운영체제 토론일지

소속 조 : 빈센조		날짜 : 2021.12.06
구성원		
학번	성명	담당교수 확인
21660021 21960025 21960027	김승현 신가영 양혜교	

토론 주제

철학자들의 만찬 해결 알고리즘

토론 내용

- JAVA 모니터를 사용한 철학자들의 만착 해결 알고리즘을 알아보자

- 철학자들의 만찬

- 운영체제의 전통적 동기화 문제
  1. 생산자-소비자 문제
  2. Reader-Writer 문제
  3. 식사하는 철학자 문제

- 식사하는 철학자 문제
  - 철학자 다섯이서 원형 식탁에 둘러앉아 생각에 빠지다가, 배고플 땐 밥을 먹는다. 그들의 양쪽엔 각각 젓가락 하나씩 놓여있고, 밥을 먹으려 할 때는 다음의 과정을 따른다.

  1. 왼쪽 젓가락부터 집어든다. 다른 철학자가 이미 젓가락을 쓰고 있다면 그가 내려놓을 때까지 생각하며 대기한다.

  2. 왼쪽을 들었으면 오른쪽 젓가락을 든다. 들 수 없다면 1번과 마찬가지로 들 수 있을때까지 생각하며 대기한다.

  3. 두 젓가락을 모두 들었다면 일정 시간동안 식사를 한다.

  4. 식사를 마쳤으면 오른쪽 젓가락을 내려 놓고, 그 다음 왼쪽 젓가락을 내려놓는다.

  5. 다시 생각하다가 배고프면 1번으로 돌아간다.

● 발생 가능한 문제
- ◆ 식사하는 철학자 문제는 교착상태의 대표적인 예제
- ◆ 식사하는 철학자 문제는 데드락 발생의 4가지 필요 조건을 모두 만족하고 있음
- ◆ 모든 철학자가 동시에 왼쪽 젓가락을 집어 든다면, 모든 철학자는 생각에 빠짐. 교착상태(Deadlock) 발생
- ◆ 한번 교착상태에 빠진 철학자들은 계속 고뇌만 하다가 기아현상(Starvation)으로 굶어 죽음

● 교착상태 4가지 필요조건
- ◆ 상호배제(Mutual Exclusion)
  – 젓가락은 한 번에 한 철학자만 사용할 수 있음

- ◆ 보유 및 대기(Hold and Wait)
  – 집어든 젓가락은 계속 든 채로 사용중인 반대쪽 젓가락을 기다림

- ◆ 비선점(No preemption)
  – 이미 누군가 집은 젓가락을 강제로 뺏을 수 없음

- ◆ 환형대기(Circular Wait)
  – 모든 철학자들이 자신의 오른쪽에 앉은 철학자가 젓가락을 놓기를 기다림

● 모니터를 이용한 해결
- • 모니터(Monitors)
  – 고급 언어의 설계 구조물로써 ,개발자의 코드를 상호 배제하게끔 만든 추상화된 데이터 형태, 공유 자원에 접근하기 위한 키 획득과 자원 사용 후 해제를 모두 처리함

- • 좌측의 도식은 전반적인 모니터의 구조를 나타냄

- • ‘entry queue’는 이미 수행 중인 프로세스가 있기에 다른 프로세스는 wait queue에 저장됨.

- • Operation 영역에서는 수행되는 프로세스가 있는데 만약 해당 프로세스가 실행 도중에 조건이 충족되지 않을 시 조건이 만족될 때까지 waiting 상태가 되는데 해당 프로세스가 shared data 영역의 queue로 진입하게 됨.

