**Pintos Project 4 : Virtual Memory**

**(설계 프로젝트 수행 결과)**

과목 명 : OS

담당 교수 : 김영재 교수님

조 / 조원 : 11조 / 장지훈 안시현

개발 기간 : 2018/12/06~12/26

**프로젝트 제목 : pintos project 2 user program**

**제출일 : 2018/12/26**

**참여 조원 : 안시현/장지훈**

1. **개발 목표**

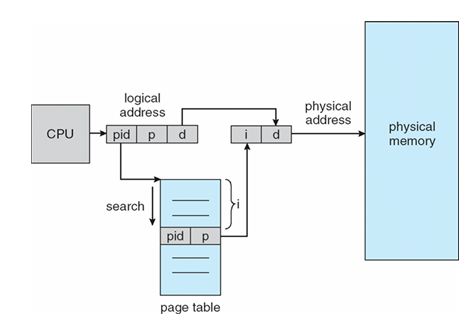
* **Supplemental page table을 통해 page의 swap in, swap out이 가능하게 한다.**
* **Lazy loading을 구현하여 크기가 큰 process의 loading이 가능하게 한다.**
* **Stack growth를 구현한다.**
* **mmap (syscall)을 구현하여 user virtual address에서 file의 read, write가 가능하게 한다.**
* **Swap table을 통해 swap disk를 효율적으로 관리한다.**

1. **개발 범위 및 내용**
   1. **개발 범위**

* **Supplemental page + frame table을 구현한다.**
* **Swap table을 구현한다.**
* **Page fault handler에서 stack growth, swap in + out, 예외 처리를 구현한다.**
* **File read, write 시 buffer의 pinning을 구현한다.**
* **mmap syscall을 구현한다.**
  1. **개발 내용**

**이전까지 Pintos는 stack size가 4KB로 고정되어 있었고, 한 process의 size도 일정 수준 이상 커질 수 없었다. 이번 project에서는 이러한 제한을 제거하여 degree of multiprogramming을 높이고, 더 효율적인 process의 실행을 가능하게 한다.**

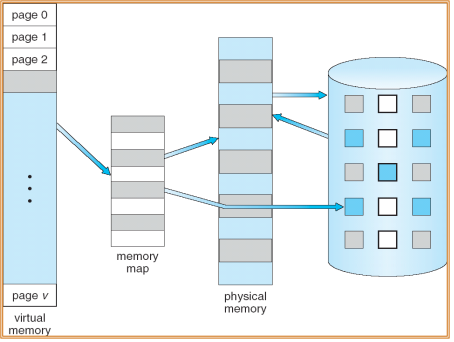
* **inverted page table (역 페이지 테이블)**



**메모리 공간을 절약하기 위해 페이지 테이블을 process마다 만드는 게 아니라 전체 physical memory에 대하여 하나만 만드는 방식으로 frame개수 만큼의 entry가 할당된다. 첫 번째 프레임은 테이블의 첫 번째에 저장**되며 **테이블은 각 프레임은 어떤 프로세스와 대응되는지에 대한 정보**를 담고 있다. **즉, 각각의 entry는 그 frame을 현재 소유하고 있는 Process ID와 page 주소를 가지고 있다. 주소 변환 시에는 frame number에 대응되는 entry를 page table을 검색하며 찾게 되고 offset을 결합하여 physical address를 알아내게 된다.**

**inverted page table을 이용했을 시에는 process별로 생성되던 page table을 모든 process가 공유하는 하나의 page table로 만들었기 때문에 memory space를 절약할 수 있다는 장점이 있지만 , 주소 변환 시에 table을 검색하여 vpn을 찾아내야 하기 때문에 주소 변환 시간이 오래 걸린다는 단점 또한 가지고 있다. 이를 보완하기 위해서 이번 프로젝트에서는 inverted hashed page table을 사용하여 더 빠르게 원하는 page table entry를 찾을 수 있도록 하였다.**

* **Swap disk**

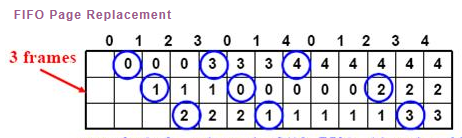
****

**System에서 process에 할당해 줄 physical memory가 부족할 때에는 사용 빈도수가 적은 불필요한 page들을 memory에서 disk로 내보내게 된다. 처리된 page들은 disk에 있는 특정한 공간인 swap disk로 보내지게 되며, 이렇게 필요없는 page들을 memory의 swap disk로 내보내는 과정을 swap-out 이라고 한다. Swap disk는 swap table을 통해 관리된다.**

