**Pintos Project 4: Virtual Memory**

담당 교수 : 박성용

이름 / 학번 : 이선호 / 20171665

개발 기간 : 22.11.15 ~ 22.11.30

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이번 프로젝트에서는 물리 메모리 뿐만이 아니라 디스크의 일부도 가상 메모리로 사용하는 것이 목표이며, 이를 위한 page fault handler와 가상 메모리 주소와 물리 메모리 주소를 mapping 시킬 수 있는 page table를 만든다. 또한 swap disk를 사용하여 필요한 프레임을 메인 메모리에 올리는 swapping을 수행하기 위해 메모리에 어떠한 페이지를 대체해야 하는지에 관한 LRU 알고리즘도 구현한다. 그리고 stack 영역에서 page fault 발생 시 stack growth를 통해 추가로 페이지를 할당할 수 있도록 한다.

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

가상 메모리 주소와 물리 메모리 주소를 mapping 할 수 있는 page table을 위한 자료구조를 추가해야 한다. 메모리에 가상 메모리 주소에 대응되는 찾고자 하는 물리 메모리 주소가 존재하지 않으면 page fault를 발생시키는데, 이때 page fault가 어떠한 이유로 나타났는지를 분석하여 디스크에서 필요한 프레임을 메모리로 가져오거나 실제 물리 메모리가 부족할 시에는 swapping, stack growth 등 적절한 메모리 처리를 통해 해당 interrupt를 처리할 수 있도록 구현해야 한다.

* 1. Disk Swap

Page fault 발생 시 프로세스를 종료하는 것이 아니라 swap disk에 thread가 원하는 프레임이 존재하는지 살펴보고, 존재하면 기존에 메모리에 있는 페이지 하나를 선택하여 디스크에서 가져오고자 하는 페이지와 교체하는 swap in과 swap out을 구현해야 한다. 이때, 메모리에서 어떠한 페이지를 택하여 교체해야 하는지 그에 대한 정책으로서 LRU replacement 알고리즘을 구현해야 한다. 이를 통해 물리적인 메모리 용량이 부족하더라도 디스크의 swap space를 사용하여 가상 메모리 관점에서 더 큰 메모리를 사용할 수 있도록 할 수 있다.

* 1. Stack Growth

기존의 PintOS에서는 stack이 단일 페이지인 4KB로 구성되어 있었으나, 만약 user stack 영역을 넘어서는 메모리에 접근할 시 page fault가 발생하면 stack의 크기를 확장시킬 수 있어야 한다. 단, 최대로 확장시킬 수 있는 stack의 크기는 8MB이다.

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

프로세스가 접근하고자 하는 가상 메모리 주소에 대응되는 물리 메모리 주소가 존재하지 않아 invalid page에 접근하려고 하면 page fault가 발생한다. 이때 OS에서 interrupt가 발생하여 이를 해결하려는 몇 가지 시도를 수행할 수 있다. 먼저 해당 가상 메모리 주소로 접근하는 것이 invalid reference인지 page lookup을 통해 확인한다. 만약 reference가 valid하면 page fault를 발생하게 한 가상 메모리 주소의 내용을 디스크에서 메모리로 가져올 수 있는지를 확인한다. 그런데 프레임을 메모리로 가져올 수 없을 만큼 남아있는 메모리 영역이 부족하면 disk swap를 수행할 수 있는지를 확인한다. Invalid reference이면 영역의 크기를 늘릴 수 있는지 확인하고, 이러한 처리가 가능한 user stack 영역이라면 stack 영역에 빈 페이지를 추가로 할당하여 stack의 크기를 증가시킨 후 프로세스를 재시작한다.

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

앞서 설명한 valid reference 처리 시 디스크에 있는 프레임을 메모리에 적재할 만큼 남아 있는 메모리 양이 충분치 않으면 기존에 메모리에 존재하는 특정 페이지를 골라서 이를 디스크로 swap out하고, 대체한 페이지에 가져오고자 하는 프레임을 swap in 해야 한다. 이때 어떠한 페이지를 축출해야 할지를 정해야 하는데, 이러한 정책으로서 가장 오래 전에 사용되었던 페이지를 고르는 LRU replacement를 사용할 수 있다. 그러나 본 프로젝트에서는 FIFO 방식과 함께 LRU replacement보다 좀 더 좋은 방법으로 알려진 second chance를 구현했다. 각 페이지에 대응되는 reference bit이 존재하여 페이지가 처음에 메모리에 적재되면 1로 set 된다. 또한 현재 메모리에 있는 페이지들이 circular list로 관리된다. 만약 축출해야 할 페이지를 택해야 할 때는 이 list를 순환하여 탐색하면서 reference bit이 1인 페이지가 한 번 확인이 되면 reference bit이 0으로 set되고, 원래 0이었던 페이지를 확인하면 그 페이지를 택하여 축출한다.