- 만약 동작했던 프로세스가 조건에 안 맞아서 대기상태가 되었다면 entry queue에서 한 프로세스가 다시 동작하고 해당 프로세스가 종료되면 entry queue에 있는 프로세스로 가는 게 아니라 먼저 shared data에서 대기하고 있는 프로세스에 Signal을 보내서 조건이 충족되지 않아 대기상태에 있는 프로세스를 먼저 확인함. 만약 프로세스가 존재하면 해당 프로세스를 우선적으로 실행

● 알고리즘 구현(Java)

- State Interface

```
package Dining_Philosopher;

public interface STATE {
    int HUNGRY=0, THINKING=1, EATING=2;
}
```

- Global class

```
package Dining_Philosopher;

public class Global {
    static int num;
    static{
        num=5;
    }

    static int Left(int i){
        return (i-1+num)%num;
    }
    static int Right(int i){
        return (i+1)%num;
    }
}
```

- Monitor class

```
package Dining_Philosopher;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class Monitor {
    private final ReentrantLock entLock;
    private final Condition self[];
    private int state[];

    //      num : 철학자가 몇명인지...
    public Monitor(int num){
        entLock=new ReentrantLock();
        self=new Condition[num];
        state=new int[num];

        for(int i=0; i<num; i++){
            self[i]=entLock.newCondition();
            state[i]=STATE.THINKING;
        }
    }

    //      who : 철학자의 ID.
    void go(int who){
        try {
            //      남남.
            pickup(who);
            putdown(who);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    void pickup(int who) throws InterruptedException {
        entLock.lock();
        state[who]=STATE.HUNGRY;
        System.out.println("Philosopher " + who + " is hungry.\n");
        Test(who);

        //      내 양쪽의 포크를 한 번에 집을 수 없었기 때문에
        //      나는 스파게티를 먹을 수 없다. 그러므로 대기 상태로 전환.
        if(state[who]!=STATE.EATING)
            self[who].await();

        entLock.unlock();
    }

    void Test(int who){
        if(state[Global.Left(who)]!=STATE.EATING &&
            state[Global.Right(who)]!=STATE.EATING &&
            state[who]==STATE.HUNGRY){
            state[who]=STATE.EATING;
            System.out.println("Philosopher " + who + " is eating.\n");

            self[who].signal();
        }
    }

    void putdown(int who){
        entLock.lock();
        state[who]=STATE.THINKING;
        System.out.println("Philosopher " + who + " is thinking.\n");

        Test(Global.Left(who));
        Test(Global.Right(who));

        entLock.unlock();
    }
}
```

- Philosopher class

```
package Dining_Philosopher;

public class Philosopher extends Thread{
    Monitor M;
    int ID;
    public Philosopher(Monitor M, int ID){
        this.M=M;
        this.ID=ID;
    }

    public void run(){
        while(true){
            M.go(ID);
        }
    }
}
```

- Main class

```
package Dining_Philosopher;

public class Main {
    public static void main(String args[]){
        // 철학자는 5명으로 가정.
        Monitor M=new Monitor(Global.num);
        Philosopher P[]=new Philosopher[Global.num];

        for(int i=0; i<Global.num; i++){
            P[i]=new Philosopher(M, i);
        }
        for(int i=0; i<Global.num; i++){
            P[i].start();
        }

        try {
            for(int i=0; i<Global.num; i++){
                P[i].join();
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

- 실행 결과

```
Philosopher 2 is thinking.

Philosopher 1 is eating.

Philosopher 2 is hungry.

Philosopher 4 is thinking.

Philosopher 3 is eating.

Philosopher 4 is hungry.

Philosopher 1 is thinking.

Philosopher 0 is eating.

Philosopher 1 is hungry.

Philosopher 3 is thinking.

Philosopher 2 is eating.

Philosopher 3 is hungry.

