**Pintos Project 4: Virtual Memory**

담당 교수 :박성용 교수님

이름 / 학번 :백승훈 / 20181641

개발 기간 : 2022.11.15 ~ 2022.12.05

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

전체 목표 - 가상 메모리 시스템을 구현한다.  
상세 목표 1. Page Table Management

페이지 테이블을 구현하여 프로그램이 로드될 때 바로 페이지를 할당하는 것이 아니라, 해당 페이지의 정보를 따로 저장해놓고 실제 해당 주소에 접근할 때 Page Fault Handler를 통해 페이지를 할당한다.

상세 목표 2. Page Swap

실제 페이징할 물리 메모리가 부족한 경우 Page replacement algorithm을 적용하여 페이지를 swap out한다.  
추후에 다시 해당 페이지에 접근하고자 하는 경우를 위해 swap in을 구현한다.

상세 목표 3. Stack Growth

현재 Pintos의 Stack 크기는 제한되어있다. 만약 프로그램이 Stack 제한을 넘어선 부분을 접근하려고 하면 Stack의 크기를 확장시켜주어 해결한다.

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

프로그램이 로드될 때 바로 페이지를 할당하지 않고, 실제로 해당 페이지에 접근하고자 할 때 페이지를 로드하게 되어 불필요한 페이지 사용이 줄어든다.

* 1. Disk Swap

물리 메모리가 부족하여 페이지를 할당할 수 없는 경우에도 페이지를 스와핑하여 원하는 페이지를 로드할 수 있게 된다.

* 1. Stack Growth

프로그램이 스택 용량 제한보다 더 큰 스택 공간을 요구할 경우 스택 크기를 늘려 프로그램이 원활하게 실행되게 할 수 있다.

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

Page Table 구현과 관련하여 Page Fault가 발생하는 이유는 “not present” 때문이다.  
우리가 process load 과정 중 palloc get page를 사용하여 페이지를 할당하지 않았으므로, 프로그램은 실제로 해당 페이지가 할당되어 있는 줄 알고 접근하지만 우리가 할당하지 않았으므로 page fault가 발생하게 된다.

이렇게 page fault가 발생하게 되면, page\_fault 함수에서 우선 not present error인지 확인한다, not present error인 경우 fault address에 대해 Page Table 내부에 해당 address 관련 정보가 있는지 확인한다. 정보가 있다면 해당 정보를 기준으로 page를 할당하여 page fault를 handling한다.

해당 address 관련 정보가 없을 때, 만약 fault address가 stack 확장이 필요한 영역이라면(Growable region), Stack을 확장시켜준다.

만약 둘 다 아니라면 프로세스를 강제로 종료시키고 할당된 자원을 회수한다.

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

Second Chance Algorithm을 사용하였다.  
우선 할당된 page들을 page list에 저장한다. page에 할당된 가상 주소에 대해, pagedir.c의 pagedir\_is\_accessed()를 사용하여 accessed 된 page인지 확인한다. 만약 accessed된 페이지인 경우 access bit를 0으로 변경한다. 만약 모든 페이지를 순회한 경우 다시 list의 첫 번째 페이지부터 순회를 시작한다.  
pagedir\_is\_accessed()가 false, 즉 access bit가 0인 경우 해당 page를 victim으로 지정하여 swap out 되게 한다.

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

프로젝트 명세서에 따르면, 많은 GNU/Linux System의 Stack Growth 의 제한이 8MB라고 한다.

또한 Stack은 가상 메모리 기준으로 PHYS\_BASE에서 시작하여 아래로 자라는 방향이므로, page fault시 해당 fault address가 PHYS\_BASE에서 아래로 8MB 이하의 주소에 대한 접근인 경우 Stack growth case로 판단하여 stack을 확장시킨다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2022/11/15 ~ 2022/11/17: 프로젝트 명세서 확인 및 구현 필요 목록 정리

