운영체제 기초

Deadlock Avoidance 과제 보고서

2022312957 조준범

목차

1. 과제 분석 및 사용 도구

본 과제에서 요구하는 바는 Deadlock Avoidance 기법을 구현하는 것이다. 구체적으로, 시스템은 N개의 프로세스와 M 종류의 자원을 가지며, 각 자원 타입별 전체 유닛 수와 프로세스별 최대 요구량 및 현재 할당 상태가 주어졌을 때, 해당 시스템이 현재 안전한 상태에 있는지를 판단하고, 안전한 순서가 존재한다면 이를 출력해야 한다.

이번 과제에서는 Habermann 알고리즘을 적용하도록 명시되어 있다. 이 알고리즘은 다수의 프로세스와 자원을 지닌 환경 아래서 각 프로세스의 잔여 필요량과 현재 시스템의 잔여 자원을 이용하여 지금까지 할당된 자원을 고려했을 때도 프로세스가 전체 요구량을 만족할 수 있는지를 판단하기 위해 반복 탐색을 통해 안전 상태를 판별하는 것이다.

즉, 각 프로세스 Pi에 대해 (특정 자원에 대한 추가 요구량 $\leq$ 특정 자원에 대한 현재 보유량 + 특정 자원의 현재 여유량) 조건을 확인해 실행 가능한 프로세스를 하나씩 선택하고, 해당 프로세스가 반환하는 자원을 받아 다음 단계로 넘어가면서, 모든 프로세스를 일련의 순서로 배치할 수 있는지를 확인함으로써 시스템이 Deadlock에 빠지지 않는 안전 상태인지 확인한다.

입력(input.txt)는 다음과 같이 구성된다 :

input.txt	내용
N M	N개의 프로세스, M개의 자원
A B C) 총 M개	각 자원별 총 보유량
A B C A B C) 총 N개, M줄	Max-Claim
A B C A B C) 총 N개, M줄	Allocation

이때 입력 오류(어떤 프로세스가 선언한 최대 요구량보다 이미 더 많은 자원을 할당받았거나, 모든 프로세스의 총 필요량 합이 시스템 전체 자원량을 초과하거나, 잔여 자원이 음수로 계산되는 경우)를 사전에 점검하여, 오류가 발견되면 탐색 로직을 수행하지 않고 오류 메시지를 출력하고 프로그램을 종료해야 한다.

2. 해결 과정과 코드 설명

문제를 해결하기 위하여 실행 속도가 비교적 빠른 C언어를 사용해 구현하고자 하며, Visual Studio 2022 플랫폼에서 코드를 작성하였다. Intel 기반 CPU가 아닌 ARM 환경에서 개발하였음을 함께 고지한다.

1) 변수 선언

```
#define MAX 100

int numProcess, numResource;
int resource[MAX] = { 0 };
int remainResource[MAX] = { 0 };
int maxClaim[MAX][MAX] = { 0 };
int allocation[MAX][MAX] = { 0 };
int need[MAX][MAX] = { 0 };

int safeSequence[MAX] = { 0 };
```

프로그램의 시작 부분에서는 MAX 변수의 값을 정수형 100으로 정의한다. 이는 프로세스 개수와 자원 종류가 최대 100개까지라는 가정 아래 나머지 배열을 선언할 수 있도록 하기 위함이다.

이어서 전역 변수들이 선언되는데, numProcess와 numResource는 각각 시스템에 존재하는 프로세스의 총 개수와 자원 종류의 총 개수를 저장한다. 해당 값들은 이후 반복문과 배열 접근 범위의 기준으로 활용된다.

다음으로 resource와 remainResource 배열이 정의되는데, 전자는 j번째 자원 종류가 보유한 전체 유닛 수를 저장하며, 후자는 현재 시스템에서 해당 자원 종류의 여유량을 나타낸다.

maxClaim은 i번째 프로세스가 j번째 자원을 최대 몇 개까지 필요로 하는지를 뜻한다. allocation은 프로세스 i가 자원 j를 얼마나 할당받았는지를, need는 특정 프로세스가 앞으로 실제로 더 필요로 하는 자원 개수를 의미한다.