**.**

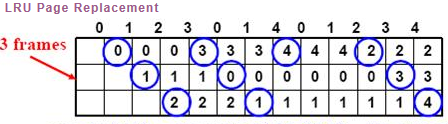
* **page replacement policy**

1. **FIFO page replacement**

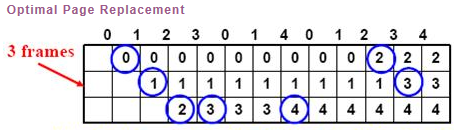
****

**FIFO(First in, Fisrt out) 페이지 교체 알고리즘은 현재 memory에 있는 page들 중 가장 오래된 페이지를 교체하는 알고리즘 이다. 알고리즘이 비교적 간단하여 구현하기 쉬우나 일반적으로 높은 page fault rate를 가지고 있다.**

**2. LRU page replacement**

**LRU(Least recently used) page 교체 알고리즘은 현재 memory에 있는 page들 중 가장 오래 전에 사용 된 page를 내보내는 알고리즘이다. LRU 방식 에서는 각 page마다 마지막 사용 시간을 유지해야 한다. LRU policy는 미래 대신 과거 시간에 대해 적용한 최적 교체 정책으로 생각 될 수 있으나, 구현이 어려워 하드웨어 적인 지원이 필요하다.**

**3. Optimal page replacement**

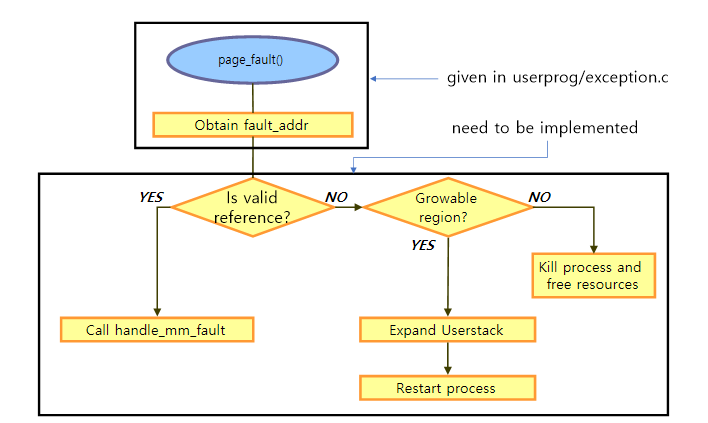
****

**Optimal page 교체 알고리즘은 앞으로 가장 오랫동안 사용되지 않을 page를 찾아서 교체하는 알고리즘이다. 실질적으로 제일 낮은 page fault rate를 보이지만, 미래에 써야할 page를 아는것은 사실상 불가능 하기 때문에 실제 구현이 어렵다. 따라서 optimal page 알고리즘은 주로 현재 알고리즘이나 새롭게 적용되는 알고리즘의 효율성을 알아보기 위해 비교 목적으로 주로 사용된다.**

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **12/6 ~ 12/10 : 매뉴얼 이해 및 stack growth 구현**
* **12/11 ~ 12/21 : page + frame table, swap table 구현**
* **12/22 ~ 12/25 : page-merge test case 관련 file IO 동기화, pinning 구현, mmap 구현**
* **12/26 : 보고서 작성**
  1. **개발 방법**

