# 운영체제 토론일지

| 소속 조 : 빈센조 | | 날짜 : 2021.11.29 |
|---|---|---|
| 구성원 | | |
| 학번 | 성명 | 담당교수 확인 |
| 21660021<br>21960025<br>21960027 | 김승현<br>신가영<br>양혜교 | |

## 토론 주제

데드락(교착상태) 해결 알고리즘

## 토론 내용

● 데드락(교착상태) 해결 알고리즘에 대해 조사해보자

● 데드락(Deadlock)

- 운영체제에서 데드락(교착상태)이란, 시스템 자원에 대한 요구가 뒤엉킨 상태를 말함.
- 즉, 둘 이상의 프로세스가 다른 프로세스가 점유하고 있는 자원을 서로 기다릴 때 무한 대기에 빠지는 상황

● 데드락(Deadlock)의 발생조건

교착 상태는 한 시스템 내에서 다음 네 가지 조건이 동시에 성립할 때 발생한다.

1) 상호 배제(Mutual Exclusion)

: 자원은 한 번에 한 프로세스만이 사용할 수 있어야 한다.

2) 점유 대기(Hold and wait)
: 자원은 한 번에 한 프로세스만이 사용할 수 있어야 한다.

3) 비선점(No preemption)
: 다른 프로세스에 할당된 자원은 사용이 끝날 때까지 강제로 빼앗을 수 없어야 한다.

4) 순환 대기(Circular wait)
: 프로세스의 집합{P0,P1, …, Pn)에서 P0은 P1이 점유한 자원을 대기하고 P1은 P2가 점유한 자원을 대기하고 ….. Pn−1은 Pn이 점유한 자원을 대기하며 Pn은 P0가 점유한 자원을 요구해야 한다.

● 데드락(Deadlock) 처리의 종류

(1) 교착 상태 예방(Prevention) 및 회피(Avoidance)

<예방(Prevention)법>
: 교착 상태 발생 조건 중 하나를 제거함으로써 해결하는 방법
1) 상호 배제 부정
2) 점유 대기 부정
3) 비선점 부정
4) 순환 대기 부정

<회피(Avoidance)법> : 교착 상태가 발생하면 피하는 방법
- 은행원 알고리즘 : 다익스트라가 제안한 방법으로, 은행에서 모든 고객의 요구가 충족되도록 현금을 할당하는 데서 유래한 방법.
프로세스가 자원을 요구할 때 시스템은 자원을 할당한 후에도 안정 상태로 남아있는지 사전에 검사하여 교착 상태를 회피하는 방법.

(2) 교착 상태 탐지 및 회복
: 교착 상태가 되도록 허용한 다음에 회복시키는 방법
<교착 상태 탐지(Detection)>
- 자원 할당 그래프를 통해 교착 상태를 탐지할 수 있다.

<교착 상태로부터 회복(Recovery)>
- 교착 상태를 일으킨 프로세스를 종료하거나, 할당된 자원을 해제함으로써 회복

● 은행원 알고리즘
<교착상태 회피>
: 프로세스의 자원 사용에 대한 사전 정보를 활용하여 교착상태가 발생하지 않는 상태에 머물도록 하는 방법

<프로세스의 상태 영역>
- 안전 상태 : 교착 상태를 회피하면서 각 프로세스에게 그들의 최대 요구량까지 빠짐없이 자원을 할당할 수 있는 상태
  + 안전 순서열이 존재
=> 순서 있는 프로세스의 집합<p1, p2, ..., pn>
각 pi에 대해 pi가 추가로 요구할 수 있는 자원 소요량이 현재 가용 상태이거나 현재 가용인 자원에 대해 pi(단, j<i)에 할당된 자원가지 포함하여 할당 가능한 경우

- 불안전 상태 : 교착 상태, 안전 순서열이 존재하지 않음
  교착상태는 불안전 상태(할당 과정에 따라 교착 상태가 될 수도 있는 상태)에서 발생