2022/11/17 ~ 2022/12/1: Page Table, PF Handler 구현

2022/12/1 ~ 2022/12/4: Disk Swapping 구현

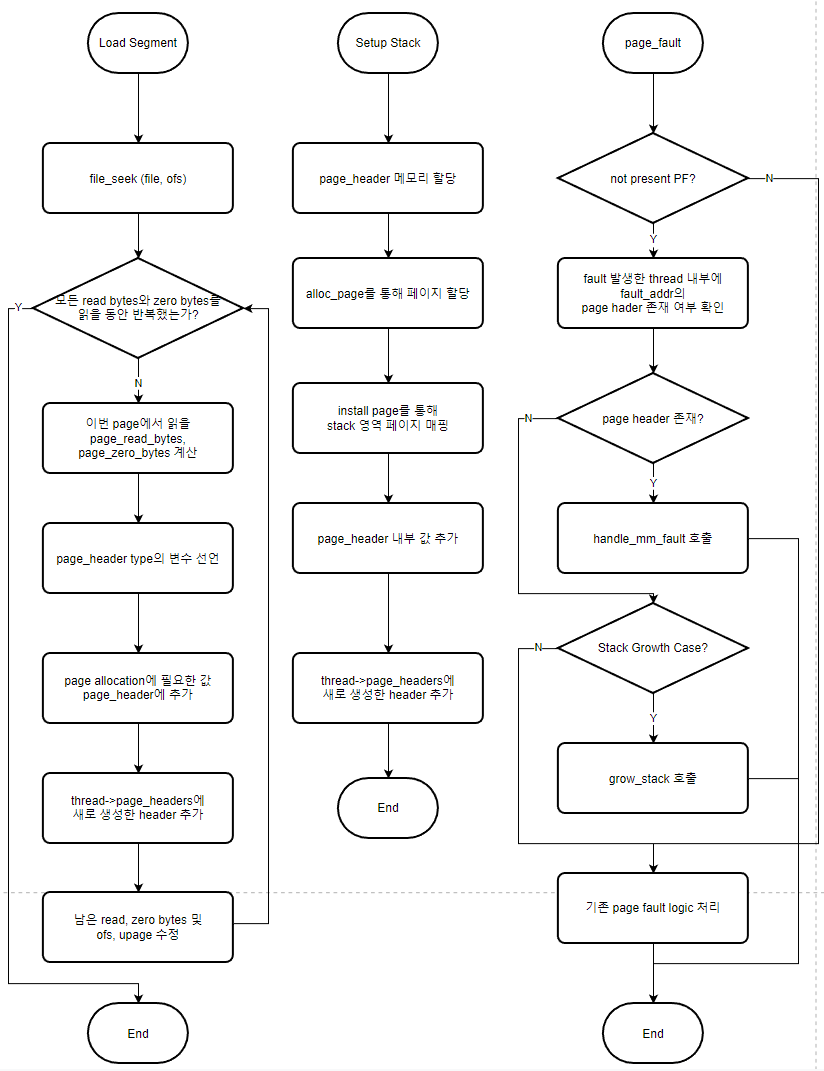
2022/12/4 ~ 2022/12/5: Stack Growth 구현

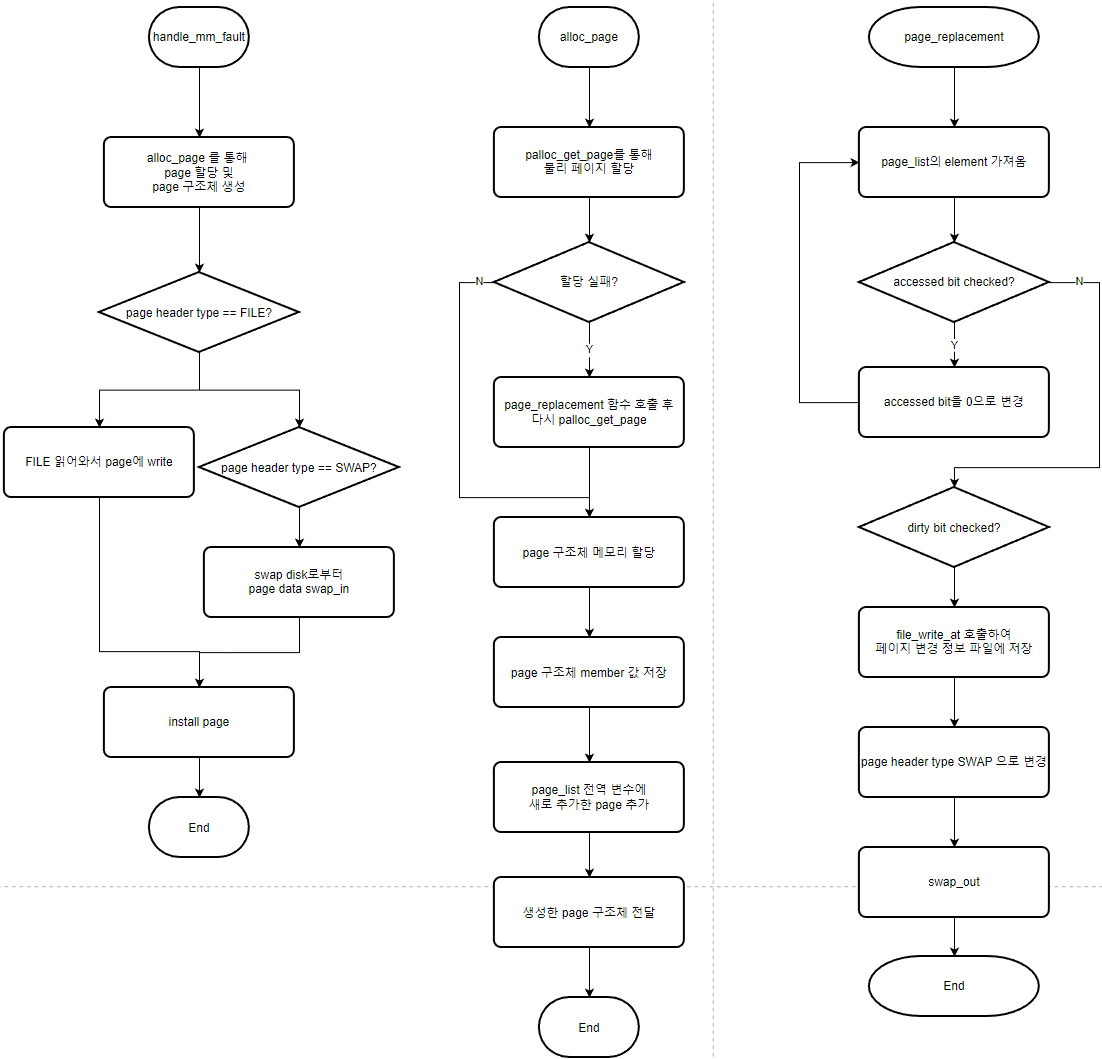
* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
* Page Table, PF Handler
  + 우선 Page Table의 추가를 위해 page와 page\_header 구조체가 필요하다.  
    page 구조체는 실제로 page를 alloc한 경우 page를 사용중인 thread와 page header 정보 및 실제 물리 page 주소를 저장한다.  
    page\_header 구조체는 process load시 실제로 page를 alloc하지 않고 대신에 만드는 데이터이다. 해당 데이터에 실제 할당되었어야 할 가상 주소, 원래 파일을 로드해왔을 file pointer, 읽어왔어야 할 파일의 byte 수를 저장한다. 또한 swap 영역으로 옮겨진 경우 몇 번째 칸에 위치해있는지 확인하기 위한 용도로 swap 이라는 멤버도 추가한다.
  + page header는 thread별로 관리한다. 이를 위해 thread 구조체에 hash type의 page\_headers 멤버를 추가한다.
    - hash를 사용하므로, hash 자료구조를 관리하기 위한 hash function과 less function을 정의해준다.
  + page 는 전역적으로 관리한다. 이를 위해 frame.c 파일을 생성하고 page\_list 변수를 list type의 전역 변수로 선언한다.
  + Page Table의 활용을 위해 Process.c의 load\_segment 함수와 setup\_stack 함수를 수정해야 한다.
    - 우선 load\_segment에서는 palloc\_get\_page 하여 페이지를 할당하고 데이터를 읽은 후 page를 install하는 부분을 제거하고, page\_header 구조체 변수를 하나 선언한 후 palloc, read 및 install page에 필요한 모든 정보를 저장하고, 현재 thread의 page header hash에 추가한다.
    - setup\_stack 함수에서는 stack을 할당하는 alloc 및 install page 부분은 그대로 두고, page\_header 변수만 선언하여 가상 주소를 담은 후 thread의 page\_headers에 추가한다.
  + 위처럼 실제 page를 할당하지 않도록 logic을 변경하면, 로드 실행 후 page fault (not present) error가 발생하게 된다. 이를 처리하기 위해 exception.c의 page\_fault 함수를 수정해야 한다.
  + page\_fault 함수에서 not present flag가 true인 경우 우선 fault 가 발생한 thread의 page\_headers가 있는 경우, process.c에 추가할 handle\_mm\_fault 함수를 호출한다.
  + handle\_mm\_fault 함수에서는 헤더의 정보를 바탕으로 페이지를 할당하고 데이터를 읽어온다. 이 때 헤더의 type 정보를 통해 파일에서 값을 읽어올지, 스왑 디스크에서 값을 읽어올지를 구분한다.
    - 파일 영역에서 읽어올 경우 header 멤버 값들을 활용하여 file\_read\_at으로 데이터를 읽어와 페이지에 할당한다.
    - swap 영역에서 값을 읽어올 경우 block\_read 함수를 통해 스왑 디스크에 있는 값을 읽어오고, 스왑 디스크 파티션을 관리하는 bitmap의 값을 정리한다.
  + handle\_mm\_fault에서 page를 할당할 때, 새로 선언할 alloc\_page 함수를 사용한다. alloc\_page에서는 전체 page list를 관리하기 위해 위해서 정의했던 page 구조체를 사용한다. palloc\_get\_page 함수를 통해 물레 페이지를 할당받고, 물리 페이지의 주소 및 해당 페이지를 사용할 thread와 해당 페이지에 대한 가상 페이지 header 정보를 구조체에 담는다. 이후 사용중인 page list에 해당 page를 넣고 해당 구조체를 반환한다.
  + 반환받은 page 구조체 내부의 물리 메모리 주소와, page header 구조체 내부의 가상 주소를 사용하여 install page를 진행한다. 성공한 경우 함수를 리턴하여 page fault 이전 프로세스가 다시 실행되게 한다.
* Disk Swapping
  + 우선 세 가지 전역 변수가 필요하다. Swap Disk 역할을 할 block\* type의 swap\_device, Swap Disk 내부의 각 블록의 사용 여부를 나타낼 bitmap type의 \*swap\_bitmap 마지막으로 bitmap 사용 시 사용할 lock type 변수 swap\_lock이다.  
    vm 디렉터리 내부에 swap.c 파일을 만든 후 전역 변수로 선언한다.
  + handle\_mm\_fault 내부 flow에서 palloc\_get\_page를 통해 페이지 할당 요청을 했을 때, 물리 페이지가 부족하여 NULL이 반환된 경우 Page Replacement Algorithm을 통해 victim page가 선정되면 이 때 새로 작성할 swap\_out 함수가 호출되게 된다.
  + swap out 함수 내부에서는 swap\_lock을 잡은 후 비트맵을 검사하여 block disk 내부의 미사용중인 slot을 찾아낸다. 그 후 해당 slot을 사용중으로 표시하고 block\_write 함수를 활용하여 page의 내용을 swap disk 내부에 저장한다. 이후 해당 page를 free하여 새로 palloc\_get\_page 호출 시 해당 페이지가 할당되도록 한다.
  + victim page가 선정된 후 swap out 전에 dirty bit을 검사해야 한다. 만약 dirty bit이 1인 경우에는 file\_write\_at을 사용하여 파일의 값을 업데이트해준다.
  + swap\_out 이후 handle\_mm\_fault 가 발생한 경우에는 page header의 page type이 SWAP으로 변경되어 있으므로, swap in을 호출하여 페이지를 파일이 아닌 swap 영역에서 가져오게 한다.
* Stack Growth
  + handle\_mm\_fault에서 fault\_address가 page\_headers 내부에 정의되어 있지 않은 경우 fault\_address가 PHYS\_BASE에서 8MB 이내의 값 접근인지 검사한다.
  + 이런 경우 Stack Growth case이므로, 이전에 선언해두었던 alloc\_page 함수를 통해 page를 할당받은 후 fault\_addr와 새로 할당한 page의 물리 주소를 install page하여 mapping시켜준다. 또한 해당 page에 대한 page header 정보를 생성한 후 thead의 page headers에 추가해준다.
  + install page를 통해 fault address를 매핑시켜주면 이후 해당 스택 영역에 접근할 때는 PF가 발생하지 않으므로 stack이 growth되었다.

1. **연구 결과**
   1. **Flow Chart**

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

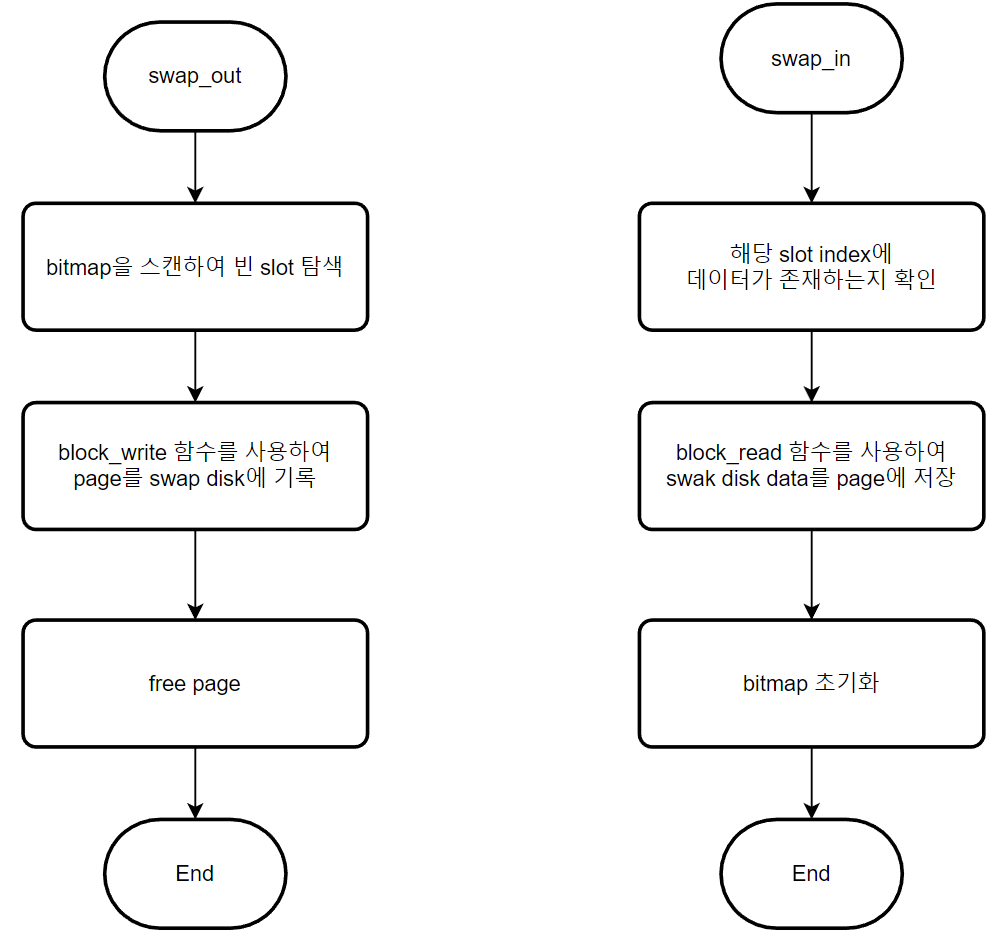
1. Page Fault Handling



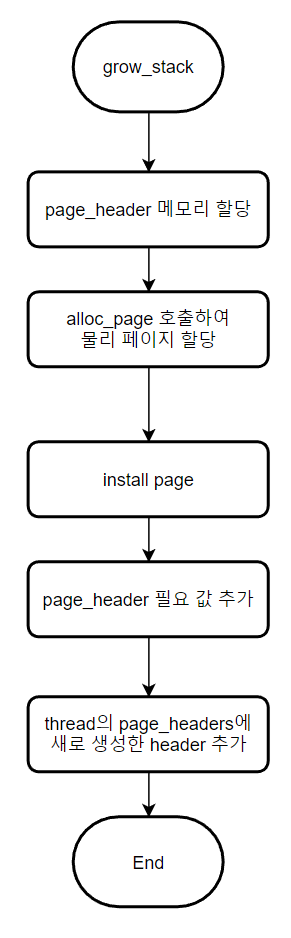


1. Disk Swapping

swap\_out, swap\_in 함수가 호출되는 경우는 page fault handling에서 다루었으므로 추가하지 않았습니다.

