**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재 교수님

이름 / 학번 : 20211584 장준영

개발 기간 : 2023-11-28 ~ 2023-12-10

1. **개발 목표**

프로젝트 3까지의 pintOS는 메모리에 대한 특별한 처리를 하지 않아도 되는 케이스만 처리했다. 즉, 메모리를 page 단위로 나눠 사용하긴 했지만 사용 방법에 대한 구현 없이 그대로 필요한 만큼 로드하기만 했다. 또, 만약 page fault가 발생하더라도 에러와 함께 프로세스를 종료했다. 이번 프로젝트 4에서는 메모리 가상화의 활용을 극대화하기 위해 paging을 제대로 구현한다. pintOS가 기본적으로 제공하는 page table에 필요한 여러 정보를 추가해 새로운 page table을 마련하고, 효율적인 메모리 사용을 위해 lazy loading을 적용한다. 이를 위해선 page fault handler에서 page fault 시에 필요한 데이터를 디스크에서 메모리로 옮기는 swapping도 구현해야 한다. Page fault는 unmapped page를 조회할 때만 발생하는 것이 아니고, 스택 영역 밖을 조회했을 때도 발생할 수 있다. 이 경우 프로세스 스택을 늘리는 stack growth 과정이 필요하다. 이 또한 구현하면 paging을 완성할 수 있다.

1. **개발 범위 및 내용**
   1. **개발 범위**

**1. Page Table & Page Fault Handler**

기본적으로 pintOS가 제공하는 page table은 담고 있는 내용이 너무 단순하다. Demand paging을 위해선 page가 디스크에 있는지, 메모리에 있는지, 어떤 파일과 mapping되어 있는지 등의 정보를 추가해 새로운 page table을 만들 것이다. 이후 page fault handler에서는 lazy loading 및 swapping 과정이 구현된다. 메모리의 효율적인 loading을 위해, page fault가 발생했을 때 필요한 page를 메모리로 loading하고 fault가 발생한 instruction을 재실행할 수 있도록 구현한다. 이 과정에서 page table을 사용할 수 있다. 이렇게 하면, demand paging을 위한 기본적인 준비 단계를 마칠 수 있다.

**2. Disk Swap**

메모리 가상화를 통해 여러 프로세스가 하나의 물리적 메모리를 공유하기 때문에, 물리적 메모리가 가득 차 free frame을 유저 프로세스에게 제공할 수 없는 상황이 발생할 수 있다. 이 경우 더 이상 참조되지 않을 것 같은 evict page를 찾고, 이를 disk의 swap space로 보내는 page replacement 과정을 통해 자리를 마련해야 한다. 이후 swap space에 있는 page를 참조해야 할 땐 다시 메모리의 영역으로 불러오는 과정도 필요하다. 이를 memory/disk swapping이라고 한다. 꾸준히 언급하고 있는 demand paging을 효율적으로 활용하기 위해 필요한 과정이다.

**3. Stack Growth**

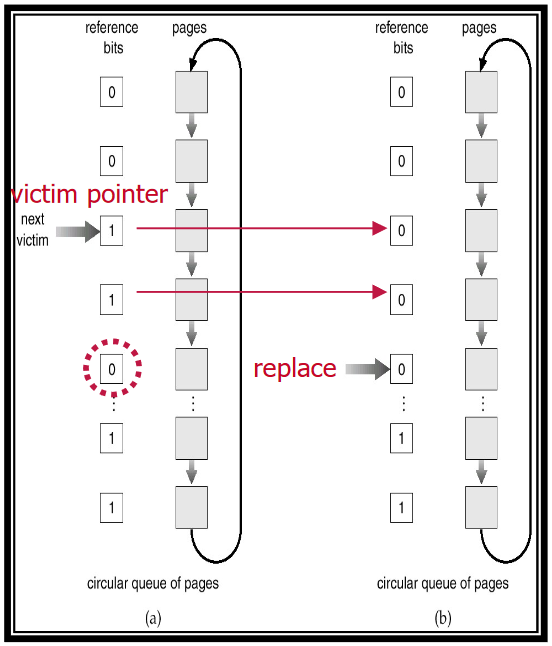
Page fault가 발생해 page fault handler가 실행됐는데, 그 주소가 프로세스의 세그먼트 중 스택 영역에 해당하는 경우 스택을 확장해야 한다. 이 과정은 앞서 설명한 page fault handling 과정과 다를 바가 없다. 프로세스가 요구하는 바에 따라 스택 세그먼트에 새로운 frame을 부여함으로써 필요로 하는 메모리를 OS가 최대한 제공하는 과정이다.

* 1. **개발 내용**

**1. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술**

프로세스가 메모리에 접근하면, 이 접근은 가상 메모리 주소에 대한 접근으로 처리된다. 넘겨 받은 가상 메모리 주소에서 VPN(Virtual Page Number)를 추출하여 해당 page가 page table에 존재하는지 확인한다. 존재한다면 해당 VPN에서 PFN(Page Frame Number)로의 관계 정보가 있으므로, 그대로 물리적인 주소를 알아내 메모리의 특정 위치에 접근할 수 있다. 하지만, 존재하지 않는 경우 VPN으로 해당하는 physical frame을 찾을 수 없다. 즉, 프로세스의 virtual address space 내 가상 주소가 가리키는 곳에 physical frame이 부재하는 상황이다. 이 경우 page fault가 발생하여 MMU exception과 함께 OS에서 page fault handler 루틴을 시행하게 된다. Handler에선 레지스터로 전달된 주소값을 받아와 해당 주소의 VPN을 확인한다. 이후, 해당 page가 valid한지 확인한다. 만약 valid 하다면, page table entry가 존재하지만 page table에 없을 뿐이므로 디스크에서 해당하는 데이터를 찾아 메모리로 올리고 그 frame을 page table에 업데이트한다. 이후, 메모리 접근을 하는 instruction을 다시 실행한다. 만약 invalid 하다면, page fault가 발생할 수 있는 또 다른 경우인 스택 공간을 확인한다. 스택 공간을 확장했을 때 그 안에 해당 주소 영역이 포함된다면 단순히 스택 영역이 작아 발생한 page fault이므로, stack growth를 통해 해결한다. 이 모든 경우에 해당하지 않는 page fault라면, segmentation fault이므로 프로세스를 중지해야 한다.

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



Page replacement algorithm으로는 pintOS 매뉴얼에서 권장하는 방법인 second chance algorithm(이하 SCA)을 사용할 것이다. 최적의 알고리즘인 Least Recently Used algorithm을 근사적으로 구현한 방법으로, 더 이상 쓰이지 않을 것으로 추정되는 페이지를 적은 오버헤드로 추정해 교체할 수 있다. SCA는 이름 그대로 두 번의 기회를 주는 방식으로 동작하는데, 교체의 대상이 첫 번째로 됐을 때는 한 번의 기회를 주고, 두 번째로 됐을 때 교체함으로써 최근에 사용된 page를 지켜낸다. SCA에선 page를 linked list 형태로 관리하며, victim pointer, reference bit을 사용하여 evict할 page를 추정한다. Reference bit는 1로 초기화된다. Page fault handler가 동작하며 page를 교체할 때가 되면, victim pointer가 linked list를 순회하며 교체될 page를 찾는다. 만약 가리키고 있는 page의 reference bit가 1이라면 이를 0으로 바꾸고(기회를 한 번 주는 과정) 다음 page로 넘어간다. Reference bit가 0이라면 이미 한 번의 기회를 사용한 page이므로, 교체한다. 위의 이미지는 강의 자료에서 SCA에 대해 설명하고 있는 그림으로, victim pointer가 reference bit가 1인 두 page를 지나쳐 0인 page를 교체하는 것을 볼 수 있다.

