Deadlock Detection 결과 보고서

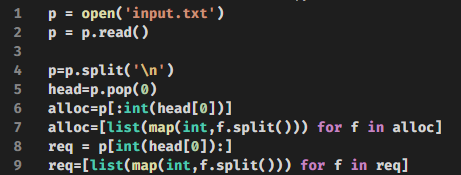
2016314726 정영준

**1) 개발환경**

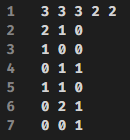
OS: Ubuntu 20.04.1 LTS

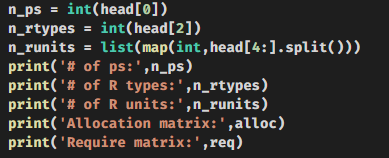
Python: 3.8.2

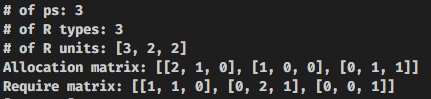
Python 사용 사유: input.txt에 문자열로 주어진 프로세스의 개수, 리소스의 unit 정보 등을 저장하여 list로 만들어 사용하려고 계획하였습니다. 동적할당이 빈번하고 더 편리한 리스트의 built-in 함수가 있는 python이 더 적합하다고 생각하였고 시스템경영공학과에서 복수전공을 하여 더 익숙한 언어였기 때문에 개발언어로 python을 사용하였습니다.

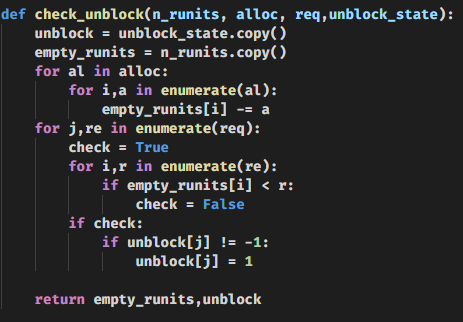
**2) 구현 / 설계**

먼저 코드의 시작은 input.txt 파일을 읽어오면서 시작합니다. p라는 변수에 읽어온 문자를 모두 할당합니다. 그리고 split함수를 사용하여 줄이 바뀌는 것을 기준으로 나누어 list로 변환시키고 이를 다시 p에 저장하였습니다. Input.txt 파일의 구조는 아래와 같습니다.

첫 줄에 프로세스의 수, 리소스의 수, 각 리소스의 유닛의 수가 주어집니다. 이 첫 줄을 head라는 변수에 따로 저장을 하였습니다. 그리고 allocation matrix를 나타내는 input.txt의 2~4번째 줄은 프로세스의 개수인 head[0]을 사용하여 나누어 따로 alloc 변수에 할당하였습니다. 마찬가지로 5~7번째 줄은 Req. matrix로 이를 req 변수에 할당하였습니다.

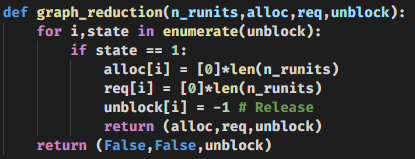
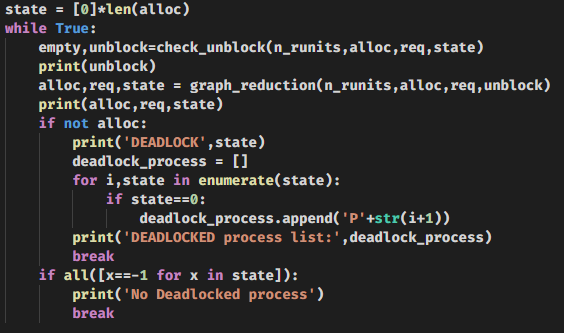
head list에 index에 해당하는 정보를 알맞은 변수에 모두 할당하였습니다. Deadlock Detection을 시작하기 전, 전체적인 프로세스 및 리소스의 정보를 print를 통해 출력하였습니다.