<은행원 알고리즘 필요 조건>
- Available[n] : 자원 n의 사용 가능 개수
- Max[m,n] : 프로세스 m의 자원 n의 최대 요구 개수
- Allocation[m,n] : 프로세스 m에 할당된 자원 n의 개수
- Need[m,n] : 프로세스 m이 추가적으로 필요로 하는 자원 n의 개수
자원 할당 알고리즘 : 자원을 할당할지 안할지 여부를 결정하는 알고리즘
안전 알고리즘 :위의 자료 구조를 이용해 안전 상태를 검사하는 알고리즘

● 자원할당 알고리즘
1. 프로세스가 요구한 자원의 개수가 필요로하는 개수보다 많은지 아닌지를 검사한다.
    Request >= Need

만약 프로세스가 필요로 하는 개수보다 운영체제에 요구한 자원의 개수가 더 많으면 에러가 있다는 의미이므로 대기시킨다. 아니라면 2번으로 넘어간다.

2. 요구한 자원의 개수가 사용 가능한 자원의 개수보다 많은지 검사한다.
Request <= Available 이상이 없다면 3번으로 넘어간다.

3. 프로세스에 요구한 자원을 할당했다고 가정하고 다음과 같이 자료 구조를 수정 후 안전 알고리즘을 통해 안전 여부를 판단한다.
Available = Available − Request
Allocation = Allocation + Request
Need = Need − Request
만약 3번에서 불안전 상태로 판정이 날 경우 대기상태로 되돌린다.
자원 할당 알고리즘에서 3번까지 간다면 안전 알고리즘을 수행한다.

● 안전 알고리즘
1. 임시 배열 변수 Finish , Work 에 초기화를 한다.
모든 프로세스의 Finish = False
Work = Available

2. 조건을 검사해 FInish 가 False이고 프로세스가 필요한 자원수보다 Work 가 크거나 같은 경우를 찾는다.
(Finish == false) && (Need <= Work)
조건을 만족한다면 3번으로 이동. 없다면 4번으로 이동한다.

3. 해당 프로세스의 FInish를 True로 변환하고 Work에 Allocation을(사용중인 자원 수) 더한다.
Finish = Ture;
Work = Work + Allocation
수행 후 2번으로 돌아간다.

4. 모든 프로세스의 Finish를 검사한다. 하나라도 False 가 있으면 불안전 상태이므로 할당하지 않고 대기한다.
전부 True면 안전상태이므로 자원을 할당해준다.

**발표자료**

# 교착상태 회피
# 은행원 알고리즘

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



## 목차 A table of contents.



Part 1

# 데드락(Deadlock, 교착상태)

데드락(Deadlock)
- 운영체제에서 데드락(교착상태)이란, 시스템 자원에 대한 요구가 뒤엉킨 상태를 말함.
- 즉, 둘 이상의 프로세스가 다른 프로세스가 점유하고 있는 자원을 서로 기다릴 때 무한 대기에 빠지는 상황

데드락(Deadlock)의 발생조건
: 교착 상태는 한 시스템 내에서 다음 네 가지 조건이 동시에 성립할 때 발생한다.
1) 상호 배제(Mutual Exclusion)
   - 자원은 한 번에 한 프로세스만이 사용할 수 있어야 한다.

2) 점유 대기(Hold and wait)
   - 자원은 한 번에 한 프로세스만이 사용할 수 있어야 한다.

3) 비선점(No preemption)
   - 다른 프로세스에 할당된 자원은 사용이 끝날 때까지 강제로 빼앗을 수 없어야 한다.

4) 순환 대기(Circular wait)
   - 프로세스의 집합{P0,P1, ..., Pn}에서 P0은 P1이 점유한 자원을 대기하고 P1은 P2가 점유한 자원을 대기하고 ..... Pn-1은 Pn이 점유한 자원을 대기하며 Pn은 P0가 점유한 자원을 요구해야 한다.

