**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재

이름 / 학번 : 우석민 / 20190408

개발 기간 : 2023-12-01 ~ 2023-12-10

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

기존까지의 핀토스 프로젝트에서는 프로그램마다 고정된 메모리를 할당하여, 그 이상의 메모리를 쓰는 프로그램들은 실행할 수 없없다. 이번에는 virtual 메모리를 구현하여 한 프로세스에서 여러 페이지를 쓸 수 있게 만들었다.

구체적으로는 page fault가 일어났을 때, 그 page에 대한 정보를 보충하는 supplementary page table을 만들고, 이를 이용해서 page fault를 처리한다. 또한 실제 physical memory를 page 단위로 나눈 frame을 관리하기 위해 각 frame이 어떤 page랑 연결되어 있는지 관리하는 frame table을 만든다. 유저 프로그램에서 스택이 커질 수 있는데, 이 때 페이지를 더 할당을 해줘야 한다. 정상적인 스택의 증가를 처리하는 stack growth를 구현한다. 마지막으로 메모리에 공간이 충분하지 않을 때 disk에 추가적인 공간인 swap space를 이용해서 lru 알고리즘으로 방출된 page 하나를 swap space에 넣는다. 이렇게 하면 당장은 메모리에 공간이 생기고, 나중에 다시 필요하다면 swap space에서 꺼내 올 수 있다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Page Table & Page Fault Handler

Page table은 page를 접근할 때 그 page가 실제로 어디에 위치해 있는지 알려주는 table이다. Page table에 항목이 없을 때 page fault handler가 불리게 된다. Page fault handler에서 supplementary page table을 이용해서 다시 필요한 page를 page table에 등록할 수 있다. 핀토스에서는 이미 page table이 구현되어 있지만, 추가적인 page를 page table에 등록하거나, page fault를 제대로 핸들링 하고 있지 않다. 만약에 이를 구현한다면 한 프로세스에서 처음에 할당된 page를 제외하고도 더 많은 page를 쓸 수 있을 것이다. 또한 page fault가 났을 때 유효한 주소라면, 적절한 handling을 통해 page를 등록할 수 있을 것이다.

* 1. Disk Swap

메모리에 공간이 없을 때 disk의 swap space를 이용해서 잠시 메모리의 내용을 디스크에 저장할 수 있다. 보통의 경우 메모리보다 디스크의 용량이 많기 때문에 메모리가 부족한 경우 프로그램을 실행할 있게 해준다. 현재 가용 메모리보다 더 많은 메모리를 유저 프로그램에서 요구한다면, 덜 쓰는 메모리 영역을 swap space에 저장하여, 추가적인 메모리 공간을 확보할 수 있다.

* 1. Stack Growth

Stack이 처음에는 고정된 메모리만 할당된다. 하지만 유저 프로그램에서 더 많은 stack 공간을 필요로 할 수 있고, 실제로는 그냥 stack을 push, sub 등을 통해 늘려서 사용한다. 이러한 경우를 구분하여, 유저가 스택을 늘리고자 하는 것이라면, 스택에 해당하는 메모리 영역을 더 할당한다. 유저는 그냥 스택을 평소에 쓰듯이 쓸 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술
  1. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

Page Fault가 발생하는 이유는 page directory에 등록되지 않은 주소에 대한 접근을 했기 때문이다. 따라서 page fault가 발생하면, page directory에 등록되지 않은 주소에 대한 정보를 알아 볼 필요가 있다. 이는 supplementary page table에 저장된다. Page directory는 실제로 physical memory랑 매핑된 가상주소 공간만 들어있다. 따라서 등록이 안되었어도 실제로 유저가 사용하는 공간일 수 있다. Supplementary page table에서 그 주소에 해당하는 공간이 어떤 것인지 먼저 확인을 하고, 종류에 따라 핸들링을 해야한다.

