**교착상태 탐지와 복구 프로젝트**

**최종보고서**

**21760029 이세영**

**21760050 황호현**

**목차**

1. **서론**
   1. **프로젝트 선택 배경 --------------------------------------------------- 3**
   2. **교착상태 정의 --------------------------------------------------------- 4**
   3. **프로젝트 수행 계획 --------------------------------------------------- 5**
2. **본론**

**2-1. 구현 방법 --------------------------------------------------------------- 7**

**2-2. 구현**

**2-2-1. Main --------------------------------------------------------- 10**

**2-2-2. Process ------------------------------------------------------ 12**

**2-2-3. Observer ---------------------------------------------------- 15**

**2-2-4. DFS ---------------------------------------------------------- 18**

**2-3. 구현 결과 보고**

**2-3-1. 교착상태가 발생하지 않은 경우 --------------------------- 20**

**2-3-2. 교착상태가 발생한 경우 ----------------------------------- 22**

**2-3-2. 교착상태가 발생한 경우 ----------------------------------- 22**

1. **결론 ------------------------------------------------------------------------- 25**

**1. 서론**

**1-1. 프로젝트 선택 배경**

여러가지 프로젝트 중 교착상태 탐지와 복구방안을 탐구하는 과제를 선택하기로 함

이 프로젝트를 선택하게 된 계기는 실제 개발을 하면서 필자가 가장 많이 맞닥뜨린 부분 중 하나가

프로젝트 수행 과정에서 발생하는 교착상태였기 때문임

교착상태가 발생하면 항상 디버그를 통해서 발생한 지점을 찾는 것도 힘들었고, 교착상태에 대한

해결방안을 강구해보는 과정도 어려웠음

어려웠던 이유에 대해서 생각해 보자면 교착상태에 대한 기본 이론이 확실하게 정립되어 있지 않음

또한, 교착상태가 발생한 상황에서 어떻게 해결하는 것이 프로그래밍적으로 또, 논리적으로

도움이 되는지에 대한 생각도 잘 정립되어 있지 않았음

이러한 이유로 프로그래밍 적인 사고와 논리적인 사고를 기르고 실제 교착상태가 발생했을 때

정확한 해결과정을 도출하는 방법을 익히기 위해

교착상태 탐지와 복구에 대한 프로젝트를 선택하게 됨.

**1-2. 교착상태 정의**

프로젝트 설명에 앞서 먼저 교착상태의 정의에 대해서 생각해볼 필요가 있음

일단 교착상태에 대한 사전적 의미를 먼저 찾아봄

사전적의미로는 두 개 이상의 작업이 서로 상대방의 작업이 끝나기 만을 기다리고

있기 때문에 결과적으로 아무것도 완료되지 못하는 상태를 가리킨다. 라고 정의 되어 있음

교착상태 발생의 필요 충분 조건으로는

상호배제, 점유와 대기, 비 선점, 환형대기가 있는 것으로 나오고

교착상태에 대한 예방기법으로는 상호배제 부정(Mutual Exclusion),

점유 및 대기(Hold and Wait) 부정, 비 선점(Non-preemption)부정,

환형 대기(Circular Wait)부정 라고 정의 되어 있음

마지막으로 교착상태 탐지 알고리즘으로

안전상태를 유지할 수 있는 요구만을 수락하고 불안전 상태를 초래할 사용자의 요구는

나중에 만족될 수 있을 때까지 계속 거절하는 Banker알고리즘이

대표적인 알고리즘으로 나옴

위와 같은 교착상태 개념들을 미리 정립해둔 뒤 이어서 교착상태 탐지와 복구방안에 대한

프로젝트 수행 계획을 작성해 보았음

**1-3. 프로젝트 수행 계획**

이 프로젝트에 대해서 수행 계획을 세우기 위해서 먼저 문제를 분석하는 과정을 수행하였음

문제에서 요구하는 요구사항을 크게 세부분으로 나누어 생각해 보고,

이 부분에서부터 다시 세부 항목으로 문제를 분석해 나가는 과정을 진행하였음

아래는 문제에서 요구하는 요구사항을 세 부분으로 나눈 기준임

**1. 동종의 자원 n개와 프로세스 m개를 포함하는 시스템을 만드는 부분**

**2. 교착상태에 대한 여러가지 처리를 하는 부분**

**3. 종료할 스레드를 선정하는 부분**

이렇게 크게 세부분으로 나눠 보았고, 각각의 부분에서 구현해야 할 사항을

다시 세부적으로 나눠보는 과정을 수행하였음

아래는 앞서 나눈 세 부분으로부터 구현해야 할 세부 사항임

* 1. **각 프로세스가 자신에 필요한 자원 집합을 생성하게 하는 부분**
  2. **임의의 순서로 한 번에 한가지유형의 자원을 요청하고, 각 유형 사이에**

