**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재 교수님

이름 / 학번 : 박성우 / 20171637

개발 기간 : 11.05~11.25

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

Page fault 가 일어났을 때 바로 프로세스를 종료 시키지 말고 새로 만들 여러 자료구조들과 paging 을 이용하여 경우에 따라서 적절히 새로운 주소공간을 할당해 주어서(필요하면 디스크 공간을 이용해서도) 프로세스가 더 원활이 돌아가게 하는 것이 이번 프로젝트의 주된 목표이다.

새로운 system call 인 mmap 과 munmap 을 구현한다. 파일을 읽어와서 메모리에 매핑 시켜주는 system call 이다.

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

기존에는 page fault 가 나면 묻지도 따지지도 않고 바로 프로세스를 종료하나 paging 기법을 통해 추가적으로 메모리를 할당할 수 있으면 그렇게 해주는 것이 옳다. 따라서 page fault handler 함수를 만들어서 상황에 따라 적절히 처리해주고 기존 page table 을 보완하여 새로운 page table 자료구조를 만들어서 page fault 를 어떻게 처리해야 하는지에 대한 정보도 같이 기록해 놓아서 적절히 page fault handler가 새로운 page table 을 참조하여 어떻게 새로운 메모리를 매핑해줄지 결정한다.

* 1. Disk Swap

정말 ‘물리적으로’ 메모리가 부족할 때가 결국 찾아오기 마련이다. 그럴 때는 main memory 이외에 추가적인 disk 에 필요한 데이터를 저장하고 거기에 읽고 쓰고 하는 과정이 필요하다. 따라서 main memory 를 초과하는 공간을 이용할 수 있다.

* 1. Stack Growth

User program 이 초기에 정해진 stack 크기를 벗어난 주소를 참조하려 할 때 적절한 heuristic 을 이용하여 이 주소 참조가 validate 한지 판단하여 validate 할 시 stack size 를 늘리고 새로운 페이지를 할당하여 참조가 가능하도록 한다.

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

기존 page directory 의 page table 에 없는 주소에 접근하려 하면 page fault 가 발생한다. 하지만 상황에 따라서 페이지를 할당하고 주소를 매핑해서 접근할 수 있게 할 수 있다. 우선 적법한 접근인지 확인을 한다. 만약 read only page 에 write 하려고 하거나 kernel memory를 참조하려고 하거나 하는 경우는 적법하지 않은 경우이므로 기존과 같이 프로세스를 종료해준다. 하지만 그 외에 경우는 추가적인 스택이 필요한지나 swap disk에 기록이 되어있는지, main memory에 여유 공간이 있어 그저 mapping 만 해주면 되는지 등등 경우에 따라서 적절한 조치를 하고 원하는 주소에 접근할 수 있도록 하게 해준다.

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

Pintos manual 에서 제시했듯이 LRU 알고리즘을 사용한다. LRU 알고리즘을 기반으로 어떤 pointer 가 clockwise 하게 각 frame 들을 돌면서 접근이 되었는지를 확인한후 접근이 되었으면 접근이 안된 상태로 변경하고 넘어간다. 접근이 안된 상태인 frame 을 만나면 과거에 분명 한번 봐준 녀석이므로 이 frame 을 evict 할 frame 으로 선정한다. 이 pointer를 전역으로 설정하여 상태를 유지해서 매번 새로운 위치에서(쫒아낸 frame 주소부터) 돌기 시작하게 한다.

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

유저 메모리의 스택의 접근에 의해서만 유저의 접근이라고 판단하기 때문에 유저 메모리 스택 포인터 esp를 확인한다. 하지만 이는 유저 모드에 국한된 경우이고, 만약 커널모드의 read나 write 시스템 콜의 신호를 받을 경우에서 page fault가 일어날 수도 있다. 이를 대비해서 thread에 current\_esp라는 포인터를 통해 시스템 콜을 불러오기 직전의 스택 포인터를 저장한 뒤 활용한다. 다음으로 적절한 heuristic 을 이용하여 적법한 stack 확장인지 확인해야 하는데 pintos manual 을 참고하여 최대 8MB 까지 확장할 수 있게 해주고 f->esp-32 이내라면(esp가 가리키는 곳 32바이트 밑보다 위쪽 주소) 적절한 주소 참조라고 heuristic 을 만들었다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

11.05~11.07: 개념과 흐름 이해, pintos 정독

11.07~11.24: vm폴더 내에 각종 자료구조와 pagefault 처리 구현

11.24~11.25: 보고서 작성

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드

**Userprog/Exception.c 수정**

- 기존에는 무조건 프로세스 종료였지만 상황에 따라 적절히 메모리 매핑을 해주어 주소를 참조할 수 있게끔 수정한다.

**Userprog/process.c 수정**

- page directory를 만들 때 새로 만든 table 인 supplemental page table 또한 같이 만들고 Lazy load 로 load 를 다시 구현하고 setup stack 할 때 새로 만들 자료구조인 frame 도 만들어지게끔 한다. Install page 할 때 또한 supplemental page table에 같이 넣어준다. Page fault 가 되었을 시 처리하는 일을 하는 함수들을 추가한다.

**Userprog/syscall.c 수정**

- Lazy load 로 했기 때문에 file read write 에 대해서 process.c 에 preload 함수를 불러 미리 load 하는 것으로 처리해주고 새로운 system call 인 mmap 과 munmap 을 구현한다.

**Thread/thread.c, init.c 수정**

- 새로 만들 자료구조들을 초기화 해준다.

* + 수정하거나 추가해야 하는 자료구조

**Threads/thread.h**

- Thread 구조체에 새로 만들 구조체 supplemental page table 포인터 를 넣는다. 또한 mmap 된 page 들을 관리하기 위한 mmap\_list 라는 list 구조체도 추가한다.

**VM/page.h**

- 기존 page table 을 보완하기 위한 supplemental page table 을 만든다. 이 구조체에는 page\_map 이라는 hash 가 있어서 여기에 새로 만들 supplemental page table entry 라는 구조들을 넣을 것이다.

- supplemental page table entry 구조체를 추가한다. 기존에 page 가 가지고 있던 정보를 포함해서 page fault 가 났을 시 어떻게 해야 하는 지를 위한 페이지 상태 등 추가 정보를 담고 있다.

**VM/frame.h**

- 기존 frame 들이 가지고 있던 정보들을 보완하기 위해 frame table entry 라는 새로운 frame 구조체를 만들고, 이 frame 들을 저장할 자료구조인 frame\_list list 와 frame\_map hash 를 만든다.

**Userprog/process.h**

- mmap 시스템 콜을 위한 mmap\_desc 라는 구조체를 추가한다.

* + 수정하거나 추가해야 하는 함수

**Userprog/process.c**

**Load 함수**

Load 함수 에서 현재 스레드의 page directory를 만든다. 그 밑에 마찬가지로 현재 스레드의 supplemental page table 도 만들어준다.

**load\_segment 함수**

기존에 필요한 메모리들을 모조리 불러와서 pagedir 의 page 들에 install\_page() 함수로 mapping 하던 것을 lazy load 로 바꿔준다. 이를 위해 일단 앞으로 참조할 주소들을 미리 supplemental page table 에 담아 놓는다. 실제로 page 들에 mapping 한것은 아니기 때문에 page fault 가 날 것이고 그 때 page\_fault\_handler가 적절히 처리 해줄 것이다.

