**Pintos Project 4: Virtual Memory**

담당 교수 : 박성용

이름 / 학번 : 이우진 / 20181669

개발 기간 : 이우진

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

본 프로젝트에서는 OS에서 virtual memory에 관련된 기능을 구현한다. 본 프로젝트를 수행하기 이전의 pintos에서는 virtual memory가 구현되어 있기는 하지만, 구현되어 있는 virtual memory는 프로그램 load 시에만 physical memory에 직접적으로 할당하고, page fault가 발생할 시의 기능이 구현되어 있지 않다. 본 프로젝트에서는 기존에 제한적으로 구현되어 있던 page fault을 구현하면서 virtual page와 physical page을 필요한 경우에만 할당할 수 있도록 구현하고, 필요할 경우 disk와 memory 간의 swap이 이루어지도록 구현한다. 이를 위해 frame table, swap table과 그와 관련된 기능을 구현한다. 그리고, virtual memory의 stack에 포함될 수 있는 적절한 위치에서 page fault가 발생했을 경우에는 stack growth가 일어날 수 있도록 구현한다.

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
  2. Disk Swap
  3. Stack Growth
* Page Table & Page Fault handler

기존의 pintos에서, page는 process을 load할 때 physical memory의 일부분을 할당 받아 mapping된다. 그런데 위와 같이 구현하게 되면 process가 최대 physical memory의 크기 만큼의 memory만을 사용할 수 밖에 없을 뿐 아니라, 실행되는 process의 개수가 많아질 수록 적은 크기의 memory만이 사용 가능하게 되어 제한적인 physical memory을 효율적으로 사용할 수 없다. 따라서, process 별로 자신이 가지고 있는 virtual page의 목록인 page table을 할당하고, 해당 page가 실제로 필요한 경우에만 해당 page을 physical memory에 mapping하여 physical memory을 효율적으로 사용할 수 있도록 한다.

위 기능을 구현하기 위해서는 page fault handler가 필요하다. process가 memory에 존재하지 않은 page 또는 잘못된 권한으로의 page 에 접근하려고 하여 page fault가 발생하고, page fault handler가 실행된다. 그러나 기존의 pintos에서는 page fault handler가 실행되게 되면 process을 종료해버린다. Page fault handler을 구현하여, page fault가 발생했을 때, 적절한 page을 참조하려고 하는 경우 physical memory에 해당 page을 불러와 프로그램이 문제 없이 실행될 수 있도록 구현한다.

* Disk Swap

Process에서 계속해서 physical memory에 physical page을 할당 받는 경우, physical memory가 제한적이므로 physical memory에 새로운 page을 할당할 공간이 부족해질 수 있다. 이 경우, disk swap을 통해 잘 사용되지 않는 page을 disk에 임시로 저장하고 disk에 저장한 page을 다시 불러오는 기능을 구현하여 physical page에 충분한 공간을 확보하여, physical page의 실제 크기보다 더 큰 memory을 사용하는 것처럼 사용할 수 있다.

* Stack Growth

기존의 pintos에서는 process의 실행에 필요한 stack을 load 시에만 정적으로 할당한다. Virtual memory에서 process별로 할당된 virtual memory에 stack이 growth할 만한 충분한 공간이 존재하므로, 해당 기능을 구현하여 process가 실행하면서 더 많은 공간을 필요로 할 경우 더 많은 공간을 사용할 수 있게 된다.

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

Page fault가 발생하는 이유에는 여러가지가 있는데, 크게 handling 가능한 종류와 handling이 불가능한 종류가 있다.

먼저, handling이 불가능한 종류에는 read-only page에 write을 수행하려고 하는 경우, user mode에서 kernel address에 접근하려고 하는 경우가 있다. 위 두 경우에는 프로세스를 종료한다.

Handling이 가능한 것에는 process가 접근 가능한 address 범위이고 적절한 연산을 수행하려고 하지만 실제 physical memory에 존재하지 않아 page fault가 발생하는 경우인데, 이 경우에는 physical page을 할당 받고, virtual memory에 mapping 함으로써 fault 상황을 해결할 수 있다.