**임의의 정지시간을 두는 부분**

* 1. **원하는 것을 모두 얻을 때까지 자원을 보유하게 하는 부분**

**2-1. 교착상태를 몇 초 간격으로 확인하는 부분**

**2-2. 교착상태가 발생하면 보고하는 부분**

**2-3. 교착상태에 연관된 스레드를 종료하는 부분**

**3-1. 종료할 스레드를 선정할 때 사용된 경험적 방법 중 최적의 결과를 내는 부분**

이런 세부사항으로 문제를 분석해 프로젝트 수행계획을 작성해 보았고,

분석한 기준에 따라서 실제 구현할 부분을 구상해보는 과정을 수행함

**2. 본론**

**2-1. 구현 방법**

앞서 분석해본 세부 사항에 따라서 실제 구현할 사항들을 작성해 봄

동종의 자원 n개와 프로세스 m개를 포함하는 시스템을 만드는 부분은

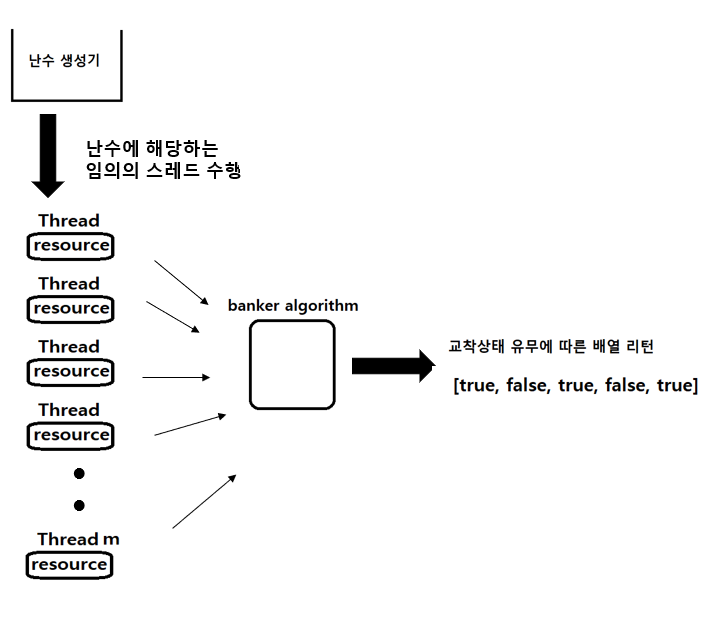
프로세스를 스레드로 놓고 스레드 내부에서 각각의 세부사항을 처리하는 방식으로 구현함