먼저, stack 주소일 수 있다. 그렇다면 frame을 할당하고, 그 주소에 frame을 등록하면 된다. 그 다음 프로그램 실행 파일일 수 있다. 이 때는 frame을 할당하고, 그 주소에 프로그램 실행파일을 읽어서 내용을 채워주면 된다. 또는 스왑공간에 있는 메모리 일 수 있다. 그렇다면 frame을 하나 할당하고, 스왑에서 다시 할당한 프레임에 덮어씌어줘야한다.

Supplementary page table에 없는 경우도 있다. 스택 크기가 증가되어야 하는 경우, supplementary page table에 정보가 없는데, 이를 해결하기 위해 heuristic을 사용해서 스택 증가 경우인지 확인한다. 만약에 맞다면 supplementary page table에 관련 정보를 넣어주고, frame을 할당하여 page directory에 등록을 해준다. Supplementary page table에 없으면서 스택 크기 증가 경우가 아니라면, 유효하지 않은 주소이기 때문에 프로세스를 종료해야 한다.

* 1. Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술

Disk swap 발생시 LRU approximation 알고리즘 중 second chance 알고리즘을 사용한다. CPU가 각 page가 access가 될 때마다 access bit을 1로 세팅을 해주는데, 이를 이용하는 것이다. LRU list를 만들어서, page replacement가 적용되는 frame들을 넣어준다. 이 list를 순회하면서 다음과 같이 진행한다. 먼저 현재 위치의 frame의 access bit이 세팅이 되어 있다면 0으로 바꾸고, 다음 frame으로 넘어간다. 만약에 access bit가 0이라면, 그 페이지를 replace를 한다.

* 1. Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술

Stack이 증가되었을 수 있는 경우가 몇가지 있고, 이를 감지하는 것을 목표로 한다. 먼저 push, pusha instruction으로 인해서 스택이 증가된 경우, fault\_addr이 esp보다 4 또는 32가 작다. 그 다음 sub 32, esp 등의 instruction을 통해서 스택이 증가된 경우, esp가 이미 fault\_addr보다 작다.

이런것과 더불어서 esp는 유저 프로그램으로부터 받은 주소이기 때문에 PHYS\_BASE 아래에 있는지 확인해야하고, fault\_addr도 마찬가지이다. 또한 stack의 크기를 8mb로 limit을 걸어 놓는다고 가정하면, phy\_base – 8mb보다 큰지 확인을 해야 한다.

이러한 관점에서 esp와 fault\_addr를 검사하면 stack growth인지 확인할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2023-12-01 ~ 2023-12-2 : page table, page fault handler 구현

2023-12-02 ~ 2023-12-4 : disk swap 구현

2023-12-04 ~ 2023-12-6 : stack growth 구현

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
  1. **Page table, page fault handler 구현**

Page table의 구현을 위해서 supplementary page table과 frame table을 만들 것이다. 이 테이블들은 각각 page.c, frame.c 파일에 구현을 한다.

Supplementary page table에는 여러 정보들을 저장해야 한다. 이를 위해 supplementary page element를 만들어야 한다. 먼저 실행 파일에 대한 정보를 저장하는 경우, open된 파일, offset, 유효한 바이트 개수, writable한지 여부 등을 저장해야 한다. 그 다음 frame address에 해당하는 faddr, 그리고 가상 주소에 해당하는 vaddr을 저장한다. 마지막으로 supplementary page element에 해당하는 타입이 필요하다. 실제 supplementary page table은 pintos에 제공하는 hash table을 이용한다. Vaddr을 key로 하여 supplementary page table element에 접근을 할 수 있다. 이러한 접근을 쉽게 하기 위해서 sup\_page\_find\_with\_vaddr이라는 함수를 통해 vaddr을 supplementary page table element struct로 변환한다.