마지막 safeSequence 배열은 만약 현재 프로세스와 자원들의 조합이 안전한 경우 프로세스가 작동할 순서를 저장해 반환하는 역할을 담당한다.

2) 파일 입력

```
fscanf(file, "%d %d", &numProcess, &numResource);

for (int i = 0; i < numResource; i++) {
        fscanf(file, "%d", &resource[i]);
        remainResource[i] = resource[i];
}

for (int i = 0; i < numProcess; i++) {
        for (int j = 0; j < numResource; j++) {
                fscanf(file, "%d", &maxClaim[i][j]);
        }
}

for (int i = 0; i < numProcess; i++) {
        for (int j = 0; j < numResource; j++) {
                fscanf(file, "%d", &allocation[i][j]);
                need[i][j] = maxClaim[i][j] - allocation[i][j];
                remainResource[j] -= allocation[i][j];
        }
}
```

프로그램의 main 함수에서 첫 번째로 호출되는 것은 input.txt 파일에서 필요한 정보를 추출해 지정된 변수에 저장하는 loadFile 함수이다. 앞에서 언급했듯이 첫 번째 줄에는 프로세스의 총 개수와 자원 종류의 총 개수가 작성되어있어 이를 읽어들여 사전에 정의된 각 변수에 저장된다.

첫 번째 반복문에서는 각 자원 종류별 전체 유닛 수를 resource[0]부터 resource[numResource - 1]까지 읽어온다. 이후 memcpy 함수를 이용해 remainResource[i]를 resource[i]로 복사하여 초기 잔여 자원 상태를 설정한다.

이어서 파일의 세 번째 줄부터 N번째 줄까지에 걸쳐 입력된 Max-Claim 행렬을 maxClaim 배열에 그대로 저장한다. 여기에서는 I번째 줄의 j번째 숫자가 maxClaim[i][j]가 되도록 2중 루프를 이용해 읽어들인다.

마지막 반복문에서는 Allocation 행렬을 읽어들여 allocation 배열에 저장함과 동시에 need와 remainResource를 계산한다. 프로세스 i이 필요로 하는 자원 j의 개수를 구하기 위해 maxClaim에서 allocation을 빼고, remainResource에서 allocation을 차감시킴으로써 현재 사용 가능한 자원이 계산된다.

3) 오류 검사

```
for (int i = 0; i < numProcess; i++) {
    for (int j = 0; j < numResource; j++) {
        if (need[i][j] < 0) {
            err = 1;
            printf(--생략--);
        }

        if (maxClaim[i][j] > resource[j]) {
            err = 1;
            printf(--생략--);
        }
    }
}

for (int j = 0; j < numResource; j++) {
    if (remainResource[j] < 0) {
        err = 1;
        printf(ㅡ생략--);
    }
}

return err;
```

loadFile 함수의 실행이 끝나면, printData()를 호출해 현재 전역 배열에 저장된 값들을 화면에 출력한다. 이 출력문 뒤에는 getError() 함수가 호출되어 입력 데이터에 논리적 오류가 없는지 검사하는데, 본 함수 내부에서는 정수형 err 오류 플래그를 0으로 초기화한 뒤, 세 가지 주요 검사를 수행한다.

첫째, '어떤 프로세스가 선언한 Max-Claim보다 더 많은 자원을 이미 할당받았는지'를 검사하기 위해 앞에서 계산한 need 배열의 값이 음수인지를 확인한다. 해당 값이 음수인 것은 allocation > maxClaim인 경우를 의미해 이미 I번 프로세스가 j번 자원을 최대 요구량보다 더 많이 할당받았음을 의미해 오류 플래그를 1으로 변경한 후 오류에 대한 상세 정보를 출력한다.

둘째로 maxClaim[i][j] > resource[j]인 경우는. I번 프로세스가 j번 자원을 최대로 요구하는 상황에서 시스템 전체가 자원 j를 요구량보다 적게 보유하고 있어 애초에 불가능한 요청을 의미한다. 이때도 err 값을 1로 설정하고 오류 메시지를 출력한다.