난수를 발생시켜서 임의의 순서로 임의의 자원을 배치시키며,

이때 처리방식을 Banker알고리즘을 사용해서 그 결과값을 Boolean 배열로

리턴 하는 방식을 사용

그림으로 표현해본다면 아래와 같이 표현할 수 있음



교착상태에 대한 여러가지 처리를 하는 부분은

처리를 위해서 따로 Observer스레드를 만들어서 Observer스레드 내부에서

각각의 세부사항을 처리하는 방식으로 구현

그림으로 표현해본다면 아래와 같이 표현할 수 있음

텍스트이(가) 표시된 사진

자동 생성된 설명

종료할 스레드를 선정하는 부분은

앞서 만들었던 Banker알고리즘의 Boolean 리턴 값을 DFS알고리즘에 적용해 봄으로서

최적이 방법을 탐색 알고리즘을 통해 찾아가는 방식으로 구현

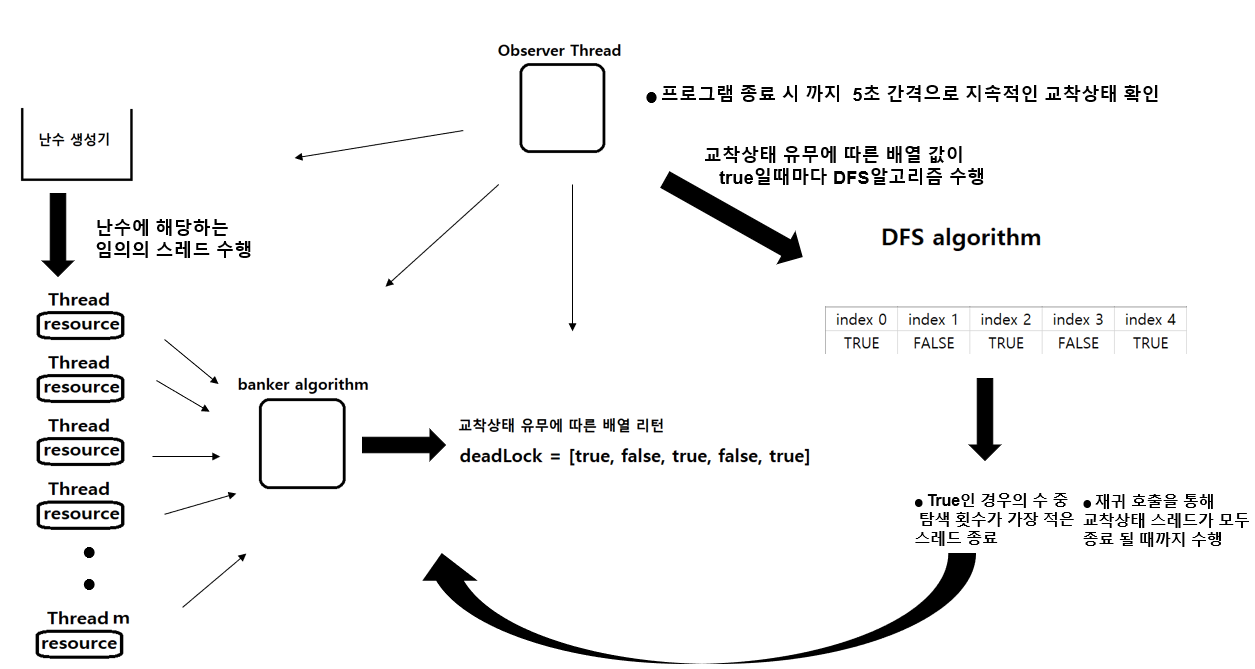
그림으로 표현해본다면 아래와 같이 표현할 수 있음

텍스트이(가) 표시된 사진

자동 생성된 설명

구현방법을 이런 식으로 정리해 보았고 앞서 설명한 모든 구현 방법을 통합해본 로직을

그림으로 표현해본다면 아래와 같이 표현할 수 있음



뒤이어 설명할 실제 구현 부분에서는

앞서 설명할 구현 방법들을 수행시킬 **Main클래스 부분**

실제 프로세스가 수행될 **Process스레드 부분**

교착상태에 대한 여러가지 처리를 수행하는 **Observer 부분**

최적의 종료 스레드를 선정하는데 사용될 **DFS알고리즘 부분**으로 나눠서

실제 구현 세부사항을 정리해 보았음

**2-2. 구현**

**2-2-1. Main 클래스**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

프로세스가 요구하는 최대 자원수를 설정하고, 현재 프로세스가 보유하고 있는 자원수를 설정하고, 임의의 순서로 프로세스 스레드를 실행시키고, 교착상태를 감시, 보고, 삭제하도록 Observer 스레드 객체를 수행시키는 Main 메서드임