**Setup stack 함수**

기존에 frame 을 만들기 위해 page 가져오던 것을 은근슬쩍 이번에 새로 만든 frame table entry 까지 같이 만들게 한다. 이 함수안에서 palloc\_get\_page 를 하던 것을 vm\_palloc\_get\_page 함수로 변경하여 frame table 도 만들고 page도 할당한다.

**Handle\_mm\_fault 함수**

Page fault 가 났을 때 handling 해주는 함수이다. 실제로 잘못된 page fault 는 exception.c 에서 이미 걸렀다. 여기 들어온다는 것은 옮은 접근 이므로 상황에 따라서 적절히 처리를 하고 페이지를 할당해 주라는 것이다. 따라서 새로 만든 구조체인 supplemental page table 을 참조해서 각 entry 들의 상태에 따라서 적절히 처리를 해주면 된다.

**Expand\_stack 함수**

Stack 을 늘려야 하는 경우에 이 함수를 호출하여 stack 을 늘려준다. 실제로는 해당하는 주소에 대한 supplemental page table entry를 만들어서 supplemental page table에 삽입해주는 게 전부이다. 이렇게 해놓고 나중에 handle\_mm\_fault 를 불러서 이 테이블을 참조하여 적당한 처리를 해줄 것이므로 모든 것은 handle\_mm\_fault 에 맡기자.

**Preload 함수**

System call read write 을 하기전에 미리 필요한 메모리를 매핑해줄 수 있도록 미리 load 하는 함수이다.

**VM/page.c**

**Supte\_hash\_func, supte\_less\_func, supte\_destory\_func 3종 세트**

Hash 를 만들 때 hash init 을 위한 3종 세트 함수들이다. user page 로 어느 hash bucket 에 담을 지 정하고 user page로 대소 비교를 할 것이고 제거할 때 어떻게 제거할 건지를 알려주는 함수이다.

**Vm\_supt\_create 함수**

Supplemental page table 을 만들어주는 함수이다. 위에서 언급하였듯 process.c 의 load 함수에서 page directory 만들 때 밑에 살짝 추가해서 같이 만들어 준다.

**Vm\_supt\_destroy 함수**

Supplemental page table 을 박살내는 함수이다.

**Vm\_supt\_install 함수 세트**

Supplemental page table 에 page 의 상태에 따라서 install 해주는 함수들이다.

**Vm\_supt\_set\_swap , vm\_supt\_set\_dirty 세트**

페이지가 디스크에 기록되어 있으면 기록되어 있다고 상태를 ON\_SWAP 으로 변경해주는 함수와 페이지가 dirty 하면 dirty 하다고 setting 해주는 함수이다.

**Vm\_look\_up , vm\_supt\_has\_entry 세트**

원하는 엔트리를 찾아 반환해주는 함수와 원하는 엔트리가 있는지 supplemental page table 에서 확인해주는 함수들이다.

**Vm\_supt\_mm\_unmap 함수**

Unmap 을 위한 함수이다.

**VM/frame.c**

**Vm\_frame\_init 함수**

Frame map hash 와 Frame list list 를 init 한다. 덤으로 frame\_lock 도 init 한다.

**Frame\_hash\_func, Frame\_less\_func 세트**

Hash 를 init 할 때 쓰이는 두 함수들이다. Kernel page를 기준으로 어느 hash\_bucket 에 담을 지와 대소 비교를 한다.

**Vm\_frame\_allocate 함수**

이 함수에서 user page 에 매핑할 memory 주소를 넘겨주게 되는데 만약 여기서 메모리가 부족하여 palloc\_get\_page 함수가 실패하게 되면 어느 frame 하나를 내쫓게 된다. 이 때 바로 위에서 언급했던 LRU 알고리즘을 통해 하나를 내쫓아버리고 palloc\_get\_page를 다시 불러서 frame을 get 한 다음에 새로 만든 frame map hash 와 frame list list 에 넣어주고 그토록 원했던 mapping 하려는 주소를 반환한다.

**Vm\_frame\_do\_free, vm\_frame\_free, vm\_frame\_remove\_entry 3종 세트**

Frame table 에서 entry를 지우기만 하거나 완전 free 까지 하거나 하는 함수들이다.

**Pick\_frame\_to\_evict, clock\_next\_frame 세트**

LRU 알고리즘을 구현한 함수 2가지이다.

**VM/swap.c**

**Vm\_swap\_init, vm\_swap\_out, vm\_swap\_in, vm\_swap\_free 4종 세트**

Disk 에 메모리 내용을 읽고 쓰고 지우고 하는 함수들이다.

