**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재

이름 / 학번 : 박태윤 / 20171646

개발 기간 : 11/05 ~ 11/27

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

src/vm디렉토리를 구현한다. 기존에는 page fault가 발생하면 프로그램이 종료가 되었다. 이를 해결하기 위해 page table을 만들어 이를 관리할 수 있는 자료구조와 함수들을 추가해주고, 물리 메모리가 가득 찬 경우 이를 해결하기 위해 swap disk라는 임시 저장 공간을 만들어 swap동작이 가능하도록 구현하고, page fault가 stack영역에서 발생하면 stack growth를 통해 추가 페이지를 할당하는 동작을 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

1.

virtual address를 physical address로 변환할 수 있게 하는 page table을 구현한다. virtual address와 physical address가 1대1로 맵핑이 되어 page table안에 자료구조로 존재하는데, 만일 특정 virtual address를 가지고 page table안에서 physical address를 찾으려고 할 때 해당되는 physical address가 없다면 이 때 page fault가 발생하게 된다. 기존에는 page fault가 발생하면 프로그램을 종료했지만, Page Fault Handler를 적절하게 구현하면 이에 해당하는 page를 할당하여 page table에 추가해주는 동작을 구현한다. 페이지 폴트 발생시 단순 종료가 아닌, 메모리의 처리를 통해 프로그램이 계속 동작할 수 있게끔 할 수 있다.

2.

만일 메모리 요청을 os가 받았을 때, 메모리가 부족한 경우, 임시 저장 공간인 swap disk에 사용하지 않는 frame을 넣고 새로운 frame을 할당받는다. 어떤 frame을 swap disk에 넣을지는 LRU알고리즘을 이용해 가장 과거에 사용이 되었던 frame을 선택할 수 있도록 한다. 이를 구현한다면 메모리에 용량이 꽉 차도 계속 프로그램을 동작시킬 수 있을 것이다.

3.

기존에는 스택의 크기가 4KB로 고정이 되어 있었는데, 해당 스택의 용량을 초과하는 영역에 접근이 발생했을 시 유효한 접근이라면 스택을 확장시키는 동작을 구현한다. 만일 유효하지 않은, 예를 들어 segment영역에 접근했을 시는 프로그램을 종료하는 등의 처리를 해준다. 현재 한 개의 page만 할당 받는 stack에서 여러 개의 데이터를 저장할 수 있도록 stack growth를 구현하면 page fault에 대한 적절한 처리가 가능할 것이다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술
  1. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술
  2. Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술
  3. Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술

1.

어떤 virtual address에 접근을 시도할 때, 이에 해당하는 physical address가 만약 존재하지 않는다면 이 때 page fault가 발생한다. 즉, page table을 사용했을 시 가상 주소에 해당하는 물리주소를 찾지 못한다면 page fault가 발생하고 이 때 page fault를 handler하는 과정을 시도하게 된다. 이는 해당 가상 주소에 해당하는 내용을 디스크로부터 읽어와 메모리에 frame으로 적재를 하는 과정이다. 이 때 해당 가상 주소가 세그먼트 영역이 아닌지 등의 유효성 검사를 실시를 해야 하며 메모리에 새로운 frame을 적재했다면 역시 page table을 업데이트를 하여page table을 통해 앞서 접근했던 가상 주소에 대응되는 물리 주소를 맵핑해준다.

2.

page fault가 발생했을 시 디스크로부터 특정 가상 주소에 해당하는 영역을 읽어와 메모리에 적재를 하는데, 이 때 메모리가 full인 상태라 할당할 수 있는 공간이 없는 경우가 생길 수 있다. 이 때, 기존에 메모리에 있던 frame을 디스크로 evict시켜 공간을 만들고 적재를 하고자 하는 내용을 해당 공간에 맵핑을 시켜주는 swap과정이 필요하다. 어떤 frame을 evict시킬 것인지를 결정하는 것은 LRU알고리즘을 따르는데, 이는 가장 최근에 사용되지 않은 frame을 select해 디스크로 evict시키는 것을 의미한다.

3.

page fault가 발생하여 stack을 growth시키려고 하였을 때, 유효한 접근인지를 판별해야 한다. 이는 user memory의 스택 영역인지, 스택 프레임에 있는 주소인지를 판별해야 하며 스택포인터인 esp를 이용하여 이를 수행한다. 만약 스택 포인터가 user memory의 스택 영역을 벗어나면 세그멘테이션 폴트가 발생하기 때문에 이에 대한 처리를 해주어야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

**11/05 ~ 11/10 : 전반적인 개념 이해 및 코드 구성**

**11/11 ~ 11/20 : page table, page fault handler구현**

**11/21 ~ 11/27 : system call, page fault handler등에서의 예외 처리**

* 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

1.