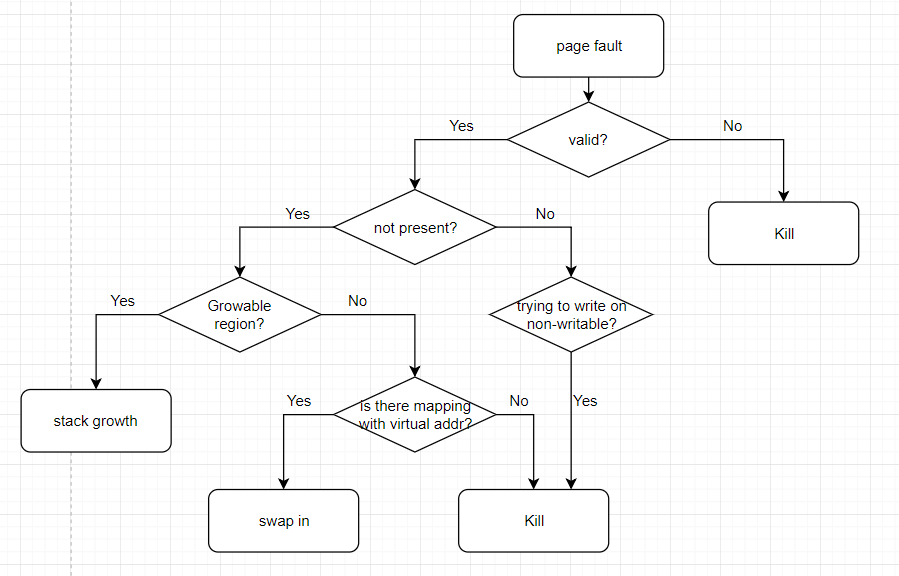
1. **stack growth : stack growth는 demand가 있을 시에만 수행된다. 즉, page fault handler에서 fault address가 stack 영역이고, write를 시도하려 할 때에 stack growth가 필요한 상황으로 판단하고 해당 fault addr에 해당하는 virtual page를 page table에 삽입하고, mapping을 한다.**
2. **load stack segment : process의 크기가 커지는 경우 (global 변수로 size가 큰 배열이 선언되어 code section의 크기가 커지는 등) process의 load 중 stack에 해당하는 page의 할당에 실패할 수도 있다. 이는 일어나서는 안 된다. 따라서 이를 방지하기 위해, stack segment의 frame 할당 시 반드시 성공하도록 (만약 frame이 부족하면 eviction을 한 후 stack의 frame을 할당하도록) 한다.**
3. **supplemental page table : inverted hashed page table로 구현했다. 즉, vpn과 pid를 함께 hashing하여 인덱스를 얻은 후, 이에 mapping된 frame number를 얻었다. 이를 통해 각 thread마다 page table을 만들 필요 없이 하나의 page table로 전체 frame을 관리할 수 있으므로, frame table을 따로 구현할 필요 없이 virtual page와 mapping된 frame들에 대해서 eviction을 수행할 수 있다.**
4. **swap disk의 관리 : swap table을 통해 swap disk의 space를 관리한다. swap disk는 몇 개의 sector로 나누어지며, 각 sector에는 index가 있고 이 index를 통해 random access가 가능하다. swap table은 어떤 index의 sector의 사용 여부를 저장하며 (배열로 구현했다.) sector가 만약 사용 중인 경우, 어떤 virtual page와 mapping되어 있는지는 위에서 구현한 page table의 entry에 저장한다.**
5. **page fault handler : 다음의 순서도를 구현하였다.**

****

1. **pinning : user program의 read, write system call의 경우 buffer ~ buffer + size에 해당하는 모든 virtual page가 frame에 mapping되어 있어야 하고, eviction의 후보가 되면 안 된다. 즉 pinning을 구현해야 하는데, page table entry의 structure에 pinned라는 boolean 변수를 추가하여 file read, write 전에 해당 virtual page들을 physical memory에 올려놓고 pinned = true로 세팅한다. eviction 시 pinned = true일 경우 eviction의 후보에서 제외하였다.**
2. **mmap : mmap (syscall)은 load\_segment에서 virtual page에 frame을 mapping하는 것과 유사한 방식으로 구현한다. 또한, child thread가 exit할 때 매핑된 virtual page의 file 내용을 다시 disk의 file에 write한다.**
   1. **연구원 역할 분담**

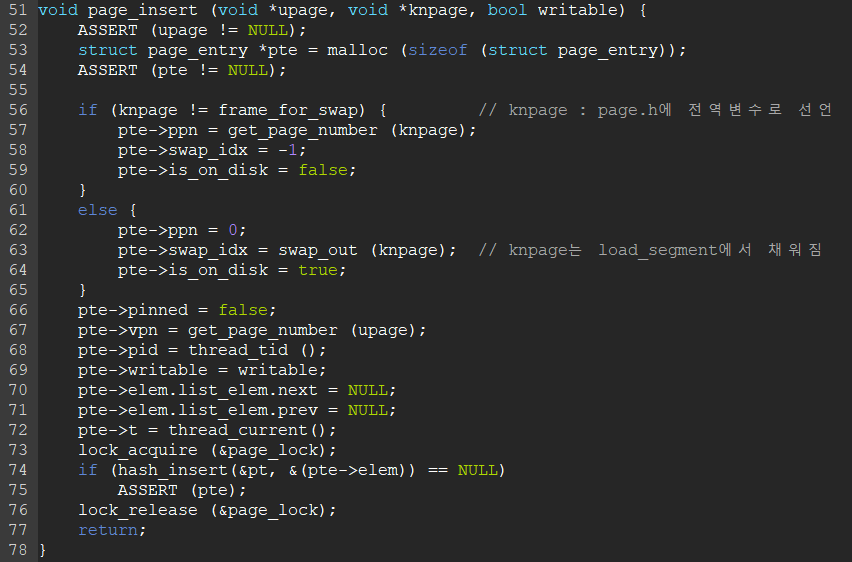
* **장지훈 : stack growth, load stack segment, supplemental page table, mmap**
* **안시현 : page fault handling, swap table, FILE I/O synchronize, pinning**