Page fault handler를 구현하기 위해서 exception.c에 있는 page\_fault 함수를 수정해야 한다. 이 함수에서는 fault\_addr을 받아서, supplementary page table에서 supplementary page element을 찾는다. 이 supplementary page element가 있는지 여부에 따라서 stack growth인지, 아니면 불러와야 하는지 알 수 있다. 만약에 supplementary page element가 있다면, supplementary page table을 이용해서 page table에 등록을 해야한다. 이는 page.c에 load\_sup\_page 함수를 만들어서 진행한다. 이 함수에서는 frame을 하나 allocation 하고, 그 frame을 들어온 주소에 등록한다. 그리고 supplementary page element에 있는 내용에 따라서 실행 파일이라면 내용을 읽어서 복사하고, stack이면 그대로 둔다.

* 1. **Disk swap 구현**

Disk swap은 swap.c에 구현을 한다. Device/block.h에 있는 함수들을 사용해서 디스크에 접근할 수 있는데, block\_get\_role(BLOCK\_SWAP)을 이용하면 swap space에 대한 block을 얻을 수 있다. 그 다음에는 swap space의 크기에 맞춰서 각 swap space를 쓸 수 있는지에 대한 bitmap을 만들어야 한다. 이 bitmap을 참고하여 비어있는 swap space를 찾는다. 이 때 배열을 써도 되지만, swap space가 크기 때문에 배열을 쓰면 공간의 낭비가 클 수 있다. 이러한 initialization 작업은 swap\_init() 함수에서 하고, threads/init.c에 swap\_init() 함수를 넣어준다.

그 다음 swap space에 특정 메모리 영역을 넣기 위해서 swap\_save\_into\_swap 함수를 구현한다. 비어있는 swap space를 bitmap에서 찾고, 그 부분에 해당하는 block의 sector을 찾아서, block\_write함수를 부른다. 이렇게 하면 swap에 저장이 된다. 이 때 저장된 swap space의 위치를 리턴해야한다.

Swap space에 저장된 특정 메모리를 다시 꺼내와서 다시 특정 physical address에 넣기 위해서 swap\_load\_from\_swap함수를 구현한다. 이 함수에서는 들어온 swap space number을 이용해서 block에서 읽고자 하는 부분이 어디인지 확인하고, block\_read 함수를 부르게 된다. 그렇게 된다면 swap에서 메모리로 다시 내용을 가져올 수 있다.

이러한 swap을 언제 할지도 중요하다. 먼저 frame을 allocation을 할 때 palloc\_get\_page를 사용하는데, 메모리가 없다면 NULL을 리턴한다. 따라서 이 때 swap에 오랫동안 사용하지 않은 메모리를 넣어야 하는 것이다. 먼저 replacement 알고리즘을 통해서 어떤 frame을 swap에 넣을지 정해야 한다. Frame을 할당할 때마다 이 때 쓰기 위해 lru\_list에 넣어둔다. 그 다음 lru\_list를 순회하면서 access bit을 확인하면서 second chance 알고리즘을 구현한다. Pagedir\_is\_accessed 함수를 이용해서 각 frame에 access bit가 설정되어있는지 확인하고, pagedir\_set\_accessed를 통해 1인 frame을 0으로 만들어 준다. 만약에 0인 frame이 있다면, 그 frame의 내용을 swap.c에서 구현한 swap\_save\_into\_swap 함수를 불러서 swap space에 저장하고, 해당하는 supplementary page table element에 swap space에 넣어주었다고 기록을 한다. 그 다음 해당 frame을 free를 시켜줘서 다시 할당될 수 있게 한다. 이 내용은 frame\_table\_evict\_frame 함수에서 구현하여, 추가적인 frame의 할당을 할 때 공간이 부족하면 이 함수를 부르면 된다.