Part 1

# 데드락(Deadlock, 교착상태)

데드락(Deadlock) 처리의 종류

(1) 교착 상태 예방(Prevention) 및 회피(Avoidance)
   <예방(Prevention)법> : 교착 상태 발생 조건 중 하나를 제거함으로써 해결하는 방법
      1) 상호 배제 부정
      2) 점유 대기 부정
      3) 비선점 부정
      4) 순환 대기 부정
   <회피(Avoidance)법> : 교착 상태가 발생하면 피하는 방법
      - 은행원 알고리즘 : 다익스트라가 제안한 방법으로, 은행에서 모든 고객의 요구가 충족되도록 현금을 할당하는 데서 유래한 방법. 프로세스가 자원을 요구할 때 시스템은 자원을 할당한 후에도 안정 상태로 남아있는지 사전에 검사하여 교착 상태를 회피하는 방법.

(2) 교착 상태 탐지 및 회복 : 교착 상태가 되도록 허용한 다음에 회복시키는 방법
   <교착 상태 탐지(Detection)>
      - 자원 할당 그래프를 통해 교착 상태를 탐지할 수 있다.
   <교착 상태로부터 회복(Recovery)>
      - 교착 상태를 일으킨 프로세스를 종료하거나, 할당된 자원을 해제함으로써 회복

Part 2

# 은행원 알고리즘

**교착상태 회피**
: 프로세스의 자원 사용에 대한 사전 정보를 활용하여 교착상태가 발생하지 않는 상태에 머물도록 하는 방법

사전 정보는 현재 할당된 자원, 가용상태의 자원, 프로세스들의 최대 요구량

**프로세스의 상태 영역**

- 안전 상태 : 교착 상태를 회피하면서 각 프로세스에게 그들의 최대 요구량까지 빠짐없이 자원을 할당할 수 있는 상태
  + 안전 순서열이 존재
    - 순서 있는 프로세스의 집합<p1, p2, ..., pn>
    - 각 pi에 대해 pi가 추가로 요구할 수 있는 자원 소요량이 현재 가용 상태이거나
      현재 가용인 자원에 대해 pi(단, j<i)에 할당된 자원가지 포함하여 할당 가능한 경우

- 불안전 상태 : 교착 상태, 안전 순서열이 존재하지 않음
  - 교착상태는 불안전 상태(할당 과정에 따라 교착 상태가 될 수도 있는 상태)에서 발생

Part 2

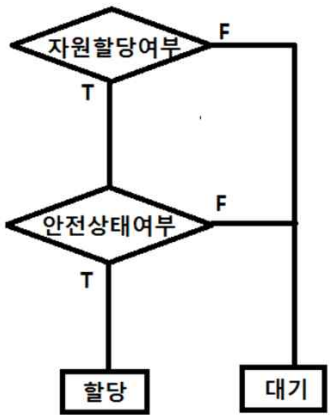
# 은행원 알고리즘

**은행원 알고리즘 필요 조건**

- Available[n] : 자원 n의 사용 가능 개수
- Max[m,n] : 프로세스 m의 자원 n의 최대 요구 개수
- Allocation[m,n] : 프로세스 m에 할당된 자원 n의 개수
- Need[m,n] : 프로세스 m이 추가적으로 필요로 하는 자원 n의 개수

자원 할당 알고리즘 : 자원을 할당할지 안할지 여부를 결정하는 알고리즘

안전 알고리즘 : 위의 자료 구조를 이용해 안전 상태를 검사하는 알고리즘



Part 3

# 알고리즘 동작 과정

자원 할당 알고리즘

1. 프로세스가 요구한 자원의 개수가 필요로하는 개수보다 많은지 아닌지를 검사한다.
   Request >= Need
   만약 프로세스가 필요로 하는 개수보다 운영체제에 요구한 자원의 개수가 더 많으면
   에러가 있다는 의미이므로 대기 시킨다. 아니라면 2번으로 넘어간다.

2. 요구한 자원의 개수가 사용 가능한 자원의 개수보다 많은지 검사한다.
   Request <= Available  이상이 없다면 3번으로 넘어간다.