1. **연구 결과**
   1. **합성 내용**

* **Flow chart**
* 
  1. **제작 내용**

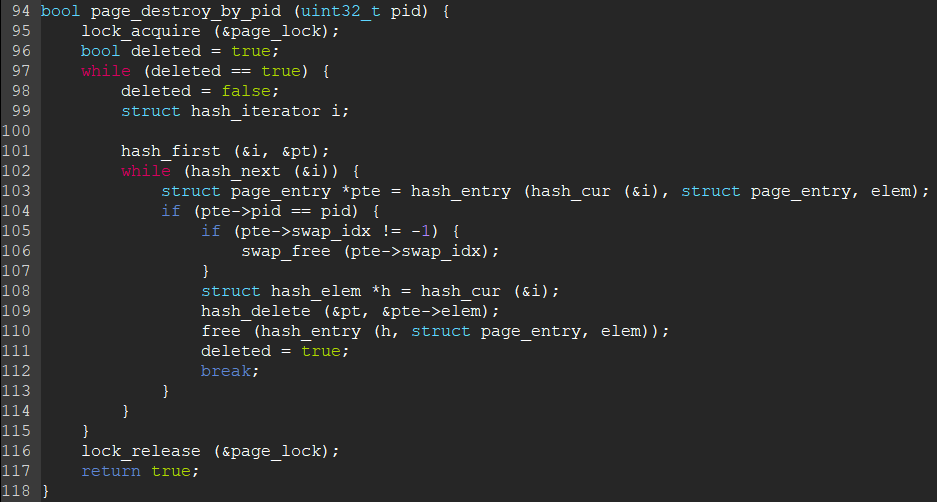
1. **supplemental page table (in ‘vm/page.c’):**

* **void page\_insert (void \*upage, void \*knpage, bool writable) : user program의 load 시 virtual address space에 frame 또는 disk의 mapping이 생길 때마다 호출된다. mmap 호출 시 file의 내용을 virtual address space에 mapping할 때도 호출된다. 코드는 다음과 같다.**

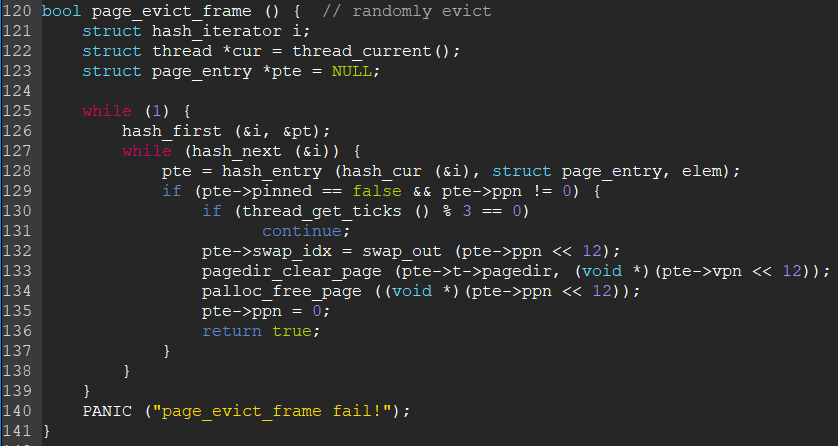
****

**frame\_for\_swap은 lazy loading을 위한 frame이다. 전역변수로 선언되어 있고, kernel의 시작 시 초기화된다. lazy하게 load하고자 하는 page의 내용을 disk에 전달하는 역할을 한다.**