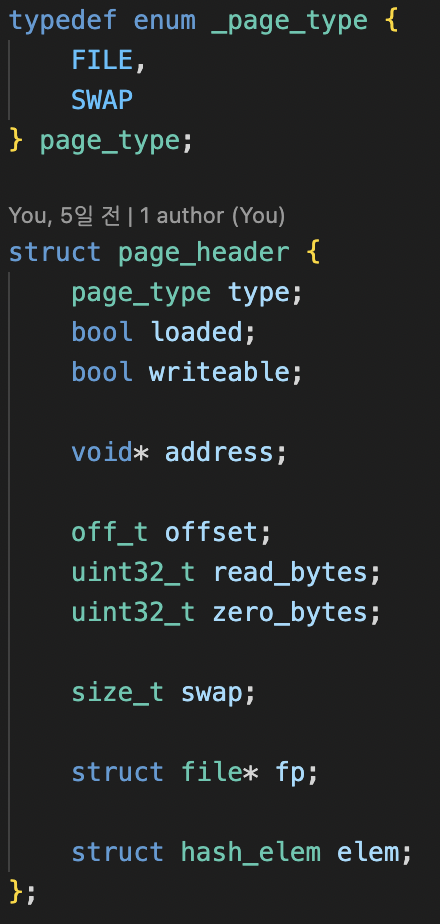


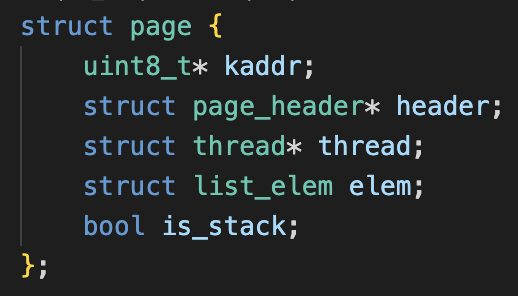
1. Stack Growth

grow\_stack 함수가 호출되는 경우는 Page Fault Handling에서 다루었으므로 추가하지 않았습니다.

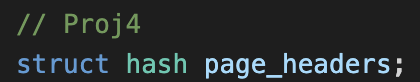
* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

우선 page\_header와 page 구조체를 정의하였다.

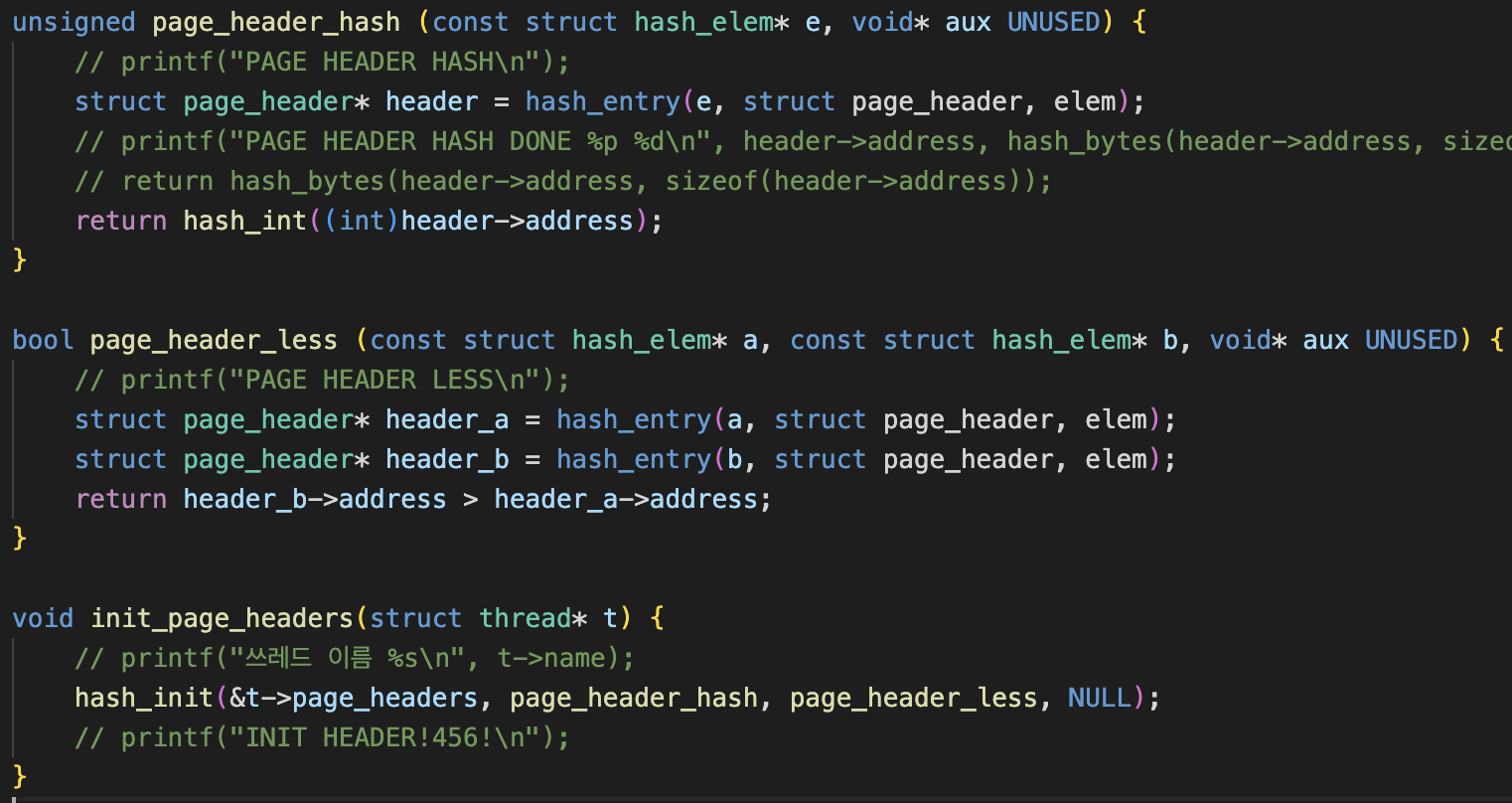




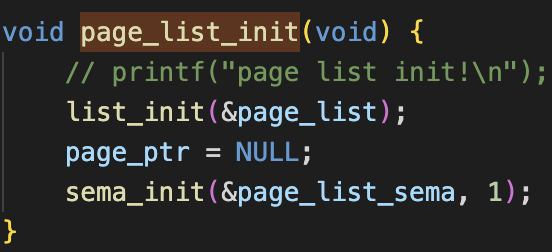
thread 구조체에 page\_header 들을 관리할 hash를 선언해준다.



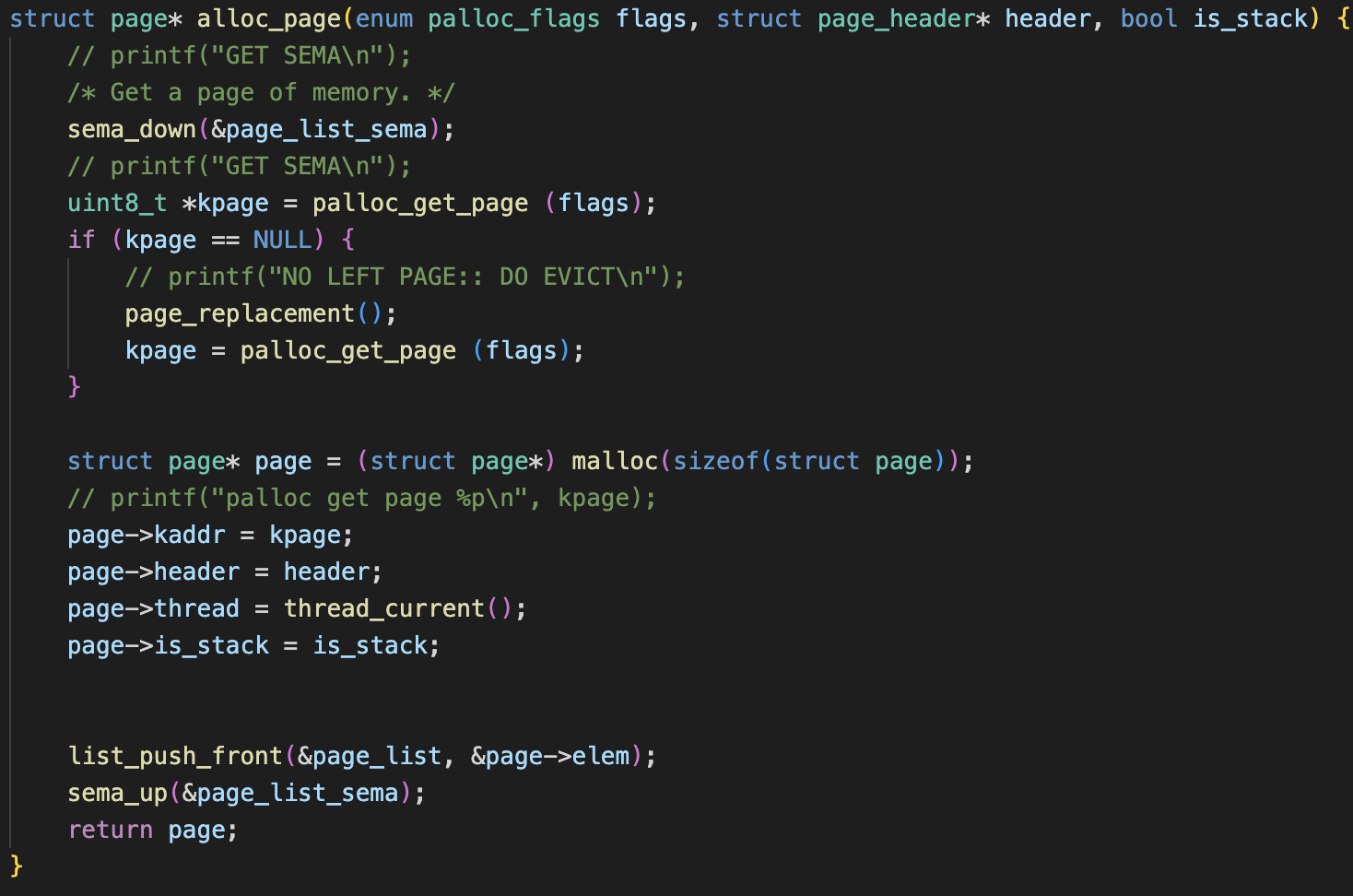
thread의 page\_headers 들에 접근하고 관리하기 위한 함수들을 선언한다.