Philosopher 0 is thinking.
```

발표자료

JAVA 모니터를 사용한 철학자들의 만찬 해결 알고리즘

SW 3B 21660021 김승현
SW 3B 21960025 신가영
SW 3B 21660027 양혜교

목차 A table of contents.

Part 1

철학자들의 만찬

운영체제의 전통적 동기화 문제
1. 생산자-소비자 문제
2. Reader-Writer 문제
3. 식사하는 철학자 문제

식사하는 철학자 문제
: 철학자 다섯이서 원형 식탁에 둘러앉아 생각에 빠지다가, 배고플 땐 밥을 먹는다. 그들의 양쪽엔 각각 젓가락 하나씩 놓여있고, 밥을 먹으려 할 때는 다음의 과정을 따른다.

1. 왼쪽 젓가락부터 집어든다. 다른 철학자가 이미 젓가락을 쓰고 있다면 그가 내려놓을 때까지 생각하며 대기한다.
2. 왼쪽을 들었으면 오른쪽 젓가락을 든다. 들 수 없다면 1번과 마찬가지로 들 수 있을때까지 생각하며 대기한다.
3. 두 젓가락을 모두 들었다면 일정 시간동안 식사를 한다
4. 식사를 마쳤으면 오른쪽 젓가락을 내려 놓고, 그 다음 왼쪽 젓가락을 내려놓는다.
5. 다시 생각하다가 배고프면 1번으로 돌아간다.

Part 1

철학자들의 만찬

발생 가능한 문제
- 식사하는 철학자 문제는 교착상태의 대표적인 예제
- 식사하는 철학자 문제는 데드락 발생의 4가지 필요 조건을 모두 만족하고 있음
- 모든 철학자가 동시에 왼쪽 젓가락을 집어 든다면, 모든 철학자는 생각에 빠짐. 교착상태(Deadlock) 발생
- 한번 교착상태에 빠진 철학자들은 계속 고뇌만 하다가 기아현상(Starvation)으로 굶어 죽음

교착상태 4가지 필요조건
1. 상호배제(Mutual Exclusion)
   - 젓가락은 한 번에 한 철학자만 사용할 수 있음

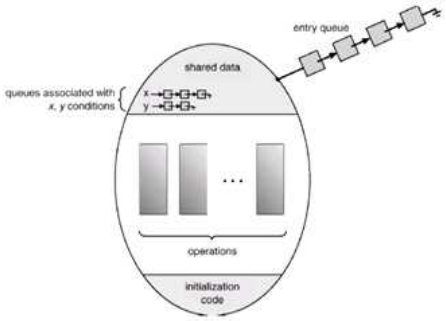
2. 보유 및 대기(Hold and Wait)
   - 집어든 젓가락은 계속 든 채로 사용중인 반대쪽 젓가락을 기다림

3. 비선점(No preemption)
   - 이미 누군가 집은 젓가락을 강제로 뺏을 수 없음

4. 환형대기(Circular Wait)
   - 모든 철학자들이 자신의 오른쪽에 앉은 철학자가 젓가락을 놓기를 기다림

Part 2

모니터를 이용한 해결



모니터(Monitors)
: 고급 언어의 설계 구조물로써 ,개발자의 코드를
상호 배제하게끔 만든 추상화된 데이터 형태
공유 자원에 접근하기 위한 키 획득과 자원 사용 후
해제를 모두 처리함

좌측의 도식은 전반적인 모니터의 구조를 나타냄
'entry queue'는 이미 수행 중인 프로세스가 있기에
다른 프로세스는 wait queue에 저장됨.

Operation 영역에서는 수행되는 프로세스가 있는데
만약 해당 프로세스가 실행 도중에 조건이 충족되지
않을 시 조건이 만족될 때까지 waiting 상태가 되는데
해당 프로세스가 shared data 영역의 queue로
진입하게 됨.

만약 동작했던 프로세스가 조건에 안 맞아서 대기상태가
되었다면 entry queue에서 한 프로세스가 다시 동작하고
해당 프로세스가 종료되면 entry queue에 있는 프로세스로
가는 게 아니라 먼저 shared data에서 대기하고 있는 프로세스
Signal을 보내서 조건이 충족되지 않아 대기상태에 있는
프로세스를 먼저 확인함. 만약 프로세스가 존재하면
해당 프로세스를 우선적으로 실행

Part 3

알고리즘 구현(Java)

- State interface

```java
package Dining_Philosopher;

public interface STATE {
    int HUNGRY=0, THINKING=1, EATING=2;
}
```

- Global class

