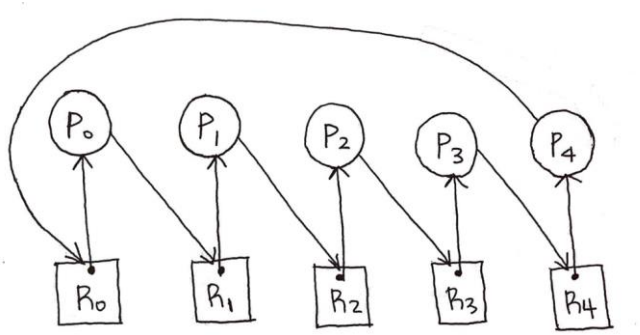
Operating System Project #3 Report

Dining Philosophers Problem

12191656 이채연

1) Dining Philosophers problem의 Resource allocation graph 및 발생 원인 분석

(1) Dining Philosophers problem의 Resource allocation graph

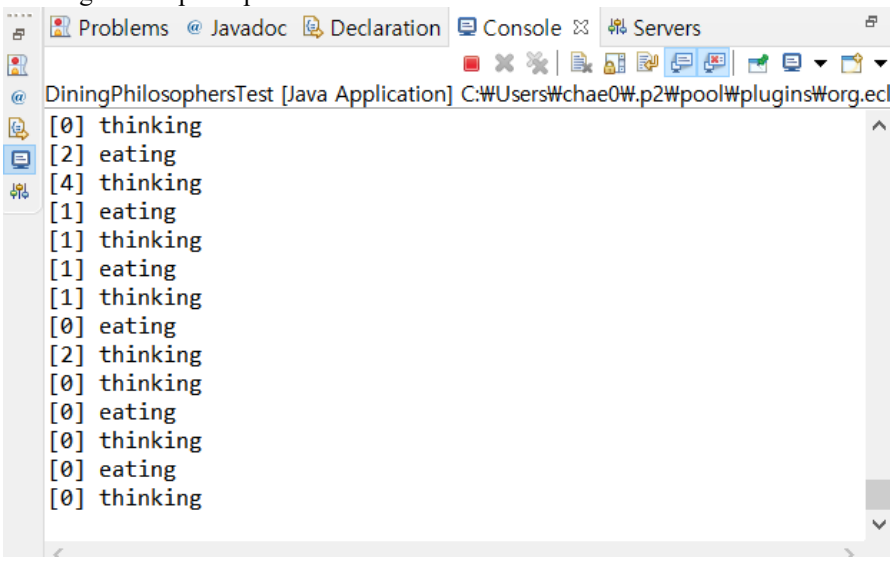


(2) Dining Philosophers problem의 발생 원인

```java
import java.util.concurrent.Semaphore;

class Philosopher extends Thread {
    int id;
    Semaphore lfork, rfork;

    Philosopher(int id, Semaphore lfork, Semaphore rfork) {
        this.id = id;
        this.lfork = lfork;
        this.rfork = rfork;
    }

    public void run() {
        try {
            while (true) {
                lfork.acquire();
                rfork.acquire();
                eating();
                lfork.release();
                rfork.release();
                thinking();
            }
        } catch (InterruptedException e) {
        }
    }

    void eating() {
        System.out.println("[" + id + "] eating");
    }

    void thinking() {
        System.out.println("[" + id + "] thinking");
    }
}

```

```java
class DiningPhilosophersTest {
    static final int num = 5;

    public static void main(String[] args) {
        int i;
        /* forks */
        Semaphore[] fork = new Semaphore[num];
        for (i = 0; i < num; i++)
            fork[i] = new Semaphore(1);
        /* philosophers */
        Philosopher[] phil = new Philosopher[num];
        for (i = 0; i < num; i++)
            phil[i] = new Philosopher(i, fork[i], fork[(i + 1) % num]);
        for (i = 0; i < num; i++)
            phil[i].start();
    }
}
```

<Dining Philosophers problem 코드>



<Dining Philosophers problem 실행결과>

위 코드가 deadlock이 발생하는 이유는 deadlock이 발생하기 위한 4가지 필요조건을 모두 만족하고 있기 때문이다.