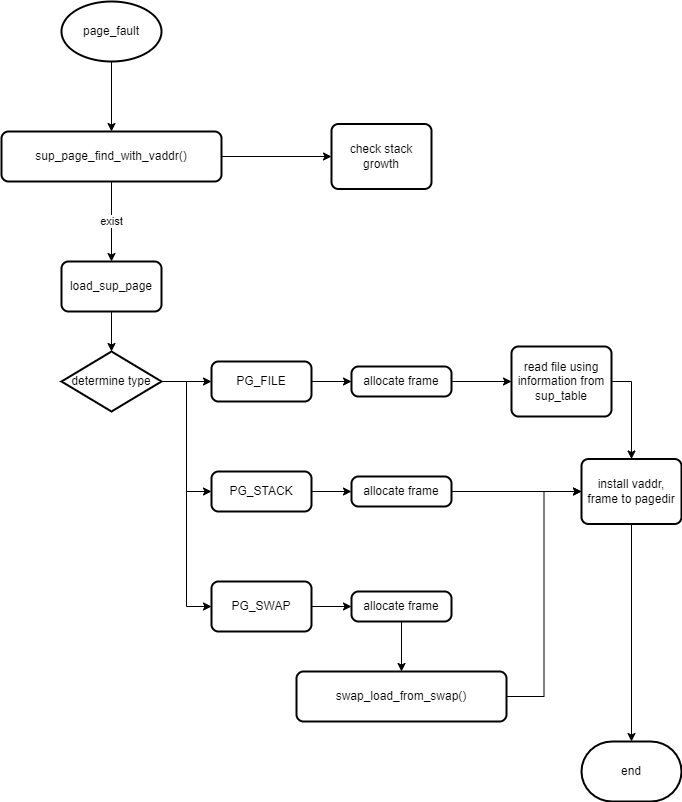
Page fault handler에서 swap에 들어있는 메모리를 접근해야 할 수도 있다. 이 때 supplementary page table element에 swap space에 있다는 것을 확인하면, swap\_load\_from\_swap 함수를 이용해서 새로운 frame에 내용을 불러오면 된다.

* 1. **Stack growth 구현**

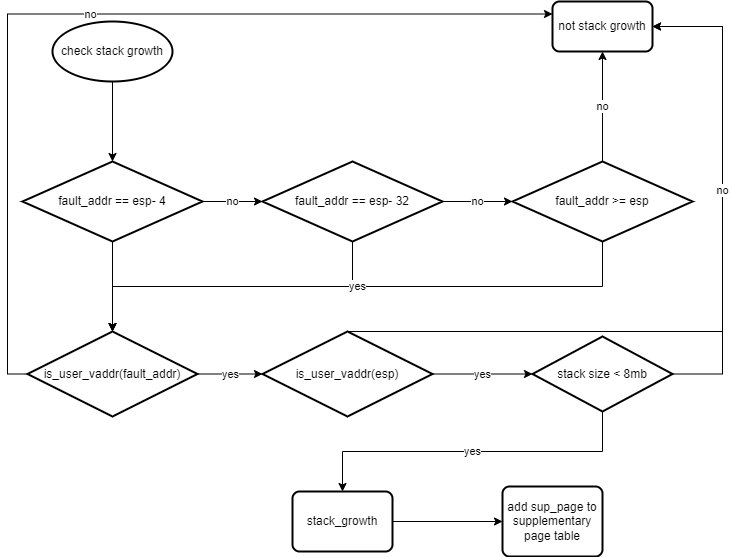
page fault handler에서 fault\_addr이 들어왔는데, 해당하는 supplementary page table entry가 없는 경우, stack growth인지 확인한다. 먼저 fault addr과 esp 주소가 유저 영역을 가리키는지 확인한다. 그 다음 유효한 stack growth인지 확인한다. Fault\_addr이 esp-4, esp-32, 혹은 esp가 fault\_addr보다 작은지 확인하면 된다. 마지막으로 fault\_addr이 스택 최대 크기보다 작은지 확인하고, 이 모든 조건을 맞추면 supplementary page table에 관련 내용을 넣는다. 이것들은 exception.c에 있는 page\_fault 함수에 구현을 하면 된다.