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

Disk swap시 현재 존재하는 frame에 대해, 최근에 access 되었는지에 대한 여부와 dirty여부를 확인한다. 최근에 access 되지 않았고, 수정되지도 않은 frame을 가장 높은 우선순위로 선택하고, 최근에 access된 frame, 최근에 수정된 frame, 최근에 access되었고 수정된 frame 순으로 우선순위를 두어 교체한다.

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

Stack growth을 할 경우, 먼저 page fault가 발생한 address가 stack 범위에 포함되는지에 대한 여부를 확인해야 한다. Fault addr이 stack이 시작하는 지점인 PHYS\_BASE와 PHYS\_BASE에서 스택의 최대 크기를 뺀 지점 사이에 있는지 확인한다. 그리고, 현재 stack과 충분히 가까운 지점에서 page fault가 발생하였는지 확인한다. 마지막으로, stack이 grow 하였을 때 stack의 최대 크기를 벗어나는 지 확인한다. 위 네 가지 조건을 만족한다면 stack 확장이 가능하다고 판단하여 stack growth을 실행한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

**11/20 – 11/26 :** frame table과 page supplement table을 구현하고, 기존 pintos 상에서 frame table과 page supplement table을 이용해 실행될 수 있도록 한다.

**11/27 – 12/03 :** page fault handler, swap table, stack growth을 구현하여 page fault handling, disk swap, stack growth가 실행될 수 있도록 구현한다.

**12/04 – 12/08 :** synchronization issue등 버그가 우려되는 상황을 처리한다.

* 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 & Page Fault Handler

먼저 Page Table을 구현하기 위해, vm 디렉토리에 page.h와 page.c 파일을 추가한다. Page.h에 page table entry을 저장하는 구조체를 선언한다. 해당 구조체에는 page가 file로부터 load되는 경우 파일에 대한 정보와, read-only 여부와 해당 page을 소유하는 thread에 대한 정보를 저장한다. Page table은 hash table로 구현할 것이므로 page.h와 page.c에 page table entry에 hash\_entry을 추가하고, hash을 정의하기 위한 함수 page\_hash( ), page\_less( )와 hash table에서 특정 page을 찾기 위한 함수인 page\_lookup( ) 함수를 정의한다. 그리고 page table의 initialization을 위한 함수인 page\_init\_table( ) 함수, page entry을 쉽게 추가할 수 있도록 page\_create\_and\_insert\_entry( ), 새로운 page entry을 반환해주는 page\_entry( ) 함수를 추가한다. 마지막으로 hash\_destroy을 이용해 page table을 삭제할 때 page table entry을 함께 삭제하기 위한 hash\_action\_func 타입의 함수 page\_free\_entry을 추가한다.

다음으로 userprog/exception.c에 위치한 page\_fault( ) 에서 page fault가 발생했을 때 적절히 handling할 수 있도록 구현하여야 하는데, 이를 위해 vm/page.h와 vm/page.c에 실제로 page fault을 handling해주는 page fault handler( ) 함수를 추가하여 page\_fault( ) 에서 호출할 수 있도록 한다. Page\_fault( )에서는 page\_fault가 발생한 원인을 확인하여 handling이 가능한 경우에만 page\_fault\_handler( )을 호출하도록 수정한다.

Page table과 page fault handler가 실제로 pinots상에서 동작하기 위해서는 threads/thread.h에 위치한 struct thread에 process별로 vm/page.h에서 정의한 page table entry을 저장하기 위해 hash table을 추가하고, userprog/process.c에서 vm/page.h에 선언한 page\_init\_table( ) 을 호출하도록 수정해야 한다. 또한, userprog.process.c에서 기존에 physical page에 직접 page을 할당해주는 코드 들을 대체하여 page\_table\_entry을 생성하고, page table에 추가해주는 코드를 넣어주어 실제로 해당 page에 접근할 경우에만 page fault가 발생해 physical memory에 mapping될 수 있도록 수정한다.

* Disk Swap

Disk swap을 구현하기 위해 vm 디렉토리에 frame table을 구현하기 위한 frame.h, frame.c을 추가하고, swap table을 구현하기 위한 swap.h, swap.c을 추가한다.