마지막으로, '어떤 자원의 현재 할당량이 보유량보다 많은지'를 점검하기 위해 사전에 자원의 전체 총량에서 이미 할당된 양을 차감해 계산된 remainResource의 값이 음수인지를 확인한다. 해당 조건문이 참이면 시스템이 특정 자원 종류를 과도하기 할당받은 상태이므로 오류로 간주한다.

검사 과정이 모두 끝나면 오류 플래그 값을 main 함수로 반환함으로써 만약 오류가 하나라도 있을 경우 다음 과정을 거치지 않고 프로그램을 종료하며, 오류가 존재하지 않는다면 다음 단계인 안전 상태 탐색을 수행한다.

4) 재귀적 안전 상태 및 시퀀스 탐색

본 과정은 두 개의 부분으로 나뉘어져 수행된다. getOptimalProcess 함수는 시스템이 안전 상태인지 아닌지를 판단하고, 안전한 경우 최종 시퀀스를 화면에 출력한다. circulate 함수는 재귀적으로 구성되어 스스로를 계속해 호출하며 백트래킹(Backtracking) 기법을 활용해 안전 시퀀스를 탐색한다. 각 함수에 대한 구체적인 내용은 다음과 같다.

```
void getOptimalProcess()
```

```
int isSafe = 0;

for (int i = 0; i < numProcess; i++) {
        int isPossible = 1;
        for (int j = 0; j < numResource; j++) {
                if (need[i][j] > remainResource[j]) isPossible = 0;
        }

        if (isPossible) {
                int progressP[MAX] = { 0 };
                progressP[0] = i;

                int tempRemain[MAX];
                memcpy(tempRemain, remainResource, sizeof(int) * numResource);

                safeSequence[0] = i;
                isSafe = circulate(progressP, 1, tempRemain);
        }

        if (isSafe) break;
}
```

함수 전반부에 선언되는 isSafe 변수는 안전 상태 탐색 결과를 저장하는 변수로, 안전 시퀀스가 하나라도 발견되면 그 값이 1로 바뀌며 탐색 반복문을 탈출한다. 그렇지 않으면 최종적으로 0인 상태로 함수가 종료된다.

본 함수는 두 가지의 반복문으로 구성되어 있는데, 첫 번째 루프를 통해 프로세스 인덱스 0부터 순서대로 i를 대입하여 현재 잔여 자원(remainResource[j])으로 i번 프로세스가 만족되는지를 검사한다. 이 검사 과정은 isPossible 플래그를 1로 초기화한 후, 내부 반복문을 통해 자원 종류 j를 순회하면서 조건을 위반하면 해당 값을 0으로 설정해 실행 불가 상태를 표시한다. 반복문이 끝난 뒤 플래그 값이 여전히 1이라면, 잔여 자원으로도 i번 프로세스를 실행할 수 있다는 뜻이 된다.

만약 isPossible이 1이라면, 이 시점에서 i번 프로세스를 안전 시퀀스의 맨 앞(progress[0])에 두고 탐색을 시작한다. 먼저 progressP 배열을 초기화하고 그 첫 번째 값을 프로세스 인덱스 값인 i로 설정한다. 이 배열은 현재까지 선택된 안전 순서를 저장하는 버퍼이며, 후속 재귀 호출 시에 각 단계에서 어느 프로세스를 선택했는지를 기록한다.

여기서 주의할 점은 remainResource가 전역 배열이므로, 이후 후술할 재귀 호출 도중 값이 변하면 원본에 영향을 미치게 된다. 따라서 tempRemain이라는 일회용 복사본 배열을 생성하고 memcpy 함수를 통해 remainResource의 값을 복사하여 circulate 함수에 파라미터로 전달함으로써, 재귀 단계별로 사용되는 잔여 자원의 상태가 서로 간섭하지 않도록 한다.

이후 전역 배열인 safeSequence에 해당 프로세스를 기점으로 탐색을 시작함을 알리기 위해 첫 번째 값으로 i를 지정한다. 이후 재귀적으로 탐색하는 과정에서 level 변수를 통해 해당 배열을 지속적으로 업데이트할 수 있다.