```java
package Dining_Philosopher;

public class Global {
    static int num;
    static{
        num=5;
    }

    static int Left(int i){
        return (i-1+num)%num;
    }
    static int Right(int i){
        return (i+1)%num;
    }
}
```

Part 3

알고리즘 구현(Java)

- Monitor class

```
package Dining_Philosopher;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class Monitor {
    private final ReentrantLock entLock;
    private final Condition self[];
    private int state[];

    //      num : 철학자가 몇명인지...
    public Monitor(int num){
        entLock=new ReentrantLock();
        self=new Condition[num];
        state=new int[num];

        for(int i=0; i<num; i++){
            self[i]=entLock.newCondition();
            state[i]=STATE.THINKING;
        }
    }

    //      who : 철학자의 ID.
    void go(int who){
        try {
            //      뭐냐.
            pickup(who);
            putdown(who);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

```
    void pickup(int who) throws InterruptedException {
        entLock.lock();
        state[who]=STATE.HUNGRY;
        System.out.println("Philosopher " + who + " is hungry.\n");
        Test(who);

        //      내 양쪽이 포크를 한 번에 집을 수 없었기 때문에
        //      나는 스파게티를 먹을 수 없다. 그러므로 먹기 상태로 전환.
        if(state[who]!=STATE.EATING)
            self[who].await();

        entLock.unlock();
    }

    void Test(int who){
        if(state[Global.Left(who)]!=STATE.EATING &&
            state[Global.Right(who)]!=STATE.EATING &&
            state[who]==STATE.HUNGRY){
            state[who]=STATE.EATING;
            System.out.println("Philosopher " + who + " is eating.\n");

            self[who].signal();
        }
    }

    void putdown(int who){
        entLock.lock();
        state[who]=STATE.THINKING;
        System.out.println("Philosopher " + who + " is thinking.\n");

        Test(Global.Left(who));
        Test(Global.Right(who));

        entLock.unlock();
    }
}
```


Part 3

알고리즘 구현(Java)

- Philosopher class

```
package Dining_Philosopher;

public class Philosopher extends Thread{
    Monitor M;
    int ID;
    public Philosopher(Monitor M, int ID){
        this.M=M;
        this.ID=ID;
    }

    public void run(){
        while(true){
            M.go(ID);
        }
    }
}
```

- Main class

```
package Dining_Philosopher;

public class Main {
    public static void main(String args[]){
        // 철학자는 5명으로 가정.
        Monitor M=new Monitor(Global.num);
        Philosopher P[]=new Philosopher[Global.num];

        for(int i=0; i<Global.num; i++){
            P[i]=new Philosopher(M, i);
        }
        for(int i=0; i<Global.num; i++){
            P[i].start();
        }

        try {
            for(int i=0; i<Global.num; i++){
                P[i].join();
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

- 실행 결과

```
Philosopher 2 is thinking.

Philosopher 1 is eating.

Philosopher 2 is hungry.

Philosopher 4 is thinking.

Philosopher 3 is eating.

Philosopher 4 is hungry.

Philosopher 1 is thinking.

Philosopher 0 is eating.

Philosopher 1 is hungry.

Philosopher 3 is thinking.

Philosopher 2 is eating.

Philosopher 3 is hungry.