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

Page fault 시에 stack growth를 해야하는 경우가 있다고 이전에 설명하였다. 이는 demand paging 방법에선 공간 효율을 위해 초기 스택 영역을 작게 잡기 때문에 발생한다. Page fault가 이러한 이유로 발생했다면, 1) 주소값 그 자체가 유저가 사용하는 데에 문제가 없어야 하고, 2) 스택 영역에 존재해야 한다. PintOS 매뉴얼을 살펴보면, 이러한 상황이 발생할 때의 조건이 설명되어 있다. 우선 당연히 주소값이 유저 스페이스 내에 존재해야 하고, pintOS 환경인 80x86 시스템에서 esp 포인터에 asm ‘pusha’ 연산을 했을 때 fault check를 하는 영역 내에 있어야 한다. 또, pintOS 매뉴얼에선 스택의 최대 사이즈를 8MB로 권장하고 있으므로, 스택의 시작 주소로부터 8MB 내에 존재하는 지도 확인해야 한다. 이러한 조건을 모두 충족한다면, page fault가 스택 영역이 작아서 일어난 것이므로 스택을 확장해주어야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

프로젝트를 하려고 pintOS 매뉴얼을 읽어보니 가상 메모리에 대한 개념이 너무 약하다는 것을 알게 되었다.. 우선 수업시간에 배운 가상 메모리와 paging에 대한 개념을 복습하고, 이후 pintOS 매뉴얼을 읽어 구현해야 될 부분에 대해 익혔다. 가장 먼저 page table과 page table handler를 구현하였다. 이후 disk swap 부분을 건드렸는데, 생각보다 까다로운 부분이 많았다. 특히 synchronization을 신경써야 된다는 것을 망각하여 디버깅에 꽤 오랜 시간을 쓰게 되었다. 또, 생각보다 상당한 양의 자료 구조를 스스로 구현해야 했기 때문에, 이 부분에서 많은 노력과 시간을 들였다. 마지막으로 stack growth를 구현하였다. 이후 까다로운 몇 개의 case를 통과하기 위해 자료 조사와 디버깅을 하여 구현을 끝마쳤다. 자세한 내용은 아래 연구 결과에서 설명하겠다.

* 1. **개발 방법**

기본적으로 pintOS 매뉴얼에서 프로젝트를 진행해야 하는 방향을 제시하기 때문에, 이를 충실히 따를 예정이다. 만들어야 하는 파일, 함수나 구조체, 수정해야 하는 파일 위치 등이 자세하게 매뉴얼에 나와있기 때문에, 어떤 파일을 왜 그렇게 수정하는 지에 대해서는 자세히 설명하지 않고 구현한 방법에 대해서만 설명하겠다. 설명이 생략된 부분에 대해서는 모두 pintOS 매뉴얼을 따랐다.

**1. Page Table & Page Fault Handler**

PintOS에서는 vm 디렉토리에 새로운 코드 파일을 만들 것을 권장한다. 그에 따라, page table과 관련된 함수 구현은 vm/page.\* 파일을 만들어 하였다. 추가적인 정보를 담은 page table의 자료구조는 hash table이다. 이 hash table의 자료형은 새로 선언한 구조체로, page table에 담고싶은 정보를 저장한다. 여기엔 VPN 정보, 메모리에 있는 페이지인지를 확인하는 flag, page replacement 때 참고해야 하는 page의 정보 등을 담는다. 이후 프로세스마다 새로운 page table을 가질 수 있도록 userprog/thread.h와 userprog/process.c에서 선언 및 초기화 과정을 거친다. 새로운 page table을 만들기 위해 hash 자료구조를 사용하였는데, 이는 page table operation의 속도를 최대한 빠르게 하기 위함이다. 이후, 그 page table operation(insert, delete, …)를 page.c에 함수로 구현하여 사용한다. Page table을 위한 준비를 마쳤다면, 이제 파일을 page 단위로 나눠 page table을 실제로 사용해야 한다. 이를 관장하는 userprog/process.c의 load\_segment 함수를 수정해 프로그램을 page 단위로 나눠 page table entry로 만들고, 이를 page table에 삽입하는 과정을 추가한다. 뿐만 아니라, stack segment 영역의 파일 역시 이 과정을 거치도록 setup\_stack 함수도 비슷하게 수정해야 한다.

마지막으로, page fault handler를 구현하기 위해 userprog/exception.c의 page\_fault 함수를 수정한다. 지금까지 계속 설명해온 page fault handling 과정을 삽입한다. 이 부분에서 userprog/process.c에 mm\_fault\_handler나 expand\_stack과 같은 subroutine을 추가하여 사용한다.