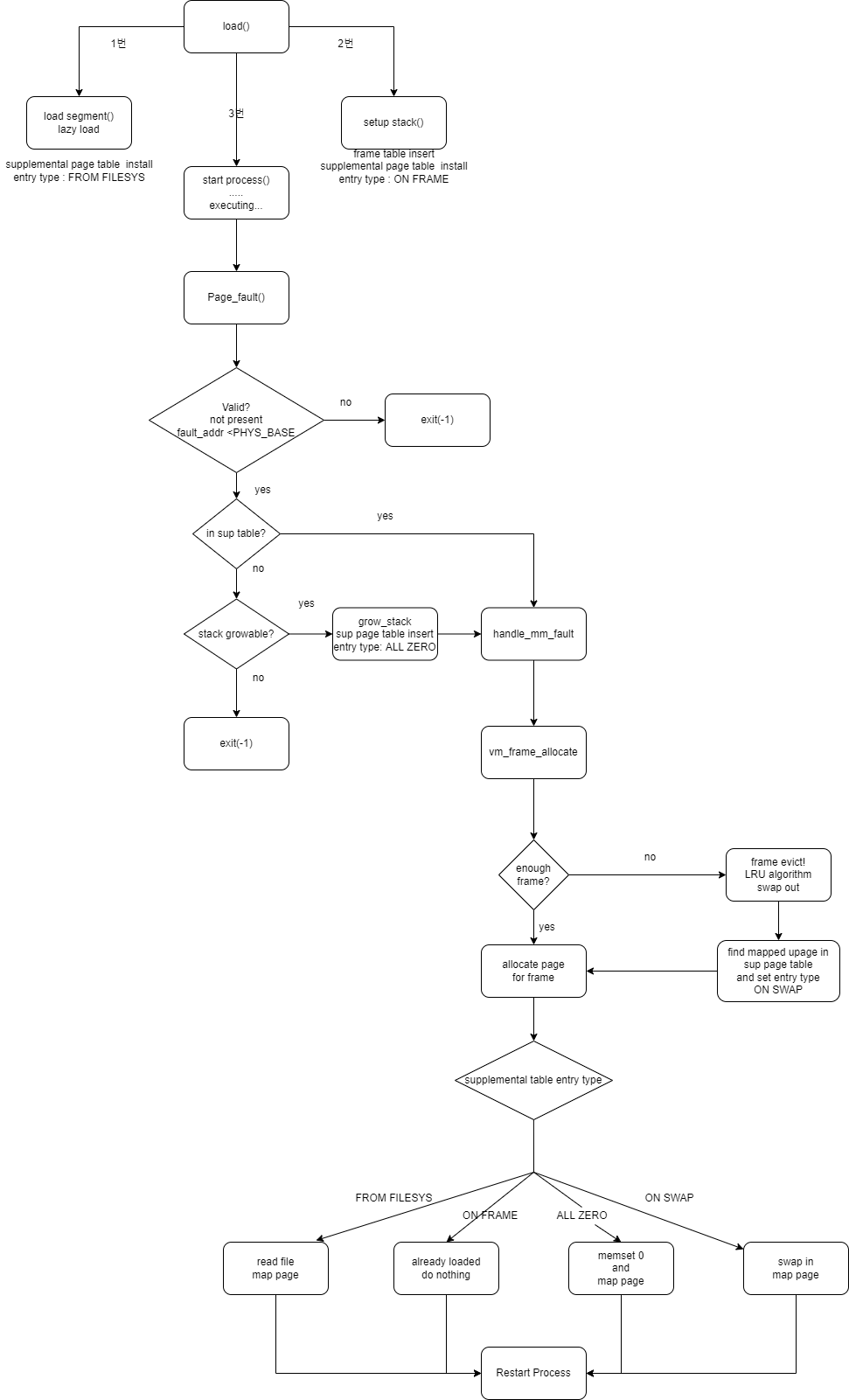
**Userprog/syscall.c**

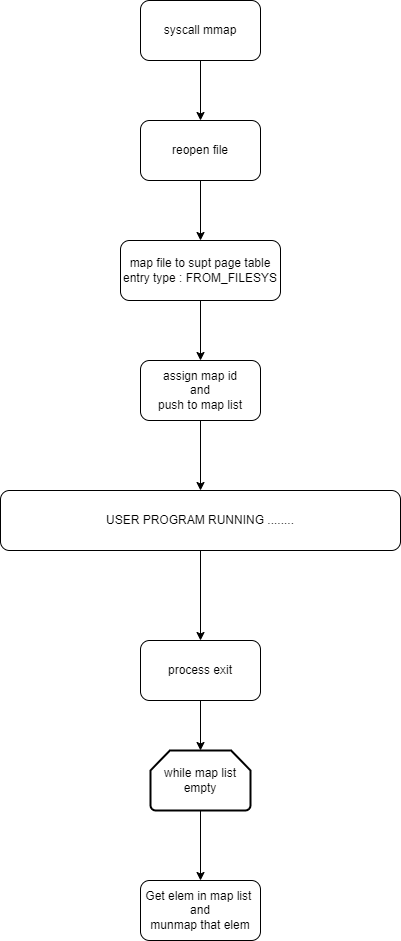
**Sys\_mmap, sys\_munmap 함수**

새로운 system call 을 실행시켜주는 함수들이다.

1. **연구 결과**
   1. **Flow Chart**

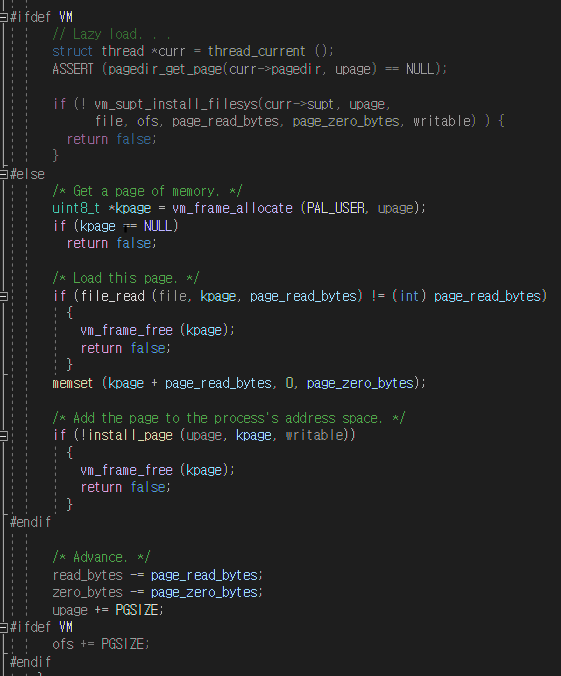
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성





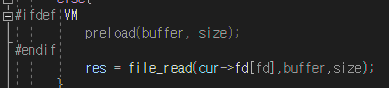
* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

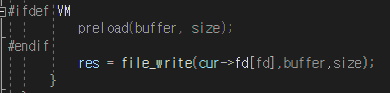
Userprog/process.c load\_segment 함수



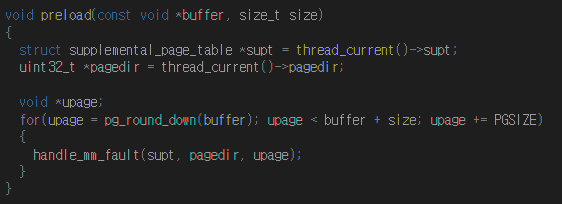
User program 을 실행하기 전에 참조해야 하는 주소들에 페이지를 할당 받고 그것들을 user page directory 안에 page table entry 에 mapping 해야 한다. pintos manual 에는 lazy load 로 구현하라 했으므로 lazy load 로 구현한다. 새로운 코드는 #ifdef VM 으로 감싸서 lazy load 로 하고 userprog 에서 make 할 때는 기존코드로 실행되게끔 하였다. 기존에는 필요한 페이지를 할당 받아 그때 그때 바로 install\_page 함수로 매핑하지만 그러지 않고 새로 만든 supplemental page table 에 필요한 페이지를 넣어 놓고 끝낸다. 추후 handle mm fault 함수에서 mapping 을 할 것이다.

다만 read 나 write system call 인 경우에는 미리 load 해야 하므로 syscall.c 에서 preload 함수를 불러서 미리 load 를 하자.

userprog/syscall.c

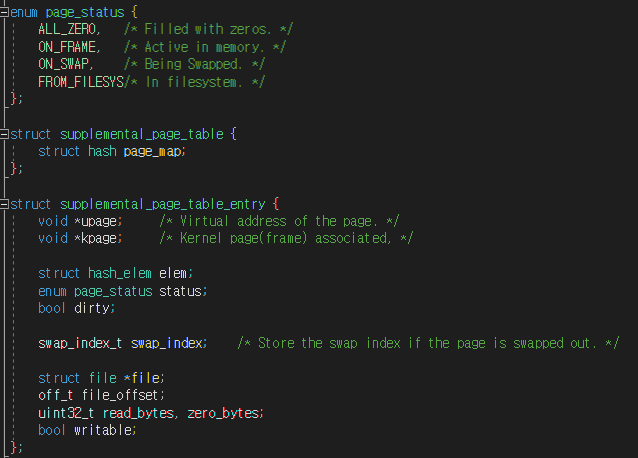
userprog/syscall.c

Userprog/process.c preload 함수



미리 handle mm fault 함수를 불러서 읽거나 쓸 것들을 mapping 을 하자.

Vm 파일에 있는 page.h 에서 supplemental page table 을 보면



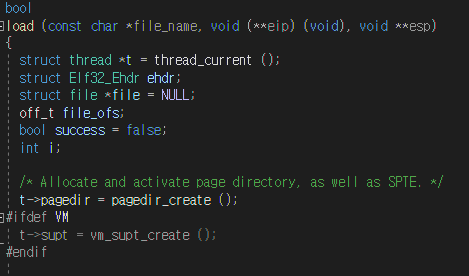
이렇게 구성되어 있다.