Philosopher 0 is thinking.
```

다른 조 발표내용

[박부성 & 구본성 조]

- 교착상태를 일으키는 철학자의 만찬 문제를 교착상태 해결 방법을 통해 문제를 해결하는 것

- 철학자의 만찬 문제
  - 철학자 thread가 작업을 완수하기 위해서는 공유자원 lFork와 rFork를 모두 점유해야 작업을 완수
  - 여기서 모두 하나의 Fork만 가지고 상대방의 포크를 기다리고 있다면 교착상태가 발생
  - 여기서 교착상태를 해결하고 모든 철학자 thread가 작업을 원활하게 완수 하도록 하는 것이 목표

- 스레드의 생성자
  - id와 lFork,rFork를 받아 저장
  - lFork와 rFork는 공유자원이며 한번에 한 스레드만 사용 가능

- run() 메소드
  - 기존의 교착상태 문제 해결법으로 홀수번째와 짝수번째가 같은 젓가락을 집도록 구현
  - 세마포어를 통해 같은 젓가락을 집고 먼저 작업을 수행 할 수 있는 스레드가 작업을 수행
  - while문을 통해 우연히 작업이 완료되는 것을 방지

- eating() 메소드와 digestion() 메소드
  - 철학자가 식사중인가 소화중인가 콘솔창에 띄워 결과를 볼 수 있도록 함

- main() 메소드
  - 세마포어 배열로 Forks를 생성
  - Forks를 생성할 때 Semaphore에 공유자원수를 1로 주어 한번에 접근할 수 있는 스레드의 수를 1개로 제한
  - 철학자 스레드를 num 개수만큼 생성 철학자스레드 들은 같은 Forks 자원을 다르게 저장하고 불러옴
  - for문을 통해 순서대로 철학자 스레드를 실행시킴

[이세영 & 황호현 조]

- 문제점
  - 만약 모든 철학자들이 동시에 자신의 왼쪽 포크를 잡는다면, 철학자들은 자기 오른쪽의 포크가 사용 가능해질 때까지 기다려야 한다. 그러나 모두 포크를 하나씩 잡고 있기때문에 모든 철학자들은 대기한다. 이 상태에서는 철학자들이 아무것도 진행할 수가 없게 되는, 교착(Deadlock)상태가 발생한다.

```
34⊖    public static void main(String[] args) {
35         int count = 5; // 철학자의 수
36
37         Semaphore[] fork = new Semaphore[count]; // 철학자의 수만큼 포크 생성
38         for(int i = 0; i < count; i++) {
39             fork[i] = new Semaphore(1); // 하나의 포크마다 한명씩만 사용할 수 있도록 세마포어 값을 1로 설정
40         }
41         Philosopher1[] phil = new Philosopher1[count]; // 철학자를 count 갯수만큼 생성
42
43         // 특정 철학자의 고유 번호와 자신의 양 옆 포크를 초기화시킴
44         for(int i = 0; i < count; i++) {
45             phil[i] = new Philosopher1(i, fork[i], fork[(i + 1) % count]);
46         }
47
48         for(int i = 0; i < count; i++) {
49             phil[i].start(); // 식사 시작
50         }
51
52     }
```

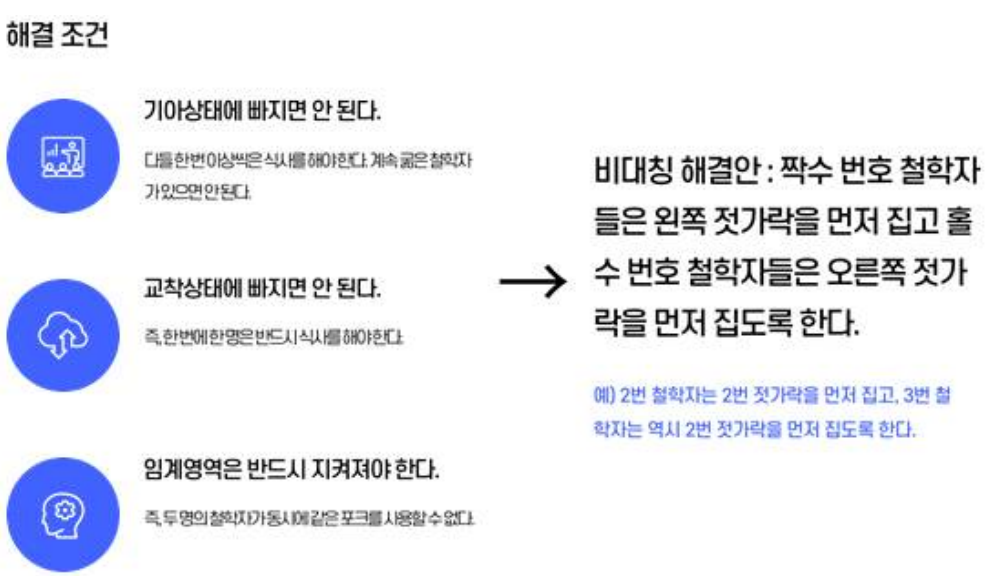
```
public void run() {
    try {
        if(id % 2 == 0) { // 고유번호가 짝수라면 자신의 왼쪽 포크부터 선점함
            left_fork.acquire(); // 왼쪽 포크에 P연산 (세마포어 값을 1만큼 감소시켜 0으로 만든다)
            right_fork.acquire(); // 오른쪽 포크에 P연산
        }
        else { // 고유 번호가 홀수라면 자신의 오른쪽 포크부터 선점함
            right_fork.acquire(); // 오른쪽 포크에 P연산
            left_fork.acquire(); // 왼쪽 포크에 P연산
        }