3. 프로세스에 요구한 자원을 할당했다고 가정하고 다음과 같이 자료 구조를 수정 후
   안전 알고리즘을 통해 안전 여부를 판단한다.
   Available = Available - Request
   Allocation = Allocation + Request
   Need = Need - Request

만약 3번에서 불안전 상태로 판정이 날 경우 대기상태로 되돌린다.
자원 할당 알고리즘에서 3번까지 간다면 안전 알고리즘을 수행한다.

Part 3

# 알고리즘 동작 과정

안전 알고리즘

1. 임시 배열 변수 Finish , Work 에 초기화를 한다.
   모든 프로세스의 Finish = False
   Work = Available

2. 조건을 검사해 FInish 가 False이고 프로세스가 필요한 자원수보다 Work 가 크거나 같은 경우를 찾는다.
   (Finish == false) && (Need <= Work)
   조건을 만족한다면 3번으로 이동. 없다면 4번으로 이동한다.

3. 해당 프로세스의 FInish를 True로 변환하고 Work에 Allocation을(사용중인 자원 수) 더한다.
   Finish = Ture;
   Work = Work + Allocation
   수행 후 2번으로 돌아간다.

4. 모든 프로세스의 Finish를 검사한다. 하나라도 False 가 있으면 불안전 상태이므로 할당하지 않고 대기한다.
   전부 True면 안전상태이므로 자원을 할당해준다.

Part 3

# 알고리즘 동작 과정

o 은행원 – 안전 도식

**은행원 알고리즘**

① $REQ_i \le NEED_i$가 거짓이면 오류
② $REQ_i \le AVAIL$이 거짓이면 $p_i$는 대기
③ $REQ_i \le AVAIL$이면
③-1 다음과 같이 할당 후와 같은 상태를 만듦
$AVAIL \leftarrow AVAIL - REQ_i$
$ALLOC_i \leftarrow ALLOC_i + REQ_i$
$NEED_i \leftarrow NEED_i - REQ_i$
③-2 이 상태가 안전상태인지를 조사
③-3 안전상태이면 $REQ_i$를 할당
③-4 그렇지 않으면 프로세스를 대기상태로.
데이터 구조는 이전 상태로 복구

**안전 알고리즘**

① 길이가 각각 $m$, $n$인 WORK와 FINISH 초기화
$WORK \leftarrow AVAIL$
$FINISH(i) \leftarrow \texttt{false}$, $i = 1, 2, \cdots, n$
② $FINISH(i) = \text{false}$이고 $NEED_i \le WORK$인 $i$ 찾기
그런 $i$가 없으면 go to ④
③ $WORK \leftarrow WORK + ALLOC_i$
$FINISH(i) \leftarrow \texttt{true}$
go to ②
④ 모든 $i$에 대하여 $FINISH(i) = \texttt{true}$이면 안전상태

o 안정과 불안정 상태의 예

■ 안정 상태의 예

Total=14 Available=2

| Process | Max | Allocation | Expect |
|---|---|---|---|
| P1 | 5 | 2 | 3 |
| P2 | 6 | 4 | 2 |
| P3 | 10 | 6 | 4 |

그림 6-17 은행원 알고리즘(안정 상태)

■ 불안정 상태의 예

Total=14 Available=1

| Process | Max | Allocation | Expect |
|---|---|---|---|
| P1 | 7 | 3 | 4 |
| P2 | 6 | 4 | 2 |
| P3 | 10 | 6 | 4 |