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

User stack에 접근할 시 기존에 할당되어 있던 stack 영역을 초과한 곳을 접근하려고 할 때 해당 영역이 적절한 stack frame 영역인지 확인하기 위해 크게 다음과 같은 조건을 확인한다.

페이지를 PHYS\_BASE에서 esp가 가리키는 영역 안에 모두 할당했을 때 stack의 크기를 초과하는지 또는 page fault가 발생한 메모리 주소가 esp – 32보다 작은지 확인하고, 해당 조건을 만족하면 stack growth를 수행할 수 없다. 이외의 조건에서는 해당 메모리 주소에 위치한 공간에 도달할 수 있을 때까지 페이지를 할당할 수 있을 만큼 stack\_pointer를 이동해 간다. 후자에서 esp – 32를 가지고 조건을 판별하는 이유는 PintOS 명세서에 안내된 바에 따라 스택 접근에 허용되지 않는 명령어인 PUSHA가 32 바이트를 한 번에 넣는 것을 막기 위해서다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2022.11.16 ~ 11.18: Page Fault Handler와 stack growth

2022.11.19 ~ 11.21: Page Table과 Disk swap

2022.11.22 ~ 11.24: process.c 코드 수정과 system call에서 예외처리 추가

2022.11.25 ~ 11.30: 오류 수정 및 보고서 작성

* 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

먼저 이전에 project 3에서 synch.c 파일의 sema\_up 함수에서 thread\_yield 함수를 호출하는 부분이 project 4의 user program 실행 부분과 충돌하는 문제를 확인하여 이를 주석 처리했다. 또한 make로 컴파일 시 extern으로 선언한 thread\_prior\_aging가 문제가 되므로 extern 선언을 지웠다.

**Page Table & Page Fault Handler**

vm 디렉토리 내 page.h, page.c, frame.h, frame.c 파일을 추가하여 페이지와 프레임, 그리고 둘을 mapping 시킬 수 있는 page table을 위한 자료구조를 생성한다. 어떤 페이지가 어떠한 프레임에 대응되는지를 mapping 시킬 수 있도록 페이지 구조체와 프레임 구조체에는 각각 virtual address와 physical adress를 서로 가리키는 포인터 멤버 변수를 선언한다. thread.h에는 page table을 PintOS에서 기본적으로 제공하는 자료구조인 hash로 선언했고, 페이지 구조체에는 각 페이지마다 page table의 원소로 저장될 수 있도록 hash\_element의 elem 멤버 변수를 추가했다.

page.h와 page.c에는 다음과 같은 함수들을 추가했다.

* page\_table\_init: 페이지 테이블을 초기화한다.
* page\_insert: 페이지 테이블에 페이지를 삽입한다.
* page\_lookup: 현재 실행 중인 thread의 페이지 테이블에서 가상 메모리 주소에 대응되는 페이지를 찾는다.
* page\_lookup\_with\_table: 원하는 페이지 테이블의 가상 메모리 주소에 대응되는 페이지를 찾는다.
* page\_table\_destroy: 페이지 테이블을 삭제한다.

frame.h와 frame.c에는 다음과 같은 함수들을 추가했다.

* frame\_lookup: 프레임 list에서 물리 메모리 주소에 대응되는 프레임을 찾는다.
* frame\_delete: 프레임 list에서 물리 메모리 주소에 대응되는 프레임을 삭제한다.
* frame\_allocate: 물리 메모리 주소와 가상 메모리 주소에 대응되는 프레임을 할당하여 프레임 리스트에 삽입한다.

userprog 디렉토리에 위치한 exception.c에서 page\_fault의 마지막의 특정 조건 시 syscall\_exit 함수를 호출하는 부분을 주석 처리하고, page\_lookup 함수를 통해 페이지가 페이지 테이블에 존재하면 handle\_mm\_fault 함수를 호출하여 프레임을 추가한다. 만약 frame을 할당할 수 있고, 찾은 페이지의 물리 메모리 주소가 존재하지 않으면 현재 물리 메모리에 존재하지 않는다는 의미이므로 swap을 수행한다.