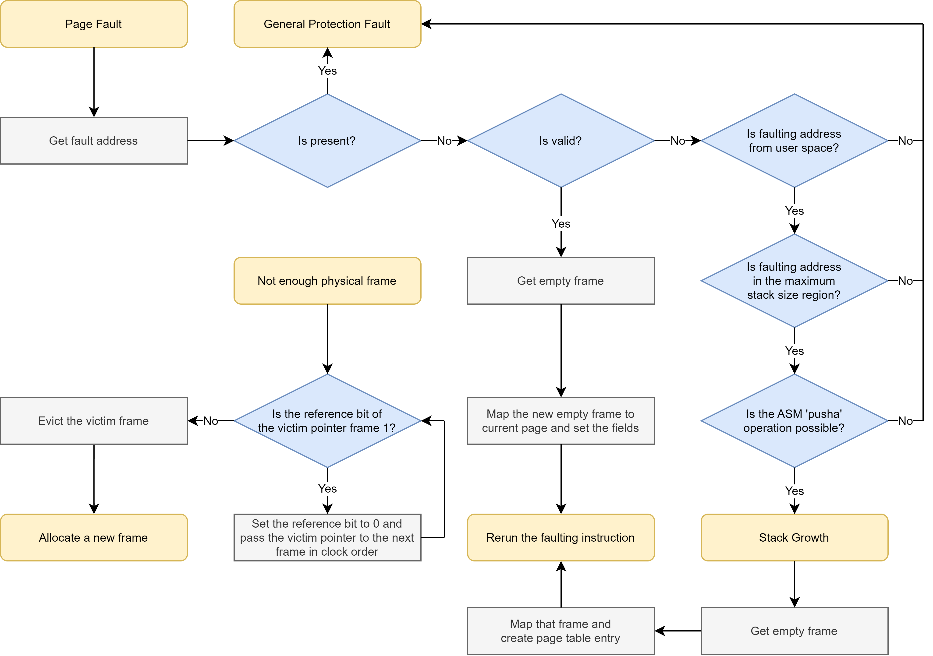
**2. Disk Swap**

역시 vm 디렉토리에 frame.\*, swap.\* 코드를 추가하여 사용한다. frame.\*에는 page replacement, swapping 등을 위해 할당된 frame을 저장하여 관리할 수 있는 frame table의 구현이 들어있다. 가장 많은 operation을 담당할 SCA를 위해 frame table은 (linked) list로 관리한다. page.c에서와 마찬가지로 frame table operation을 함수로 작성하고, SCA를 위해 victim pointer를 순환시키는 함수나 frame eviction을 위한 함수도 작성한다. Frame table이 완성되면, vm의 test case를 통과하기 위해 threads/palloc.h에서 제공하던 메모리 할당 함수를 frame table을 이용해 다시 작성한다(allocate page, free page utils). 이를 통해 frame이 가득 찬 경우 page replacement가 실행되고, evict 된 frame이 free되면서 swapping을 지원할 수 있다. swap.\*에는 swapping을 실질적으로 실행하는 함수가 작성되어 있다. pintOS 매뉴얼을 보면, swap slot을 bitmap 자료구조로 관리할 것을 권장한다. 따라서, bitmap으로 swap slot을 관리하며 block.h의 시스템 함수를 이용해 swapping의 전반적인 과정을 이행하였다.

**3. Stack Growth**

앞서 설명한 스택 영역 판정 기준을 확실히 이해하고 나면, 이 부분의 구현은 어렵지 않다. 복잡한 판정 과정을 is\_user\_vaddr 함수나 PHYS\_BASE 연산 등을 이용해 구현하면, 그 이후 상황에 맞게 스택을 키우면 된다. 먼저 threads/vaddr.h의 pg\_round\_down을 사용하여 사용 가능한 VPN을 얻고, 그에 따른 새로운 page entry를 만들수 있다. 새로 생성한 page entry에 frame을 매핑하고 이를 삽입하기만 하면 된다. 주의할 점은, syscall routine 등 stack growth가 필요한 모든 시점마다 이 판정을 해주어야 한다. 이는 후술한다.

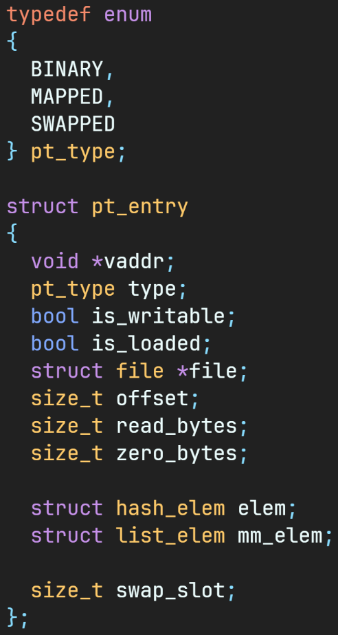
1. **연구 결과**
   1. **Flow Chart**



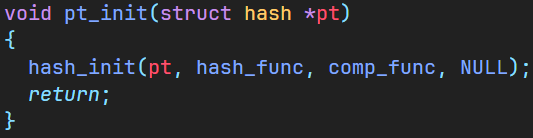
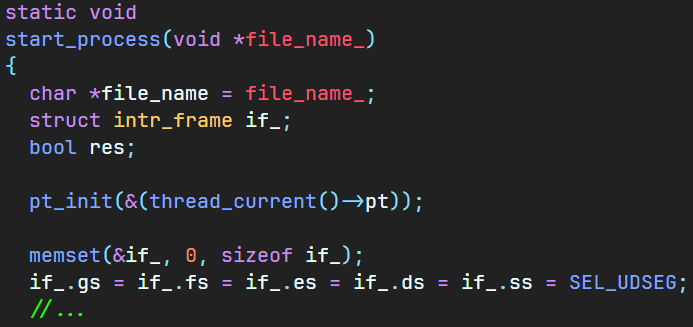
(※ Page fault handler, Page replacement algorithm, Stack growth 세 가지를 한 이미지에 정리하였음.)

* 1. **제작 내용**

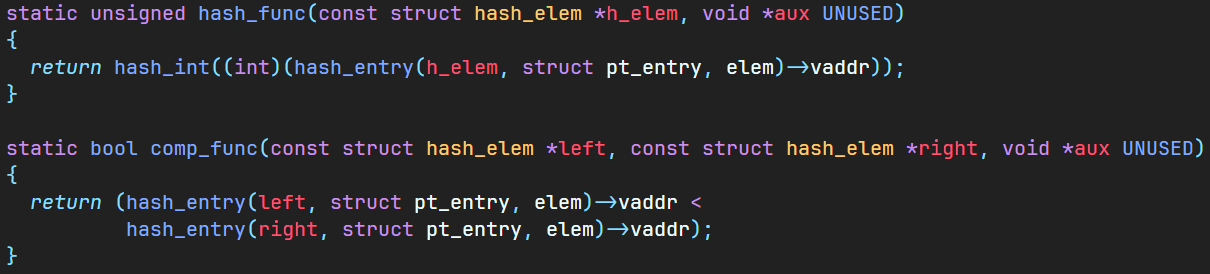
우선 vm/page.\*의 함수를 보자.



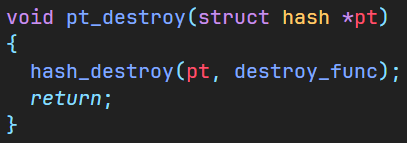
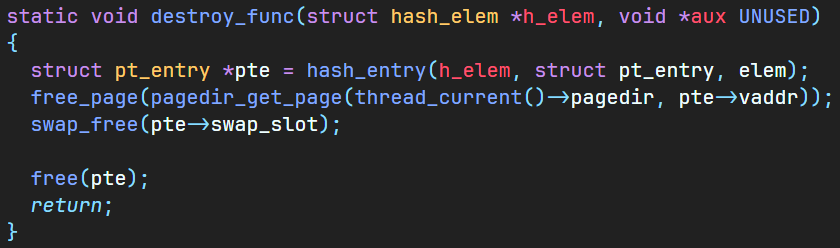
page.h에 정의되어 있는 page table entry인 pt\_entry이다. 필드로는 필요한 정보를 담고 있다. 이는 모두 이후에 설명할 page table operation에 적극적으로 활용된다. page type(pt\_type)은 세 가지가 있는데, BINARY는 일반적인 binary file에서 생성된 page를, MAPPED는 memory mapping 과정에서 생성된 page를, SWAPPED는 stack segment 영역의 page나 이미 swap 된 적이 있는 page를 의미한다. 이는 이후 여러가지 루틴에서 활용된다. 다음은 page.c이다.

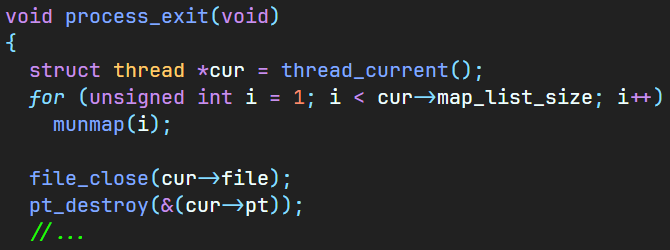
 

Page table의 자료구조로 hash를 사용하기 때문에, 사용 이전에 초기화해주어야 한다. page table이 thread마다 존재하기 때문에 threads/thread.c의 start\_process에서 호출하여 초기화 할 수 있다. (이를 위해, threads/thread.h의 struct thread 내에 pt 필드를 만들어주어야 한다.)