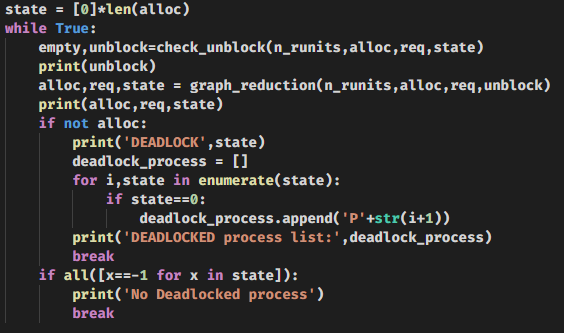
위의 예시 input.txt 파일의 프로세스 및 리소스 정보를 출력한 결과입니다.



다음으로 unblock 상태인지 체크하는 check\_unblock 함수의 선언 부분입니다. 인자로는 n\_runits(리소스 유닛의 수), alloc(allocation matrix), req(request matrix), unblock\_state(현재 unblock 상태)가 있습니다. 먼저 unblock 변수에 unblock\_state를 복사하고 시용가능한 리소스 유닛을 나타내는 empty\_runits변수에는 전체 리소스 유닛의 개수를 복사합니다. Allocation matrix의 정보를 통해 이미 할당된 리소스 유닛을 각 empty\_runits의 개수에서 빼줍니다. 이 과정을 마치면 empty\_runits list에는 각 리소스 타입별 할당되지 않은 리소스 유닛의 개수가 기록됩니다.

다음으로 request matrix의 정보를 활용하여 프로세스가 요청한 리소스 유닛 개수가 사용가능한 남아있는 리소스의 유닛 개수보다 크면 check가 False로 바뀌어 unblock list에서 그 프로세스에 해당하는 변수를 1로 바꾸어 줍니다. 여기서 1은 unblock상태 0은 block상태를 의미합니다. -1에 대한 처리는 Graph Reduction을 통해 삭제된 프로세스를 뜻하며 추후 설명하겠습니다. 예를 들어 프로세스 P1, P2, P3가 unblock, block, unblock 상태일 경우 반환하는 unblock list의 값은 [1,0,1]이 됩니다.

다음으로 Graph Reduction Method를 적용하는 함수입니다. 위의 check\_unblock 함수와 같은 인자를 전달 받습니다. 먼저 각 프로세스의 unblock 정보가 저장된 unblock list를 순서대로 탐색하면서 시작합니다. 만약 unblock process가 존재한다면 해당하는 allocation matrix의 원소를 모두 0으로 바꾸고 request matrix의 원소 또한 모두 0으로 변환시킵니다. 이는 가지고 있는 모든 리소스를 release 하고 더 이상 다른 리소스를 요청하지 않는 상태로 전환 시킨 것을 의미합니다. Graph Reduction 과정에서 Edge를 제거하는 부분을 코드로 구현한 것 입니다. 그리고 unblock list에서 해당 프로세스의 상태를 -1로 전환 시킵니다. 이는 Release (Delete all edge)상태를 의미합니다. -1로 상태가 기록된 프로세스는 추후 unblock을 확인하는 check\_unblock 함수의 탐색 대상에서 제외됩니다. 만약 하나의 프로세스의 Edge가 모두 제거 되었다면 allocation matrix, request matrix, unblock list를 반환하고 함수를 끝마칩니다. 만약 unblock process가 없다면 False가 포함된 튜플을 반환합니다.



마지막으로 Graph Reduction method를 실행시키는 코드입니다. Unblock process가 존재할 경우 반복적으로 Edge를 제거하여야 하기 때문에 while True를 사용하였습니다. 탈출조건으로는 2가지가 있습니다. 먼저 graph\_reduction 함수의 반환값이 False인 경우이고 이는 모든 프로세스가 block 상태입니다. 다른 조건으로는 모든 process의 Edge가 삭제된 경우입니다. 먼저 모든 프로세스의 unlbock 상태를 모두 block 상태로 가정하고 이를 check\_unblock 함수에 전달하여 각 프로세스의 unblock 여부가 기록된 list인 unblock을 변수에 저장합니다. 이를 graph\_reduction 함수에 전달하여 allocation matrix, request matrix, unblock list를 하나의 unblock 프로세스의 Edge가 제거된 상태로 업데이트 합니다. 만약 모든 프로세스가 block 상태일 경우 deadlock 발생 메시지를 출력하고, block 상태의 프로세스 목록을 출력합니다. 만약 모든 프로세스의 Edge를 제거하는데 성공하였으면 ‘No Deadlocked process’를 출력합니다.