코드 부분으로는

51번 라인에서 임의의 프로세스 스레드를 실행시켜 주기 위해서 0~4에 해당하는 번호를 중복없이 받아옴

74번 라인은 운영체제가 프로세스에게 가용해줄 수 있는 자원들을 work변수에 넣어주는 과정

78번 라인은 프로세스 개수만큼 반복하며 랜덤에 저장되어 있는 순서대로 프로세스 번호를 담아 실행시켜주고, 그 순서대로 queue에 담아 줌

85번 라인은 교착상태를 감시, 보고, 삭제를 수행하는 Observer 감시자 스레드를 실행시켜주는 부분임

**2-2-2. Process 클래스**

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

실제로 작업을 수행하는 Process 스레드 객체임. 크게 동작방식으로는

임의의 순서대로 프로세스 스레드를 수행한 객체들은 run 메서드를 실행하게 될 것이며 이 때 run 메서드는 세마포어를 이용해 큐에 들어온 순서대로 프로세스를 하나씩 빼주고, 해당 프로세스는 isSafe라는 메서드를 실행하면서 교착상태가 발생하는지, 안하는지 체크하고 발생한다면 옵저버에 의해 처리되고 발생하지 않는다면 안전 순서열에 등록될 것임

코드 부분으로는

우선 92~94번 라인에서 프로세스 번호와 최대 요구 자원수 및 현재 보유한 자원수를 담을 변수를 선언함 이는 Main 메서드에서 프로세스를 실행하기전, 생성자에 의해 초기화됨

121번 라인에서는 각 스레드가 실행되면 생성자에 의해 초기화되었던 값들을 이용해

calculateNeed 메서드를 수행하게 되는데 이 메서드에서는 최대 요구자원수와 현재 보유한 자원수를 이용해 현재 프로세스 스레드가 요구하는 자원수를 계산하여 저장함

123번 라인은 세마포어를 이용해 큐에 삽입된 순서대로 하나씩 빼주어 임계영역에 접근하게됨

134번 라인의 메서드는 임계영역 메서드로 파라미터로 들어온 프로세스를 현재 가용해줄 수 있는 자원으로 할당해주었을 때 교착상태가 발생하는지 확인하는 메서드임

136번 라인은 조건문으로 현재 파라미터로 들어온 프로세스 번호에 해당하는 finish 배열의 인덱스값이 false인지 확인함. false값은 프로세스 스레드가 종료되지 않았음을 의미하며 그렇기 때문에 교착상태가 발생하는지 확인해야함

138번 라인은 자원의 개수만큼 반복을 수행하며 만약 현재 가용해줄 수 있는 자원(work)보다 프로세스 스레드가 요구하는 자원(need)가 많다면 해당 프로세스에게 자원을 할당할 시 교착상태가 발생하게 됨. 따라서 반복문을 중단시키고 수행한 j값을 가지고 144번 라인에서 조건문으로 확인함. 이 때 자원의 개수만큼 다 돌고도 모든 자원들을 할당해줄 수 있다면 즉, 반복문 수행중 break에 걸리지 않는다면 j값은 최종적으로 R값과 동일해질 것임. 그럼 144번 라인의 조건문을 수행함

144번 라인은 교착상태가 발생하지 않는 프로세스 스레드만 들어올 수 있으며 해당 프로세스에게 자원을 다 할당해주고 프로세스가 보유한 자원을 회수해옴. 그리고 safeSeq 배열에 프로세스 번호를 담아주고 finish 배열을 통해 작업이 마쳤음을 의미해주고 교착상태가 발생하지 않았으므로 deadlock 배열의 프로세스 번호에 해당하는 인덱스의 값을 False로 설정해줌

157번 라인은 만약 프로세스 자원을 할당하려다가 교착상태가 발생하는 프로세스라면 deadlock 배열의 프로세스 번호에 해당하는 인덱스 값에 True를 줌으로써 교착상태가 발생한 스레드임을 알리고 deadLockProcesses 배열에 프로세스 번호에 해당하는 인덱스 값에 현재 프로세스를 담음

163번 라인은 큐에 담겨있는 모든 프로세스들이 실행을 마치고 난 뒤 안정상태 사이클을 출력함

**2-2-3. Observer 감시자 클래스**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

텍스트이(가) 표시된 사진

자동 생성된 설명

규칙적으로 프로세스들을 확인하면서 교착상태가 발생한 스레드가 존재하는지 확인하고, 교착상태가 발생한 스레드를 발견하게 되면 출력문을 통해 보고하고 삭제할 스레드를 종료할 때 최적의 결과를 내기 위해서 DFS 탐색 알고리즘을 수행함