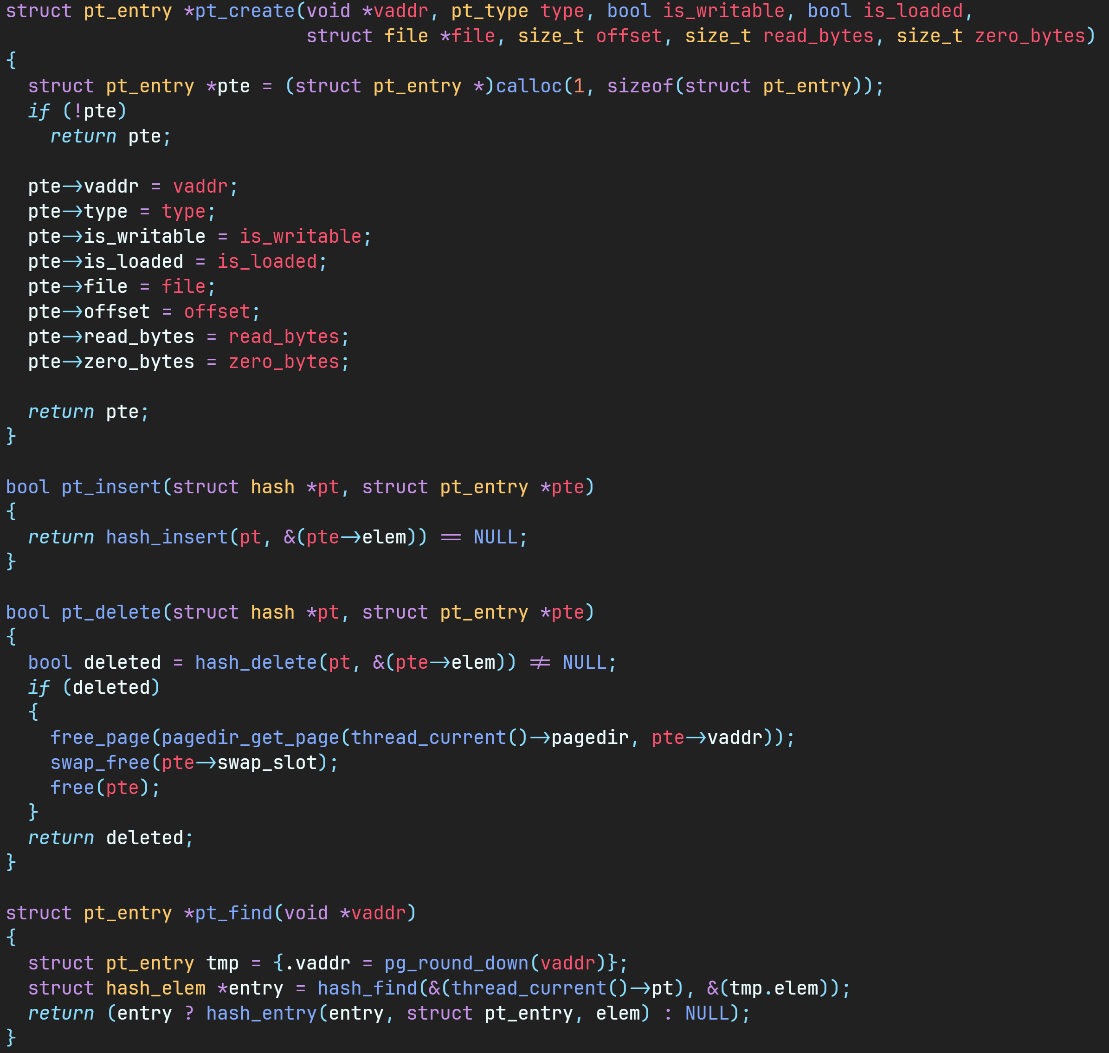


초기화에 사용되는 hash\_func와 comp\_func는 위와 같다. 단순히 vaddr을 통해 해시 값을 구하고, vaddr 값을 통해 대소 비교를 하는 것을 확인할 수 있다.

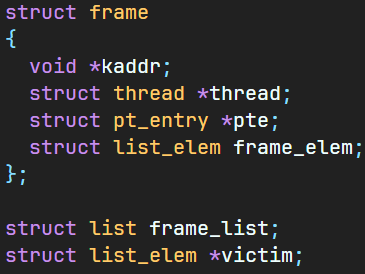
 



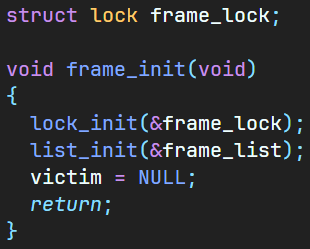
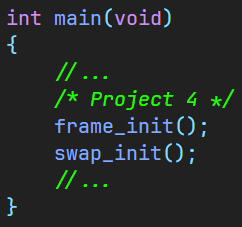
pt\_destroy에선 destroy\_func에 따라 해시를 삭제한다. destroy\_func 역시 직관적으로 해당 pte의 값을 free하는 과정임을 알 수 있다. pt\_init과 마찬가지로 프로세스가 종료되면 호출하여 hast table을 free 해주어야 한다. 이는 threads/thread.c의 process\_exit에서 수정해준다.



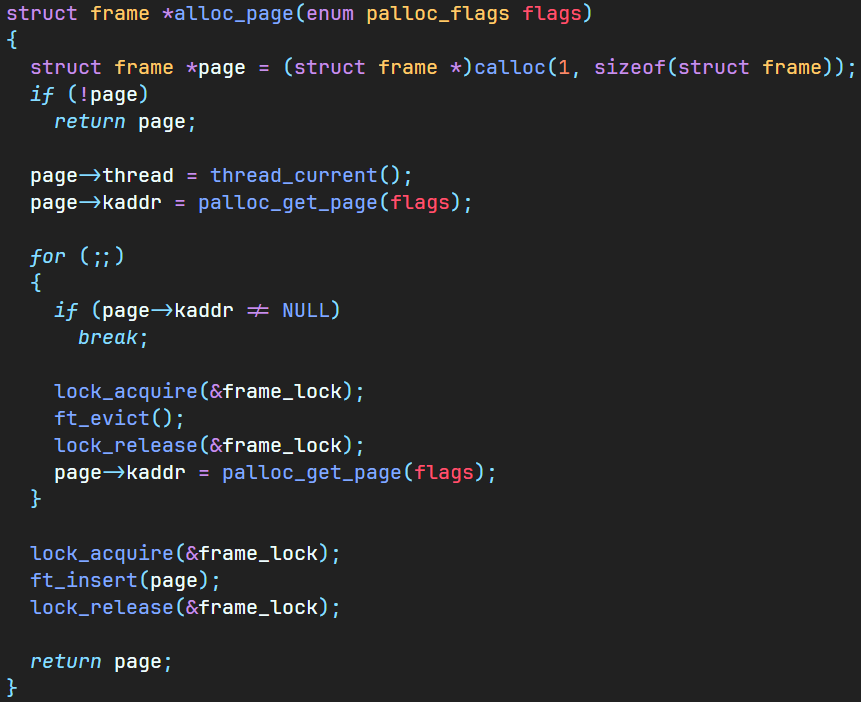
다음으로 기본 operation인 create, insert, delete, find이다. pt\_create에선 단순히 pte를 하나 만들어 전달 받은 값을 할당한다. pt\_insert에선 전달 받은 pte를 hash\_insert한다. pt\_delete는 hash\_delete를 통해 entry를 삭제하고, 삭제에 성공했다면 destroy\_func에서 본 것처럼 free 과정을 거친다. pt\_find에선 단순히 hash를 이용해 entry를 찾고 반환한다. 이렇게 page table의 구현을 마칠 수 있다.