1. **연구 결과**
   1. **Flow Chart**

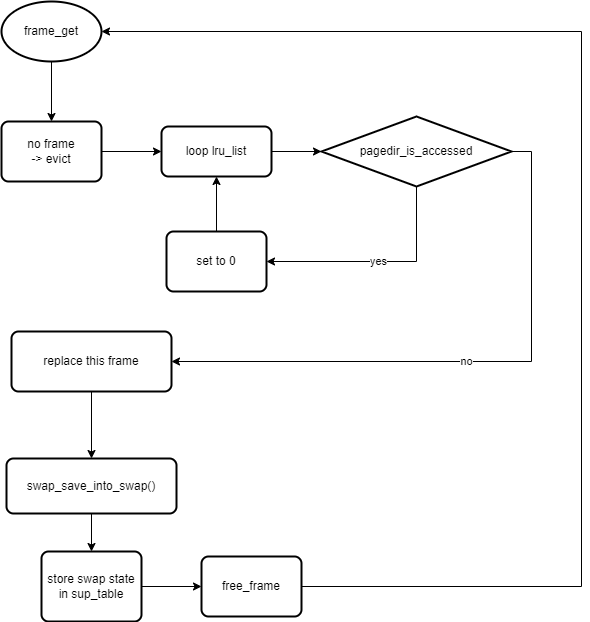
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성



Page fault에 대한 내용이다. Page fault가 났으면 sup\_page\_find\_with\_vaddr 함수를 이용해서 해당하는 sup\_page를 찾는다. 만약에 없으면 stack growth를 확인한다. 이 내용은 아래에 따로 flow-chart를 그렸다. 만약에 항목이 있다면 load\_sup\_page 함수를 부르고, 여기에서 type에 따라 처리 방법을 다르게 한다. PG\_FILE(실행 파일인 경우)라면, supplementary page table element에 파일의 어디서부터 어디까지 읽어야 하는지에 대한 내용을 참고하여 읽는다. PG\_STACK인 경우에는 그냥 frame만 할당해주면 된다. 마지막으로 pagedir에 vaddr을 frame과 연결시켜주고 handling이 끝난다. Swap 상태에 있는 PG\_SWAP은 아래에 swap과 관련된 flow chart와 함께 설명하였다.



Stack growth 같은 경우는 위의 6가지 조건을 확인한다. Fault\_addr이 esp-4, esp-32 또는 esp보다 큰지 셋 중 하나에 해당되고, 동시에 fault\_addr, esp가 user주소 영역이여야 하고, stack size를 초과해서는 안된다.



Swap을 구현하기 위해 2 부분을 수정해야 한다. 먼저 frame을 allocation을 할 때 더 이상 없으면, second chance 알고리즘으로 swap space에 넣을 frame을 정한다. 정해졌으면, swap\_save\_into\_swap()으로 swap space에 넣고, supplementary page table에 내용을 저장한다.

그리고 다시 swap sapce에 저장된 메모리를 불러올 때 page fault에서 PG\_FAULT이면 swap\_load\_from\_swap 함수를 이용해서 frame에 swap에 있는 데이터를 불러온다.

* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
  1. Page table and page fault handling

텍스트, 스크린샷, 폰트, 메뉴이(가) 표시된 사진

자동 생성된 설명

Supplementary page table의 각 element는 다음과 같은 struct로 저장하였다. 또한 page\_type은 enum으로 저장하였다. (mmap은 매뉴얼 보고 하다가, 채점 기준에 없는 것을 나중에 확인하였습니다.)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

그 다음 frame table에 저장되는 각 frame은 다음과 같이 저장하였다. Not\_evict나 has 같은 항목은 synchronization을 위해서 따로 저장을 해 두었다. Not\_evict 같은 경우는 syscall 등에서 쓰고 있는 frame 같은 경우 evict를 하면 안되기 때문에 표시하기 위해 넣었고, has는 연결된 page table entry가 있는지 확인하기 위해 넣었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Page\_fault 함수에서는 다음과 같은 내요을 넣었다. 먼저 sup\_page\_find\_with\_vaddr 함수를 이용해서 supplementary page table 항목이 있는지 확인하고, 없다면 위에서 설명한 기준을 가지고 stack growth인지 확인한다. Stack growth가 맞다면, sup\_page\_table\_stack\_growth 함수를 이용해서 supplementary page table에 크기가 증가된 stack관련 내용을 넣는다.