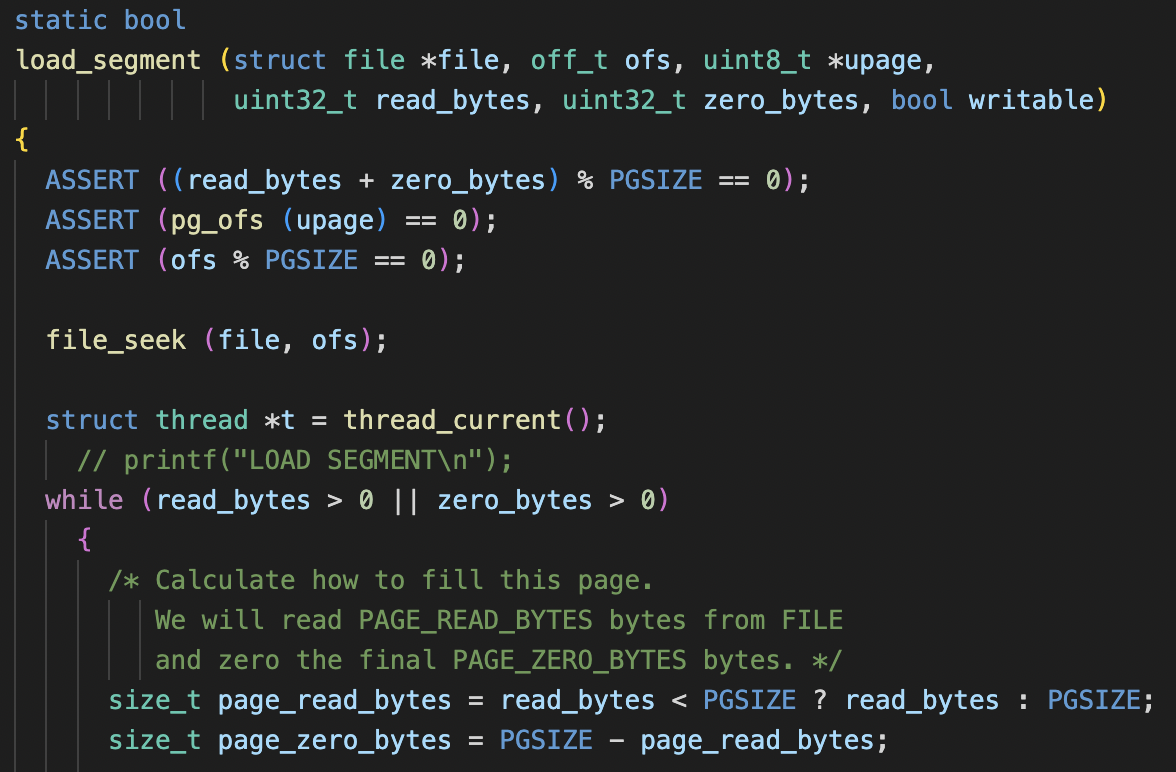
page list들을 관리하는 전역 변수 page\_list 를 초기화하는 함수를 추가했다.

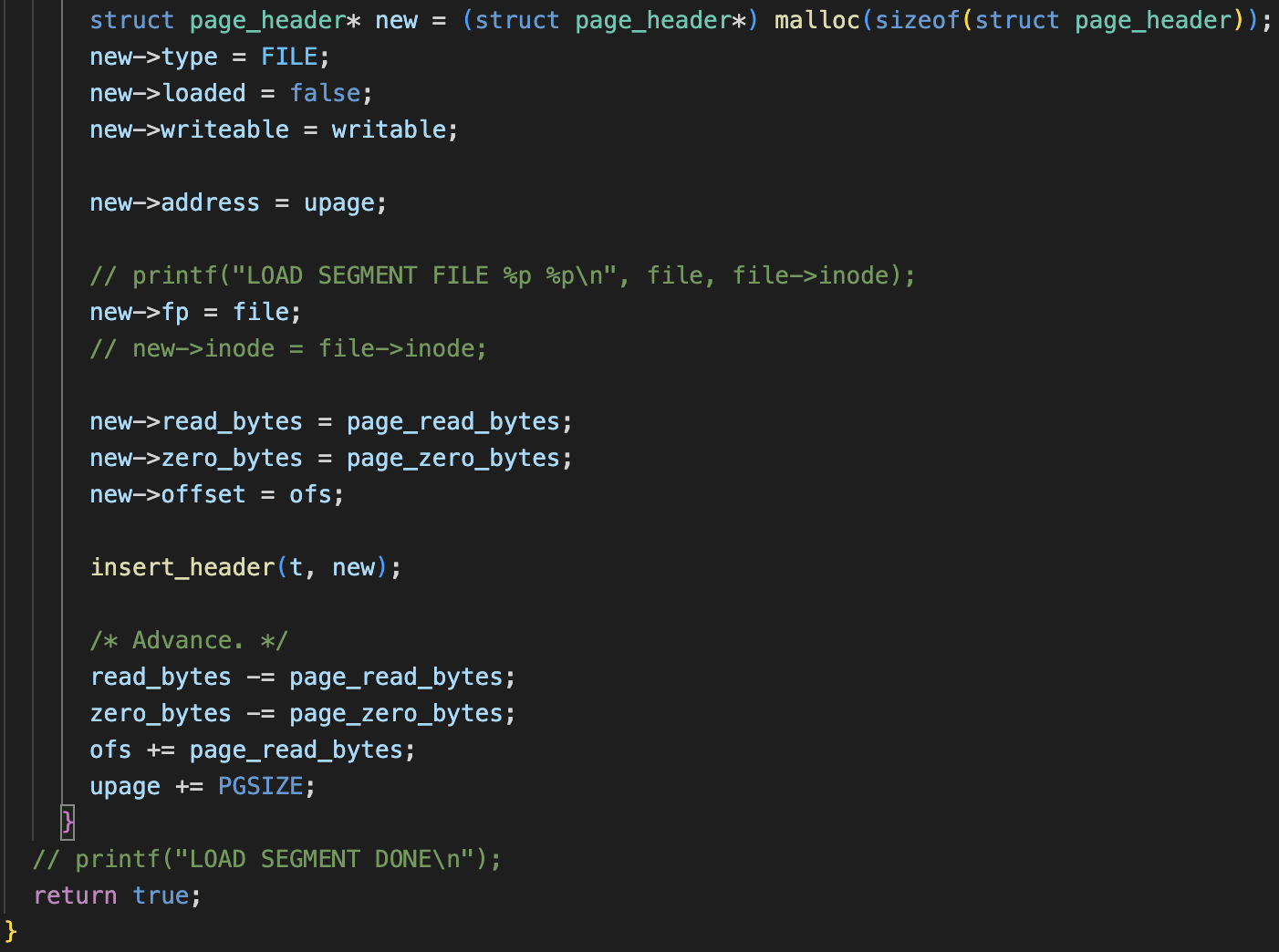


palloc\_get\_page 대신 사용할 alloc\_page 함수를 선언한다.

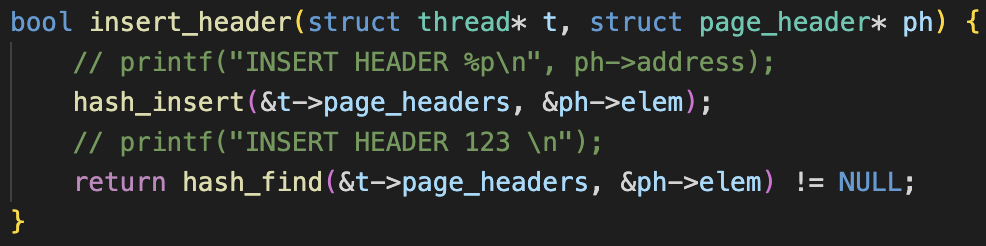


Load Segment 함수를 수정하였다. page alloc 하는 부분을 제거하고 page\_header를 생성하여 thread의 page\_headers hash에 추가해 주었다.