Frame table은 page table과 마찬가지로 hash table로 구현한다. 먼저 vm/frame.h에 frame\_entry을 추가한다. Frame\_entry에는 physical page의 주소와 현재 frame에 mapping된 page table entry, hash table에 저장하기 위한 hash\_entry가 저장되어야 한다. 이후, hash table을 생성하기 위해 frame.h와 frame.c에 hash을 정의하기 위한 함수 frame\_hash( ), frame\_less( )와 hash table에서 특정 frame을 찾기 위한 함수인 frame\_lookup( ) 함수를 정의한다. 그리고 frame table의 initialization을 위한 함수인 frame\_init( ) 함수를 추가하고, page fault handler에서 새로운 frame을 할당받기 위한 함수인 frame\_get\_page( ), 반대로 frame을 제거하기 위한 frame\_free\_page( ) 을 추가하고 page table과 마찬가지로 hash\_destroy을 이용해 frame table을 삭제할 때 frame entry을 함께 삭제하기 위한 hash\_action\_func 타입의 함수 frame\_free\_entry( )을 추가한다. Frame\_get\_page( ) 에서는 처음에는 physical\_page에서 page을 새롭게 할당받는 것을 시도하고, 만일 physical memory에 공간이 없다면 disk swap을 통해 얻은 빈 frame을 반환한다. Frame table은 모든 thread가 함께 사용하는 것이므로, synchronization을 위해 semaphore을 통해 한번에 하나의 thread만이 frame table을 수정할 수 있도록 frame.h에 semaphore을 추가해주고 frame\_init( ) 함수에서 초기화할 수 있도록 하였다.

Swap table은 bitmap과, pintos에서 제공하는 struct block을 사용하여 구현한다. 먼저 swap.h에 swap을 위한 block을 저장하기 위한 변수 swap\_block과 bitmap swap\_table을 추가한다. 그리고, bitmap과 swap\_block을 초기화 하기 위해 init\_swap\_table( ) 함수를 추가하고, 반대로 제거하기 위해 free\_swap\_table( ) 함수를 추가한다. 그리고, swap\_in( ) 과 swap\_out ( ) 함수를 추가하여 각각 disk에서 physical memory로, physical memory에서 disk로 오가는 기능을 구현한다. Frame table과 마찬가지로 swap table도 모든 thread가 사용하는 자료구조이므로 semaphore을 추가하고 semaphore을 초기화하는 함수를 init\_swap\_table( ) 함수에 추가한다.

위에서 구현한 frame\_get\_page( ) 함수를 vm/page.c의 page\_fault\_handler( ) 에 추가함으로서, disk swap을 구현할 수 있다.

* Stack Growth

Stack Growth는 위에서 구현한 함수들을 이용해 비교적 쉽게 구현할 수 있다. Userprog/excepthion.c 파일의 page\_fault\_handler( ) 에서 stack growth하는 조건을 만족하는 fault addr인지 확인하고, page\_table에 추가한 후 page fault\_handler( ) 을 호출해주도록 수정해준다.

1. **연구 결과**
   1. **Flow Chart**

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

텍스트, 스크린샷, 폰트, 그래픽 디자인이(가) 표시된 사진

자동 생성된 설명

스크린샷, 텍스트, 그래픽 디자인, 디자인이(가) 표시된 사진

자동 생성된 설명

* 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, Disk Swap

**Page table 관련**

Vm 디렉토리에 page.c, page.h 파일을 추가하고, *vm/page.h*에 page table의 entry을 저장하기 위한 타입인 *struct page\_table\_entry* 을 추가하였다. *Struct page\_table\_entry*에 선언한 변수들과 기능은 다음과 같다.