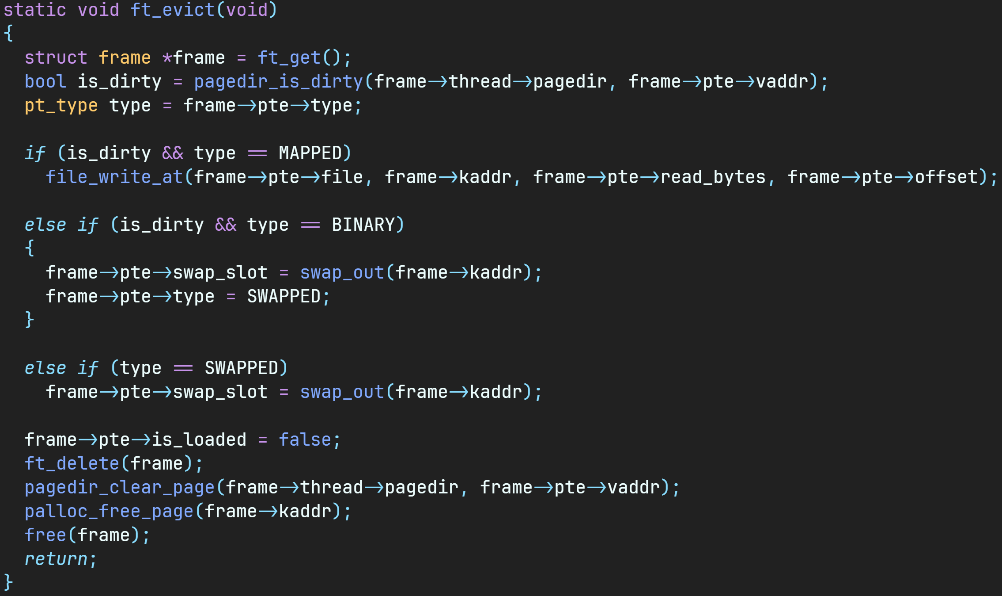
다음은 vm/frame.h이다. frame은 다음과 같은 형태로, physical address(kaddr), frame을 사용중인 thread(thread), 매핑된 pte(pte), iterator(frame\_elem)을 담고 있다. frame\_list는 SCA를 위한 frame table이고, victim은 victim pointer이다.

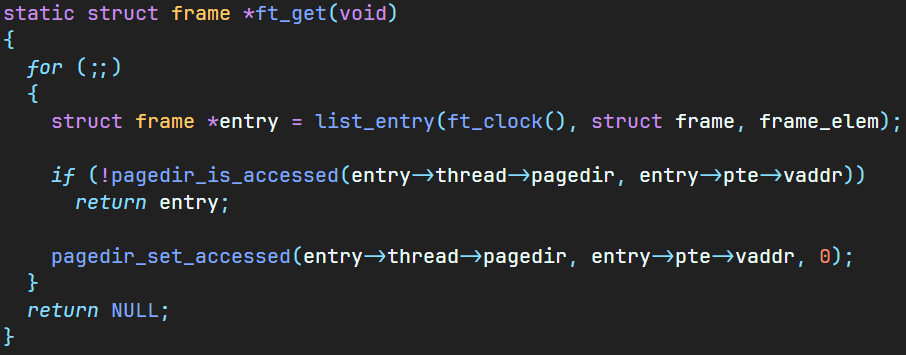
vm/frame.c에는 함수들이 구현되어 있다. frame operation에선 동기화가 매우 중요하므로, 그때 사용할 frame\_lock을 선언한다. frame\_init에선 이를 위해 필요한 lock, list, victim pointer를 초기화한다. frame table은 system-wide하므로 threads/init.c의 main에서 호출하여 초기화한다. 이후 frame 관련 operation을 정의하는데, 모두 wrapper의 형태로 매우 단순하다. 따라서, 중요한 함수만 몇 가지 설명하겠다.



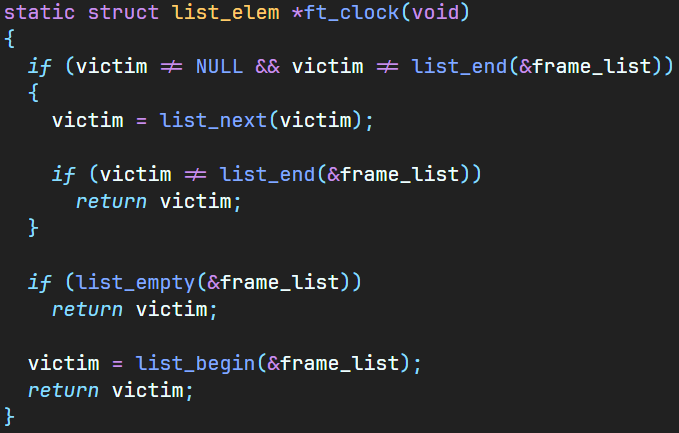
alloc\_page는 단순히 page를 할당하는 과정이다. page를 위한 공간을 할당받고, 매핑하고, 공간을 만들어 삽입하는 과정이다. 기존 pintOS의 palloc\_get\_page를 대신한다. 여기서, ft\_evict는 다음과 같다.



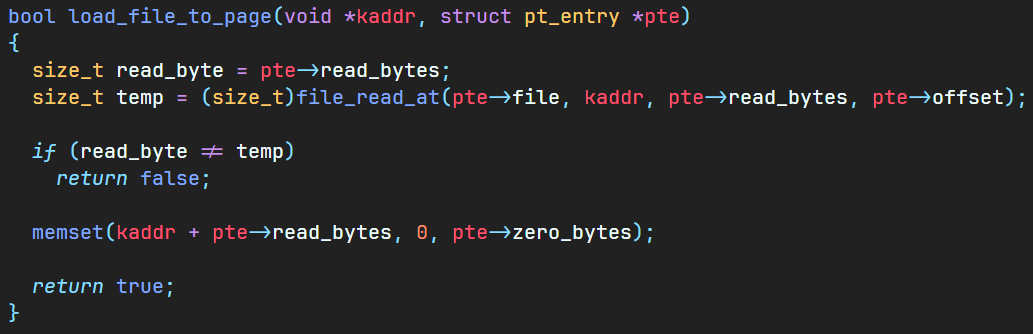
ft\_get을 통해 빈 frame을 찾고, 해당 frame의 mapped page가 dirty인지 확인한다. frame이 dirty이고 BINARY type이라면 일반적인 page이므로 swap out하고 type을 SWAPPED로 바꾼다. 이미 swap out 된 적이 있거나 stack segment 내의 page인 경우 dirty의 여부를 따지지 않고 단순히 swap out만 해준다. frame이 dirty이고 type이 MAPPED인 경우엔 그대로 write하여 업데이트한다. 이후 free 과정을 거쳐 frame을 완전히 evict 할 수 있다. (설명하지 않은 함수들은 중요한 경우 후술된다.)