Deadlock이 발생하기 위한 4가지 필요조건은 다음과 같다.

1. 상호배타(Mutual Exclusion) : fork는 한번에 한 철학자만 사용할 수 있다.
2. 보유 및 대기(Hold and Wait) : 집어든 fork를 계속 들은 채로 다른 철학자가 사용중인 반대쪽 fork를 기다린다.
3. 비선점(No Preemption) : 이미 다른 철학자가 집어든 fork를 강제로 뺏을 수 없다.
4. 환형대기(Circular Wait) : 모든 철학자들이 자신의 오른쪽에 앉은 철학자가 fork를 놓기를 기다린다. (1)의 Resource allocation graph를 보면, cycle이 형성 되어있음을 알 수 있다.

2) Dining Philosophers problem 해결을 위한 3가지 방법

(1) DeadLock Prevention – critical section에 들어갈 수 있는 process를 4개로 지정하여 cycle이 일어나지 않도록 하여 Circular Wait 조건을 제거한다.

```java
import java.util.concurrent.Semaphore;

class Philosopher extends Thread {
    int id;
    Semaphore lfork, rfork, room;/* 추가 */

    Philosopher(int id, Semaphore lfork, Semaphore rfork, Semaphore room) {
        this.id = id;
        this.lfork = lfork;
        this.rfork = rfork;
        this.room = room;/* 추가 */
    }

    public void run() {
        try {
            while (true) {
//critical section에 들어갈 수 있는 process를 최대 4개로 설정
                room.acquire();/* 추가 */
                lfork.acquire();
                rfork.acquire();
                eating();
                lfork.release();
                rfork.release();
                thinking();
                room.release();/* 추가 */
            }
        } catch (InterruptedException e) {
        }
    }

    void eating() {
        System.out.println("[" + id + "] eating");
    }

    void thinking() {
        System.out.println("[" + id + "] thinking");
    }
}
class DiningPhilosophersTest {
    static final int num = 5;

    public static void main(String[] args) {
        int i;
        /* forks */
        Semaphore[] fork = new Semaphore[num];
        for (i = 0; i < num; i++)
            fork[i] = new Semaphore(1);
        Semaphore room = new Semaphore(4);/* 추가 */
        /* philosophers */
        Philosopher[] phil = new Philosopher[num];
        for (i = 0; i < num; i++)
            phil[i] = new Philosopher(i, fork[i], fork[(i + 1) % num], room);
        for (i = 0; i < num; i++)
            phil[i].start();
    }
}
```

(2) **DeadLock Prevention** – 왼쪽 fork와 오른쪽 fork를 둘 다 얻을때까지 아무도 못들어오도록 원자적 처리를 하여 **hold and wait** 조건을 제거한다.

```
1  import java.util.concurrent.Semaphore;
2
3  class Philosopher extends Thread {
4      int id;
5      Semaphore lfork, rfork, once;/* 추가 */
6
7      Philosopher(int id, Semaphore lfork, Semaphore rfork, Semaphore once) {
8          this.id = id;
9          this.lfork = lfork;
10         this.rfork = rfork;
11         this.once = once;/* 추가 */
12     }
13
14     public void run() {
15         try {
16             while (true) {
17 //왼쪽 fork와 오른쪽 fork를 얻을 때까지 아무도 못들어오도록 원자적 처리
18                 once.acquire();/* 추가 */
19                 lfork.acquire();
20                 rfork.acquire();
21                 once.release();/* 추가 */
22                 eating();
23                 lfork.release();
24                 rfork.release();
25                 thinking();
26             }
27         } catch (InterruptedException e) {
28         }
29     }
30
31     void eating() {
32         System.out.println("[" + id + "] eating");
33     }
34
35     void thinking() {
36         System.out.println("[" + id + "] thinking");
37     }
38 }
40 class DiningPhilosophersTest {
41     static final int num = 5;
42
43     public static void main(String[] args) {
44         int i;
45         /* forks */
46         Semaphore[] fork = new Semaphore[num];
47         for (i = 0; i < num; i++)
48             fork[i] = new Semaphore(1);
49         Semaphore once = new Semaphore(1);/* 추가 */
50         /* philosophers */
51         Philosopher[] phil = new Philosopher[num];
52         for (i = 0; i < num; i++)
53             phil[i] = new Philosopher(i, fork[i], fork[(i + 1) % num], once);
54         for (i = 0; i < num; i++)
55             phil[i].start();
56     }
57 }
```