*uint8\_t \*upage* : page table entry에 저장될 virtual page address  
*uint8\_t \*kpage* : physical memory에 mapping된 physical page address  
*struct file \*file, off\_t ofs, size\_t page\_read\_bytes, size\_t page\_zero\_bytes*: page\_table\_entry에 할당된 virtual page가 file로 부터의 load을 필요로 하는 경우, 할당된 frame에 file을 load하기 위한 정보  
*bool readonly* : virtual page가 readonly인지 여부를 저장하는 변수  
*bool loaded* : virtual page가 physical page을 할당 받은 적이 있는지 여부를 저장하는 변수로, swap\_out 시에 disk에서 데이터를 불러와야 하는지 확인하기 위해 필요하다.  
*size\_t swap\_idx* : swap out 된 page가 disk에 swap table의 어떤 위치에 저장되어 있는지에 대한 변수  
*struct thread \*th* : virtual page을 소유하고 있는 thread에 대한 변수  
*struct hash\_elem h\_elem* : page table entry을 page table에 저장하기 위한 변수

Hash table인 page table을 생성하기 위해서는 hash table을 정의하기 위해 hash값에 관한 정보를 전달하는 함수인 hash와 비교 함수인 less가 필요하다. 이를 위해 *page\_hash ( )* 함수와 *page\_less ( )* 함수를 추가하였다. Page table에서는 *upage* 변수를 이용하여 hashing을 하도록 구현하였다.

Page table에서 원하는 데이터를 쉽게 찾을 수 있도록 하는 함수인 *page\_lookup ( )* 함수를 추가하였다. 해당 함수는 *upage* 값을 인자로 받아 pintos에서 제공하는 *hash\_find ( )* 함수를 이용해 데이터를 탐색하는데, 존재하지 않는 경우 NULL을 반환하도록 구현하였다.

위에서 선언한 page table을 initialization하기 위해 *page\_init\_table ( )* 함수를 *vm/page.h*에 선언하고, *vm/page.c*에 구현하였다. *page\_init\_table ( )* 함수에서는 hash table을 새롭게 malloc을 통해 할당하고, pintos에서 제공하는 *hash\_init ( )* 함수와 위에서 선언한 함수들을 이용해 page table을 initialization 한다.

외부에서 *page\_table\_entry*을 쉽게 생성하고 page table에 추가할 수 있도록 *page\_table\_entry*을 생성해주는 함수인 *page\_entry ( )* 함수와, 생성부터 page table로의 삽입까지 한번에 수행해주는 *page\_create\_ans\_insert\_entry ( )* 함수를 추가하였다. 두 함수 모두 *page\_table\_entry*을 생성하는데 필요한 정보들을 입력 받는데, *page\_create\_ans\_insert\_entry ( )* 함수에서는 *page\_entry ( )* 함수에 해당 인자를 전달해 *page\_table\_entry ( )* 을 생성하고 추가로 *page\_table*을 인자로 받아 해당 page table에 생성한 *page\_table\_entry*을 추가할 수 있도록 구현하였다.

마지막으로 핀토스에서 제공하는 *hash\_destroy ( )* 함수를 이용해 page table에 저장된 데이터들을 모두 할당 해제 해주기 위해 *hash\_action\_func* 타입의 함수 *page\_free\_entry* 을 추가하였다. 해당 함수에서는 *hash\_elem*을 인자로 받는데, *hash\_elem*을 이용해 *page\_table\_entry*을 구해 *page\_table\_entry*을 free 해 준다. 이 때, page table entry와 mapping된 physical page도 process을 종료하는 시점에서 모두 free되므로, 해당 physical page을 가지는 frame 도 마찬가지로 이 함수에서 할당 해제 해준다. 이 때 아래에서 구현할 *frame\_free\_page ( )* 함수를 이용하였다.

**Frame table 관련**

Vm 디렉토리에 frame.c, frame.h 파일을 추가하고, vm/frame.h에 frame table의 entry을 저장하기 위한 타입인 *struct frame\_entry* 을 추가하였다. *Struct frame\_entry*에 선언한 변수들과 기능은 다음과 같다.