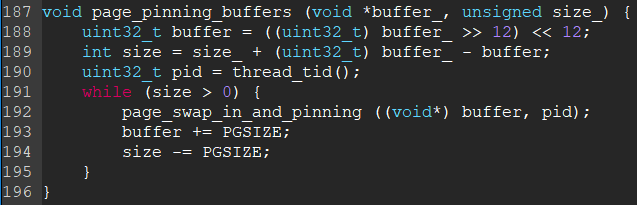
* **bool page\_destroy\_by\_pid (uint32\_t pid) : parameter로 넘어온 pid의 page entry를 supplemental page table에서 모두 지우는 역할을 한다. thread\_exit() 함수에서 호출된다.**

****

* **bool page\_evict\_frame () : supplemental page table에 있는 모든 pinning되어 있지 않은, frame과 mapping되어 있는 entry 중 random하게 하나를 선택하여 swap\_out 시키고 frame을 free시킨다. 코드는 다음과 같다.**

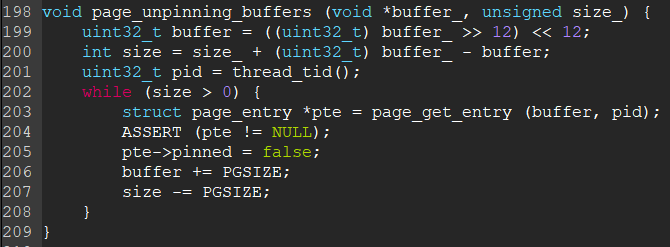
****

* **void page\_pinning\_buffers (void \*buffer\_, unsigned size\_) : read, write syscall 함수에서 호출된다. parameter로 전달된 buffer ~ buffer + size에 해당하는 virtual address space에 대해, swap out된 virtual page는 swap in 해주고 모든 virtual page에 대해 pinning을 해줘 evict의 후보가 되지 않도록 한다. 같은 thread에 의해 lock이 2번 걸리지 않게 하기 위해 반드시 필요하다. 코드는 다음과 같다.**

****

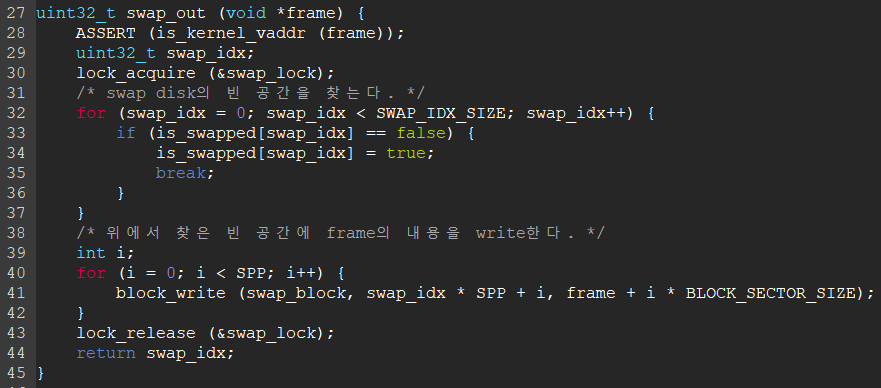
**page\_swap\_in\_and\_pinning 함수는 인자로 전달된 virtual page number와 pid에 해당하는 page entry를 찾아서 swap in 해주고 pinning을 해준다.**

* **void page\_unpinning\_buffers (void \*buffer\_, unsigned size\_) : file read, write가 끝난 후 page를 unpinning해준다.**

****

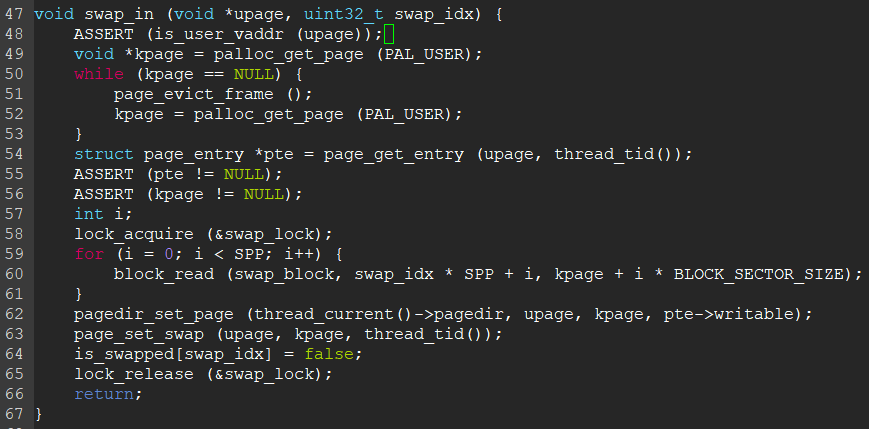
**2) swap disk (in ‘vm/swap.c’)**

* **uint32\_t swap\_out (void \*frame) : parameter로 넘겨진 frame을 disk의 연속된 빈 sector에 write한 후, write한 sector의 첫 번째 index를 return한다. 코드는 다음과 같다.**