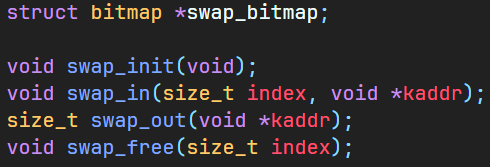
빈 frame을 반환하는 ft\_get은 위와 같다. victim이 가리키는 frame을 ft\_clock()을 통해 가져와서, unaccessed frame을 발견하면 이를 반환하고 그렇지 않다면 해당 frame을 unaccessed로 변경하는 것을 계속 반복한다. (reference bit)



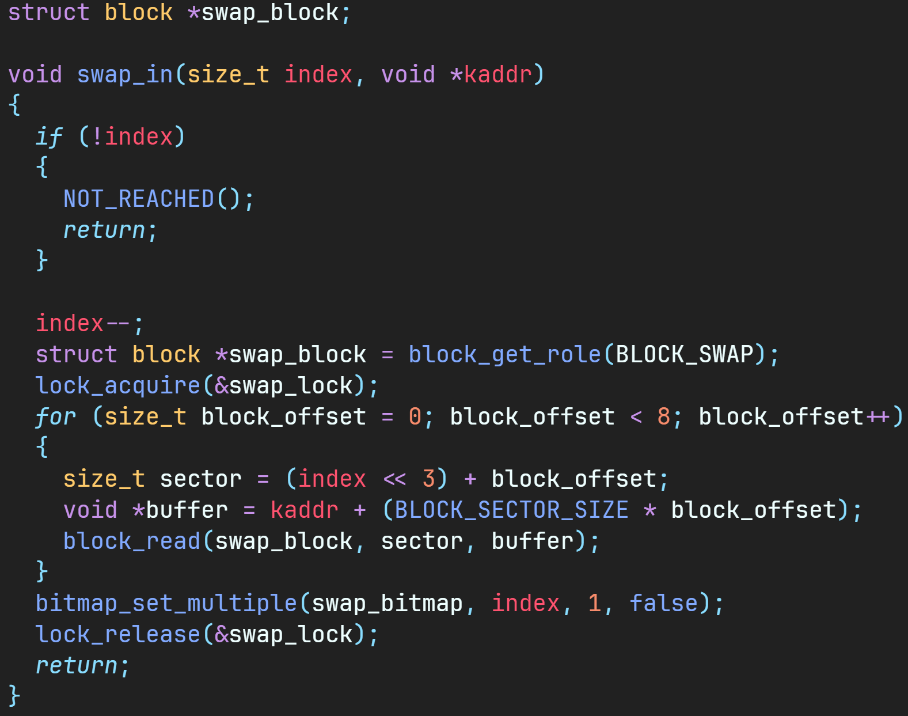
ft\_clock은 clock order 순환을 지원한다. 단순히 victim pointer를 다음 frame으로 넘겨가면서, 마지막에 도달했을 때 맨 처음으로 바꿔주는 동작을 한다고 보면 된다.

vm/frame.c의 또 다른 interface인 free\_page와 load\_file\_to\_page는 각각 free (deallocation), file load를 담당한다. free\_page는 특정 physical address의 pte를 찾아 free하고, load\_file\_to\_page는 특정 physical address에서 pte의 read\_byte만큼 읽어 저장한다.



마지막으로 swap interface이다. vm/swap.h에는 swap slot으로 사용할 swap\_bitmap이 존재한다. swap\_init은 단순히 사용하는 bitmap과 lock을 초기화하는 루틴으로, 역시 system-wide하기 때문에 threads/init.c에서 초기화해준다.



in, out, free 모두 직관적인 동작을 하므로, swap\_in만 설명하겠다. (swap\_out은 해당 swap slot의 index를 반환한다는 사실만 알면 좋다.) index는 1부터 시작하므로 0인 경우 NOT\_REACHED()(General fault)를 호출하고 중지한다. 아닌 경우, index에서 1을 빼 실질적인 index로 바꿔준다. 이후, block\_get\_role을 통해 swap\_block에 블록을 가져온다. 이후, 블록의 8개의 섹터에 대해 sector에는 디스크 스왑 스페이스의 섹터 번호를, buffer에는 데이터를 저장할 메모리의 위치를 계산하여 저장하고, block\_read를 통해 swap\_block에서 sector의 데이터를 buffer로 읽어온다. 마친 후에는 bitmap\_set\_multiple을 통해 bitmap의 해당 index 비트를 false로 설정하여 이후 사용하지 않는다. 이렇게 하면, vm 디렉토리에 page fault handling을 위해 작성한 코드에 대해 설명을 마칠 수 있다.

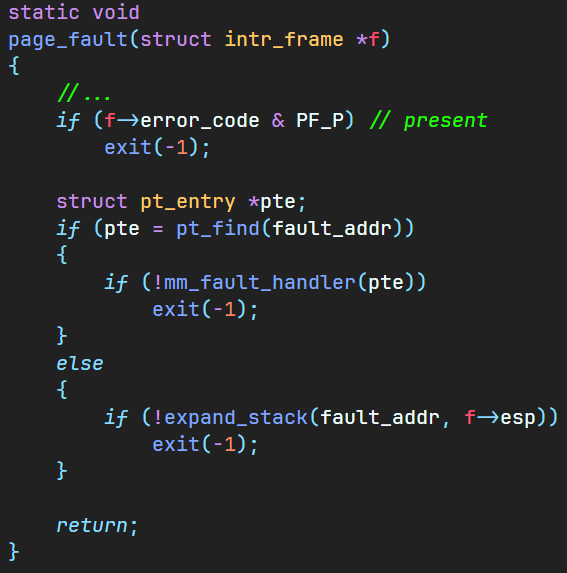
이제 page fault handler에선 demand paging을 구현해야 한다. 우선, userprog/process.c의 load\_segment 함수를 보자.