*void \*page\_ptr* : frame에 저장된 physical page의 address을 관리하기 위한 변수  
 *struct page\_table\_entry \*pte* : frame에 저장된 physical page에 mapping된 virtual page을 가지고 있는 page table entry  
 *struct hash\_elem h\_elem* : hash table인 frame table에 저장하기 위한 변수

위 Page table과 마찬가지로, hash table인 frame table을 구현하기 위해 *frame\_hash ( )* 함수와 *frame\_less ( )* 함수를 추가하고, *frame\_lookup ( )* 함수도 추가하였다. Page table에서는 *upage*을 이용해 hashing을 했던 것과 달리 *page\_ptr* 을 이용해 hashing을 한다는 것을 제외하면 구현은 page\_table의 것과 같다. 마찬가지로, hash table을 initialization하기 위한 함수인 *frame\_init ( )*과 *hash\_destroy( )* 에서 사용할 frame\_free\_entry 함수 또한 추가하였다.

page table과는 달리 하나의 frame table을 여러 thread들이 같이 사용하는 구조이기 때문에, 한번에 하나의 thread만 접근할 수 있도록 해야 하므로 *vm/frame.h*에 *struct semaphore* 변수인 *frame\_table\_lock* 을 추가하고, *frame\_init ( )* 에 초기화하는 코드를 추가하였다. 그리고, *vm/frame.h*에 *struct hash* 타입의 hash table 변수 *frame\_table*을 추가하였다.

Frame을 frame table에서 제거하고, 할당 해제 해주는 함수인 frame\_free\_table을 구현하였다. 구현 방식은 frame\_free\_entry ( ) 함수와 거의 동일하다.

**Swap table 관련**

Vm 디렉토리에 swap.c, swap.h 파일을 추가하고, vm/swap.h에 frame table의 entry을 저장하기 위한 *struct bitmap* 타입의 변수 *swap\_table* 을 선언하였다. 그리고, swap in 또는 swap out 시에 disk와 데이터를 주고 받기 위해 *struct block* 타입의 변수 *swap\_bloc*k을 선언하였다. 또한, frame table과 마찬가지로 swap table 또한 모든 thread가 함께 사용하는 자료구조 이므로, block을 access할 때를 위한 semaphore인 *swap\_blk\_lock*, swap table을 access할 때를 위한 semaphore인 *swap\_tb\_lock* 을 각각 선언하였다.

Bitmap은 0과 1로 구성된 일종의 array로, bitmap을 이용해 disk의 어느 부분이 비어 있고 어느 부분이 사용 중인지 쉽게 알 수 있으므로 사용하였고, block은 pintos에서 disk 와의 소통을 목적으로 제공하는 자료구조이다.

위에서 선언한 *swap\_block*과 *swap\_table*을 initialization해주는 함수 *init\_swap\_table ( )*을 추가하였고, *swap\_table*을 정리하여 할당 해제를 해주기 위한 함수 *free\_swap\_table ( )* 을 추가하였다.

Swap in과 swap out의 처리를 위한 함수들인 swap\_in ( ) 과 swap\_out ( ) 함수를 vm/swap.h와 vm/swap.c에 추가하였다. 두 함수 모두 swap 하고자 하는 *page\_table\_entry* *pte* 을 인자로 받아, pte로부터 swap 하고자 하는 physical page의 주소를 구하여 처리한다.

Swap out의 경우, *pte*로부터 thread의 page directory을 구해 *pagedir\_clear\_page ( )* 함수를 이용해 virtual page와 physical page와의 mapping을 제거한다. bitmap\_scan\_and\_flip( ) 함수를 사용해 swap\_table에서 비어 있는 index을 찾아 표시해주고, swap\_block에서 해당 index의 위치에 기존에 page에 존재하던 데이터를 저장해준다. 이 때, block에서 데이터를 읽고 쓰는 단위가 page 크기의 1/8 이므로, index에 8을 곱하고, 곱한 index부터 block에서 8개의 위치에 데이터를 써서 block에 데이터가 잘못 쓰여지는 일이 없도록 하였다. 이 때 pte에 bitmap에서의 index을 저장해주어 swap in 을 할 때 사용하도록 하였다.

Swap in의 경우, swap out과 반대로 위에서 저장해 두었던 index을 이용해 block에서 데이터를 읽어 들이고, bitmap에 표기 해 두었다.