Part 4

# 알고리즘 구현(Java)

GfGBankers.java

```
public class GfGBankers {
        int n = 5; // 프로세스의 수
        int m = 3; // 자원의 수
        int need[][] = new int[n][m];
        int [][]max;
        int [][]alloc;
        int []avail;
        int safeSequence[] = new int[n];

        void initializeValues()
        {
        // P0, P1, P2, P3, P4 프로세스
        // Allocation 행렬
        alloc = new int[][] {
                        { 0, 1, 0 }, //P0
                        { 2, 0, 0 }, //P1
                        { 3, 0, 2 }, //P2
                        { 2, 1, 1 }, //P3
                        { 0, 0, 2 } }; //P4

        // MAX 행렬
        max = new int[][] {
                 { 7, 5, 3 }, //P0
                 { 3, 2, 2 }, //P1
                 { 9, 0, 2 }, //P2
                 { 2, 2, 2 }, //P3
                 { 4, 3, 3 } }; //P4

        // 가용 자원들
         avail = new int[] { 3, 3, 2 };
        }
```

```
void calculateNeed()
{
for (int i = 0;i < n; i++)
{
    for (int j = 0;j < m; j++)
     {
    need[i][j] = max[i][j]-alloc[i][j];
     }
}
}
```

Part 4

# 알고리즘 구현(Java)

GfGBankers.java

```
void isSafe(){
    int count=0;
    //이미 할당된 프로세스를 찾기 위해 방문한 배열

    boolean visited[] = new boolean[n];
    for (int i = 0;i < n; i++){
        visited[i] = false;
        }

    //사용 가능한 리소스의 복사본을 저장하는 작업 배열
    int work[] = new int[m];
    for (int i = 0;i < m; i++){
        work[i] = avail[i];
    }

    while (count<n){
        boolean flag = false;
        for (int i = 0;i < n; i++){
            if (visited[i] == false){
                int j;
                for (j = 0;j < m; j++){
                    if (need[i][j] > work[j])
                        break;
                }
            if (j == m){
                safeSequence[count++]=i;
                visited[i]=true;
                flag=true;
                for (j = 0;j < m; j++){
                    work[j] = work[j]+alloc[i][j
                }}}}
        if (flag == false){
            break;}}
```

```
if (count < n){
    System.out.println("The System is UnSafe!");
}
else{
    System.out.println("The given System is Safe");
    System.out.println("Following is the SAFE Sequence");
    for (int i = 0;i < n; i++){
        System.out.print("P" + safeSequence[i]);
        if (i != n-1)
        System.out.print(" -> ");
    }
}
}
```

```
public static void main(String[] args)
{
  int i, j, k;
  GfGBankers gfg = new GfGBankers();

  gfg.initializeValues();
  //필요 행렬 계산
  gfg.calculateNeed();

  // 시스템이 안전한 상태인지 검사
  gfg.isSafe();
}
```

결과

```
The given System is Safe
Following is the SAFE Sequence
P1 -> P3 -> P4 -> P0 -> P2
```

**다른 조 발표내용**

[박부성 & 구본성 조]

- Soshani와 Coffman 알고리즘을 통해 교착상태 탐지
  - 교착상태 해결책으로 교착 상태 발생 여부를 탐지

- 알고리즘 수행 흐름

1단계 : Work와 Finish는 각각 길이가 m과 n인 벡터이다.
Work = Available로 초기화하고, Allocation[i] 이 0이 아니면
Finish[i] = False로 아니면 True로 초기화 한다.

2단계 : 다음 조건을 만족하는 i값을 찾는다.
Finish[i] == False , Request <= Work

3단계 : 다음 조건과 일치하는지 여부를 판단하여 2단계로 이동한다.
Work = Work + Allocation[i] , Finish[i] = True

4단계 : Finish[i] == False 라면, 1 <= i <= n 인 범위에서 시스템은 교착상태에 있으며 프로세스 Pi 또한 교착상태에 있다.

- 교착상태 탐지 알고리즘으로 교착 상태를 탐지 할 수 있으나 근본적인 교착상태 방지 해결책이 아니다.

[이세영 & 황호현 조]

- 은행원 알고리즘을 이용한 교착상태 해결

은행원 알고리즘은 '최소한 고객 한명에게 대출해줄 금액은 항상 은행이 보유하고 있어야 한다'는 개념에서 나온다.

- 은행원 알고리즘이란?