이제 현재까지 안전하다고 판단된 시퀀스 순서를 담은 progressP, 현재 백트래킹이 몇 번째 단계에 있는지 나타내는 level, 현재 단계에서 각 자원별 남은 유닛 개수를 의미하는 tempRemain을 파라미터로 하는 circulate 함수를 호출한다. 여기서 level 파라미터가 1로 전달된 것은 이미 해당 함수 실행 과정에서 안전한 첫 번째 프로세스를 찾았으며 해당 프로세스로 탐색을 시작하겠다는 의미를 담고 있다. circulate 함수의 기능 및 흐름은 다음과 같다.

```
int circulate(int progressP[MAX], int level, int remain[MAX])

if (level == numProcess) return 1;

int finish[MAX] = { 0 };
for (int i = 0; i < level; i++) finish[progressP[i]] = 1;
for (int i = 0; i < numResource; i++) remain[i] += allocation[progressP[level-1]][i];
for (int i = 0; i < numProcess; i++) {
        if (!finish[i]) {
                int isPossible = 1;
                for (int j = 0; j < numResource; j++) {
                        if (need[i][j] > remain[j]) isPossible = 0;
                }

                if (isPossible) {
                        progressP[level] = i;

                        int tempRemain[MAX];
                        memcpy(tempRemain, remain, sizeof(int) * numResource);

                        safeSequence[level] = i;
                        if (circulate(progressP, level + 1, tempRemain)) return 1;
                }
        }
}

return 0;
```

첫 줄의 (level == numProcess) 조건은 이미 안전 시퀀스 배열에 numProcess개의 인덱스가 모두 채워진 상태를 의미한다. 즉, 프로세스를 전부 차례대로 할당하여 실행한 상태라는 뜻이므로 1을 반환해 탐색을 종료한다. 본 단계에서 마지막 프로세스가 요구하는 자원량과 각 자원별 남은 유닛 개수를 비교하지 않는 것은 앞선 오류 검출 과정에서 요구량만큼 충분히 자원을 할당할 수 있음이 보장되었기 때문이다.

종료 조건에 해당하지 않으면, 아직 안전 시퀀스 배열에 채워야 할 프로세스가 남아 있다는 뜻이므로 다음 단계를 수행한다. 우선 finish 배열을 선언해 0으로 초기화하는데, 해당 배열은 이미 앞서 선택된 프로세스들이 종료되었음을 표시되는 기능을 담당하게 됨으로써 향후 탐색 과정에서 불필요한 접근을 막는다.

이후, 직전에 선택된 프로세스가 해제되면서 반환해야 할 자원을 현재 잔여 자원에 더하여 반영한다. 다시 말해 바로 직전 단계에서 실행이 완료된 프로세스가 점유하고 있던 모든 자원을 돌려받는 것이며, 이 과정을 통해 직전 프로세스가 안전하게 종료되었으며, 사용 완료된 자원을 다른 프로세스가 접근하여 사용할 수 있음을 가능케 한다.

이제 반환된 자원이 반영된 상태에서, 다음 단계 순서에 포함될 후보 프로세스를 탐색한다. 앞서 정의된 finish 배열을 순환하며 그 값이 0인 프로세스만 고려하며, 실행 가능한 각 프로세스에 대하여 isPossible 실행 가능성 플래그를 1로 설정한다. 이어서 반복문을 통해 특정 자원을 할당받을 여유가 되는지 각 자원에 대한 검사를 진행한다. 만약 요구량이 가용 가능한 자원 유닛 수보다 많을 시 플래그를 0으로 지정하고 결과적으로 조건문 및 반복문을 탈출하며 불가능함을 반환한다.