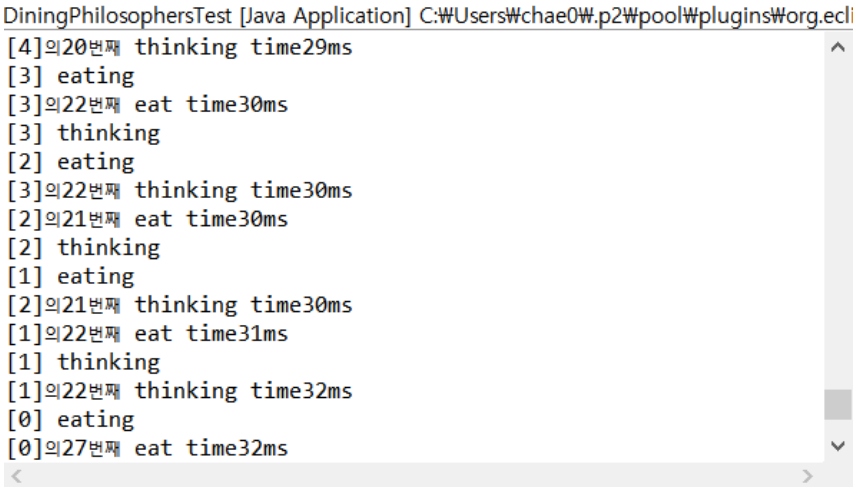
(3) DeadLock Prevention – fork를 요청할 때 증가하는 방향으로만 요청할 수 있도록 하여 circular wait 조건을 제거한다. ID가 4일때만 4번째 fork 다음에 0번째 fork를 요청하여 감소하는 방향이므로 ID가 4일 때 lfork와 rfork를 바꾸는 코드를 추가로 넣어준다.

```
1  import java.util.concurrent.Semaphore;
2
3  class Philosopher extends Thread {
4      int id;
5      Semaphore lfork, rfork;
6
7      Philosopher(int id, Semaphore lfork, Semaphore rfork) {
8          this.id = id;
9          this.lfork = lfork;
10         this.rfork = rfork;
11     }
12
13     public void run() {
14         try {
15             while (true) {
16 /*resource가 증가하는 방향으로만 request할 수 있도록 하기 위해
17 ID가 4 일때는 lfork와 rfork를 바꾼다.*/
18                 if (id < 4) {
19                     lfork.acquire();
20                     rfork.acquire();
21                 } else {
22                     rfork.acquire(); // r0
23                     lfork.acquire(); // r4
24                 }
25                 eating();
26                 lfork.release();
27                 rfork.release();
28                 thinking();
29             }
30         } catch (InterruptedException e) {
31         }
32     }
33
34     void eating() {
35         System.out.println("[" + id + "] eating");
36     }
37
38     void thinking() {
39         System.out.println("[" + id + "] thinking");
40     }
41 }
43 class DiningPhilosophersTest {
44     static final int num = 5;
45
46     public static void main(String[] args) {
47         int i;
48         /* forks */
49         Semaphore[] fork = new Semaphore[num];
50         for (i = 0; i < num; i++)
51             fork[i] = new Semaphore(1);
52         /* philosophers */
53         Philosopher[] phil = new Philosopher[num];
54         for (i = 0; i < num; i++)
55             phil[i] = new Philosopher(i, fork[i], fork[(i + 1) % num]);
56         for (i = 0; i < num; i++)
57             phil[i].start();
58     }
59 }
```

