호 배타(Mutual Exclusion)는 이 임계구역에 꼭 하나의 프로세스(또는 쓰레드)만 진입해야함을 뜻한다.
진행(Progress)는 이 임계구역에 들어갈 프로세스들의 순서가 유한 시간 내에 정해져야 한다는 것이다.
유한 대기(Bounded waiting)은 어떤 프로세스(또는 쓰레드)라도 기다리고 있으면 유한한 시간 내에 임계구역에 들어갈 수 있어야 한다는 것이다.

이를 쉽게 이해하기위해 강사님이 말씀하신 화장실예제를 만들어 보았다.

```
class pee{
    //해솔, 진욱, 한나세사람이 술마시다 화장실간다.
   //변기는 하나

    {//해솔 쓰레드
        진욱, 한나 술마시기
        if ( 오줌마렵다 )
        {
            RestRoom.Lock();
            소변보기;//CriticalSaction
            RestRoom.Unlock();
        }
    }
    {//진욱 쓰레드
        해솔, 한나 술마시기
        if ( 오줌마렵다 )
```

```
        {
            RestRoom.Lock();
            소변보기;//CriticalSaction
            RestRoom.Unlock();
        }
    }
    {//한나쓰레드
        진욱, 해솔 술마시기
        if ( 오줌마렵다 )
        {
            RestRoom.Lock();
            소변보기;//CriticalSaction
            RestRoom.Unlock();
        }
    }
```

3.Mutex(Synchronized)

프로세스는 컴퓨터에 주어진 자원을 분할해서 쓴다.

쓰레드는 프로세스마다 주어진 전체 자원을 공유

위의 개념을 다시 잡고 위의 술마시는 예제를 다시 보면 여러 술집에는 다 화장실이 있지만(전체자원) **우리가 술을 마시는 곳에는 하나의 변기만(분할된 자원)** 있는 화장실이다

그리고 우리는 같은 화장실을 다 같이 쓰지만 한번에 한명만 가야한다.

근데 만약 술집에 다른 손님들도 있다 이상황에서는 우리끼리는 누가 화장실을 갔는지 정보 공유가 가능하지만, 다른 손님들은 우리 맴버중 누가 화장실을 갔는지 모르고 화장실을 들어가려 할 수도 있다(2개의 쓰레드가 부디치는 상황(에러)).

이때 사용하는 것이Mutex(Synchronized) 이다.

이정도만 이해하고 나중에 설명에서 자세히 다뤄보겠다.

4.위의 개념을 이해하기위한 간단한 예제 만들기

1)우린 분식집 요리사이다.

2)김밥과, 라면, 돈까스같은 일식을 만든다.

-그럼 김밥을 만드는 곳, 라면을 만드는 곳, 일식을 만드는 곳 이 **각각의 Saction들이 프로세스** 이다.

-만약 일식을 만드는 섹션에서 돈까스를 만들고 샐러드도 만드는 작업을 한다면 이 **각 작업이 쓰레드**이다.(첫번째1번 목차에있던 Context Switching)

3)라면 주문이 들어왔다 화구는 4개밖에없는데 10개를 끓여야한다.