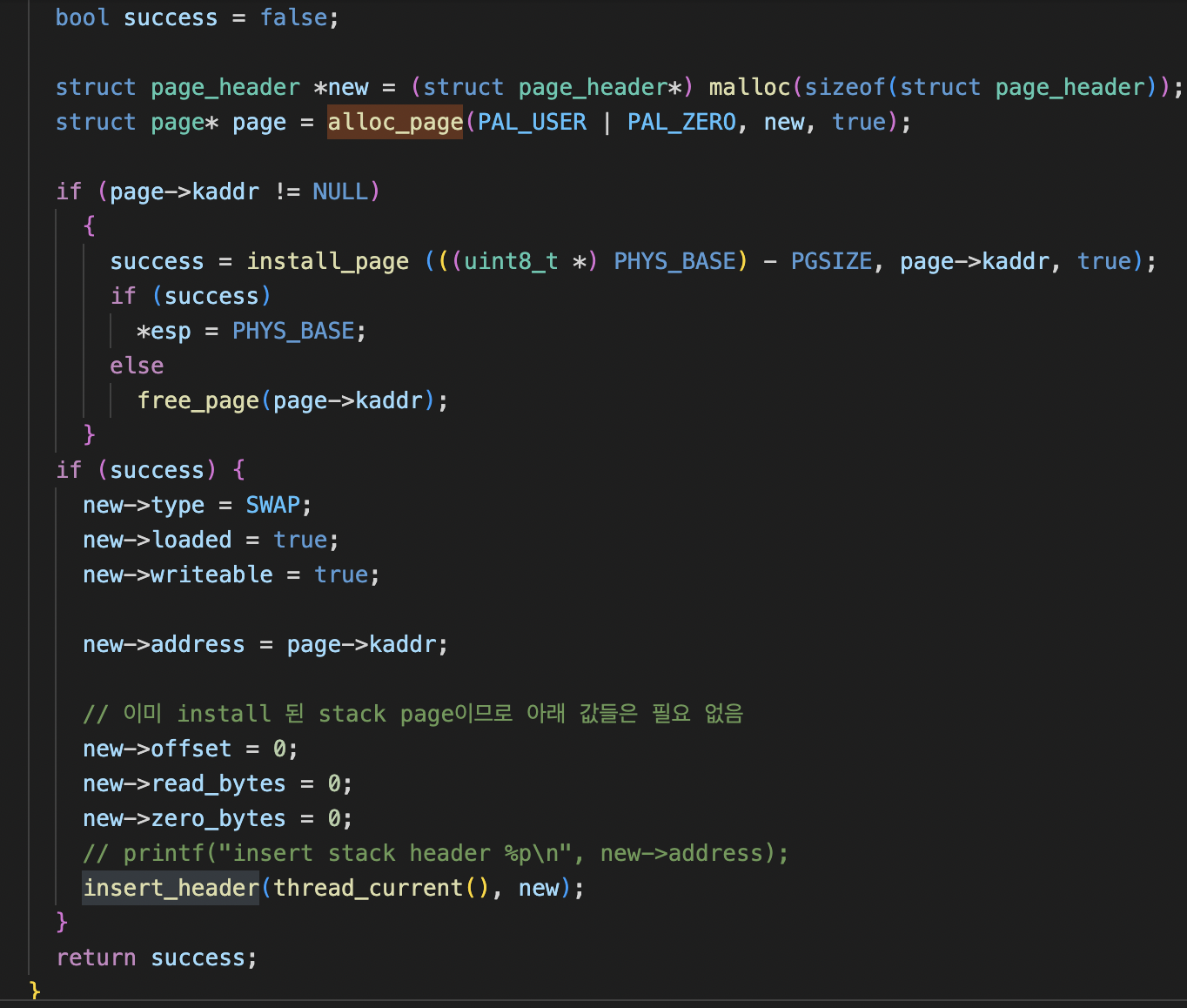




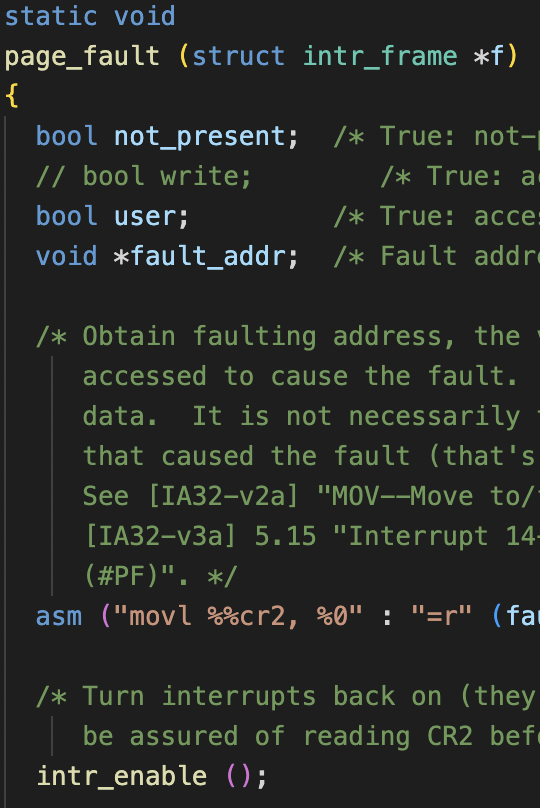
insert header는 다음과 같이 구현했다.



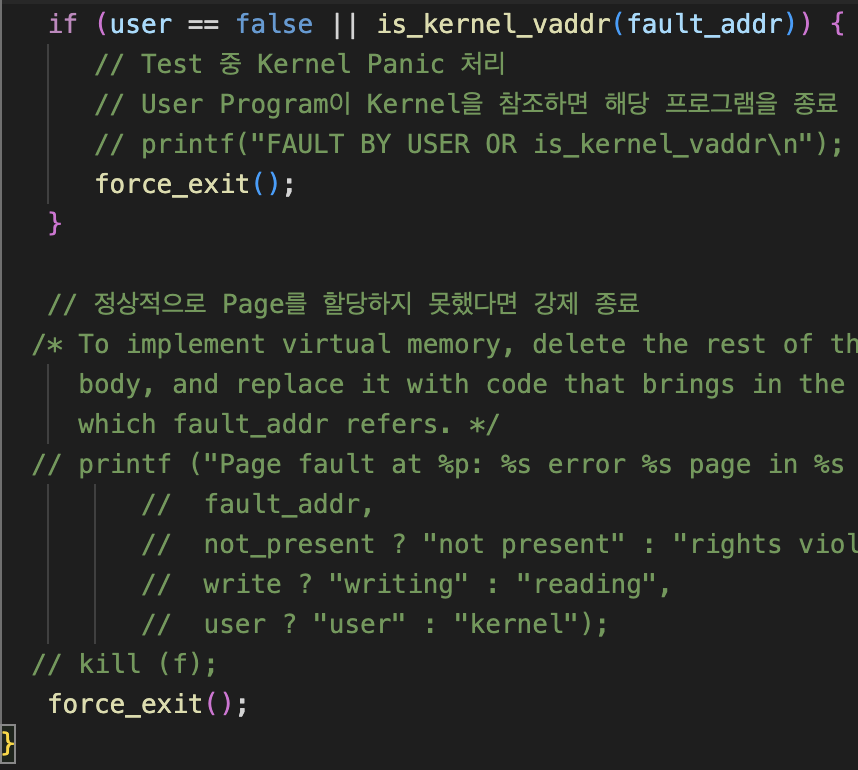
setup\_stack도 아래와 같이 수정하였다.



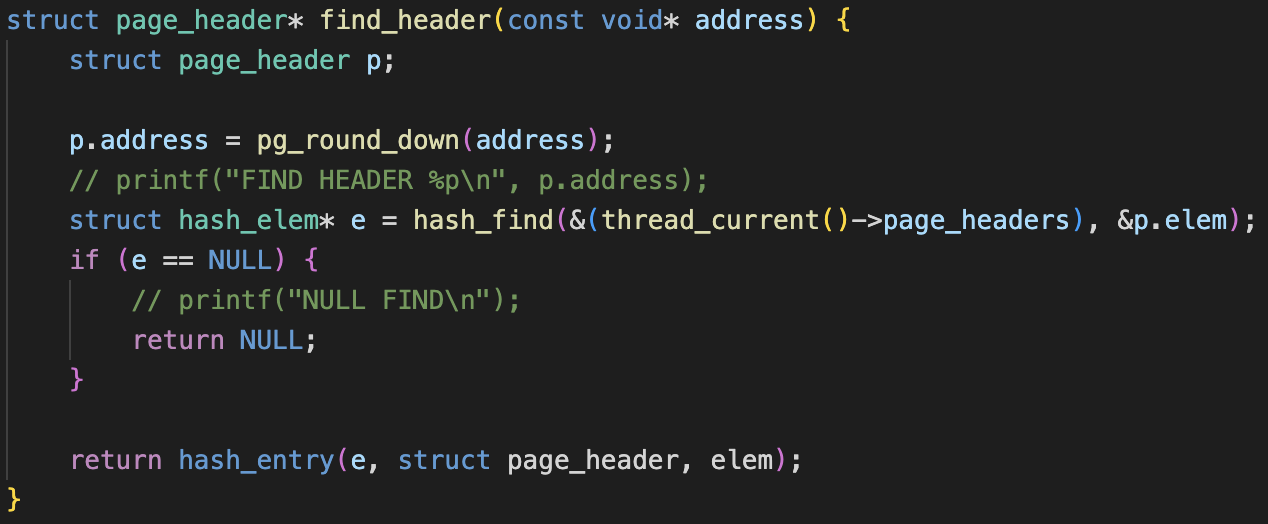
page fault를 handling할 page\_fault 함수는 아래와 같이 수정하였다.





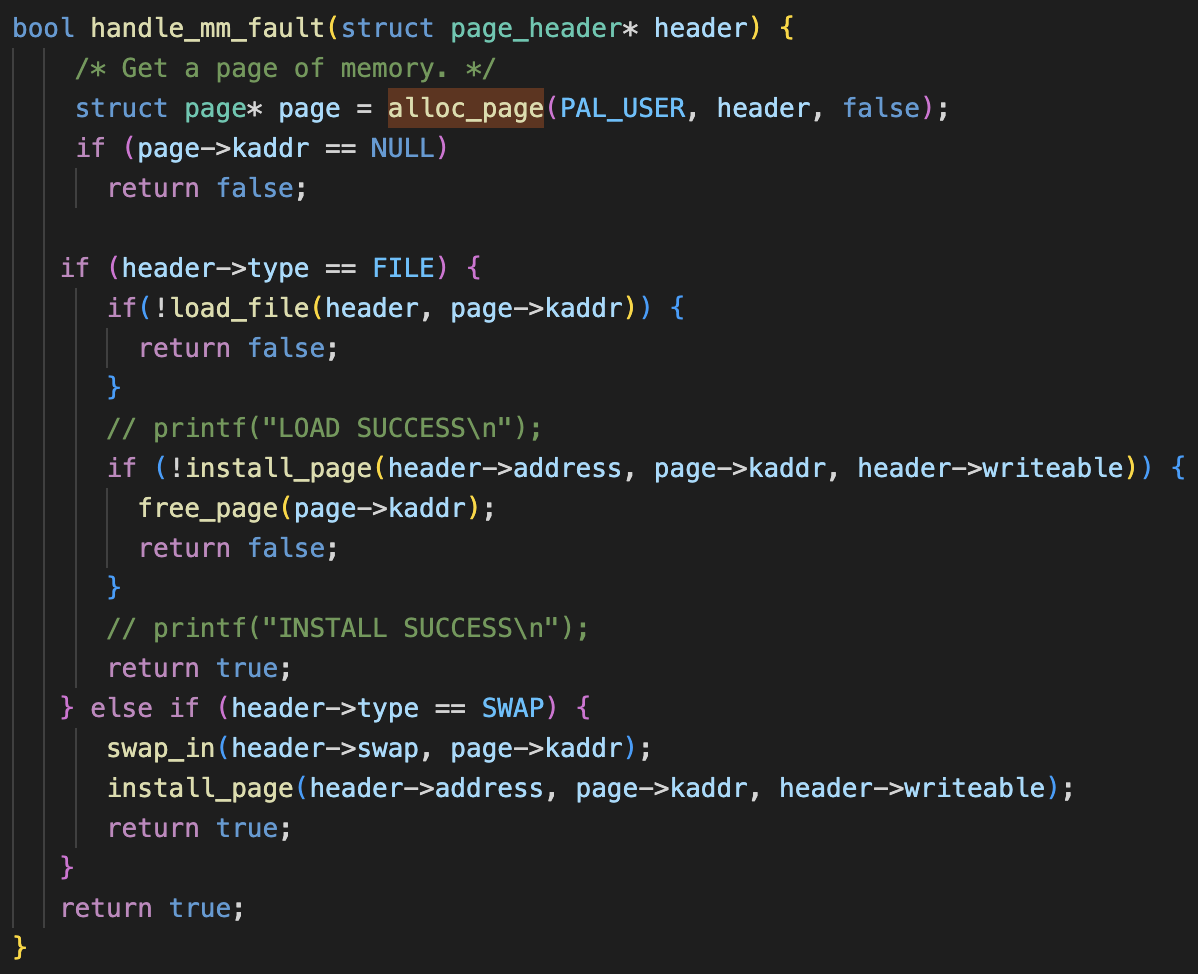


not present일 때 사용한 find\_header, handle\_mm\_fault, grow\_stack 함수는 아래와 같이 구현하였다.