Page\_status 는 현재 페이지의 상태이다. 이 상태에 따라 page fault 가 났을 때 다른 방식으로 처리를 해주어야 하므로 나올 수 있는 4가지 상태를 만든다. 기존 page table 을 보완하기 위해 만든 supplemental page table 은 hash 로 만들기 때문에 sup page table 안에 page map 이라는 hash 를 넣는다. 그리고 이 sup page table 안에 들어갈 요소인 sup page table entry 가 있다. 사진에서 보다시피 upage 는 user virtual address 고 kpage 는 frame 주소이고 status 는 page 의 상태이고 swap 을 했을 때 disk 에 저장되는 위치를 기록하는 swap\_index 가 있고 file 구조체, offset 등등이 들어간다.

Page directory 가 process 당 하나 있듯이 sup page table 또한 그렇게 만든다 그래서 thread 구조체에 원소로 넣어준다. 보고 가면

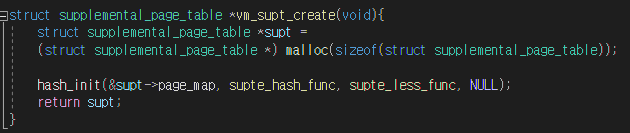


이렇게 thread 구조체 안에 넣어주고 시작한다. 이 table 의 본질은 hash 이므로 hash 를 init 해줘야한다. Table 은 결국 기존 page directory 와 page table 을 보완하기 위함이므로 딱 pagedir 을 만드는 곳에 옆에 초기화를 해주자. 그곳은 process.c 의 load 함수에 있다.

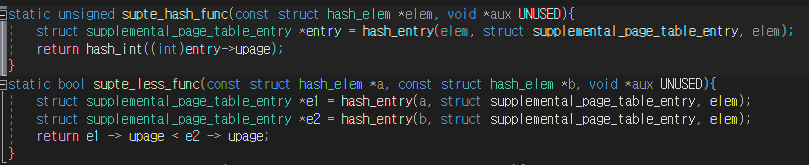


보면 VM 으로 감싸고 vm supt create 함수를 통해 table을 만들어 주었다는 것을 알 수 있다.

vm supt create 함수를 보고 가면

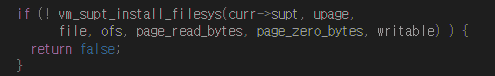


이렇게 hash init 을 한다. 덤으로 hash init 을 위해 필요한 hash func 와 less func 도 보고 가면

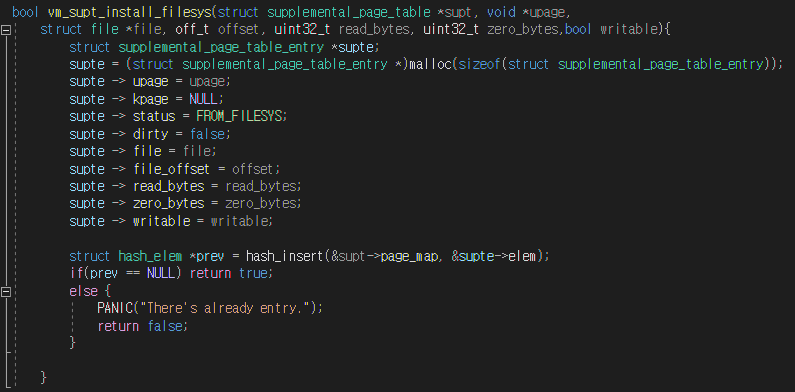


이렇게 되어있다. 전부 entry 구조체 안에 upage 변수를 통해 hash bucket 을 정하고 대소비교를 한다.

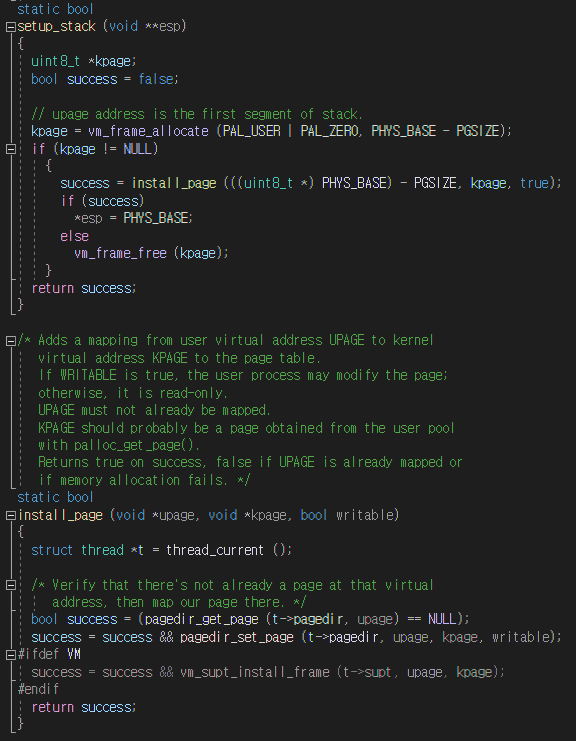
다시 load\_segment 코드를 살짝 보면 vm supt install filesys 함수를 불러서 sup page table entry를 table에 넣는데



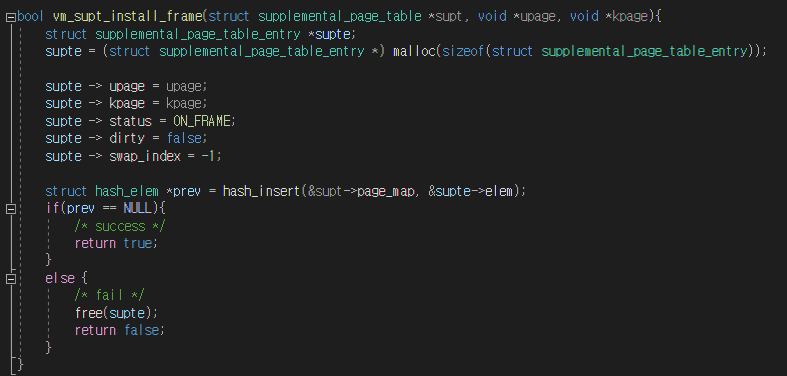
이 함수는 페이지 상태를 FROM FILESYS 상태로 넣어주는 함수이다. 이렇다.



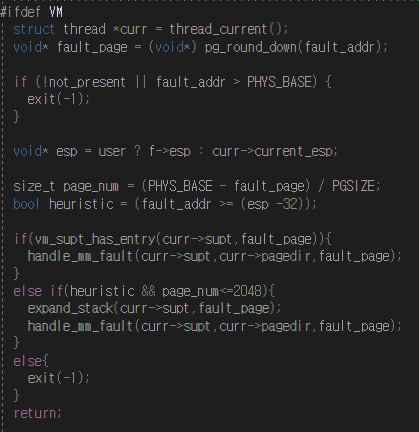
다음은 setup stack 함수와 install\_page 함수이다.



기존에는 kpage 를 palloc get page 로 frame을 받아왔지만 그것에 추가로 새로 만든 자료구조인 frame table 을 구성하기 위해 vm\_frame\_allocate 라는 함수를 대체한다. 추후 handle mm fault 함수가 나올 때 자세히 살펴볼 것이다. Install page 를 할 때 supplemental table 에도 같이 넣어준다. Entry는 ON\_FRAME 타입으로 넣는다.