-주의 : 한 냄비에 라면을 두개 이상 넣어도 안되고,

한 화구에 냄비를 두개 이상 탑 쌓아도 안된다.(Critical Saction)

```
import java.util.EmptyStackException;
                    //Runnable이라는 인터페이스를 사용
                        //(해당 클래스에는 필수적으로 run함수 필요)
class RamenCook implements Runnable{
                //라면의 수
    private int ramenCount;
                //버너의 상태 (버너 = 쓰레드)
    private String[] burners = {"_","_","_","_"};
                //끓일 라면의 개수를 입력받는다.
    public RamenCook(int count){
        ramenCount = count;
    }
    //Thread에서 진행할 작업을 여기에 정의한다.
    @Override
    public void run(){
        //끓일 라면의 수가 0이 될때까지
        while(ramenCount > 0) {

            synchronized (this) {
                //쓰레드가 동시에 라면 두개를 가져가는 경우
                  // 카운트는 하나만 줄어들게된다.
                //이를 방지하기 위해 synchronized를 만듦.
                  //(한번에 한 쓰레드만 손댈 수 있다.)
                ramenCount--;
                System.out.println(
                        Thread.currentThread().getName()
                                + ": " + ramenCount + "개 남음");

            }
```

```
                //버너 4개중 빈 것을 찾는
            for (int i = 0; i < burners.length; i++) {
                if (!burners[i].equals("_")) continue;
                //라면을 두개 못 가져가게 하는게 아닌
                //한 쓰레드가 두개의 버너를 차지하지 않게 하기 위해 사용
                synchronized (this) {
                    //if(burners[i].equals("stop")){
                    //해당 쓰레드(버너)의 이름으로 버너를 차지하고 라면을 얹음.
                    burners[i] = Thread.currentThread().getName();
                    System.out.println(
                            "            "
                                    + Thread.currentThread().getName()
                                    + ": [" + (i + 1) + "]번 버너 on");
                    //버너의 상태 출력
                    showBurners();
                    //}
                }
                //라면 끓는 시간을 줌 굳이 필요한 건 아님 그냥 편하게 보기 위해
                try {
                    Thread.sleep(2000);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                //라면을 다 끓이고 해당 버너를 끄고 비우기 위해
                synchronized (this) {
                    burners[i] = "_";
                    System.out.println(
                            "                                "
                                    + Thread.currentThread().getName()
                                    + ": [" + (i + 1) + "]번 버너 OFF");
                    showBurners();
                }
                break;
            }
            //다음 라면을 끓이기까지 정지해야 하는 시간을 랜덤
            //쓰레드 정지(버너 잠시 불끔).
            try {
                Thread.sleep(Math.round(1000 * Math.random()));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
            //실행될 때마다 버너의 상태를 알려줌
private void showBurners(){
```

```
        String stringToPrint
                = "                                                    ";
        for(int i = 0; i < burners.length; i++){
            stringToPrint += ("  " + burners[i]);
        }
        System.out.println(stringToPrint);
    }
}
public class selfThreadSyncTest {
    public static void main(String[] args) {
        try {
            RamenCook ramenCook = new RamenCook(10);
            new Thread(ramenCook, "A").start();//쓰레드 A
            new Thread(ramenCook, "B").start();//쓰레드 B
            new Thread(ramenCook, "C").start();//쓰레드 C
            new Thread(ramenCook, "D").start();//쓰레드 D
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

위의 코드를 실행!

```
A: 9개 남음
            A: [1]번 버너 on
                                                A  _  _  _
D: 8개 남음
            D: [2]번 버너 on
                                                A  D  _  _
C: 7개 남음
            C: [3]번 버너 on
                                                A  D  C  _
B: 6개 남음
            B: [4]번 버너 on
                                                A  D  C  B
                            B: [4]번 버너 OFF
                                                A  D  C  _
                            C: [3]번 버너 OFF
                                                A  D  _  _
                            D: [2]번 버너 OFF
                                                A  _  _  _
                            A: [1]번 버너 OFF
```

```
                                                                _  _  _  _
C: 5개 남음
            C: [1]번 버너 on
                                                                C  _  _  _
B: 4개 남음
            B: [2]번 버너 on
                                                                C  B  _  _
A: 3개 남음
            A: [3]번 버너 on
                                                                C  B  A  _
D: 2개 남음
            D: [4]번 버너 on
                                                                C  B  A  D
                                    C: [1]번 버너 OFF
                                                                _  B  A  D
                                    B: [2]번 버너 OFF
                                                                _  _  A  D
                                    A: [3]번 버너 OFF
                                                                _  _  _  D
                                    D: [4]번 버너 OFF
                                                                _  _  _  _
```

```
C: 1개 남음
            C: [1]번 버너 on
                                                             C  _  _  _
A: 0개 남음
            A: [2]번 버너 on
                                                             C  A  _  _
                                    C: [1]번 버너 OFF
                                                             _  A  _  _
                                    A: [2]번 버너 OFF
                                                             _  _  _  _