이 시점에서 isPossible이 1로 판명된 프로세스 i는 현재 단계에서 순서에 포함될 후보로 결정된다. 따라서 먼저 progressP 배열의 다음 순서(level+1)에 인덱스 i를 기록한다. 이로써 재귀 단계별로 프로세스를 순서에 추가하는 형태가 완성된다. 동시에, 앞선 getOptimalProcess 함수에서처럼 각 자원별 잔여 자원을 담은 remain 배열을 tempRemain 배열에 복사한다. 이는 재귀적으로 다음 단계를 탐색하는 과정에서 같은 배열에 접근해 그 값을 수정함으로써 나중에 다른 후보를 시도할 수 없게 되는 부작용을 방지하기 위함이다. 따라서 탐색이 끝난 이후에도 이전 상태를 유지할 수 있게 되는 것이다.

이와 동일한 맥락에서 safeSequence 전역 배열에도 현재 단계에 선택된 i번 프로세스 인덱스를 기록해 둔다. 이때 safeSequence와 progressP의 차이점은 전자는 나중에 실제 결과를 사용할 때 최종적인 순서를 출력하기 위함이며, 후자는 재귀 호출 간에 순서를 임시로 저장하며 탐색 경로를 유지하는 역할을 수행하게 된다.

마지막으로, 적절한 파라미터들을 대입하여 자기 자신을 재귀적으로 호출하게 된다. 여기서 (level+1)을 전달함으로써 다음 단계에 안전 시퀀스 배열의 다음 인덱스를 탐색하는 중이라는 사실을 알리고, tempRemain을 인자로 주어 다음 단계에서 사용할 잔여 자원을 넘긴다. 만약 하위 재귀 호출이 1을 반환하면, 반복적인 재귀 호출을 통해 안전한 시퀀스 탐색이 완료되었음을 나타냄으로

즉시 상위 호출에도 1을 반환하여 탐색을 종료한다. 이렇게 하여 한 번이라도 전체 프로세스를 안전하게 종료할 수 있는 순서를 찾게 되면, 더 이상 불필요한 후보를 검사하지 않고 바로 모든 재귀를 빠져나옴으로써 탐색 효율을 높인다.

반면, 하위 호출이 0을 반환하면 해당 후보 i번 프로세스로 시작하는 경로는 안전한 순서를 만들지 못했음을 의미한다. 이 경우 전체 프로세스를 순환하는 반복문이 계속 진행되어 다음 프로세스를 대상으로 검사를 수행한다. 후보를 모두 탐색했음에도 하위 호출로부터 1을 반환받지 못하면, 이 시점에서 모든 프로세스 조합에 대해 안전 시퀀스를 완성 할 수 없다는 결론이 내려진 것이다. 다시 말해, 특정 단계에서 남은 자원만으로는 더 이상 실행 가능한 프로세스를 찾는데 실패했으며 deadlock 가능성이 존재하는 불안전 상태임을 상위 호출에 반환하는 것이다.

circulate 함수를 통한 재귀 탐색이 끝난 뒤, 최상위 호출인 getOptimalProcess에서는 isSafe 변수에 그 결과가 저장되며 안전한지 불안전한지 판단하게 된다. 만약 isSafe의 값이 1이면, deadlock 없이 종료 가능한 상태이므로 'SAFE'를 출력하고 이어서 safeSequence 배열에 저장된 순서대로 인덱스를 출력하며 안전 시퀀스를 함께 보여준다.

허나 isSafe 값이 0이면 안전한 순서가 없으므로 'UNSAFE"를 출력하고 본 시스템에 deadlock 가능성이 있음을 사용자에게 알려준다. 이와 같은 두 가지 결과를 출력한 이후 프로그램 작동을 종료한다.

3. 예제 실행 결과

예제 1: 안전 상태 (5개의 프로세스와 4개의 자원)

출력결과

총 프로세스 수 : 5
총 자원 수 : 4

각 자원별 유닛 수 :
1번 자원 : 10개
2번 자원 : 5개
3번 자원 : 7개
4번 자원 : 8개

각 프로세스별 최대 자원 할당 요구량 :
1번 프로세스 : 7개 | 5개 | 3개 | 4개 |
2번 프로세스 : 3개 | 2개 | 2개 | 2개 |
3번 프로세스 : 9개 | 0개 | 2개 | 2개 |
4번 프로세스 : 2개 | 3개 | 3개 | 1개 |
5번 프로세스 : 4개 | 2개 | 2개 | 2개 |