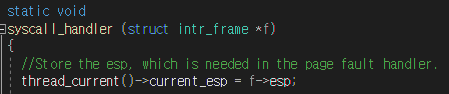


이제 load 가 끝나고 프로그램이 돌아가다가 page fault 가 뜨게 되면 기존에는 그대로 processs 를 kill 했지만 이제는 다르다. Userprog/exception.c 를 보면



정말로 invalid 한 reference 는 칼 같이 걸러낸다. 그리고 pintos 에 나와 있듯이 제대로 된 esp 를 받아와야 한다. 이를 위해 thread 구조체 안에 current esp 변수를 만들고 system call 이 들어올 때 미리 esp 포인터를 current esp 변수에 저장해둔다.

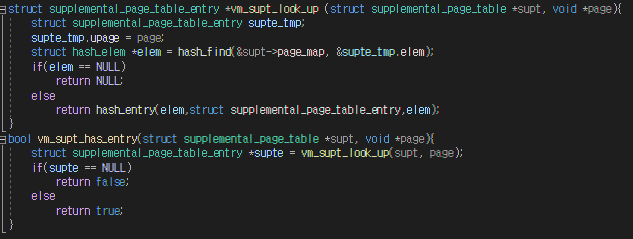
threads/thread.h thread 구조체 안

syscall.c

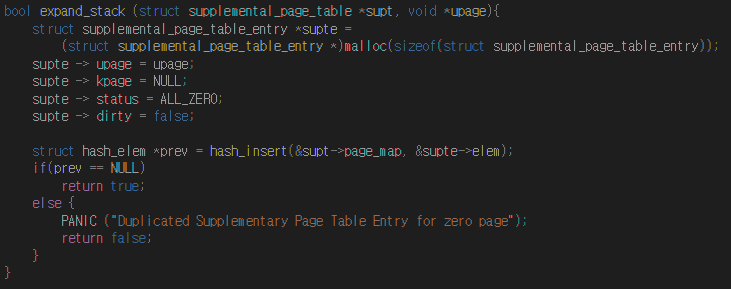
Stack 이 모자랄 경우를 위해 stack growable 할 수 있는지 보는 변수 두개 page\_num 과 heuristic 을 만든다. Page num 은 page 의 개수이다. 최대 스택 사이즈일 때, 즉 stack 이 8MB 일 때 page 의 개수는 2048개이다. 따라서 page num 이 2048 보다 크면 안된다. 또한 pintos manual 에 나와 있듯이 f->esp -32 정도 까지는 가능한 stack grow 이므로 이 정도 까는 허용해준다.

이제 경우를 나눠서 page fault 를 처리해보자.

모든 필요한 주소는 supplemental page table 에 들어가 있다. 문제를 일으킨 fault addr 로 pg\_round\_down 함수로 fault page 주소를 구한 후 이 주소를 vm supt has entry 함수로 sup page table 에 있는 지 확인한다.

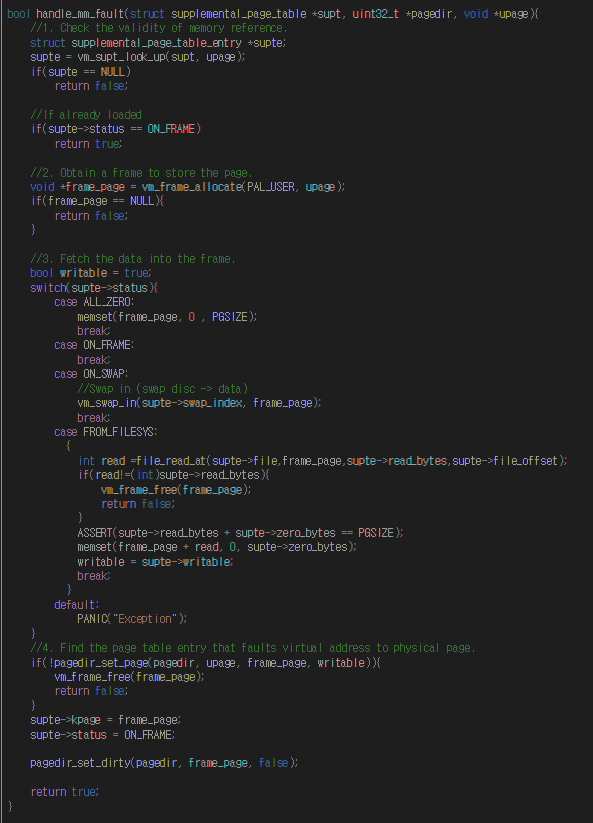


만약 있으면 mapping 이 문제이므로 handle mm fault 함수를 불러 page 의 상태를 확인한 후 적절히 처리해준다. Sup page table에 없으면 stack grow 가 가능한지 확인해본다. 아까 위에서 만든 heuristic 변수가 참이고 page num 이 2048 이하이면 가능한 것으로 간주하고 expand stack 함수를 부른다. Userprog/process.c 의 Expand stack 함수를 보자.



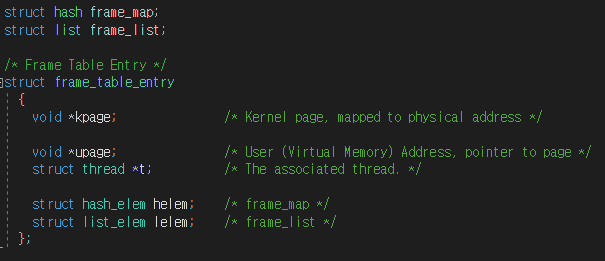
함수 이름은 expand stack 이지만 사실 하는 일은 page 상태를 ALL ZERO 로 하고 supplemental page table 에 넣어주는 일만 한다. 프로그램을 돌릴 때 필요한 주소 참조는 모조리 다 sup page table 안에 들어가 있어야 한다. 그래서 이 함수에서는 sup page table 안에 넣어주기만 하고 나머지 모든 일은 handle mm fault 함수가 sup page table 을 참조해서 적절히 처리한다. 믿고 맏기자.

Handle mm fault 함수를 보자.

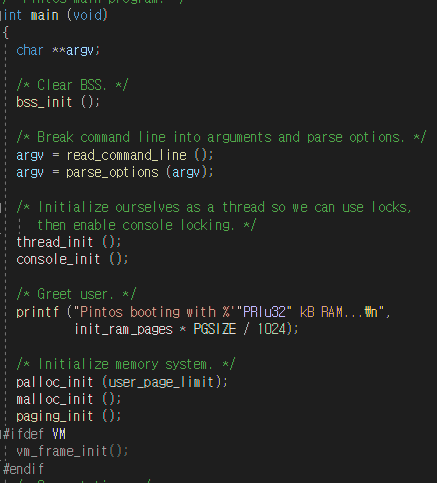


이번 프로젝트에서 가장 중요한 함수이다. 프로그램을 돌릴 때 필요한 주소들은 전부 다 supplemental page table 에 들어가 있으므로 이 table 을 참조해서 entry 상태를 보고 적절히 처리를 할 것이다. 이 함수의 인자인 Upage 는 fault page 를 넘겨 받았다. 이 페이지 주소로 vm frame allocate 함수를 호출한다. 이 함수 내부에서 palloc get page 를 통해 기존대로 frame을 얻어내기는 하는데 여기에 frame table entry를 구성하고 frame table 에 넣어주는 부분이 추가적으로 들어가 있다. 더불어 기존에는 만약 자리가 없어 palloc get page 가 실패하고 null 을 반환하면 그거로 process 종료하겠지만 이번 프로젝트에서는 그럴 수 없다. 필요하다면 기존 frame 중 하나를 쫓아내 버리고 frame을 할당 받아야만 한다.