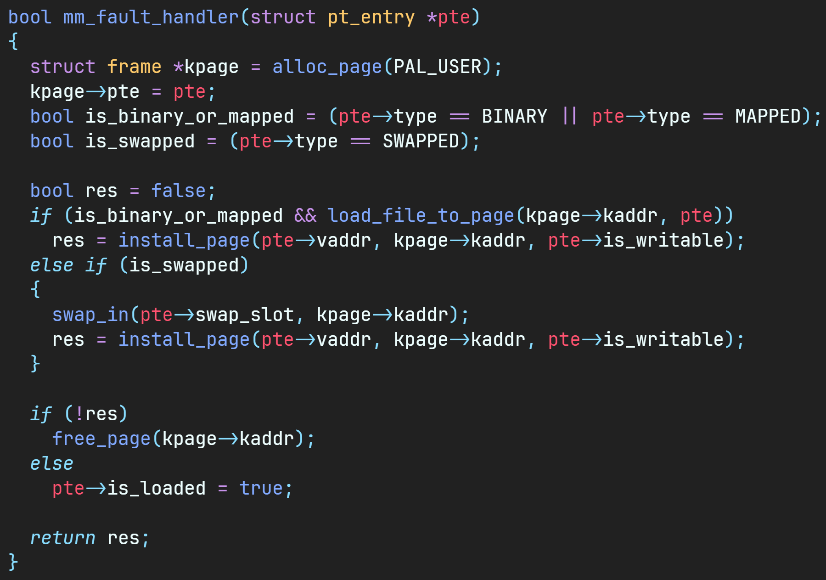
수정된 load\_segment 함수를 보면, 실질적으로는 어떠한 loading도 하지 않는 것을 볼 수 있다. 대신, pte라는 page table entry를 만들어 현재 프로세스의 page table에 삽입한다. 이는 실행 이전에 모조리 loading 하지 않고, 이후 page fault가 발생했을 때 부분적으로 필요한 page를 loading 할 수 있도록 위탁하기 위함이다.



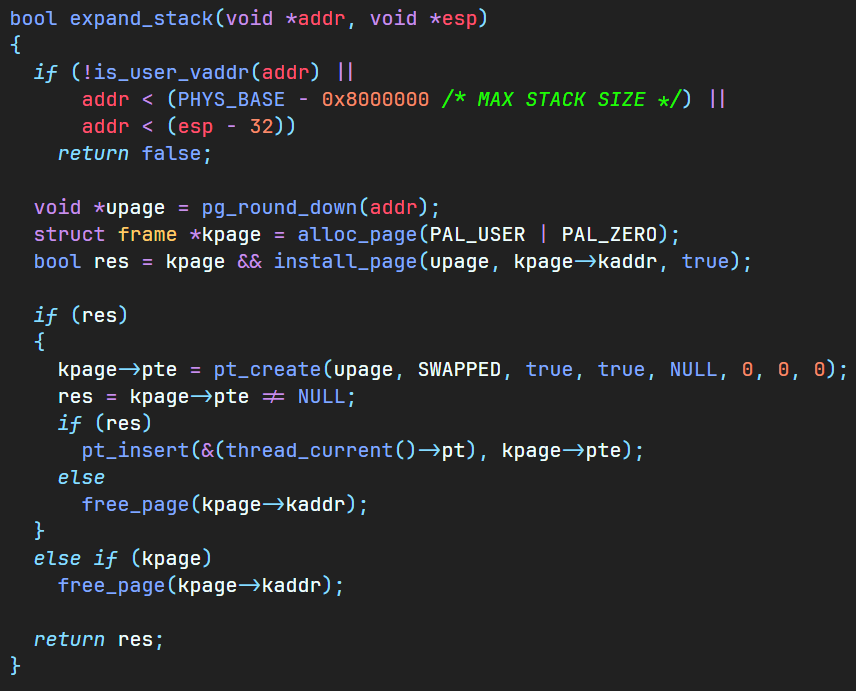
setup\_stack도 마찬가지로 수정한다. stack segment의 page를 생성할 때도 pte만 생성하여 삽입하고 별다른 loading은 하지 않는다. kpage를 alloc\_page로 할당하는 이유는, stack segment page의 가장 기본적인 영역인 user stack을 위해 최소한의 page 공간을 보장하기 위해서이다. 또, 앞서 설명했듯이 stack segment의 page이므로 pt\_type을 SWAPPED로 설정하는 것을 잊어선 안된다.



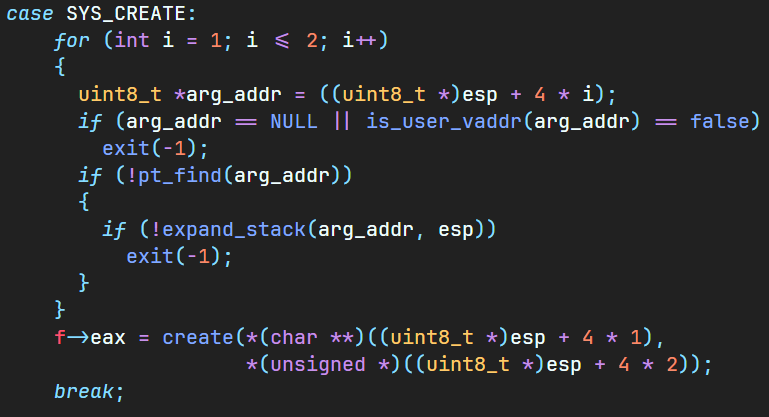
이제 userprog/exception.c의 page\_fault를 확인하자. Flow chart에서의 체크 루틴을 그대로 거친다고 보면 된다. 우선, fault\_addr의 frame이 present라면 에러 코드와 함께 중지한다. 그렇지 않다면 fault\_addr의 pte가 존재하는지 pt\_find로 확인한다. 만약 존재한다면, load\_segment에서 미리 생성한 pte가 존재한다는 뜻이기 때문에 mm\_fault\_handler를 호출한다. 그렇지 않다면, stack segment를 늘려 새로운 공간을 할당해야 하는 상황이기 때문에 expand\_stack을 호출한다.



일반적인 page fault 상황을 handling하는 userprog/process.c의 mm\_fault\_handler이다. 우선 kpage에 alloc\_page를 통해 frame을 할당한다. fault\_addr의 page type이 BINARY이거나 MAPPED인 경우 해당 file의 데이터를 page로 옮겨온 후 pte에 install 하면 된다. SWAPPED인 경우 일반적인 page fault 상황이므로 swap out 된 데이터를 swap in 하고 install 한다. 오류 없이 install을 마쳤다면 load 되었음을 기록하고, 실패했다면 kpage를 free한다.

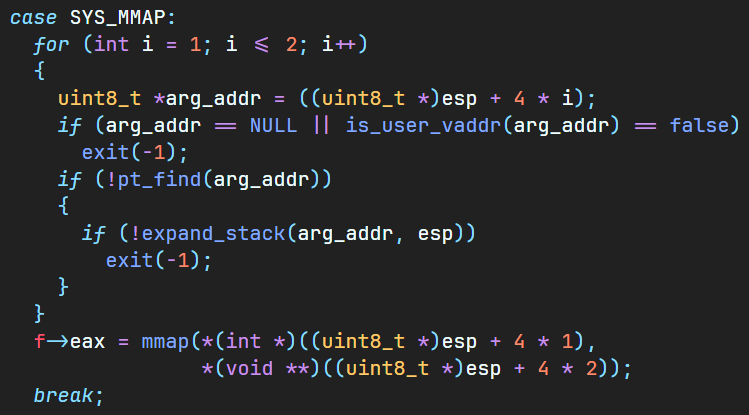
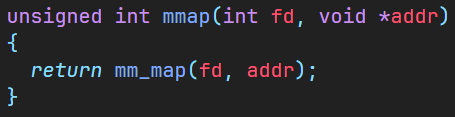


expand\_stack에서는 가장 먼저 fault\_addr validation을 진행한다. user space의 주소인지, 최대 스택 크기를 넘어서지는 않는지, esp로부터 32 이상 벗어나 asm pusha 연산이 불가능하진 않는지를 체크한다. 이후, 앞선 과정과 동일하게 frame을 할당하여 page와 매핑한 후 pte를 생성해 page table에 삽입하는 과정을 거친다. 이 expand\_stack 루틴은 page fault handling 외에도 syscall argument passing (by stack pushing) 과정에서도 실행되어야 한다. user stack에 argument를 push하는 과정에서 스택의 공간이 모자라질 수 있기 때문이다. 따라서, argument validation 과정에 스택 공간을 체크하여 확장하는 루틴을 추가한다.