각 프로세스별 할당된 자원 개수 :
1번 프로세스 : 0개 | 1개 | 0개 | 1개 |
2번 프로세스 : 2개 | 0개 | 0개 | 1개 |
3번 프로세스 : 3개 | 0개 | 2개 | 0개 |
4번 프로세스 : 1개 | 2개 | 1개 | 0개 |
5번 프로세스 : 0개 | 0개 | 1개 | 1개 |

각 프로세스별 필요한 자원 개수 :
1번 프로세스 : 7개 | 4개 | 3개 | 3개 |
2번 프로세스 : 1개 | 2개 | 2개 | 1개 |
3번 프로세스 : 6개 | 0개 | 0개 | 2개 |
4번 프로세스 : 1개 | 1개 | 2개 | 1개 |
5번 프로세스 : 4개 | 2개 | 1개 | 1개 |

각 자원별 남은 유닛 수 :
1번 자원 : 4개
2번 자원 : 2개
3번 자원 : 3개
4번 자원 : 5개

RESULT : SAFE
시퀀스 : 2 -> 3 -> 4 -> 1 -> 5

안전 시퀀스가 존재하여 SAFE를 출력하고 하단에 프로세스 접근 순서를 표시한다.

[input.txt]
5 4
10 5 7 8
7 5 3 4
3 2 2 2
9 0 2 2
2 3 3 1
4 2 2 2
0 1 0 1
2 0 0 1
3 0 2 0
1 2 1 0
0 0 1 1

예제 2: 불안전 상태 (5개의 프로세스와 4개의 자원)

출력결과

총 프로세스 수 : 5
총 자원 수 : 4

각 자원별 유닛 수 :
1번 자원 : 10개
2번 자원 : 5개
3번 자원 : 7개
4번 자원 : 8개

각 프로세스별 최대 자원 할당 요구량 :
1번 프로세스 : 7개 | 5개 | 3개 | 4개 |
2번 프로세스 : 3개 | 2개 | 2개 | 2개 |
3번 프로세스 : 9개 | 2개 | 2개 | 2개 |
4번 프로세스 : 2개 | 3개 | 3개 | 1개 |
5번 프로세스 : 4개 | 2개 | 2개 | 2개 |

각 프로세스별 할당된 자원 개수 :
1번 프로세스 : 2개 | 2개 | 2개 | 2개 |
2번 프로세스 : 2개 | 1개 | 1개 | 1개 |
3번 프로세스 : 2개 | 1개 | 1개 | 1개 |
4번 프로세스 : 2개 | 0개 | 2개 | 1개 |
5번 프로세스 : 1개 | 0개 | 1개 | 1개 |

각 프로세스별 필요한 자원 개수 :
1번 프로세스 : 5개 | 3개 | 1개 | 2개 |
2번 프로세스 : 1개 | 1개 | 1개 | 1개 |
3번 프로세스 : 7개 | 1개 | 1개 | 1개 |
4번 프로세스 : 0개 | 3개 | 1개 | 0개 |
5번 프로세스 : 3개 | 2개 | 1개 | 1개 |

각 자원별 남은 유닛 수 :
1번 자원 : 1개
2번 자원 : 1개
3번 자원 : 0개
4번 자원 : 2개

RESULT : UNSAFE

안전 시퀀스가 존재하지 않아
UNSAFE를 출력하고 종료한다.

[input.txt]
5 4
10 5 7 8
7 5 3 4
3 2 2 2
9 2 2 2
2 3 3 1
4 2 2 2
2 2 2 2
2 1 1 1
2 1 1 1
2 0 2 1
1 0 1 1

예제 3: 오류가 존재해 탐색 불가한 경우

출력결과

각 자원별 유닛 수 :
- 1번 자원 : 10개
- 2번 자원 : 5개
- 3번 자원 : 7개
- 4번 자원 : 8개

각 프로세스별 최대 자원 할당 요구량 :
- 1번 프로세스 : 7개 | 5개 | 3개 | 4개 |
- 2번 프로세스 : 3개 | 6개 | 2개 | 2개 |
- 3번 프로세스 : 9개 | 0개 | 2개 | 2개 |
- 4번 프로세스 : 2개 | 3개 | 3개 | 1개 |
- 5번 프로세스 : 4개 | 2개 | 2개 | 2개 |