Process finished with exit code 0
```

5.Semaphore

Semaphore의 사전적의미는 수기 신호 혹은 신호로 알려주다.

그리고 사전적의미와 같은 역할을 한다. 기차의 교차로 같다고 생각하면 된다. 1번 기차와 2번기차가 있는데 교차로에서 서로 같이 지나가게 되면 당연히 사고난다.

그래서 수신기호를 줘서 누구 먼저 들어와 누구 먼저 들어와 이렇게 만들어주는 역할이다.

5-1.Semaphore와 Syncronized의 차이

Syncronized와 Semaphore은 진입을 막고 여는 것에 있어서 기능은 같지만 다르다.

공항을 예제로 생각해보자.

Syncronized.

어떤 공항은 시설이 낙후되어 입국심사하는 창구가 하나다. 그렇다면 아래그림과 같을 것이다. (한 명, 한 명, 입장하고 심사하는 모든 과정이 쓰레드)

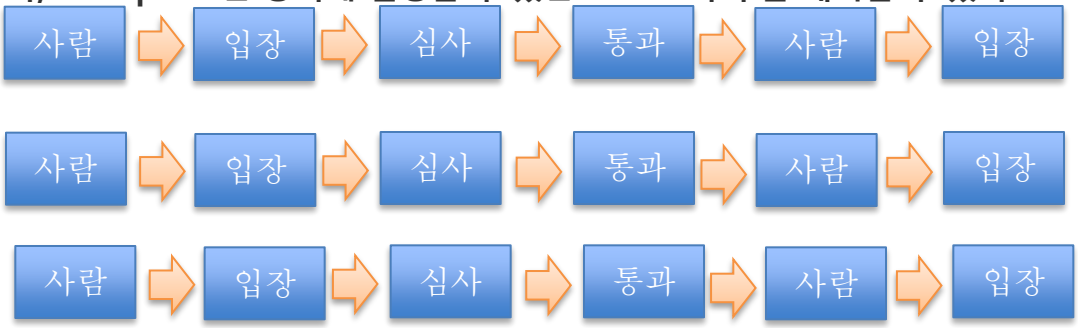
한 명을 심사하며 다른 사람이 못 들어가고 그 사람이 나가고 문이열려야지만 다른 사람이 들어간다.

즉, 하나의 Thread만 수행 가능.

Semaphore.

같은 공항이라도 이 구역은 입국심사하는 곳이 하나지만 통로는 하나지만 양쪽으로 방향이 있어서 여러 명이서 심사를 볼 수 있다

즉, Semaphore는 동시에 실행할 수 있는 Thread의 수를 제어할 수 있다



6.Memory Hierarchy

제대로 된 설명을 찾기 힘들어 유튜브를 통해 조사하였다.
일단은 메모리에는 여러 종류가 있는데 속도와 메모리가 반비례하다는 점이 특징이다.
속도가 빠르면서 메모리가 많은 것은 현재 존재하지 않는다.

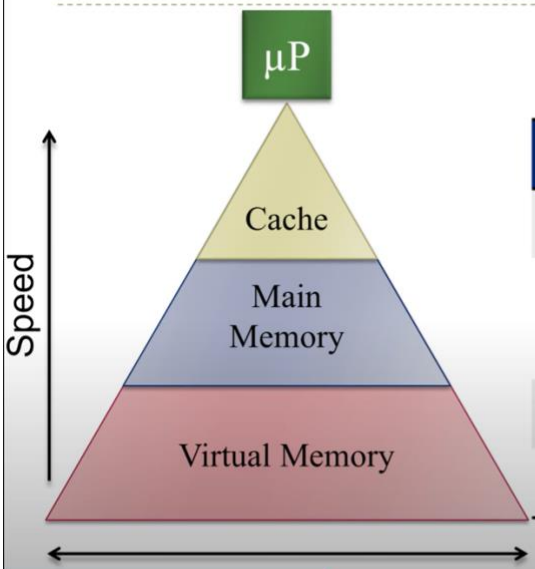
Memory System Challenge

- Ideal memory is fast, cheap, and large
- No technology provides all three
- Solution: Use hierarchy of memories to approximate

	Speed	Bit Area	Data lifetime	Cost/bit
SRAM	Fast (Larger arrays are slower)	Large (≥6T)	Data stable while powered	Expensive
DRAM	Slow (Multiple steps)	Small (1T & capacitor)	Must be periodically refreshed	Inexpensive
Disk	Very Slow (mechanical)	Very small	Non volatile	Very cheap

속도를 상위계층으로 두어 순차적으로 정렬한 아래의 그림과 같은 시스템 or계층 구성을 memory hierarchy라고한다.(사진엔없지만Cache보다 위인 래지스터도 있음)

Solution: The Memory Hierarchy



	Speed	Bit Area	Cost/GB
SRAM	~1ns	Large (≥6T)	~$10k
DRAM	~10-50ns	Small (1T & capacitor)	~$10
Disk (SSD)	~10^5ns	Very small	$1
Disk (HDD)	~10^7ns	Very small	$0.1

Numbers roughly from 2012