크게 동작방식으로는

Observer 감시자 스레드를 while문을 통해서 무한 반복하며 실행 매 감시마다 Thread.sleep을 주어 5초에 한 번씩 감시하도록 구현하였으며 감시중 교착상태를 발견하면 출력문으로 보고, 삭제를 수행할 프로세스를 찾아 교착상태 프로세스별로 각각 DFS 알고리즘을 수행시켜 각각의 프로세스를 삭제하였을 때의 패널티 점수를 가져와 가장 적은 패널티 점수에 해당하는 프로세스 번호를 받아 옴. 그리고 이번엔 삭제 가정이 아닌 실제로 삭제를 수행하면서 자원을 회수하고 그로 인한 다른 교착상태 프로세스들을 수행시켜보며 교착상태를 해결할 수 있는지, 그리고 아직도 해결할 수 없다면 다시 DFS 알고리즘을 통해 최적의 스레드를 찾아 삭제하는 방식으로 구현

코드 설명으로는

위에서 대부분 설명했기 때문에 조건문이나 반복문 위주로 설명함

255번 라인에서 먼저 각각 프로세스에 해당하는 패널티 점수를 받아오기 위해서 score 배열의 모든 값을 (int)1e9로 초기화해줌. 이는 무한을 의미하며 패널티 값이 가장 적은 프로세스를 가져와야 하기 때문에 모든 score 배열의 초기값은 크게 만들어 놓음

258번 라인에서는 프로세스 개수만큼 반복문을 수행하며 만약 i번째 인덱스에 해당하는 프로세스가 교착상태라면(deadlock[i]가 True라면) isDead 값을 True로 줌으로써 DFS 탐색 알고리즘을 수행해야됨을 의미하고 DFS 알고리즘의 파라미터로 해당 프로세스를 넘겨 버림

만약 하나라도 교착상태가 발견되지 않았다면 isDead 값은 초기값인 false값 그대로 일 것이며, 이는 265번 라인의 조건문에서 교착상태가 아무것도 없으니 무한 반복문인 While문을 종료시켜 감시자 스레드 객체가 수행 중단됨

270번 라인에서는 교착상태가 발생한 프로세스들의 패널티 점수들 중에서 가장 작은 패널티값을 가진 프로세스의 번호를 deleteProcess 변수에 담아주며, 이 deleteProcess 변수는 실제로 삭제가 수행될 프로세스의 번호를 담게됨

277번 라인에서는 자원의 개수만큼 반복 수행하며 deleteProcess 변수에 담긴 프로세스 번호에 해당하는 프로세스의 자원을 모두 회수해버리며 삭제를 수행하고 해당 프로세스의 수행을 인터럽트 시킴으로써 중단시키고 교착상태를 해제하고 finish 배열에 True를 줌으로써 해당 프로세스는 수행을 마침

285번 라인에서는 다시 프로세스 개수만큼 반복 수행하면서 회수하였던 총 자원으로 다른 교착상태를 해결할 수 있는지 체크함. 그렇게 모든 교착 프로세스들이 해결이 된다면 다시 5초뒤 옵저버가 감시할 때 교착상태 프로세스가 존재하지 않다고 출력하며 감시가 종료되고, 만약 아직도 교착 프로세스가 존재한다면 옵저버에 의해 다시 DFS 탐색 알고리즘으로 최적의 종료할 스레드를 탐색할 것임

**2-2-4. DFS 알고리즘**

텍스트이(가) 표시된 사진

자동 생성된 설명

교착상태에 연관된 스레드를 종료하기 위해서 종료할 스레드를 선정하기 위한 경험적 방법 중 최적의 결과를 내는 방법을 찾기 위해서 DFS 알고리즘을 사용하였음

크게 동작방식으로는

한 개의 스레드에서 교착상태가 발생했을 경우엔 그냥 삭제시켜주면 되지만, 두 개 이상의 스레드에서 교착상태가 발생한 경우에는 어느 스레드를 종료하는 것이 최적의 결과를 내는지를 고민해야 함. 따라서 위의 DFS 알고리즘을 통해 교착상태가 발생한 각각의 스레드를 하나씩 모두 종료시켜보면서 어느 스레드를 종료했을 시에 최적의 결과를 내는지 발견하기위해 DFS 탐색알고리즘을 사용한 것임

코드 설명으로는

파라미터로 들어오는 Process 객체는 자신이 교착상태일 때 들어오는 프로세스임.