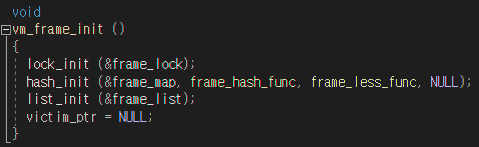
frame 구조체를 살펴보면



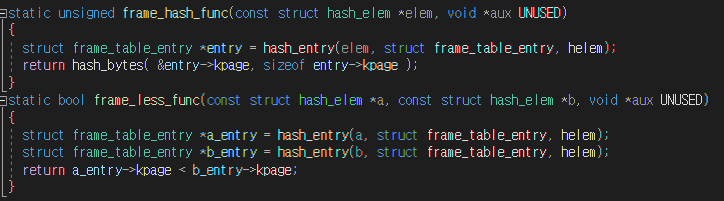
매우 간단하다. Frame table 을 list 형태, hash 형태로 하나씩 만들어 두고 entry들을 여기에 넣어줄 것이다. Entry를 보면 당연하게도 kpage 가 있고 upage 도 넣어주고 현재 어느 쓰레드인지를 나타내는 thread 구조체도 있다. Frame table은 thread 당 하나 있는 게 아니라 공통적으로 하나 있기 때문에 thread 구조체를 원소로 넣어주어야 한다. 그래서 또한 threads/init.c 에서 초기화를 박아버리고 시작해야 한다. 그것을 확인해보면



Threads/init.c 의 main 함수에서 vm\_frame\_init 으로 초기화를 해주었다. 그렇다면 또 vm\_frame\_init 함수를 확인해보면

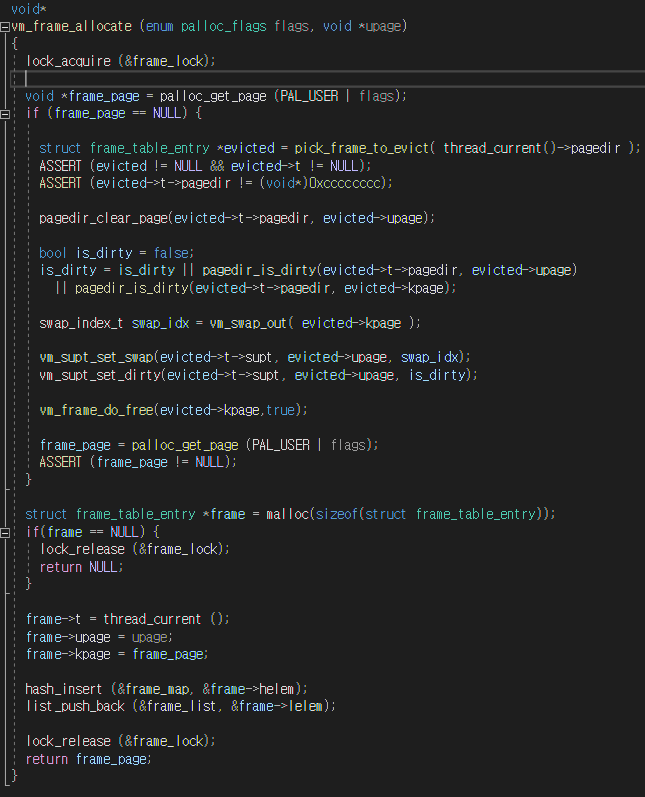


락을 하나 만들고 frame table을 hash 로 만든 것을 위한 hash init 을 하고 list 로도 하나 만들었으므로 list init 도 해준다. 마찬가지로 hash func 와 less func 를 보면

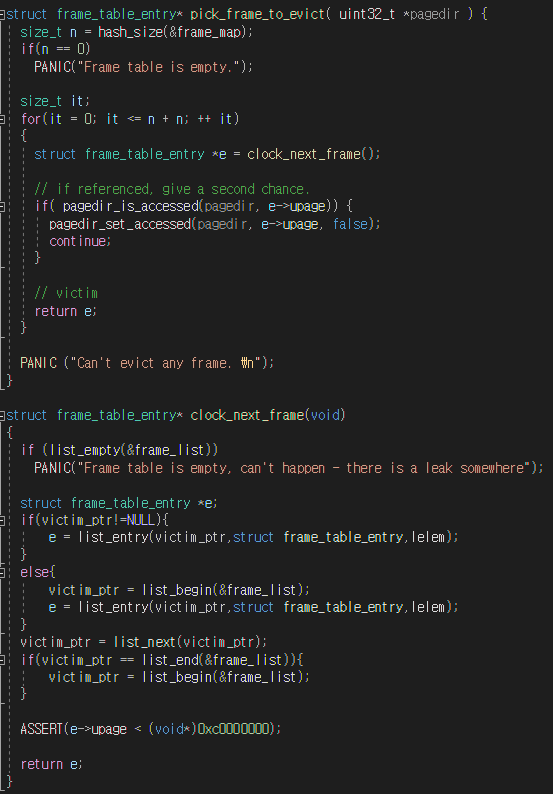


이렇다. 이번엔 kpage 를 이용한다.

이제 제대로 vm frame allocate 함수를 보자.



처음에 palloc get page 로 frame을 받아온다. 성공하면 그 페이지를 반환하는데 덤으로 frame table entry를 만들고 그것을 frame table 넣어준다. 실패했을 때가 문제인데 이제 LRU 알고리즘을 통해 frame 하나를 내쫓아버려야 한다. 그것을 위한 함수가 frame.c 안의 pick frame to evict 와 clock next frame 함수이다.

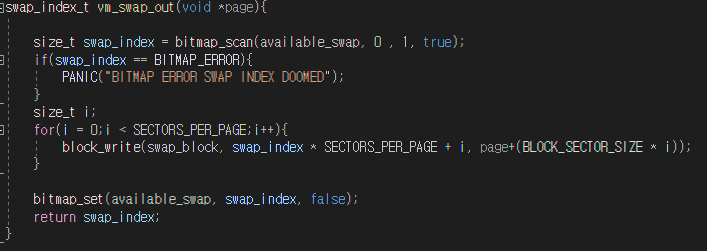


수업 시간에 배운 그대로이다. 전역 포인터를 하나 만들고 쫓아낼 frame을 정하는데 한번은 봐주고 그 다음이 니놈이다 하고 결정한다. 이것을 위해, 즉 swap 을 위해 frame table list 를 만들었다. 이 list가 바로 쫓아낼 놈을 정하기 편하게 palloc get page 로 할당받은 모든 frame 들을 가지고 있다!

다시 vm frame allocate 함수로 돌아가서 쫓아낼 페이지를 결정하면 vm swap out 함수를 통해 swap out 을 해준다.



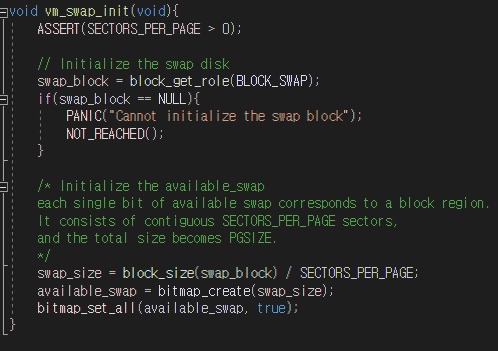
VM/swap.c의 swap out 을 보고 가면



디스크에 내쫓는 frame을 기록하고 어디에 기록했는지 swap index 에 기록하고 swap index 를 반환한다.