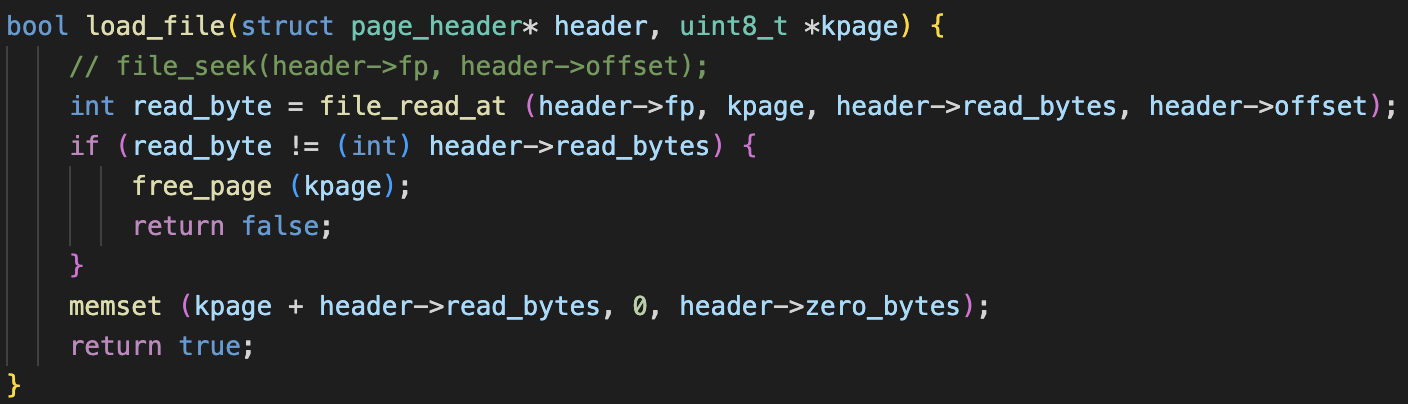
현재 thread의 hash buckets 안에 fault address 에 대한 정보를 담은 header가 있는지 확인한다.

handle\_mm\_fault 함수는 아래와 같이 구현하였다.



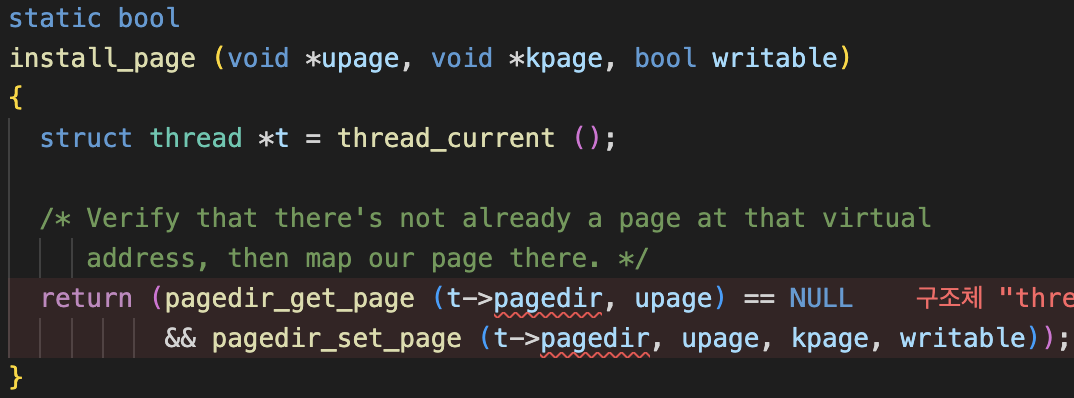
위에서 find\_header로 찾은 header에 type에 따라 페이지의 데이터를 다른 방식으로 로드한다.

load\_file 함수는 다음과 같이 구현하였다.



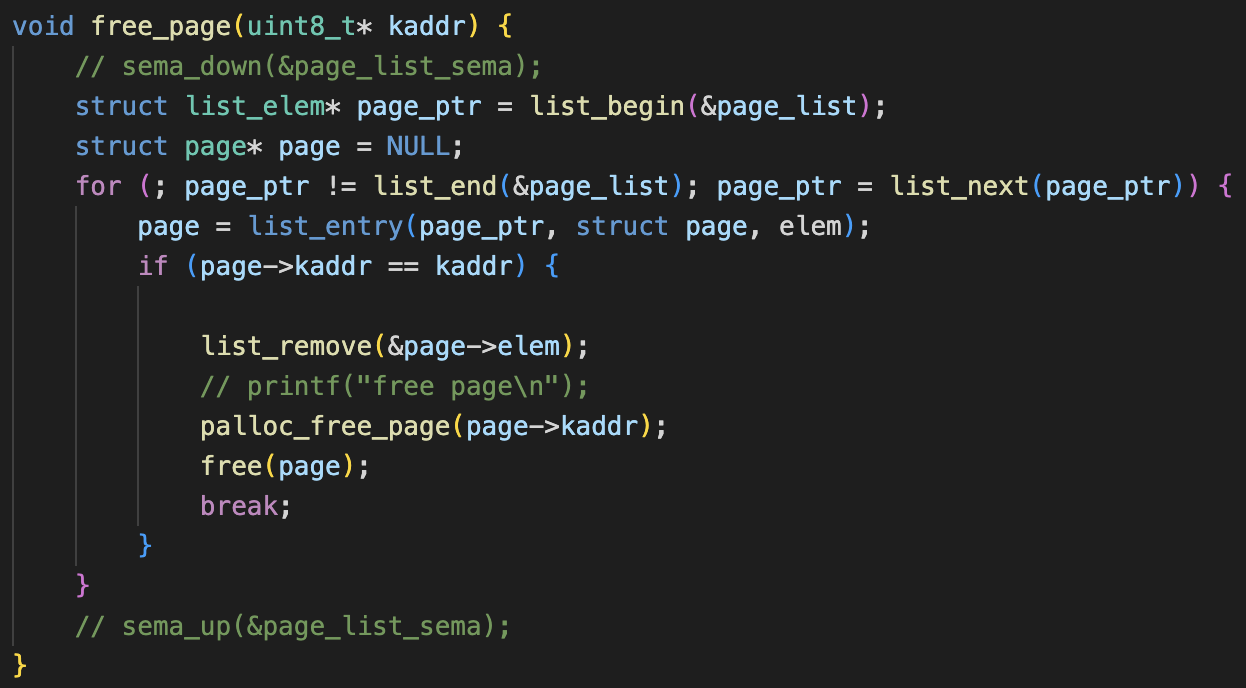
header의 정보를 통해 file\_read\_at 함수를 호출하여 파일에서 값을 읽어와 page에 저장한다.

install\_page 함수는 따로 변경하지 않으며 내용은 아래와 같다.



물리 페이지 (kpage)와 가상 페이지 주소(upage) 간의 관계를 매핑해주는 함수이다.

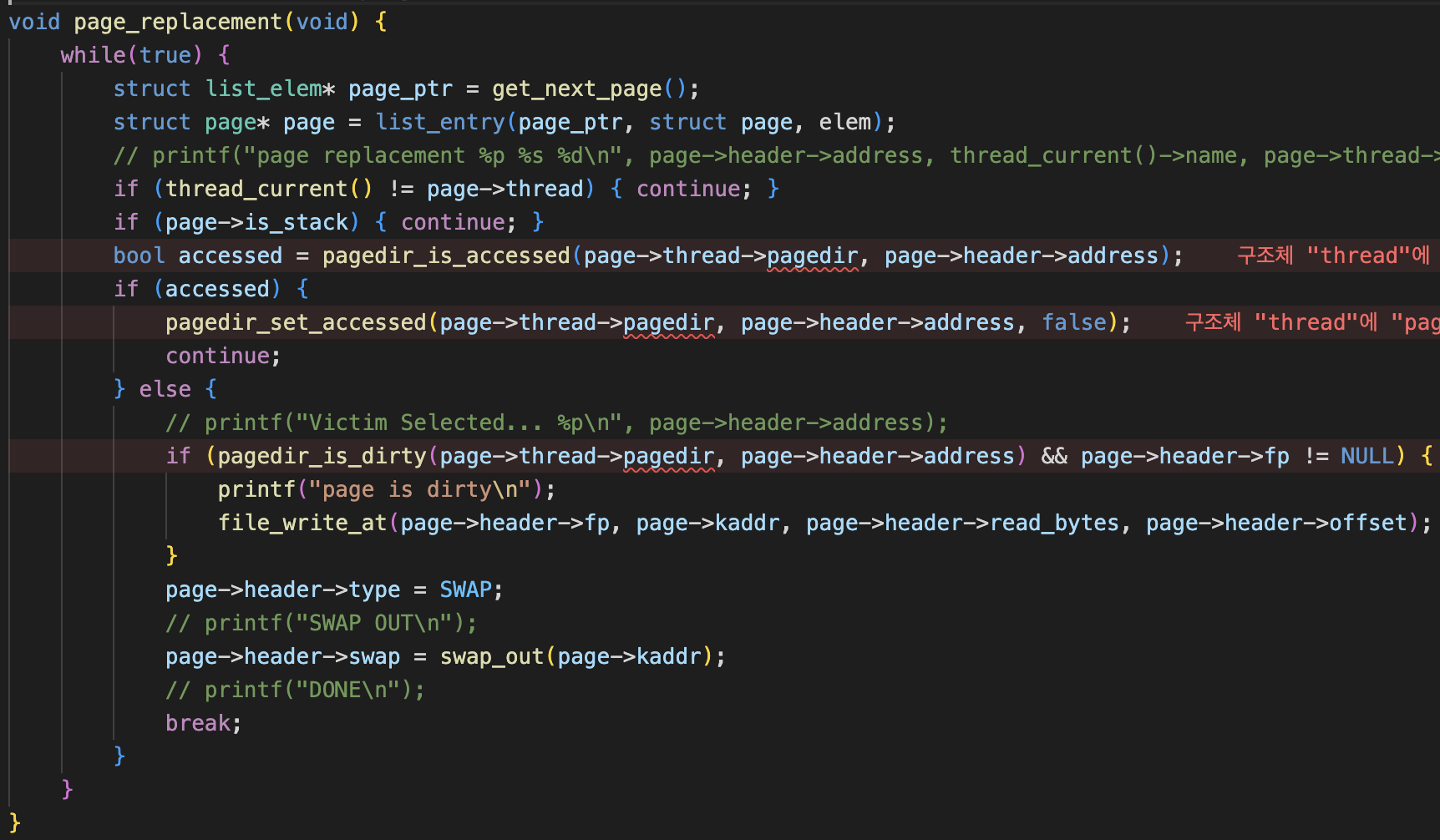
해당 과정에 문제가 발생했을 때 page를 반환하기 위해 사용한 함수 free\_page는 아래와 같이 구현하였다.