**Disk Swap**

vm 디렉토리 내 swap.h, swap.c 파일을 추가하고, swap disk에서 어떠한 block을 swap할 수 있는지를 PintOS에서 기본적으로 제공되는 자료구조인 bitmap type의 swap\_bitmap을 통해 관리한다. 또한 Disk swap을 수행할 때 어떠한 물리 메모리의 프레임을 대체할지를 정하는 LRU replacement 알고리즘을 구현하기 위해 frame.h와 frame.c에 다음과 같은 함수들을 추가했다.

* frame\_table\_init: 프레임 테이블 리스트를 초기화한다. 여기서의 프레임 테이블은 LRU 알고리즘을 수행하기 위해 가장 오래 전에 사용된 프레임을 찾고자 하는 리스트이다.
* select\_victim\_frame: 어떠한 프레임을 축출해야 할지를 LRU replacement에 근사한 second chance 알고리즘을 통해 구한다.

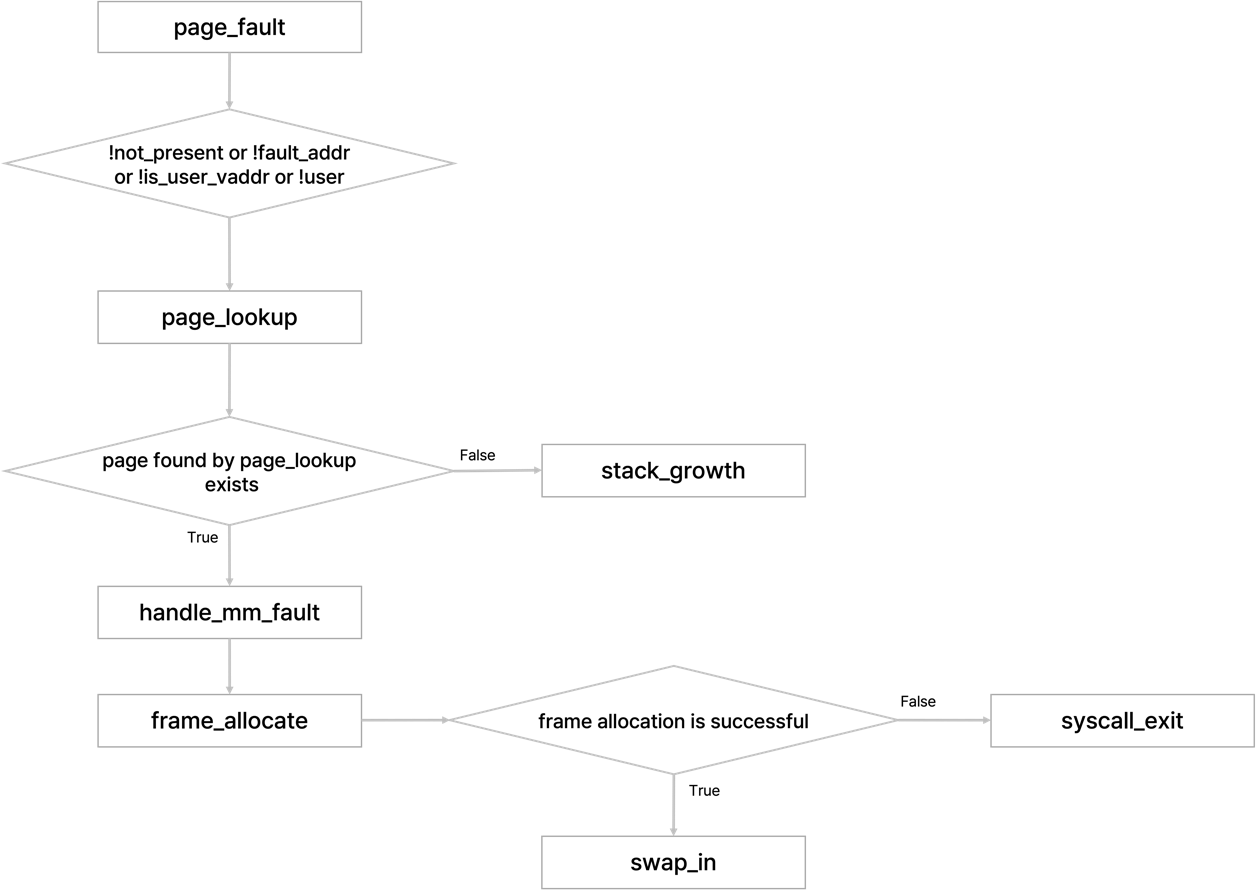
**Stack Growth**

userprog 디렉토리 내 exception.c 파일에 stack\_growth 함수를 추가했다. stack\_growth를 수행하기 위해서는 stack 영역의 top을 가리키는 esp가 필요한데, 이를 위해 thread.h의 멤버변수로 stack\_pointer를 추가했다. page fault를 통해 온 경우 stack 프레임 안에는 존재하지만 stack top을 벗어나 위치한 메모리 영역을 가리키는 경우이므로, 해당 메모리 주소까지 페이지를 할당할 수 있는지 확인하여 가능한 만큼 페이지를 할당하는 작업을 stack\_growth 함수에서 수행한다.