만약 항목이 있다면 load\_sup\_page 함수를 호출하여 주소에 해당하는 내용을 frame을 allocation 하여 load를 한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

위 처럼 supplementary page에 저장된 type을 switch하여 각 타입에 맞는 방법으로 로드를 해 주었다. 먼저 frame\_table\_get\_frame() 함수를 통해 frame을 하나 얻는다. PG\_FILE인 경우 file에서 읽어와야 하기 때문에 file\_seek를 하고, file\_read를 하는 방식으로 frame에 저장한다. Stack 같은 경우는 frame에 저장할 내용은 따로 없다. 마지막으로 pagedir에 vaddr과 frame에 할당된 주소를 이어준다.

* 1. Swap 구현

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Swap을 구현하기 위해 swap\_save\_into\_swap 함수와 swap\_load\_from\_swap 함수를 구현하였다. 이 함수들에서는 swap\_used라는 bitmap에서 swap number을 확인하고, swap\_block에서 해당하는 sector을 접근한다. Swap\_save\_into\_swap은 block\_write를 써서 swap space에 write를 하고, swap\_load\_from\_swap은 block\_read 함수를 써서 swap space로부터 다시 메모리에 복사해온다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Frame\_table\_get\_frame 함수에서는 만약에 palloc\_get\_page가 실패한 경우 frame\_table\_evict\_frame() 함수를 호출하여 frame을 swap space에 넣는다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

그 다음 lru\_list를 순회한다. Lru\_list에 저장된 frame의 virtual address의 access bit을 pagedir\_is\_accessd 함수를 이용해서 확인한 다음, true이면 false로 바꿔준다. 아니라면 victim\_frame을 이 frame으로 지정한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

그 다음 eviction을 진행한다. Swap\_save\_into\_swap 함수를 불러서 현재 frame에 있는 내용을 swap space에 저장하고, swap\_num을 저장해서 나중에 다시 불러올 수 있게 한다. 그 다음에 supplementary page table element의 타입을 PG\_SWAP으로 변경하여 나중에 page fault에서 swap space에 있는 내용인지 확인할 수 있게 한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

page\_fault에서 supplementary page의 타입을 switch할 때 PG\_SWAP 항목을 추가하여 새롭게 할당받은 frame에 원래의 내용을 복사하기 위해 swap\_load\_from\_swap을 부른다.

* 1. Stack growth

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Page\_fault에서 supplementary page table에 내용이 없다면 (sp==null) stack growth인지 확인해야한다. 위에서 언급한 heuristic으로 stack grow를 해야하는지 확인한다. Fault\_addr이 esp-4, esp-32 또는 esp보다 더 큰지 셋 중에 하나여야 한다. 그 다음 esp와 fault\_addr 모두 유저 주소 공간에 위치해야 하고, fault\_addr은 stack max size보다 작아야 한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명Sup\_page\_table\_stack\_growth 함수를 불러서 supplementary page table에 관련 내용을 넣어준다. 이 때 stack의 크기가 PGSIZE보다 더 크게 커졌을 경우를 고려하여 while문을 활용해 초기 스택이 있는 부분까지 supplementary page table에 등록이 안된 부분이 있으면 추가를 해준다.

* 1. **시험 및 평가 내용**
* (채점 대상 테스트 케이스에 해당하는) make check 수행 결과를 캡처하여 첨부
* 텍스트, 스크린샷, 폰트, 흑백이(가) 표시된 사진

  자동 생성된 설명