**Page fault 처리**

위에서 구현한 swap table 관련 함수를 이용하여 vm/frame.h와 vm/frame.c에 frame\_get\_page ( ) 함수를 구현하였다. 비어있는 physical page을 반환하는 함수인데, 만약 physical memory에 공간이 충분하다면 새로운 frame을 생성하여 frame table에 추가하고, 반환한다. Physical memory에 공간이 충분하지 않은 경우 swap out과 swap in을 수행하는데, pintos에서 제공하는 pagedir\_is\_accessed ( ) 함수와 pagedir\_is\_dirty ( ) 함수를 이용해 page가 access 되었는지, dirty 한 page인지 확인할 수 있다. 이를 이용해 hash의 모든 entry을 순회하며 최근에 access 되었는지에 대한 여부와 dirty여부를 확인한다. 최근에 access 되지 않았고, 수정되지도 않은 frame을 가장 높은 우선순위로 선택하고, 최근에 access된 frame, 최근에 수정된 frame, 최근에 access되었고 수정된 frame 순으로 우선순위를 두어 교체한다.

Page fault을 처리하기 위한 함수인 page\_fault\_handler ( ) 함수를 vm/page.h에서 선언하고 vm/page.c에서 구현하였다. Page\_fault\_handler ( ) 함수는 page fault가 발생한 address와 접근하려는 목적(read 또는 write) 을 인자로 받는다. 핀토스에서 제공하는 함수인 pg\_round\_down ( ) 을 사용하면 address로부터 page의 시작주소를 알 수 있으므로, 해당 함수를 이용해 page의 시작주소를 구해 page table entry을 얻을 수 있다.

먼저 pte가 readonly page 인데, write을 하고자 하는 접근일 경우 잘못된 접근이므로 false을 반환한다. 이후, 위에서 *구현한 frame\_get\_page( )* 함수를 이용해 새롭게 mappig될 frame을 얻는데, 이 때 pte의 *loaded*을 확인하여 저장된 disk에 저장된 적이 있는 경우 swap\_in을 호출해준다. 이후, pagedir\_set\_page( ) 함수를 이용해 thread의 Pagedir에 할당 받은 physical page을 추가해준다.

**기존 pintos에서의 변경점**

Page fault을 처리하기 위해 위에서 구현한 *page\_fault\_handler ( )* 을 *userprog/exception.c*에 있는 *page\_fault ( )* 에서 호출한다. 이 때, *not\_present* 상황에서만 page fault handler을 호출하도록 하였다.

기존에 userprog/process.c의 load( ) 와 *load\_segment( )* 에서 *palloc\_get\_page( )* 을 통해 physical page을 할당 받았던 것을 대체하여, page\_create\_and\_install\_entry ( ) 함수을 호출하였다. 그리고, load( ) 에서 *page\_init\_talbe( )* 을 호출하였고, process\_exit( ) 함수에서 *hash\_destroy( )*함수를 호출해 page table을 할당 해제 하도록 하였다.

*Threads/init.c*에서 *frame\_init( ),* *init\_swap\_table ( )* 함수를 호출하여 frame table과 swap table을 initialization 해 주었다.

* Stack Growth

Stack growth의 경우, *userprog/exception.c*에서 *page\_fault( )* 함수 내부에 관련 코드를 추가하여 구현하였다. 만약 *page\_fault\_handler( )* 로 처리가 불가능한 fault\_addr의 경우, stack growth로 처리가 가능한 fulat addr인지 확인한다. 상기했듯이, 조건은 다음과 같다.

*fault\_addr < PHYS\_BASE  
PHYS\_BASE – MAX\_STK\_SIZE <= fault\_addr  
f->esp <= fault\_addr + 32  
PHYS\_BASE – MAX\_STK\_SIZE <= f->esp - PGSIZE*

위 조건을 모두 만족하는 경우, *page\_create\_and\_insert\_entry ( )* 함수를 이용해 새롭게 *page table entry* 을 할당하고, *page\_fault\_handler( )* 을 호출하여 처리한다.

* 1. **시험 및 평가 내용**
* (채점 대상 테스트 케이스에 해당하는) make check 수행 결과를 캡처하여 첨부
* 텍스트, 흑백, 문서, 패턴이(가) 표시된 사진

  자동 생성된 설명