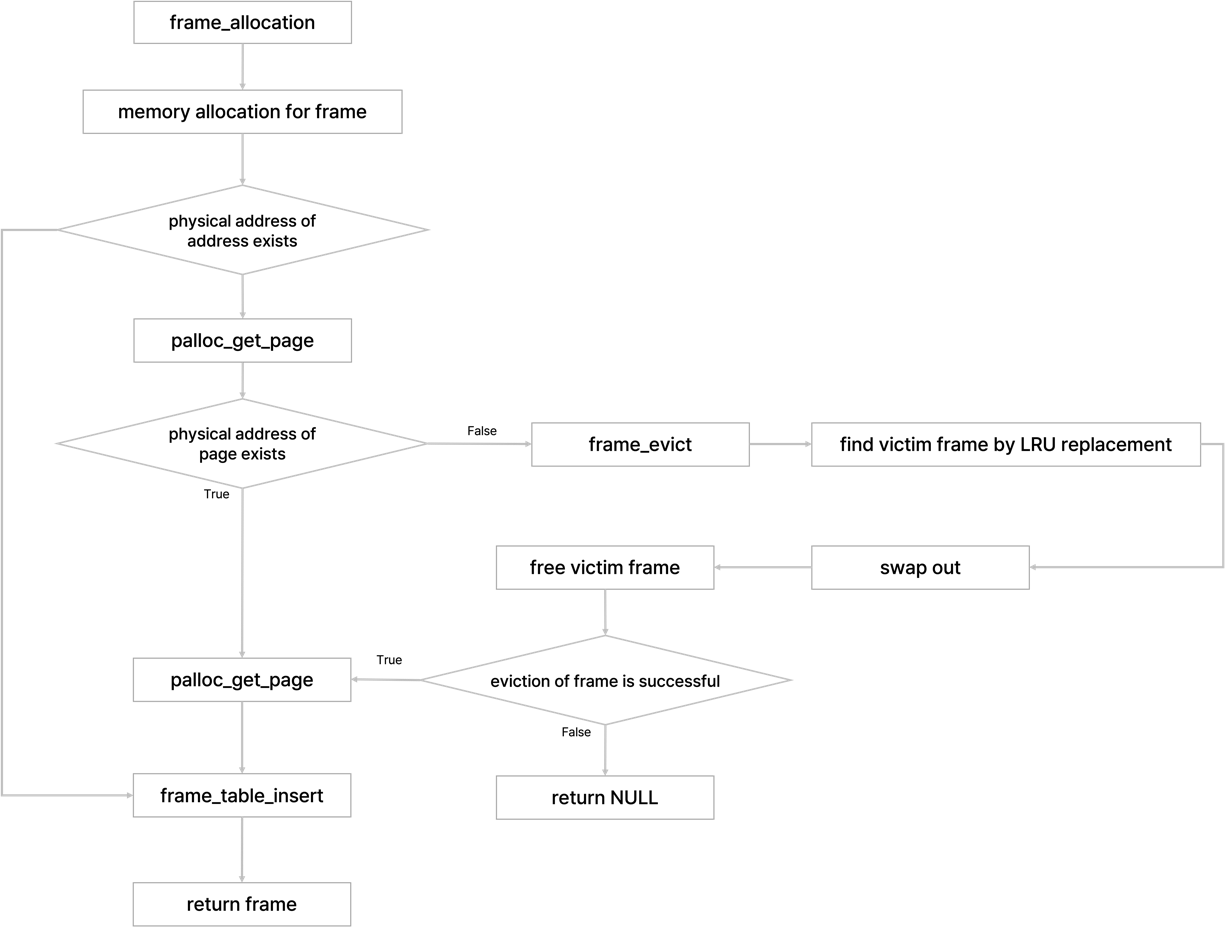
1. **연구 결과**
   1. **Flow Chart**

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

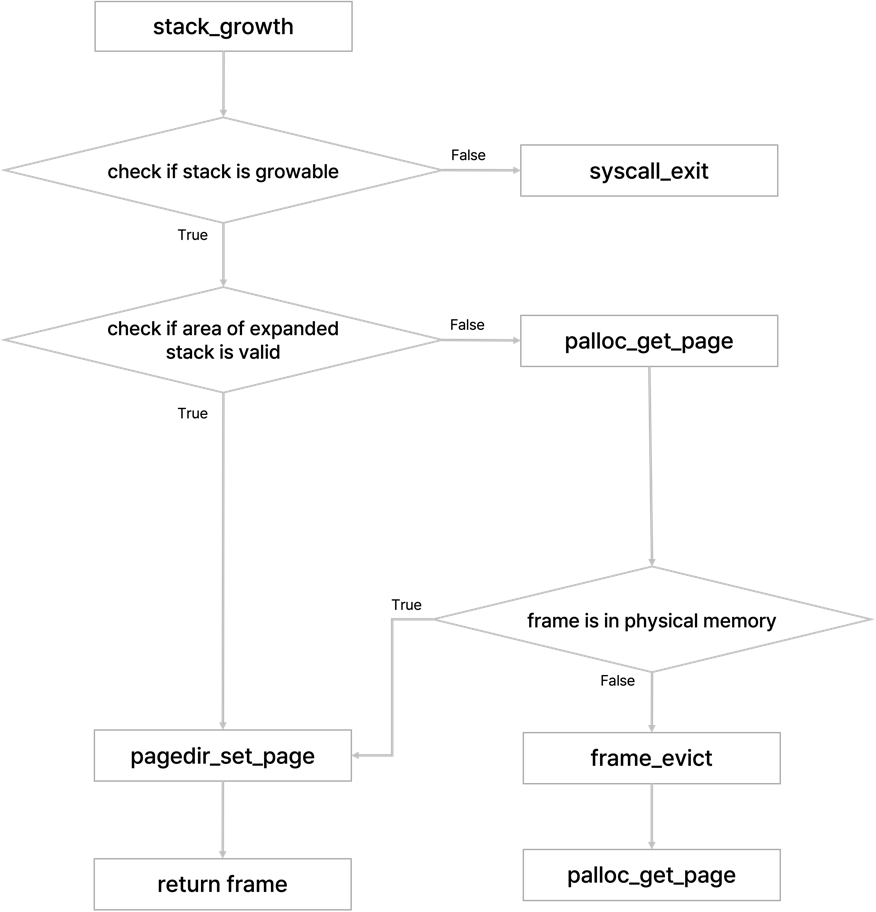
**Page Table & Page Fault Handler**



**Disk Swap**



**Stack Growth**



* 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

**Page Table & Page Fault Handler**

텍스트이(가) 표시된 사진

자동 생성된 설명

페이지와 프레임을 가리키는 별도의 page table entry를 만들기보다는 page와 frame이 모두 virtual\_address와 physical\_address 멤버변수를 지니도록 하여 어떠한 가상 또는 물리 메모리 주소가 요청되었을 때 page\_lookup, page\_lookup\_with\_table 또는 frame\_lookup 함수를 통해 그 주소에 대응되는 페이지 또는 프레임에 접근할 수 있도록 하기 위함이다. page\_entry 구조체에는 swap\_index라는 멤버변수를 추가했는데, swap space에 존재하지 않고 메모리에 적재된 페이지라면 swap\_index는 -1이고, swap space에 존재하면 swap space에 대응되는 swap\_block의 index를 갖는다. writable은 현재 page가 read-only가 아닌지를 확인하는 용도이며, loaded는 swap space가 아니라 물리 메모리에 적재되어 있는지 여부, pinned는 page-merge-stk와 page-merge-mm을 통과하기 위해 시도한 멤버변수로서 실제 최종 제출에서 사용한 코드에서는 큰 의미가 없다. 페이지 테이블을 각 thread 별로 hash type의 변수로 갖게 했으므로 이에 대응되는 page table의 entry를 가리킬 수 있도록 hash\_elem type의 멤버 변수를 선언했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