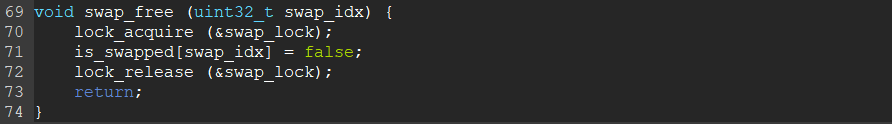
****

**is\_swapped[]는 bool 배열로, swap table의 역할을 한다.**

* **void swap\_in (void \*upage, uint32\_t swap\_idx) : parameter로 넘겨진 swap\_idx부터 PGSIZE ( = 4,096KB)만큼을 upage로 read한 후, 해당 frame을 인자로 넘어온 upage에 매핑시켜준다. 코드는 다음과 같다.**

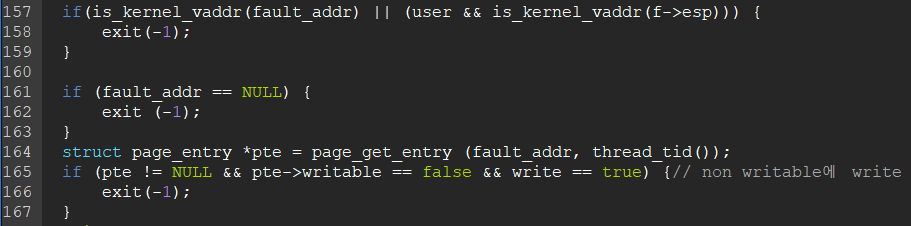
****

* **void swap\_free (uint32\_t swap\_idx) : parameter로 넘겨진 swap\_idx를 free시킨다. 코드는 다음과 같다.**

****

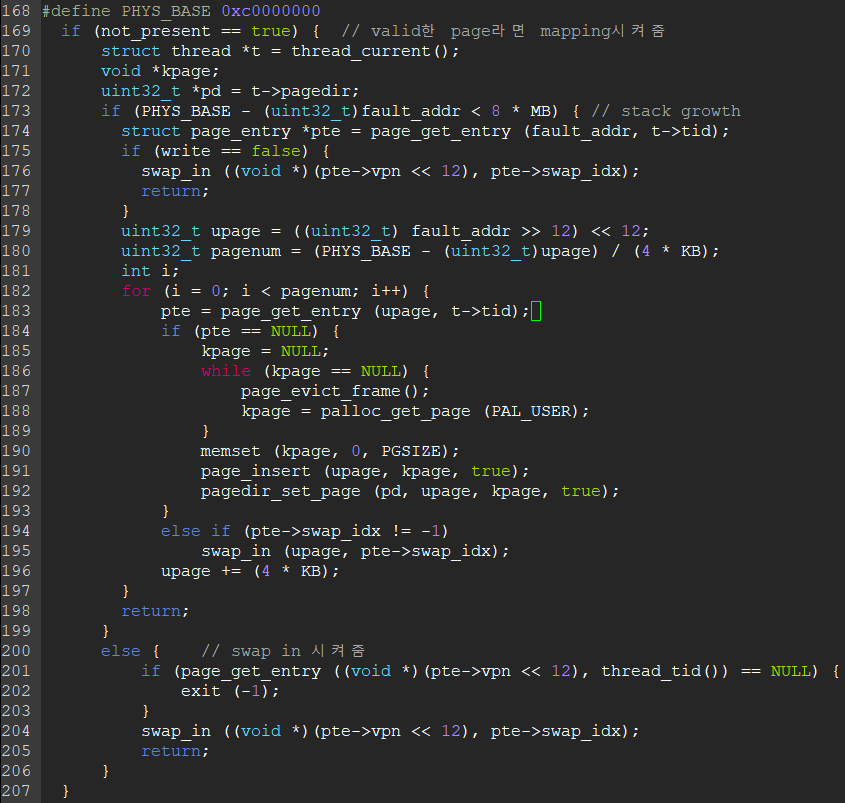
**3) page fault handler (in ‘userprog/exception.c’)**