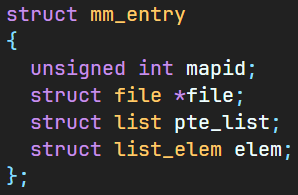


SYS\_CREATE의 예시를 보자. pt\_find를 통해 argument의 주소값에 대해 pte가 존재하는지 확인하고, 그렇지 않다면 expand\_stack을 통해 스택을 확장한다. 여기까지 하면 이번 프로젝트에서 요구하는 demand paging을 모두 성공적으로 구현할 수 있다.

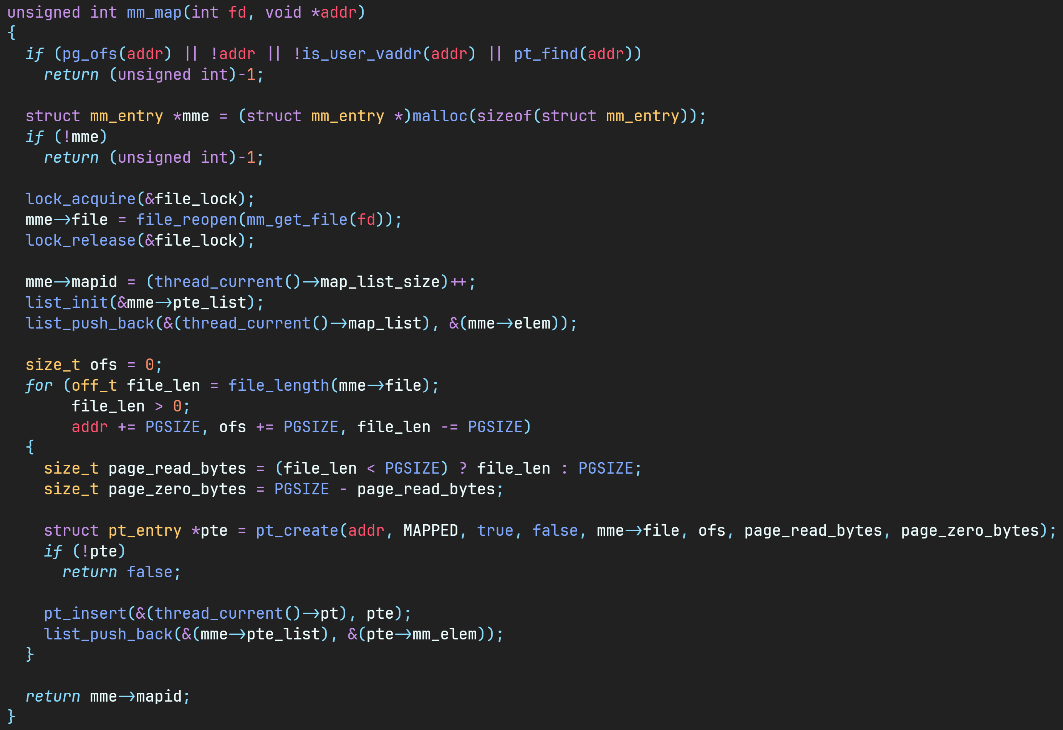
여기까지 구현하면 page-merge-mm을 제외한 모든 요구 test case에 대해 pass를 받을 수 있다. page-merge-mm에선 mmap이라는 새로운 system call을 호출하는데, 이를 lib/syscall-nr.h 및 lib/user/syscall.\*에서 확인해보니 pintOS 내에서 정의는 되었지만 구현은 되지 않은 상태였다. 따라서, userprog/syscall.c에 mmap syscall을 구현해준다. (이 과정은 프로젝트 문맥에 크게 관여하지 않으므로 자세한 설명은 생략하지만, memory mapping에 대한 전반적인 이해와 구현을 필요로 한다. 이는 vm/mmap.\*에 인터페이스로 구현하였다.)

다른 syscall과 동일하게 체크 루틴 이후 mmap이라는 wrapper를 호출해 mm\_map을 실행하게 한다. 이는 vm/mmap.h에 정의되어 있다.



mmap은 메모리 매핑을 위한 코드이므로, 매핑에 대한 정보가 필요하다. 각 프로세스는 개별 메모리 매핑 이후 그 정보를 기억함으로써 mmap syscall로 인해 생성된 page를 관리할 수 있다. 매핑된 file을 저장하고, 매핑으로 인해 만들어진 page를 pte\_list에 삽입하여 관리한다.



실질적으로 mmap의 동작을 관장하는 mm\_map 함수이다. 우선 pintOS 매뉴얼의 mmap 부분에 설명된 조건에 따라 validation을 진행한다. 이후, 메모리 매핑 루틴을 시행한다. mme의 file 필드를 채우는 과정에서, file을 reopen하므로 userprog/syscall.\*에서 사용하는 file\_lock을 extern으로 외부 참조하여 사용해준다. 이후엔 지금까지 진행해왔던 loading 루틴으로 pte를 만들어 삽입하면 된다. 이 과정을 추가하여 page-merge-mm까지 pass를 받을 수 있다.



최종적으로, pintos 디렉토리 내의 Makefile.build에 새로 만든 vm 디렉토리의 코드 파일들을 추가해준다. 이를 통해 cmake의 참조 파일에 새로운 파일을 추가할 수 있다.

* 1. **시험 및 평가 내용**

|  |
| --- |
| cse20211584@cspro10:~/pintos/src/vm$ make check  cd build && make check  …  pass tests/vm/pt-grow-stack  pass tests/vm/pt-grow-pusha  pass tests/vm/pt-grow-bad  pass tests/vm/pt-big-stk-obj  pass tests/vm/pt-bad-addr  pass tests/vm/pt-bad-read  pass tests/vm/pt-write-code  pass tests/vm/pt-write-code2  pass tests/vm/pt-grow-stk-sc  pass tests/vm/page-linear  pass tests/vm/page-parallel  pass tests/vm/page-merge-seq  pass tests/vm/page-merge-par  pass tests/vm/page-merge-stk  pass tests/vm/page-merge-mm  pass tests/vm/page-shuffle  …  All 113 tests passed.  make[1]: Leaving directory '/sogang/under/cse20211584/pintos/src/vm/build'  cse20211584@cspro10:~/pintos/src/vm$ |

채점 항목을 포함한 113개의 vm test case를 모두 통과하였다. (캡쳐 이미지가 너무 커 채점 결과를 복사한 텍스트로 대체한다.)