213번 라인에서 해당 프로세스가 존재하지 않으면 그냥 리턴 해주고 종료 시킴

그 후에 index변수에 프로세스에 해당하는 번호를 담아 줌

그리고 deadLock 변수만큼 반복문을 실행하는데 이때 deadlock 배열에는 각 프로세스 번호에 해당하는 인덱스에 boolean값이 들어가 있는데 여기서 True값은 프로세스가 교착상태라는 것을 나타냄. 즉, 이 문제에서는 프로세스를 5개 선언했기 때문에 0~4 인덱스에 각각 프로세스가 교착상태인지 boolean값으로 나타날 것임

그리고 219번 라인 if문에 보면 !checkDeadLock[index] 라는 조건문이 있는데 checkDeadLock 변수는 교착상태가 발생한 프로세스 중에서 DFS탐색을 진행하며 해당 프로세스를 삭제를 했는지 여부를 나타내는 Boolean 배열임

따라서 if문에서는 현재 파라미터로 들어온 프로세스가 교착상태이면서 탐색 알고리즘을 실행한적 없으면 조건문 내부로 진입하게 됨

그후로 checkDeadLock[index]에 true값을 주며 해당 프로세스가 탐색 알고리즘을 수행했다는 값을 주고 min\_score를 한 개 늘림. 이때 min\_score 변수는 일종의 패널티로 탐색 알고리즘을 수행해가면서 스레드를 하나씩 삭제한다고 가정할 때마다 하나씩 늘려주게 되는데, 운영체제가 프로세스를 삭제하는 것은 사용자 입장에서 패널티로 작용할 수 있기 때문에 최종적으로 240번 줄에서 if문을 통해 패널티가 최소가 되는 즉, 특정 프로세스를 삭제했을 경우에 삭제되는 프로세스들이 최소가 되면서 모든 프로세스들의 교착상태를 해결할 수 있는 방법을 발견해서 문제를 해결하기 위해서 선언된 변수임

225번 라인에서는 삭제를 가정했기 때문에 해당 프로세스가 가지고 있는 자원을 work변수에 모두 회수함. 그렇게 해당 프로세스를 탐색했다는 checkDeadLock 배열에 true값을 주고 자원까지 모두 회수해오는 과정을 거치면 형식적으로는 교착상태였던 해당 프로세스는 삭제되었음을 의미함. 그리고나서 DFS문으로 다시 재귀 호출하며 다시 교착상태이면서 아직 탐색하지 않은 프로세스들을 골라 삭제를 가정하며 패널티를 늘리고, 자원을 회수하는 과정을 반복하며 탐색을 수행함

231번 라인부터는 재귀함수에 들어갔었던 작업들이 하나씩 빠져나오면서 다시 삭제하느라 회수했었던 자원들 만큼을 복구해주고 패널티를 줄여주며 checkDeadLock에 false값을 주면서 다시 복구해줌. 이 과정들을 반복하면서 결국 모든 교착상태 스레드들을 탐색해가며 최적의 방법을 찾는 것임

**2-3. 구현 결과 보고**

**2-3-1. 교착상태가 발생하지 않은 경우(교착상태가 1개일때)**

**기본적인 프로세스 스레드들의 자원 정보 (요구 최대자원, 현재 보유한 자원)**

텍스트이(가) 표시된 사진

자동 생성된 설명

**출력문**

텍스트이(가) 표시된 사진

자동 생성된 설명

가용해줄 수 있는 자원을 나타내는 work의 초기값은 [3, 3, 2]임

가장 먼저 3번 프로세스가 들어오는데 3번 프로세스가 요구하는 자원은

[0, 1, 1]이므로 [3, 3, 2] 자원으로 가용해줄 수 있음

따라서 3번을 수행해주고 3번의 자원을 회수하여 work는 [5, 4, 3]이 됨

그 다음으로 2번 프로세스가 들어오는데 2번 프로세스가 요구하는 자원은

[6, 0, 0]인데 가용해줄 수 있는 work는 [5, 4, 3]이기 때문에 해당 프로세스에게 자원을 할당하게 된다면 교착상태가 발생할 것임. 따라서 해당 프로세스를 실행시키지 않고 deadLock 배열에 교착상태가 발생한다는 true값을 줄 것임