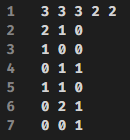
**3) 기타 고려사항, Rule**

초기의 unblock 상태를 조사할 때 2개의 프로세스가 모두 unblock 상태일 경우 input txt 파일에 먼저 기록된 프로세스의 Edge를 제거합니다.

아무 프로세스도 요청하거나 할당되지 않은 리소스가 존재하더라도 정상적으로 작동하도록 설계 하였습니다.

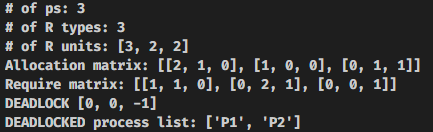
프로세스의 Edge가 제거되어 그 프로세스를 제외하고 함수를 실행시키면 결과 출력 시 프로세스의 id가 초기의 순번과 달라 변하게 됩니다. 이를 해결하기 위해 -1 상태를 도입하여 프로세스를 제외하는 대신 unblock 상태 탐색 시 이미 Edge가 제거된 프로세스를 건너 뛰는 형태로 구현하였습니다.

**4) Testing**

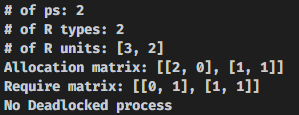
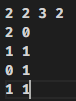
3 Process, 3 Resource types, Each Resource units: 3 2 2

Allocation matrix P1: [2, 1, 0] P2: [1, 0, 0] P3: [0, 1, 1]

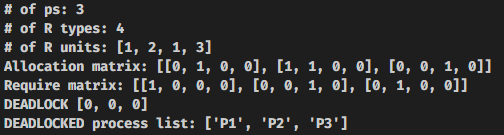
Request matrix P1: [1, 1, 0] P2: [0, 2, 1] P3: [0, 0, 1]

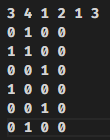
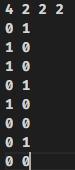
인 초기 상태가 Deadlock 상태인 예시입니다.

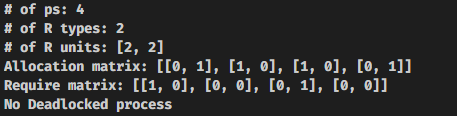
Deadlock list를 보면 P3의 경우 Edge가 제거되었지만 다른 P1, P2가 모두 block 상태로 P1, P2는 Deadlocked process입니다.



다음 예시와 그 결과입니다. P1은 초기에 unblock process이기 때문에 먼저 Edge를 제거하면 P2또한 unblock process가 되어 모든 프로세스의 Edge가 제거 됩니다. 따라서 No Deadlocked process를 출력합니다.



마지막으로 R4를 제외하고 모두 할당이 되었고 R4는 아무도 요청하지 않는 상태의 input txt입니다. 초기 상태에서 모든 프로세스가 block 상태로 바로 Deadlock Detection에 성공하였습니다. P1, P2, P3모두 Deadlocked process list에 포함됩니다.



마지막으로 프로세스가 4개일 때의 테스트 입니다. Cycle이 포함된 경우에도 Deadlock detection이 정상적으로 이루어지는 것을 확인하였습니다. 순서대로 4개의 프로세스가 하나씩 Edge를 제거하는데 성공하여 모든 Edge가 제거되고 Deadlock이 검출되지 않았다는 메시지를 정상적으로 출력합니다.

**5) 제출물 내 파일 설명**

**main.py : Deadlock Detection 구현의 모든 함수와 실행을 포함한 python 파일**

**input.txt: 프로세스 및 리소스 정보가 기록된 txt파일 (main.py와 동일한 폴더에 위치)**

**2016314726\_정영준\_P2\_결과보고서.docs : 결과보고서 word 파일**