frame\_entry 구조체에도 추가적인 멤버 변수를 사용했는데, disk swap 발생 시 어떠한 프레임을 축출해야 하는지를 고르는 과정에서 프레임의 리스트를 순환하여 탐색하기 위해 f각 frame이 프레임 리스트에서 어떠한 원소에 속하는지를 가리키는 list\_elem type의 elem 멤버 변수를 추가했다. 또한 frame이 어떠한 thread에 속하는지를 알 수 있도록 owner\_thread라는 멤버 변수를 추가했는데, 이는 frame을 메모리에서 축출하거나 삭제할 때 이에 대응되는 페이지도 clear 되어야 하는데, 이때 페이지 테이블을 thread별로 갖도록 했으므로 이를 조회하기 위한 용도로 사용된다. unused\_count는 LRU replacement에 근접한 알고리즘인 second chance로 축출할 페이지를 택할 때 필요한 reference bit을 구현한 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

페이지를 추가로 생성하는 함수인 page\_insert는 위와 같이 구현했다. 먼저 malloc으로 페이지를 heap에 할당하고, page의 멤버 변수인 virtual\_address와 physical\_address에 가장 가까운 페이지 시작 위치로 내림한 주소를 할당한다. 그리고 현재 실행 중인 thread의 page\_table에 접근하여 요청한 가상 또는 물리 메모리주소를 지니는 페이지가 page\_table에 존재하는지 확인하고, 존재하면 페이지 테이블에 hash\_entry로 만들어서 이를 page\_table에 삽입해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

탐색하고 싶은 페이지를 가리키는 페이지 테이블이 어떠한 thread에 속한 page\_table인지를 인자로 전달받으면 그 가상 메모리의 주소를 지니고 있는 페이지를 page\_table에서 hash로 탐색하며, 만약에 존재하면 그 hash\_entry를 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

페이지 테이블을 초기화할 때 페이지 정렬이 필요한데, 이때 페이지의 가상 주소의 오름차순으로 정렬될 수 있도록 그 기준을 정의한 함수이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

페이지 테이블에서 조회한 hash\_entry를 가지고 해당 페이지의 가상 주소에 대응되는 hash의 bucket 번호를 구하는 함수이다. 명세서에서는 구현이 요구되었으나 실제 구현 과정에서 사용되지는 않았다.

텍스트이(가) 표시된 사진

자동 생성된 설명

프레임을 할당하는 함수의 내용도 페이지를 추가하여 삽입하는 과정과 크게 다르지 않다. 다만 인자로 받은 physical\_address가 NULL일 때 이는 메모리에 존재하지 않는 page라는 의미이므로 swap disk에서 대응되는 프레임을 메모리로 적재해 와야 한다. 이를 위해 evict\_frame 함수를 호출하여 LRU replacmenet 알고리즘에 의해 페이지를 축출하고 앞서 생성한 새로운 프레임을 프레임 리스트에 할당하여 메모리에서 접근 가능하도록 한다.

텍스트, 스크린샷, 화면이(가) 표시된 사진

자동 생성된 설명

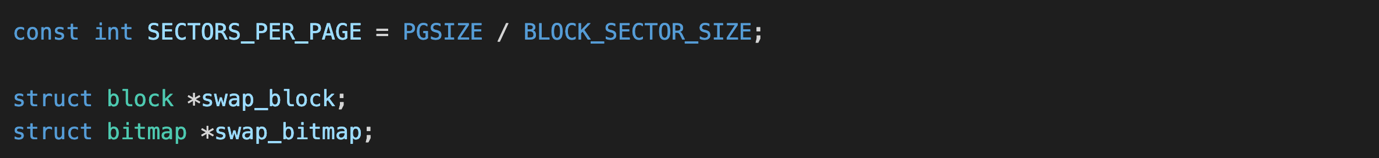
page\_handler의 마지막 부분에서 기존에 invalid한 주소인지를 확인하는 부분을 주석 처리하고 사용자 영역의 메모리 주소인지를 파악할 수 있는 is\_user\_vaddr 함수를 이용하여 가상 메모리 주소 자체가 invalid한지를 파악한다. 만약에 valid reference이면 해당 가상 메모리를 지니고 있는 페이지를 page table에서 탐색하고, 그러한 페이지가 존재하면 handle\_mm\_fault 함수를, 존재하지 않으면 stack의 크기를 늘릴 수 있도록 했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