프로세스가 자원들을 요청하면 시스템은 그것을 들어주었을 때
시스템이 계속 안전 상태에 머무르게 되는지 판단하고
안전하게 된다면 그 요청을 들어주는 알고리즘

● 은행원 알고리즘의 장/단점

• 장점

– 항상 안전상태를 유지할 수 있다.

(최소한 고객 한명에게 대출해줄 금액은 항상 은행이 보유하고 있어야 하기 때문에)

• 단점

– 최대 자원 요구량을 미리 알아야 한다. (안전 상태인지 불안전 상태인지를 판단하는데 필요하다.)

– 항상 불안전 상태를 방지해야 하므로 자원 이용도가 낮다.

(불안전 상태가 될 가능성이 있는 자원은 이용하지 않으므로 이용도가 낮아짐

[허원석 & 조은새 조]

● '식사하는 철학자들' 문제를 이용한 데드락

● 데드락 상황

다섯 명의 철학자가 원형테이블 주위에 앉아있다. 스파게티 한 접시가 각 철학자에게 주어진다. 이를 먹기 위해서는 **두개의 포크가 필요**하다.
접시와 접시 사이에 하나의 포크가 있다.
만약 5명의 철학자들이 서로 사이좋게 양보하며 자신의 왼쪽, 오른쪽에 있는 두개의 포크를 잡고 골고루 식사를 할 수 있다면 그들에겐 아주 행복한 저녁이 될 것이다. 하지만 이를 소프트웨어, 특별히 멀티 스레드 기반의 어플리케이션이라고 보았을 때 이것을 '영원히' 보장 할 수 있는가? 가령, **5명의 철학자가 동시에 왼쪽 또는 오른쪽 포크를 잡는다면 어떻게 될까?**

● Fork 클래스

lock을 선언하여 포크를 사용할 때(useFork)와 안할 때(unUseFork)의 메소드 설정

● Tableware 클래스

테이블에 5명의 철학자가 있으므로 5개의 포크를 배열로 선언

● Philosopher 클래스
철학자가 아무것도 하지 않는 상태(think)
음식을 먹는 상태(eat)
포크를 드는 행위(takeFork)
포크를 내려놓는 행위(putFork)

● DiningPhilosopher 클래스
메인 함수에서 철학자 5명을 각각 선언한 후,
스레드풀을 활용하여 클래스를 실행한다

[김찬혁 & 송지현 조]

● 자원 할당 그래프 알고리즘을 이용한 데드락 해결

● 자원 할당 그래프란?
프로세스의 자원 할당 상태를 표현해주는 그래프이며,
프로세스는 동그라미, 자원은 네모로 표현하고,
요청 화살표와 할당 화살표가 존재한다.

- 요청 화살표 : 프로세스에서 자원으로 연결된 선으로 프로세스가 어떤 자원을 쓰고 싶은지를 나타낸다.
- 할당 화살표 : 자원에서 프로세스에 연결된 선으로 자원이 이 프로세스가 쓰고 있음을 나타낸다.

● 알고리즘 원리
시스템은 프로세스로부터 자원에 대한 요청을 받으면 순환 대기 상태가 발생하지 않는 경우에만 자원을 할당 한다. 만일 어떠한 프로세스가 자원을 추가 요청함으로써 순환 대기 상태가 발생한다면 시스템은 이를 막기 위해 애초부터 요청을 거절해버림으로써 데드락을 회피한다.

[박동환 조]

● 명시적 lock을 이용한 데드락 해결

● 명시적 lock?

JAVA의 명시적인 Lock은 표준 JDK 에서 총 3가지로 제공된다.

- ReentrantLock
- ReentrantReadWirteLock
- StampedLock

• 명시적인 Lock은 Java의 Object에 숨어있는 암묵적인 Lock보다 명확함.
• 블록이 아닌 lock / unlock의 명시적인 호출로 Lock을 걸기 때문에 위험하기도 함

● 명시적인 Lock들은 시한부 혹은 interrupt가 가능한 Lock을 제공하여 Dead Lock이나 기아상태로 인해 장시간 wait하는 상황에 대한 해결 방법 제시한다.