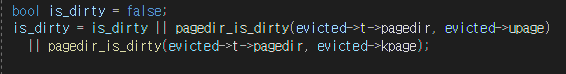
더 진행하기 전에 이 swap 을 위해 무엇을 했는지 보고 가야한다. 이것 또한 swap disk 또한 process 당 하나가 아니기에 threads/init.c 에서 초기화를 해주고 가야한다. 보고 가자.

init.c 함수에서 vm swap init 함수를 호출하여 초기화 하였다.

VM/swap.c 에 있다.

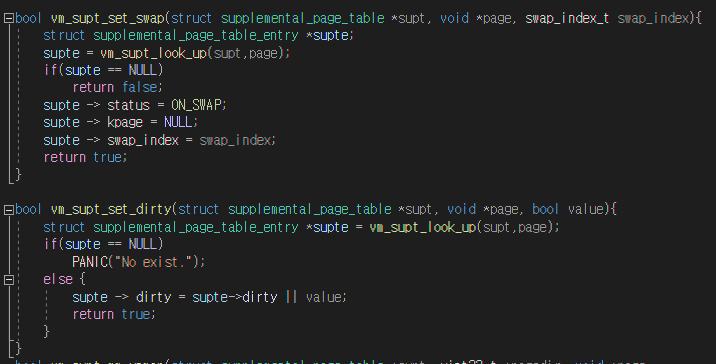
Swap 을 위한 bitmap 을 세팅하고 블록을 설정해주는 초기화 함수이다.

이제 다시 vm frame allocate 함수로 돌아와서 swap out 으로 disk 에 frame을 기록했으니 user 측에 이 사실을 알려주어야 한다. 그걸 위해 setting 하는 부분을 확인해보자.





혹시나 쫓아내는 page가 write 을 통해 변경되었을 수 있다. 더럽혀졌으므로 dirty 를 set 해주기 위해 is\_dirty 변수를 만들어서 세팅해주고 이제 진짜 user 측에 알리기 위해 supplemental page table 쪽으로 set 함수들을 통해 간다.



Swap out 이 되었으면 supplemental page table 에서 그 주소를 찾아서 swap out 되었다고 말해주어야 한다. 말 안하면 엉뚱한 곳을 찾아가서 결국 이상한 주소를 참조하여 터져 죽을테니 꼭 알려주어야 한다. 그래서 swap out 된 page를 supplemental page table 에서 찾아 그 페이지의 상태를 ON\_SWAP 으로 바꾸어주고 kpage 를 null로, disk 를 참조할 수 있게 swap\_index에 디스크 어디에 써져 있는지 알려준다. 혹시 page가 더럽혀졌는지도 알려줘야 한다. 그래서 마찬가지로 더럽혀졌으면 더럽혀졌다고 그 page 의 dirty 변수를 변경한다.

이제 다시 한번 vm frame allocate 로 돌아가서

지금 현재 한 frame table 의 한 frame 이 내쫓긴 상태다. 고로 frame table 은 그 entry 를 가지고 있을 필요가 없다. 고로 free 해버린다.



드디어 frame 을 받기위한 준비를 다 맞추었다 이제 자리가 생겼으므로 한번 더 palloc get page 를 통해 page 를 할당받아보자.



분명 자리를 비우고 page를 할당 받으려 했는데 또 실패하면 이건 중대한 하자가 있는 것이므로 assert 문으로 터트려 버린다.

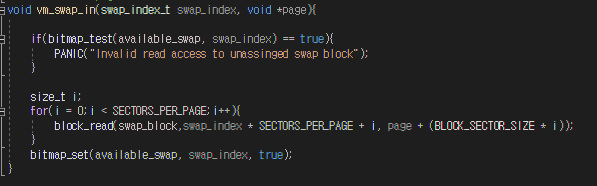
이렇게 할당 받고 나면 성공했을 때 한 것처럼 frame table entry를 만들고 frame table 에 넣어주고 드디어 vm frame allocate 가 끝나게 된다.

흐름을 잃지 말고 다시 handle mm fault 함수로 돌아가자. 이제 supplemental page table entry에 mapping 할 frame\_page 가 준비가 되었다. Sup page table entry의 page 상태를 보면 4가지 경우가 있다.

ALL\_ZERO 인 경우가 stack grow 로 새로 mapping 해야 하는 경우이다. Frame\_page를 0으로 초기화 해주고 pagedir set page 함수로 mapping 을 해주자!



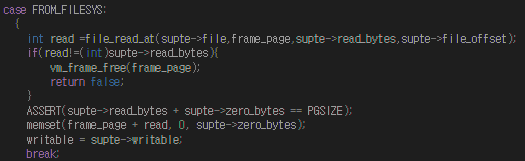
ON SWAP 인 경우는 위에서 봤듯이 원하는 주소가 지금 현재 disk 로 나가 있는 경우이다. 가져와야 하므로 swap in 함수를 호출해서 frame page 에 다시 data를 받아오자. Vm/swap.c 에 있는 Swap in 함수 한번 보고 가자.



전에 swap out 했을 때 디스크 어디에 저장했는지 swap index 가 가지고 있다. 이 swap index 와 frame page를 넘겨서 disk 의 swap index 주소를 참조하여서 내용을 frame\_page에 다시 받아온다. 이제 pagedir set page 함수로 mapping 해주면 끝이다!

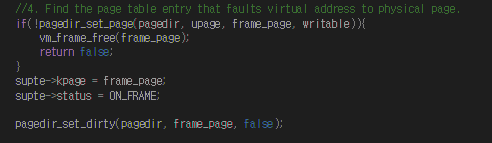


FROM\_FILESYS 인 상태는 원래대로 라면 lazy load 가 아니어서 미리 전부 mapping 되었겠지만 lazy load 하는 바람에 미리 mapping 하지 않았다. 따라서 이때 mapping 을 하는 것이다. 원래 load segment 함수에서 했던 것 처럼 sup page table entry 의 file 을 읽어 frame\_page 에 기록한다. 이제 pagedir set page 함수를 통해 mapping 해주면 끝이다!



ON FRAME 이미 되어있는 부분이므로 애초에 switch 문 들어오기 전에 true 를 반환하면서 handle mm fault 함수를 끝내버린다.

이 부분은 각각 page 상태에 따라 적당히 처리를 한 후 mapping 하는 부분이다.



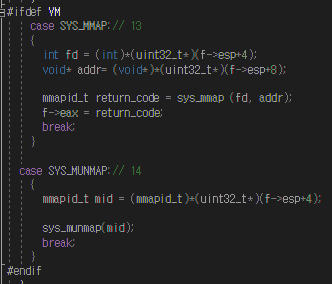
이제 원하는 주소를 user program 에 준 것이다. 다시 handle mm fault 함수가 끝나고 page fault 함수가 끝나고 process 는 재시도를 하게 되면서 program 을 계속 해 나간다. 끝이다.

이제 page-merge-mm test case 를 위한 mmap 과 munmap system call 구현한 부분을 볼 것이다.