* **우선 page fault의 원인에 따라 처리해야 할 page fault인지, 비정상 종료시켜야 할 page fault인지 확인한다.**

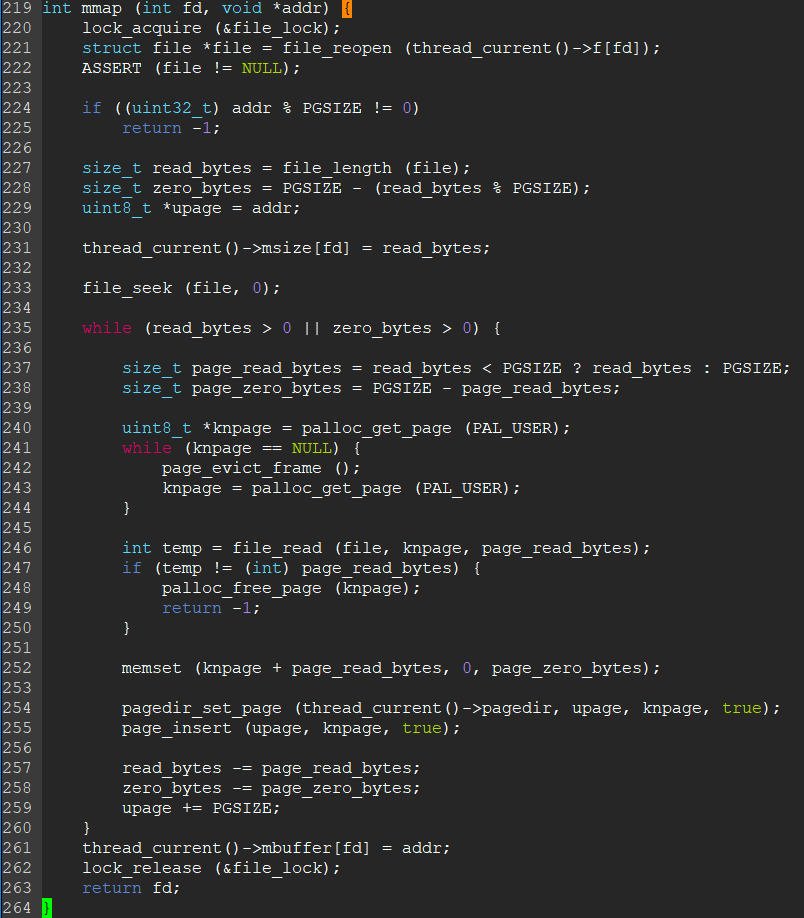
****

* **not present = true인 경우, 다음의 2가지 상황을 고려할 수 있다.**

**만약 stack growth를 demand한다면 해당 virtual page에 frame을 매핑시켜준다. 또한 swap out된 virtual page에 access하려 하는 경우에는 swap in 시켜준 후 user program을 계속 진행한다.**

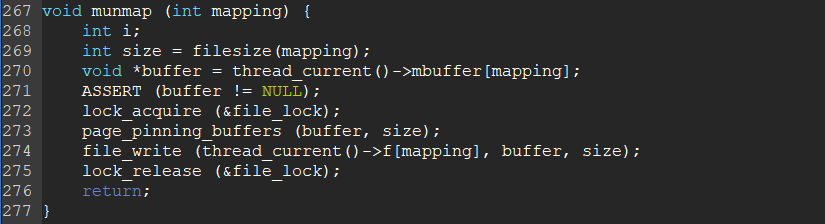
****

**4) void mmap (int fd, void \*addr) (in ‘userprog/syscall.c’) : fd에 해당하는 file을 user address space인 addr에 mapping시킨다. fd에 해당하는 파일은 user program에서 이미 open system call을 통해 오픈되어 있다. addr ~ addr + filesize(fd) 만큼의 user address space에 file을 frame에 write한 후 mapping시키는 것을 반복한다. load\_segment에서와 같은 방식으로 mapping을 진행하며, 코드는 다음과 같다.**

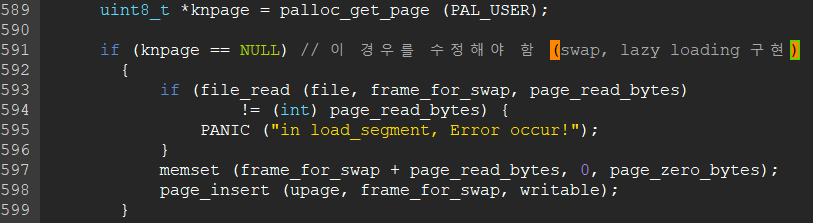
****

**mmap의 return값은 fd와 같은 값을 취했으며, thread structure 안에 mbuffer, msize를 정의하여 fd에 해당하는 파일의 길이, 파일이 mapping된 user address의 시작 주소를 저장하였다.**