앞서 page\_handler 수행 시 가상 메모리에 대응되는 페이지가 page table에는 존재함이 확인되었으므로 물리 메모리에는 해당 페이지에 대응되는 프레임이 존재하지 않다는 의미이다. 그래서 handle\_mm\_fault 함수에서는 찾고자 하는 페이지에 해당되는 프레임을 새롭게 할당하고, 프레임이 제대로 할당되었으면 swap in을 수행하여 디스크에서 물리 메모리로 이동시킨다.

**Disk Swap**



디스크의 swap space를 block(sector) 단위로 관리하는데, 이를 위해 한 페이지에 몇 개의 block이 대응될 수 있는지를 구하기 위해 전체 page 크기를 한 block 크기로 나눈 값을 전역 변수로 사용했다. 또한 swap\_block은 block swap space를 의미하고, swap\_bitmap은 swap disk에 있는 block이 swap이 되었는지 안 되었는지를 판단하기 위한 용도로서 사용된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

swap\_bitmap을 모든 block에 관해 처음에 false로 초기화해주기 위한 용도이다. 기존에 PintOS에서 제공되는 bitmap data type을 사용해서 굳이 비트 연산을 하거나 배열로 선언하지 않아도 bitmap\_create, bitmap\_set\_all 등 사전에 제공된 함수로 쉽게 관리할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

swap\_in은 swap disk에 있는 프레임을 물리 메모리로 적재하기 위한 함수이다. Swap disk에서 인자로 받아 온 페이지의 swap\_index가 가리키는 block의 내용을 읽어들이는 block\_read 함수를 사용하고, 해당 swap\_index가 가리키는 block에 대응되는 swap\_bitmap의 원소를 false로 만든다. 이후 page\_table에서 인자로 받아 온 가상 메모리 주소에 대응되는 페이지를 찾아서 페이지 테이블에 swap\_index의 내용을 swap disk에서 고른 block의 인덱스로 업데이트 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

swap\_out 함수에서는 처음에 swap\_bitmap의 원소가 false로 오는 block을 찾아 해당 block부터 시작하여 페이지 크기만큼 프레임의 내용을 써서 swap out을 수행한다.