다음으로는 1번 프로세스가 진입하게 되며 1번 프로세스가 요구하는 자원은

[1, 2, 2]이므로 [5, 4, 3]으로 가용해줄 수 있기 때문에 1번을 수행해주고 1번의 자원을 회수하여 work는 [7, 4, 3]이 됨

다음으로 0번 프로세스가 진입하면서 0번 프로세스가 요구하는 자원량은

[7, 4, 3]인데 work 값도 [7, 4, 3]이기 때문에 자원을 할당해줄 수 있고 0번을 실행한 후 자원을 회수하여 work값은 [7, 5, 3]이 될 것임

마지막으로 4번 프로세스가 진입하며 4번 프로세스가 요구하는 자원량은

[4, 3, 1]로 work가 가용해주기에 충분하여 4번 프로세스를 수행한 뒤 자원을 회수함

그럼 안전 순서열은 3 -> 1 -> 0 -> 4가 될 것이며 아까 교착상태가 발생했던 2번 프로세스는

감시자 스레드 객체인 Observer 객체에 의해서 교착상태 감지에 발견되고 이 객체에 의해서

최적의 결과를 내기위한 교착 프로세스를 삭제하는 수행을 진행하게 될 것임

옵저버 감시자 객체가 교착상태 프로세스를 탐색하면서 deadLock 배열값이 true인 것을 발견하면 교착상태가 발생하였다는 메시지를 띄워서 보고하고, 이때 deadLock 배열값은 위의 출력문처럼

[false, false, true, false, false]가 될 것임. 즉 2번째 인덱스 번호를 가진 프로세스가 교착상태가 발생하였다는 의미이고 옵저버 객체에 의해 이 2번째 프로세스는 삭제가 될 것임. 그리고 난 다음 5초뒤에 옵저버 감지가 객체가 다시 모든 프로세스를 감시하게 되는데 이젠 교착상태인 프로세스가 존재하지 않을 것이고 이때 deadLock 배열값은 [false, false, false, false, false]가 될 것임

그럼 더 이상 교착상태 프로세스는 존재하지 않는다는 메시지를 띄우며 옵저버 객체는 감시를 중단함

**2-3-2. 교착상태가 발생한 경우 (교착상태가 여러개일때)**

**기본적인 프로세스 스레드들의 자원 정보 (요구 최대자원, 현재 보유한 자원)**

텍스트이(가) 표시된 사진

자동 생성된 설명

**출력문**

텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명

가용해줄 수 있는 자원을 나타내는 work의 초기값은 [3, 3, 2]임

가장 처음으로 들어온 4번 프로세스가 요구하는 자원량은

[4, 3, 1]로 가용해줄 수 있는 자원인 [3, 3, 2]보다 많이 요구함. 따라서 이 프로세스에게 할당해줄 경우엔 교착상태가 발생해서 해당 프로세스에게 자원을 주지 않음

다음은 0번 프로세스가 진입하게 되는데 0번 프로세스가 요구하는 자원량은

[7, 4, 3]으로 work보다 많은 자원을 요구함. 따라서 이 프로세스에게도 자원을 주지 않음

다음으로는 2번 프로세스가 진입하게 되는데 2번 프로세스가 요구하는 자원량은

[6, 0, 0,]으로 마찬가지로 할당시 교착상태가 발생하기 때문에 자원을 주지 않음

다음으로는 1번 프로세스가 진입하고 1번 프로세스가 요구하는 자원량은

[1, 2, 2]로 할당해줄 수 있음. 그럼 1번 프로세스를 실행하고 자원을 회수하여

work 값이 [5, 3, 2]가 됨

마지막으로 3번 프로세스가 진입하고 3번 프로세스가 요구하는 자원량은

[0, 1, 1]로 할당해줄 수 있음. 그렇게 할당해주고 자원을 회수하면 work는 [7, 5, 4]가 됨

그럼 안전 사이클은 1 -> 3이 되며 0, 2, 4 번째 프로세스는 교착상태인 프로세스들임

문제에서는 교착상태인 프로세스를 종료시킴으로써 최적의 결과를 도출하라고 하였기 때문에

0, 2, 4번 프로세스를 종료시켜야 하지만, 프로세스를 모두 종료해버리면 사용자 입장에서는 좋은 결과가 되지 못하므로 그중에 최소한의 프로세스만 종료하고 나머지 프로세스들을 다시 실행시킬 수 있다면 그것이 최적의 방법일 것임