전체 page list를 순회하여 해당 page를 찾아 palloc\_free\_page를 호출한다.

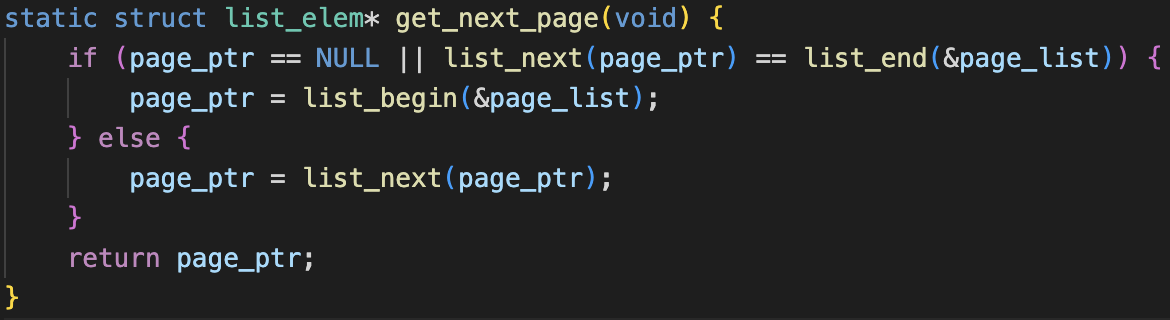
위에서 구현하고 사용한 alloc\_page 내부에서, palloc\_get\_page를 했을 때 page가 할당되지 않은 경우가 있다. 이 경우는 물리 메모리 공간이 부족하여 페이지를 할당하지 못한 경우인데, 이때는 위에서 본 것 처럼 page\_replacement 함수를 호출하여 페이지를 swap out 하고 다시 페이지를 할당한다.

page\_replacement 함수는 아래와 같이 구현하였다.



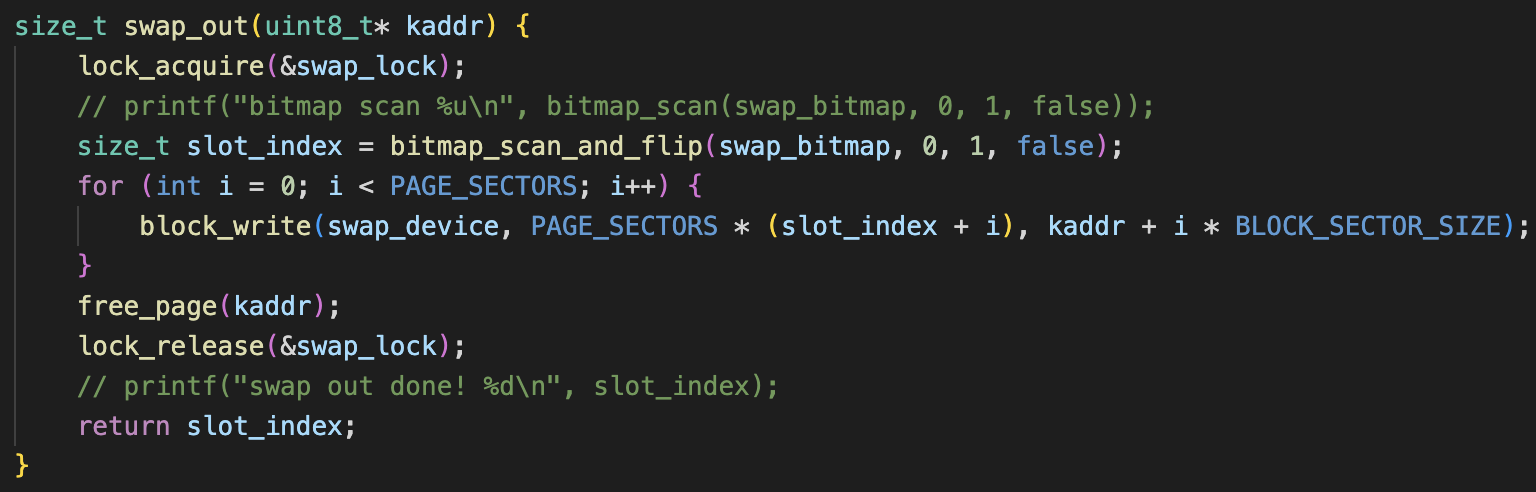
dirty bit과 accessed bit 검사와 변경을 위한 함수는 pagedir.c에 선언되어있던 함수들을 사용하였다.  
Second Chance Algorithm을 사용하여 victim page를 선정하였다.  
이후 test case를 test함에 있어서, 현재 thread가 아닌 다른 thread의 page를 swap out 하거나, stack page를 swap out 한 경우 프로그램이 비정상적으로 종료되는 문제가 있었다. 그래서 위 두 경우에는 해당 page를 victim으로 선정하지 않도록 구현하였다.

page list 순회를 위해 사용한 get\_next\_page 함수는 아래와 같이 구현하였다.



list의 끝에 도달하였을 때 다시 첫 번째 노드를 가져오게 하여 list를 circular하게 순회하도록 구현하였다.

swap\_out 함수는 아래와 같이 구현하였다.



swap\_bitmap을 검사하여 빈 슬롯을 찾고 해당 슬롯에 page의 정보를 block\_write를 사용하여 기록해주었다.

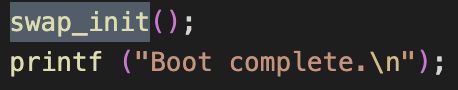
위 함수에서 사용한 swap\_bitmap은 swap.c에 선언한 bitmap type의 전역 변수이다.  
swap\_device는 swap\_disk이며, swap.c에 전역 변수로 선언되어 있다.  
아래와 같이 선언하고 초기화하였다.



PAGE\_SECTORS 라는 매크로를 추가하였는데, 이는 페이지의 값을 저장하기 위해 사용해야 하는 BLOCK SECTOR의 수이다. swap out 시 해당 값을 이용하여 page의 값을 swap disk에 저장하였다.

또한 swap\_init 함수에서 위에서 구현하였던 page\_list\_init 함수를 호출하였고, 해당 swap\_init 함수는 pintos 시작 시 호출하도록 구현하였다.

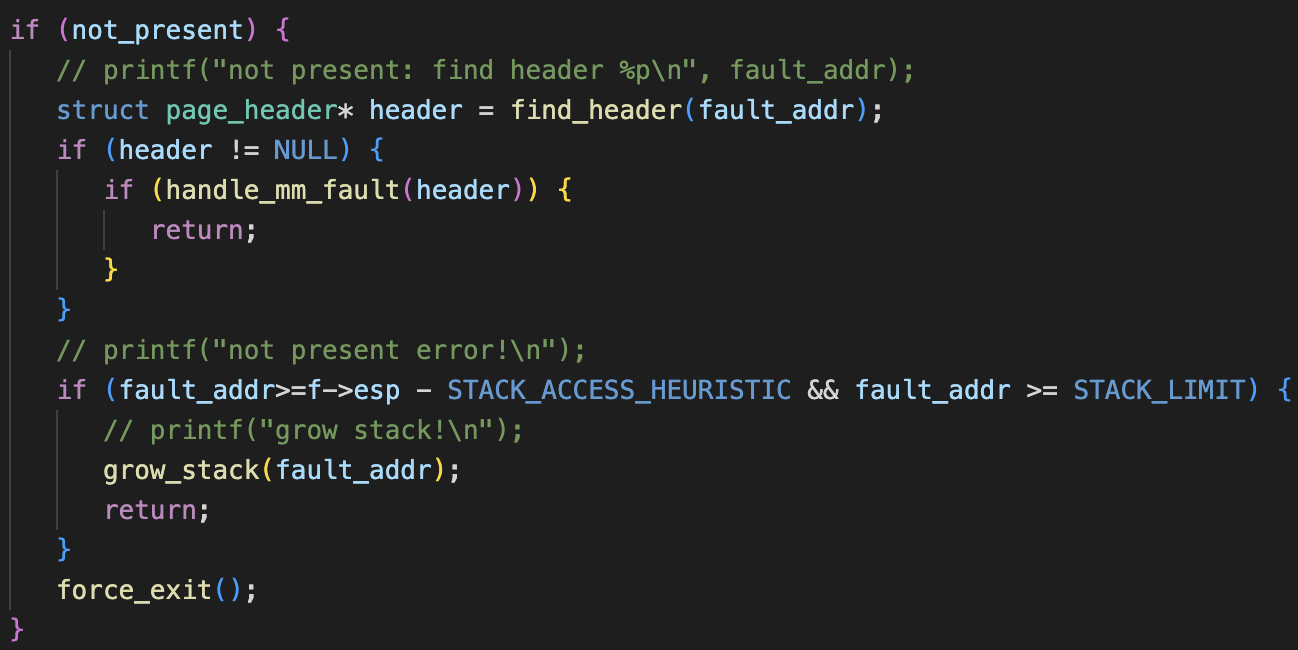
아래와 같이 init.c의 int main() 함수에 함수 호출을 추가하였다.

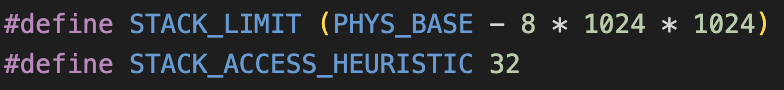


handle\_mm\_fault에서 사용한 swap\_in 함수는 아래와 같이 구현하였다.