**Stack Growth**

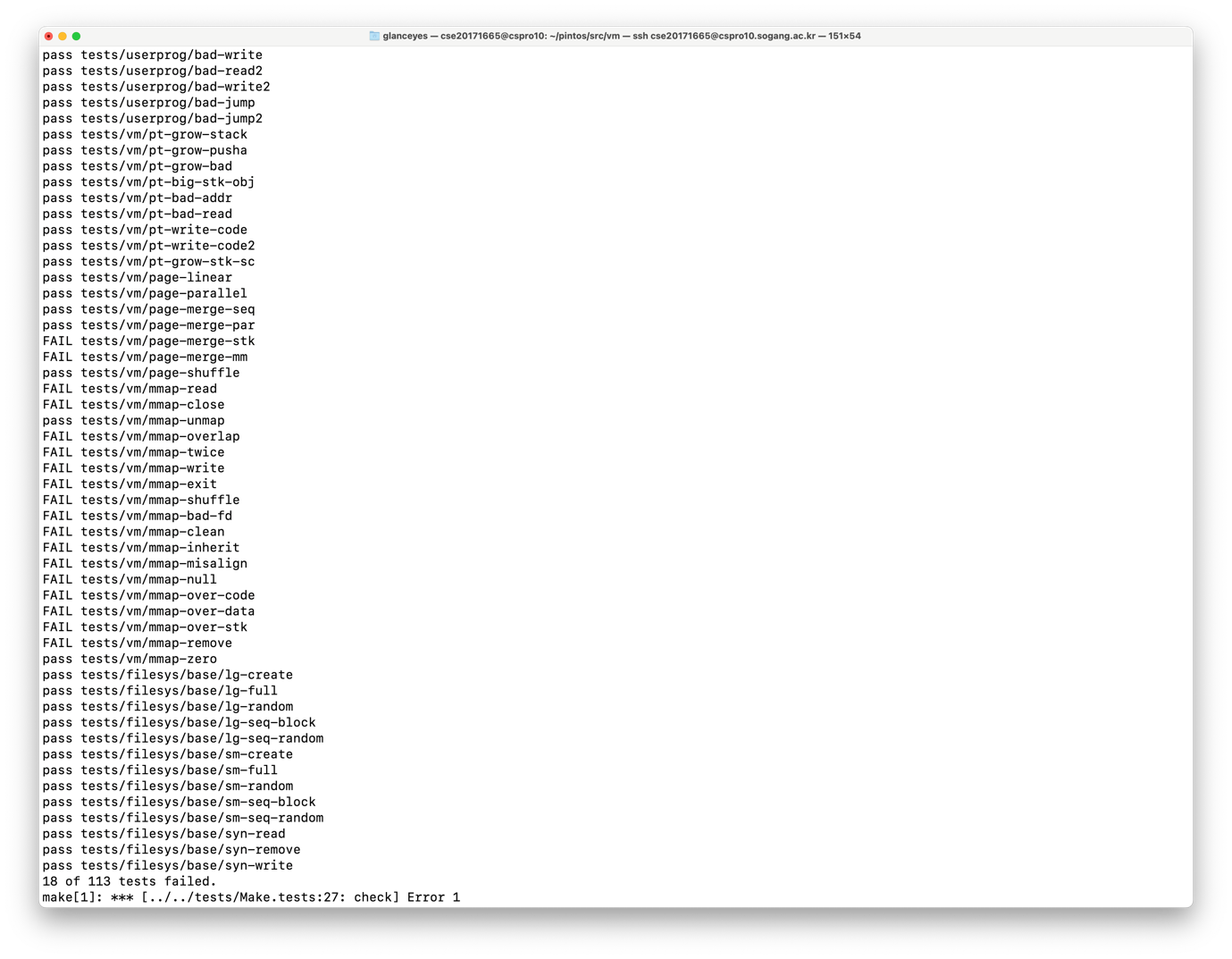
텍스트이(가) 표시된 사진

자동 생성된 설명

페이지를 모두 합한 영역의 크기가 전체 stack의 크기를 초과하거나 PintOS 명세서에 따라 PUSHA 명령어와 같이 stack 접근에 사용되지 않아야 할 명령어를 사용한 경우 syscall\_exit를 호출한다. Stack의 top을 가리키는 stack\_pointer를 페이지 단위로 줄여가면서 해당 페이지에 할당이 가능한지를 확인하고, 다시 stack\_pointer의 top을 이동해 가면서 페이지를 stack에 할당하는데, 페이지에 대응되는 프레임이 물리 메모리에 존재하지 않으면 frame\_evict 함수를 통해 임의의 프레임을 하나 축출하여 swap을 수행한다.

명세서와 여러 자료를 살펴 본 결과 page-merge-stk와 page-merge-mm을 위해서는 read와 write의 system call을 호출하는 부분을 유의하라는 내용이 있었는데, buffer의 크기만큼 모두 올바른 address인지를 확인해 주어야 하는 점에 따라서 이를 직접 구현했는데도 불구하고 예상과는 다르게 page-merge-stk와 page-merge-mm에서 오류가 발생했다. 또한 디스크의 파일로부터 buffer로 데이터를 읽어올 때 해당 buffer가 존재하는 페이지는 swap out이 발생하지 않도록 pinning 되어야 한다는데, 이를 구현해도 마찬가지로 오류가 발생했다. 이에 대한 궁금증으로 며칠 간 오류 해결을 시도했으나, 보고서 작성 시점 기준으로 결국 어떠한 것이 오류의 원인인지 구하지 못한 채 마무리하기로 했다.

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



page-merge-stk와 page-merge-mm을 제외한 프로젝트 4의 채점 대상인 모든 케이스를 통과했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Functionality의 page-merge-mm과 page-merge-stk만 통과를 하지 못한 것을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Robustness의 채점 항목은 모두 통과되었다.