        System.out.println("Philosopher " + id + " eating"); // 두개의 포크을 들고 식사
        right_fork.release(); // 오른쪽 포크에 V연산 (0이었던 세마포어 값을 1만큼 증가시켜 1로 만든다)
        left_fork.release(); // 왼쪽 포크에 V연산
        System.out.println("Philosopher " + id + " thinking");

    } catch (InterruptedException e) {}
```

[허원석 & 조은새 조]

- 교착상태(Deadlock)의 식사하는 철학자들

- 해결 방법
  - 철학자 스레드의 id는 정수이므로, 이를 이용하여 id가 짝수인 철학자는 왼쪽부터, id가 홀수인 철학자는 오른쪽부터 젓가락을 집어 들게 하면 교착상태가 일어나지 않는다.

[김찬혁 & 송지현 조]



1. 뮤텍스, 상태값을 생성하고 초기화 한다.
2. 포크의 사용 여부를 모두 가능하게 한다.
3. 스레드를 0~4번 순서대로 생성함과 동시에 동작하도록 한다.
4. 짝수 철학자는 왼쪽 포크를 먼저 집고, 홀수 철학자는 오른쪽 포크를 집도록 한다.
5. 철학자가 포크를 집을 때, 먼저 해당 포크에 대한 뮤텍스를 잠궈 다른 철학자가 접근하지 못하게 한다.
6. 만약 집으려는 포크가 이미 다른 철학자 손에(사용 불가능 상태)있다면 포크가 다시 사용 가능 할 때까지 대기 하도록 한다. 대기 할 때는 포크번호에 대한 상태값을 가진다.
7. 대기 중에 해당 번호의 포크가 사용가능 해지면(signal을 받으면) 다른 철학자가 사용하지 못하도록 사용 불가능 하도록 하고, 뮤텍스를 풀어준다.
8. 철학자는 양손에 포크를 들어야 식사를 시작할 수 있다. 식사를 마치고 나면 왼쪽 철학자는 오른쪽 포크를 먼저 놓고, 오른쪽 철학자는 왼쪽 포크를 먼저 놓는다.
9. 모든 철학자가 식사를 마치면 각 스레드는 동작을 멈추고, 마지막에 뮤텍스와 상태값을 파괴하여 메모리를 반납한다.
10. 각 철학자는 5번 식사를 하도록 한다.

[박동환 조]

- 교착상태
  - 자원을 자유롭게 할당한 결과 자원부족 상태
- 기아상태
  - 교착상태를 예방하기 위해 무한히 기다림
    -작업이 결코 사용할 수 없고 계속 기다려야 하는 자원을 할당할 때 발생하는 결과

```
do {
wait (chopstick[1]);
wait (chopstick[(i+1) % 5]);

//eat

signal(chopstick[i]);
signal(chopstick[(i+1) % 5]);

// think

} while (true);
```

```
monitor DP {
   enum {THINKING; HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING) self [i].wait;
   }

   void test(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if((state[(i+4)%5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i+1)%5] != EATING) && {
            state[i] = EATING;
            self[i].signal();
         }
   }

   initailization_coid() {
      for(int i = 0; i<5; i++)
      state[i] = THINKING;
   }
}
```