3) 해결책들에 대한 설명 및 간단한 비교평가 결과

```
class Philosopher extends Thread {
    int id;
    int eatNum, thinkNum;
    Semaphore lfork, rfork;
    long sx;
    long ex;

    Philosopher(int id, Semaphore lfork, Semaphore rfork) {
        this.id = id;
        this.lfork = lfork;
        this.rfork = rfork;
        this.eatNum = 0;
        this.thinkNum = 0;
    }
    public void run() {
        try {
            sx = System.currentTimeMillis();
            while (true) {
                lfork.acquire();
                rfork.acquire();
                eating();
                lfork.release();
                rfork.release();
                thinking();
            }
        } catch (InterruptedException e) {
        }
    }

    void eating() {
        System.out.println("[" + id + "] eating");
        eatNum++;
        ex = System.currentTimeMillis();
        System.out.println("[" + id + "]의" + eatNum +"번째 eat time" + (ex-sx) + "ms");
    }

    void thinking() {
        System.out.println("[" + id + "] thinking");
        thinkNum++;
        ex = System.currentTimeMillis();
        System.out.println("[" + id + "]의" + thinkNum +"번째 thinking time" + (ex-sx) + "ms");
}

class DiningPhilosophersTest {
    static final int num = 5;

    public static void main(String[] args) {
        int i;
        /* forks */
        Semaphore[] fork = new Semaphore[num];
        for (i = 0; i < num; i++)
            fork[i] = new Semaphore(1);
        /* philosophers */
        Philosopher[] phil = new Philosopher[num];
        for (i = 0; i < num; i++)
            phil[i] = new Philosopher(i, fork[i], fork[(i + 1) % num]);
        for (i = 0; i < num; i++) {
            phil[i].start();

        }
    }
}
```

이런식으로 코드를 추가하여 각 해결책들을 비교평가 해 보았다. 우선 기본 코드에 대해서는 결과가 다음과 같다.



(1) **critical section**에 들어갈 수 있는 **process**를 **4**개로 지정하여 **cycle**이 일어나지 않도록 하여 **Circular Wait** 조건을 제거한다. 이렇게 하면 **deadlock**은 발생하지 않지만 **multiprocessing**의 정도가 낮아져 성능이 떨어질 수 있다. 또한 **semaphore**가 더 추가되었으므로 오버헤드가 증가하여 속도가 약간 느려질 수 있다. 하지만 코드 실행 결과 속도에 큰 차이는 보이지 않았다.

(2) 왼쪽**fork**를 가지고 있으면서 오른쪽 **fork**를 요청하는 것을 방지하기 위해 왼쪽 **fork**와 오른쪽 **fork**를 둘 다 얻을 때까지 아무도 못 들어오도록 원자적 처리를 하여 **hold and wait** 조건을 제거한다. 이렇게 하는 방법 또한 **deadlock**은 발생하지 않지만 **multiprocessing**의 정도가 낮아지며 오버헤드가 증가하여 성능이 약간 떨어질 수 있다. 하지만 코드 실행 결과 속도에 큰 차이는 보이지 않았다.

(3) **fork**를 요청할 때 증가하는 방향으로만 요청할 수 있도록 하여 **circular wait** 조건을 제거한다. **ID**가 **4**일때만 **4**번째 **fork** 다음에 **0**번째 **fork**를 요청하여 감소하는 방향이므로 **ID**가 **4**일 때 **lfork**와 **rfork**를 바꾸는 코드를 추가로 넣어준다. 하지만 코드 실행 결과 속도에 큰 차이는 보이지 않았다.

4) 결론 : 요약 결과 및 느낀점

세 가지 해결방안은 미세한 속도 차이가 발생할 것이지만, 코드 상으로 실행 속도를 비교해 봤을 때 눈으로 보이는 뚜렷한 차이는 나타나지 않았다.
이번 과제를 통해 deadlock에 대해 깊게 이해할 수 있어서 좋은 경험이 되었다. 나중에 현장에 나갔을 때 deadlock을 활용해야 하는 상황이 생기면 이번 과제에서 배운 것을 바탕으로 잘 관리해보고 싶다.