각 프로세스별 할당된 자원 개수 :
- 1번 프로세스 : 8개 | 1개 | 0개 | 1개 |
- 2번 프로세스 : 2개 | 0개 | 0개 | 1개 |
- 3번 프로세스 : 3개 | 0개 | 2개 | 0개 |
- 4번 프로세스 : 1개 | 2개 | 1개 | 0개 |
- 5번 프로세스 : 0개 | 0개 | 1개 | 1개 |

각 프로세스별 필요한 자원 개수 :
- 1번 프로세스 : -1개 | 4개 | 3개 | 3개 |
- 2번 프로세스 : 1개 | 6개 | 2개 | 1개 |
- 3번 프로세스 : 6개 | 0개 | 0개 | 2개 |
- 4번 프로세스 : 1개 | 1개 | 2개 | 1개 |
- 5번 프로세스 : 4개 | 2개 | 1개 | 1개 |

각 자원별 남은 유닛 수 :
- 1번 자원 : -4개
- 2번 자원 : 2개
- 3번 자원 : 3개
- 4번 자원 : 5개

오류!
1번 프로세스의 1번 자원에 대한 현재 할당량이 요구량을 초과함

오류!
2번 프로세스의 2번 자원에 대한 최대 요구량이 시스템 자원 총량을 초과함

오류!
1번 자원의 할당량이 실제 보유량을 초과함

[input.txt]
```
5 4
10 5 7 8
7 5 3 4
3 6 2 2
9 0 2 2
2 3 3 1
4 2 2 2
8 1 0 1
2 0 0 1
3 0 2 0
1 2 1 0
0 0 1 1
```

요구량보다 더 많이 할당받은 경우,
요구량이 시스템 총 보유량보다 많은 경우,
시스템의 자원 보유량보다 더 많이 할당된 경우
세 가지에 대한 오류를 검출하여
오류 메시지를 표시하고 종료한다.

4. 실행 방법

본 과제를 C언어로 수행했기에 프로젝트 파일 전체를 함께 첨부한다.

Visual Studio 2022와 같은 컴파일러를 이용해 이를 불러오고, [2022312957_조준범_DeadlockAvoidance.sln] 솔루션 파일을 실행함으로써 (main.c를 단독 실행하면 아무 출력도 발생하지 않는다.) 전체 프로그램의 흐름과 출력 결과를 확인할 수 있다.

프로그램은 input.txt 파일을 기반으로 동작하므로, 실행 전 동일 디렉터리에 해당 입력 파일이 존재해야 한다.

실행 결과는 콘솔 창에 순차적으로 출력되며, 초기 상태, 오류 존재 유무, 안전 상태 판별 결과 및 안전 시퀀스를 확인할 수 있다.

5. 결론

본 과제에서는 Habermann 알고리즘을 기반으로 하는 Deadlock Avoidance 기법을 C언어로 구현하고, 다양한 입력 예제를 통해 동작을 검증하였다. 구현 과정에서는 전역 배열과 재귀 호출을 적절히 설계하여 배열 복사본을 사용함으로써 백트래킹 시 상태를 안전하게 유지할 수 있었으며, 입력 단계에서의 오류 검출 로직을 통해 잘못된 자원 요구나 할당 상황을 사전에 차단하도록 하였다.

실제 실행 결과를 통해 시스템이 안전 상태일 때 안전 시퀀스를 출력하였고, 오류나 불안전 상태에서는 그 정보를 표시함으로써 의도대로 작동하였음을 확인할 수 있었다. 이 과정을 통해 Deadlock Avoidance 기법이 자원 요구량과 할당 가능한 유닛 수를 비교해 한 단계씩 안전 시퀀스를 만드는 구조로 동작함을 체감하였고, 운영체제 수준의 자원 관리 기법이 어떻게 동작하는지에 대한 이해도를 높일 수 있었다.