이번 프로젝트에는 mmap 관련된 test case 들이 page-merge-mm 하나 빼고 다 빠져있다. 그러나 이 page-merge-mm test case 단 1개 때문에 mmap 과 unmap system call 을 둘 다 구현해야 한다. 의도적으로 mmap 으로 시작하는 test case 를 다 빼고 page-merge-mm test case 만 살짝 남긴 건지 아니면 mmap 으로 시작 하는 test case 를 다 빼서 mmap 관련된 test case 는 다 빠졌다고 생각한 것인지는 모르겠지만 project 4 ppt 에는 mmap 에 대한 설명이 전혀 없으므로 주어진 pintos manual 을 이용해 mmap 과 munmap system call 을 구현해야 한다.

Src/lib syscall-nr.h 와 src/lib/user syscall.c 에는 이미 mmap 과 munmap 이 처리가 되었다. 따라서 userprog/syscall.c 에서부터 처리를 해주면 된다.

Userprog/syscall.c 로 가보자.



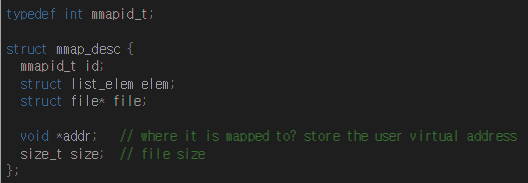
기본 처리를 해주고 sys mmap 과 sys munmap 함수를 호출해서 처리한다.

Mmap 을 위한 구조체들을 보고 가야한다.

Thread.h 에 thread 구조체 안에 mmap 을 위한 list 를 만든다

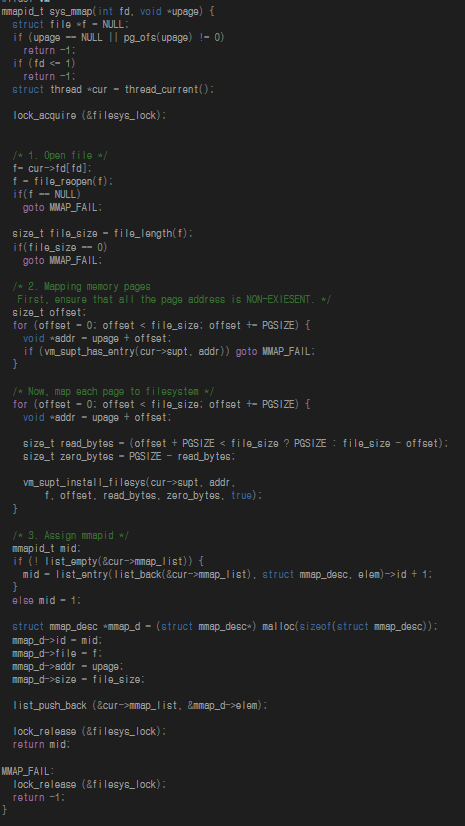


process.h 에 있는 mmap\_desc 구조체를 보자.



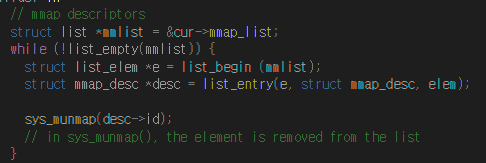
이 구조체 들을 mmap list 에 넣을 것이다. Mmap 은 file 의 내용을 memory 에 기록하는 system call 이다. 따라서 구조체 멤버로써 file 구조체와 memory 어디에 기록할 건지 주소와 크기, 각각의 mmap descriptor 를 구분하기 위한 id를 갖는다.

이제 Sys mmap 함수이다.

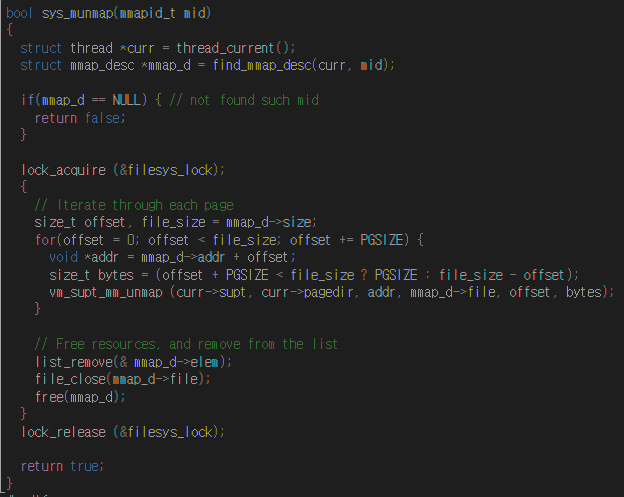


복잡해 보이지만 간단하다. 파일을 열어서 파일을 페이지 사이즈별로 나누어서 supplemental page table 에 page 상태 FROM FILESYS 로 넣어주는 게 전부다. 그리고 각각의 mmap desc 들을 구별하기 위해 id 를 assign 해준다.

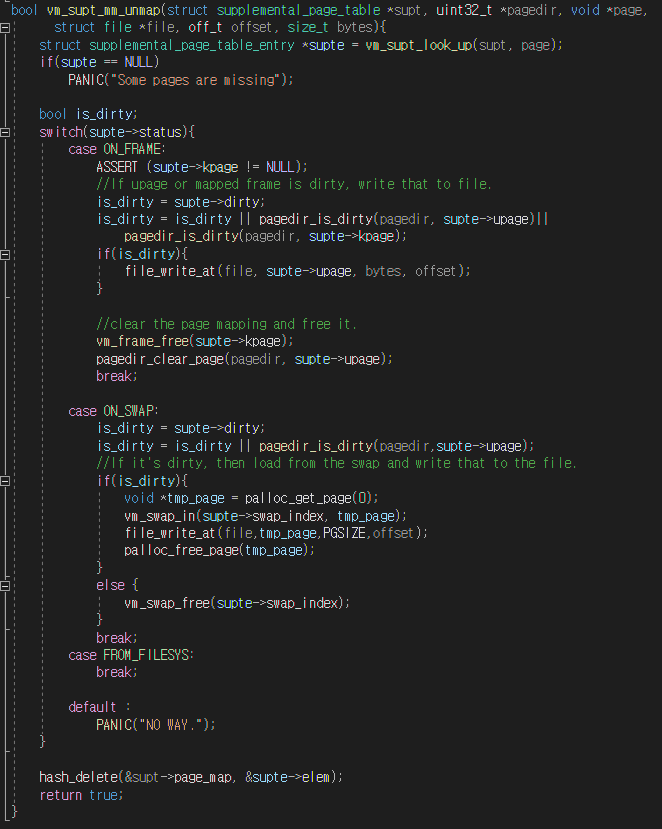
Process 가 끝날 때 process exit 함수가 불러지는 데 여기서 mmap 한 것을 다 unnmap 한다.



Sys munmap 함수를 보면



여기 또한 page 별로 vm supt mm unmap 을 불러 sup page table 에 관련된 일을 처리해주고 mmap list 에서 제거하고 file 을 닫아주고 free 해주는 게 다이다. 중요한 것은 vm supt mm unmap 이므로 그 함수를 보자.



Supplemental page table entry 에 dirty 인지 아닌지를 저장한 것이 여기서 빛난다. 만약 write 에 의해 해당 주소의 data 가 바뀌었을 경우 해당 주소의 상태에 따라서 처리를 해주어야 한다. ON FRAME 이면 file 에 변경 사항을 write 해주어야 하고 ON SWAP 이면 disk 에 저장된 변경된 데이터를 가져와서 file 에 작성을 해주어야 한다.

이로서 모든 test case 들을 해결하였다.

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