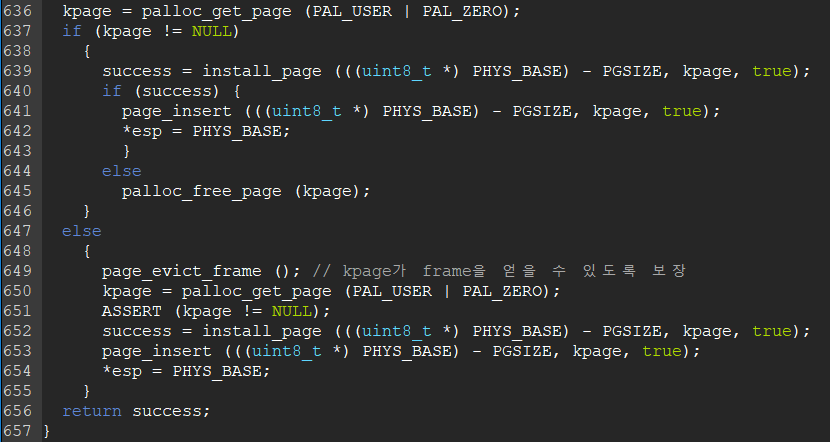
**5) void munmap (int mapping) (in ‘userprog/syscall.c’) : exit에서 호출되며, mmap에서 mapping된 파일이 user address space 내에서 수정되었을 때 이를 다시 file에 write해주고 mapping을 지운다. parameter로 넘어온 mapping은 mmap에서 return되었던 값이여야 한다.**

****

**6) lazy loading (in ‘userprog/process.c’) : load\_segment에서 virtual page와 mapping 가능한 frame을 할당할 수 없는 경우, 위에서 언급했던 frame\_for\_swap을 이용하여 swap disk에 읽은 내용을 저장한다. load\_segment에서 이를 수행하는 코드는 다음과 같다.**

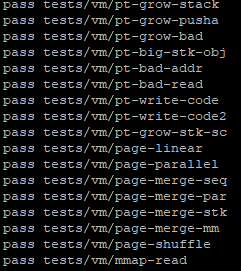
****

**7) stack segment에 대한 frame 할당 보장 (in ‘userprog/process.c’) : load 함수에서 stack에 argument를 쌓기 위해서는 setup\_stack 함수에서 stack segment에 해당하는 frame이 mapping됨을 보장해야 한다. 이를 위해서, 만약 stack segment에 할당할 frame이 없는 경우 page\_evict\_frame 함수를 통해 새로운 frame을 얻을 수 있도록 보장하였다.**

****

* 1. **시험 및 평가 내용**

**Make check 결과 첨부**

* 

1. **기타**
   1. **연구 조원 기여도**

* **안시현(50%)**
* **장지훈(50%)**
  1. **소감**
* **안시현 : 이번 pintos project를 통해 virtual page와 page replacement 관련 알고리즘을 구현하면서 수업시간에 배운 page table이나 swap-in swap-out, 그리고 memory와 disk의 상호작용 등이 정확히 어떤식으로 진행되는지에 관해 더 자세히 이해할 수 있었다. 또한 , Swap table을 구현하면서 그동안 수업에서 배웠던 page replacement 방식에 대해 조금 더 이해 할 수 있었다. 마지막 project까지 잘 끝내게 되어서 뿌듯하고 모든 것을 배운 상태에서 pintos project를 다시 진행해 보면 배운것을 잘 점검 할 수 있을 것 같다.**
* **장지훈 : 이번 프로젝트를 통해 이론으로만 배운 page table을 기초적인 수준에서 구현함으로써, page table이 OS에서 메모리의 효율적인 관리를 위해 어떤 역할을 수행하는지, 어떻게 degree of multiprogramming을 높이는지 확실히 이해할 수 있었다. 또한 physical memory, disk로의 메모리 계층 구조를 고려하여 어떻게 process를 더 빨리 실행시킬 수 있을지, 효율적으로 메모리를 사용할 수 있을 것인지에 대해서도 생각해볼 수 있었다.**