우선 옵저버 감시자 객체에 의해서 deadLock 배열을 검사함

출력문을 보면 처음 deadLock 배열의 값음 [true, false, true, false, true]로 0, 2, 4번 프로세스가

교착상태가 일어난다는 true값을 가지고 있음

그럼 true값을 발견한 옵저버 객체는 0번 프로세스를 먼저 DFS 탐색 알고리즘을 수행시킴

그럼 0번 프로세스는 탐색을 수행하며 페널티 값을 가지게 될 것이고 그 페널티 점수는

5개의 배열을 10억으로(1e9) 초기화한 score 배열에 담기게 될 것임

그렇게 0번 프로세스를 삭제할 시에 페널티는 1로 되며 다음으로 2번 프로세스가 DFS 탐색 알고리즘을 수행하러 들어가면서 페널티 점수를 기록함

처음에 2번 프로세스의 페널티 점수는 2지만 탐색을 거듭하면서 페널티 점수가 1로 변경됨

마지막으로 4번 프로세스가 DFS 탐색 알고리즘을 수행하며 페널티 점수를 기록하게 되는데

4번 인덱스는 처음에 페널티 점수가 3점으로, 삭제했을시에 최악의 페널티를 가진 프로세스였지만 탐색을 거듭하면서 4번 프로세스도 페널티 점수가 1점으로 모두 동일한 페널티를 가지게 됨

그렇게 score 배열의 페널티 값들 중 최소값이 되는 프로세스를 즉, 삭제했을 때 가장 좋은 효율을 내는 프로세스를 삭제해야 하는데 위처럼 모두 동일한 페널티를 가질 경우 프로세스 번호가 가장 낮은 프로세스부터 삭제하도록 설정해 주었음

그렇기 때문에 출력문에는 삭제한 프로세스 번호가 0으로 뜨게 되고 0번 프로세스가 보유했던

자원을 work로 회수해와서 work의 자원은 [7, 5, 4] + [0, 1, 0] = [7, 6, 4]가 됨

그러고 나서 다시 0번부터 4번까지 반복문을 수행하면서 아직 교착상태인 프로세스들에게 자원을 할당해도 되는지 검사함. 그럼 현재 deadLock은 [false, false, true, false, true] 일 것이며 2번째와

4번째 프로세스가 아직 교착상태이기 때문에 2번째 프로세스 먼저 검사를 진행함

이때 2번째가 처음에 요구했던 자원량은 [6, 0, 0]으로 현재 work인 [7, 6, 4]가 할당해줄 수 있는

프로세스가 되었음. 따라서 2번 프로세스에게 자원을 할당해주고 실행이 완료된 프로세스 자원을

다시 회수함으로써 work 값은 [7, 6, 4] + [3, 0, 2] = [10, 6, 6]이 됨

그렇게 2번 프로세스를 안전 상태로 만들었기 때문에 안전 사이클은 1 -> 3 -> 2로 변경되었음

마지막으로 4번 교착 프로세스를 검사함

4번 프로세스가 처음에 요구했던 자원량은 [4, 3, 1]로 현재 work가 가용해줄 수 있는 프로세스이기 때문에 4번 프로세스가 성공적으로 수행이 완료되어 안전 상태가 됨

최종적으로 안전 사이클은 1 -> 3 -> 2 -> 4가 되며 0번 프로세스는 종료되었음

**3. 결론**

프로세스가 교착상태가 되었을 때 해결방안을 고민해보고

여러가지 경험적 방법들을 이용하여 최적의 결과를 도출하는 과정을 진행하면서

교착상태의 문제점을 알게 되고 이를 어떻게 해결하는 것이 더 좋은 방법인지 알게 되었음

앞서 작성했던 프로그램 선택 배경에서

위 프로젝트를 수행한 이유에 대한 긍정적이고 유의미한 결과를 도출하였고

앞으로 진행해 나갈 수많은 프로젝트 수행 중에서 교착상태가 발생하더라도

유연하게 대처할 수 있는 사고과정을 학습함