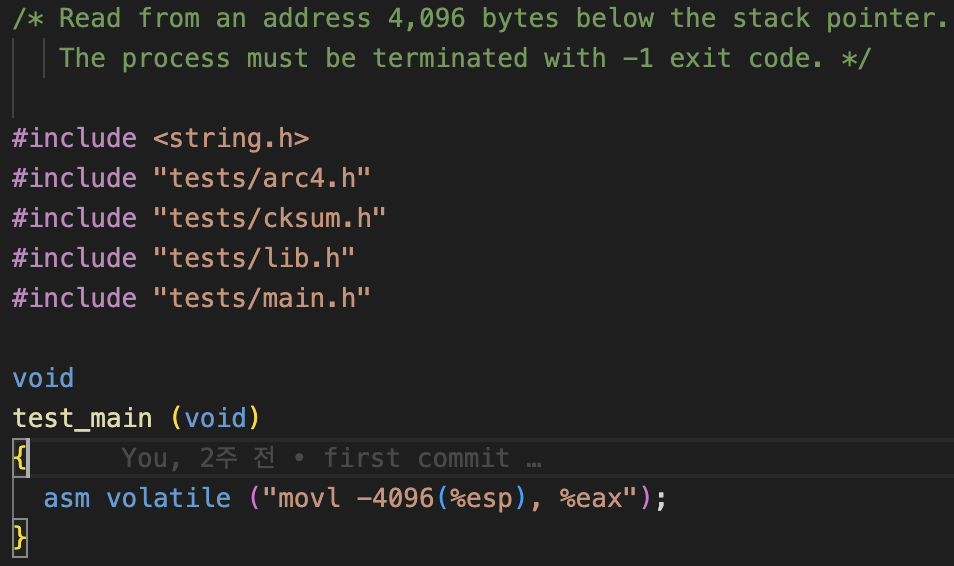


위와 같이 block\_read 함수를 이용하여 swap disk에 있는 정보를 page에 할당하도록 구현하였다.

find\_header, handle\_mm\_fault 함수와 이에 대한 상세 내용들은 위에서 확인하였다.  
grow\_stack 함수를 확인하기 전에 우선 grow\_stack이 호출되는 부분부터 다시 보면  


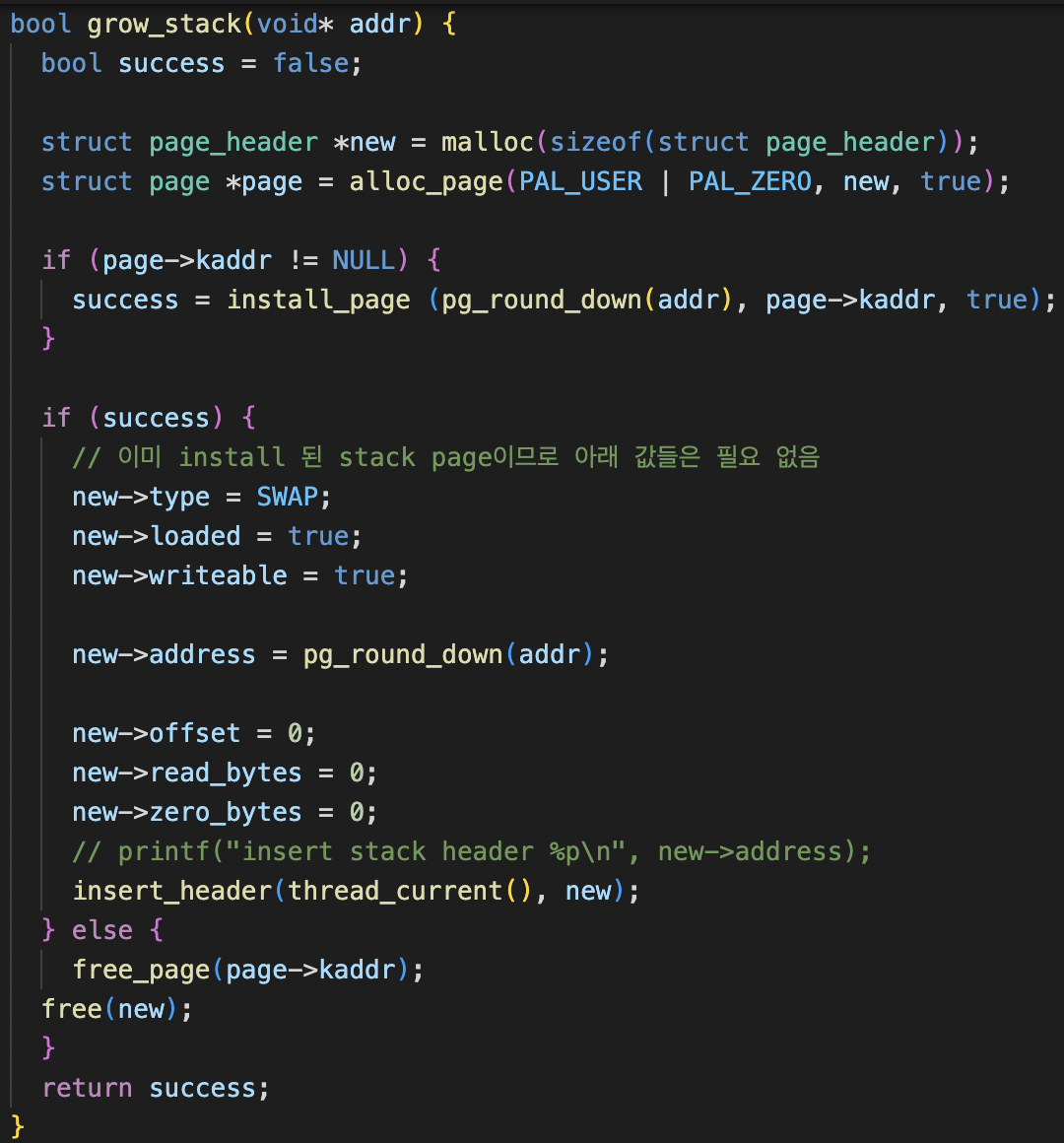
위처럼 우선 fault\_address에 대해 page\_header가 없을 때 stack growth case인지 검사하게 된다.  
Stack Growth case 판별을 위해서는 우선 fault address가 Stack의 크기 제한 이내의 접근인지 확인하여야 한다.  
ppt에 보통의 GNU/Linux OS의 Stack 크기 제한이 8MB라고 명시되어 있었으므로, 스택 최대 크기 제한도 아래와 같이 정의하였다.  


또한 메뉴얼에 따르면 해당 접근이 Stack에 대한 접근이 맞는지 휴리스틱을 통해 판단해야 한다는 부분이 있었다.  
이 부분을 test하는 case가 pt-grow-bad인데, 테스트 코드를 보면 아래와 같이 접근한다.



위 case를 통과하기 위해 휴리스틱을 32로 정의하고 stack 접근인것처럼 접근하는 경우만 grow stack을 호출하게 하여 해당 case를 통과하였다.

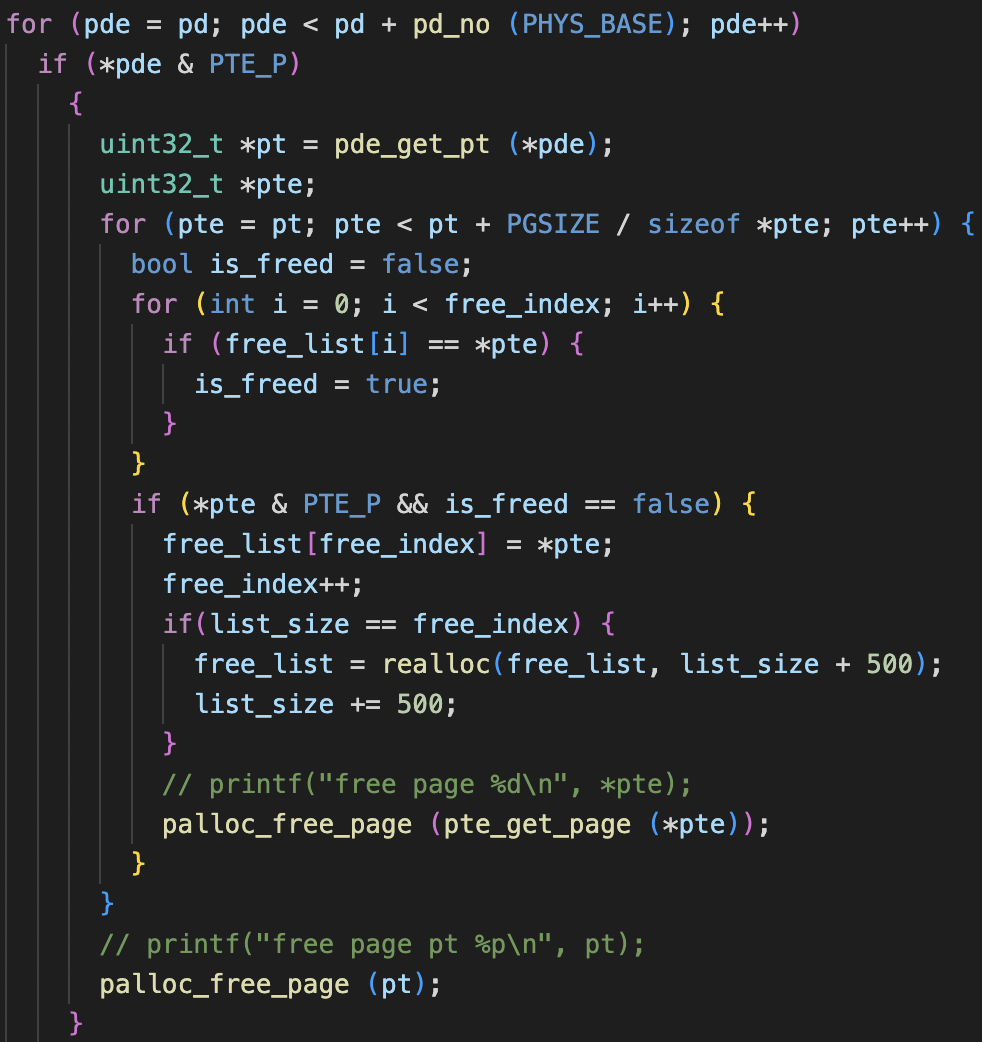
grow stack 함수는 아래와 같이 구현하였다.



page를 할당하고 접근하려 했던 fault\_address에 install page를 통해 페이지를 매핑시켜 stack growth를 구현하였다,

여기까지 구현하고 test 했을 때 page-parallel 을 통과하지 못하는 문제가 있었다.  
그래서 디버깅해보니 process\_exit에서 pagedir\_destory를 처리하는 중 문제가 발생함을 알게 되었고, 정확한 원인은 모르겠으나 alloc\_page와 free\_page를 통해 페이지를 할당하고 해제하는 과정을 내가 직접 구현하게 되면서 기존 page 처리 구조와 충돌이 발생하여 이미 해제했던 페이지를 다시 해제하려고 하는 문제가 발생했다.

그래서 page\_destory 함수를 아래와 같이 수정하였다.



해제한 pte 값들을 저장하는 배열을 선언하여 해제 후 값을 추가하였고, 이후 동일한 pte에 대한 페이지를 해제하려고 시도하면 is\_freed flag를 검사하여 동일한 페이지에 대한 중복 해제가 실행되지 않도록 처리하였다.

* 1. **시험 및 평가 내용**
* (채점 대상 테스트 케이스에 해당하는) make check 수행 결과를 캡처하여 첨부