기존에는 page fault가 나서 userprog디렉토리 exception.c의 page\_fault함수에 들어가게 되면 경고메시지를 출력한 후 프로세스를 kill하였다. 이를 주석처리 하고 이 함수에 조건에 맞으면 load\_page를 할 수 있도록 코드를 작성한다. 또한 page table이 필요하므로 page.h에 page\_table과 이에 대한 엔트리(유저페이지, 커널페이지 등을 포함)를 나타내는 pt\_entry구조체를 추가해준다. 페이지 테이블은 유저페이지와 커널페이지가 1대1로 대응이 되기 때문에 해시테이블로 구현을 하였다. 또한 page.h에 페이지 테이블 초기화 함수, 엔트리 삽입 함수, 제거 함수, 탐색 함수 등을 추가를 해준다. 물리메모리의 frame을 나타내는 ft\_entry를 frame.h에 추가를 해주고 역시 이에 대한 초기화 함수, 삽입 함수, 제거 함수, 탐색 함수 등을 추가를 해준다.

2.

swap관련 동작들은 vm디렉토리의 swap.h, swap.c에 구현이 되어 있다. 스왑되는 블락을 나타내는 swap\_block, 가능 여부를 나타내는 swap\_available등 swap에 필요한 변수들을 swap.h에 선언하였으며 이 변수들과 비트맵을 초기화 해주는 swap\_init함수, 비트맵을 초기화 해주는 swap\_free함수, 디스크에 넣고 가져오는 함수인 swap\_in, swap\_out함수를 구현하였다. evict를 하고자 하는 frame은 lru clock 알고리즘으로 선택이 되는데, clock알고리즘으로 frame을 순회하는 함수는 frame.c에 clock\_frame\_next로 구현을 하였고 pagedir\_is\_accessed함수를 이용하여 accessed bit를 확인하면서 clock방향으로 순회를 해 evict를 하고자 하는 frame을 select해서 return하는 pick\_frame\_to\_evict함수를 구현하였다.

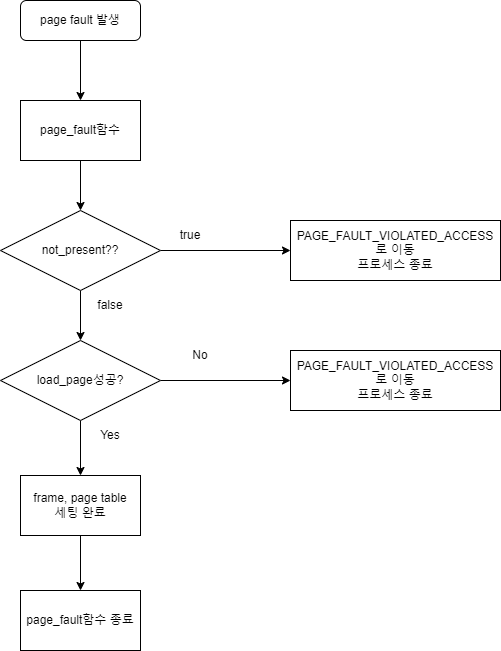
3.

stack growth는 userprog디렉토리 exception.c의 page\_fault함수에서 이루어질 수 있다. 여기서 스택 영역에서의 포인터를 나타내는 esp에 대해 유효한지를 검사를 해야하는데, user mode에서는 page\_fault함수에서 인자로 받은 intr\_frame f에서 f->esp로 바로 받아올 수 있지만 kernel mode에서는 그럴 수 없기 때문에 미리 thread.h의 쓰레드 구조체에 current\_esp를 추가하여 시스템 콜이 호출되는 경우 현재 esp를 이 변수에 계속 추가를 하였다. 따라서 kernel mode에서는 현재 쓰레드의 current\_esp에 대해 스택 포인터의 유효성을 검사한다. 스택포인터와 페이지 폴트가 발생한 fault\_addr에 대해 두 가지 측면에서 유효성 검사를 하는데 하나는 fault\_addr가 user memory의 스택 영역인지, 다른 하나는 스택 포인터가 스택 프레임 위에 존재를 하는지에 대해 유효성 검사를 한다. 이에 대해 계산을 하여 각각 stack\_on\_frame, stack\_is\_addr 불리언 변수로 표현을 하였고 이 두 변수가 모두 true인 경우에만 stack growth를 하였다.

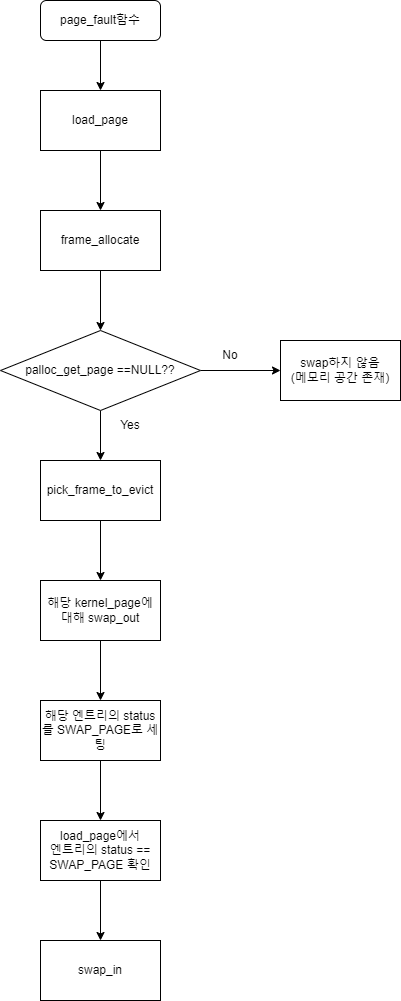
1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성
  + Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술
  + Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술
  + Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술

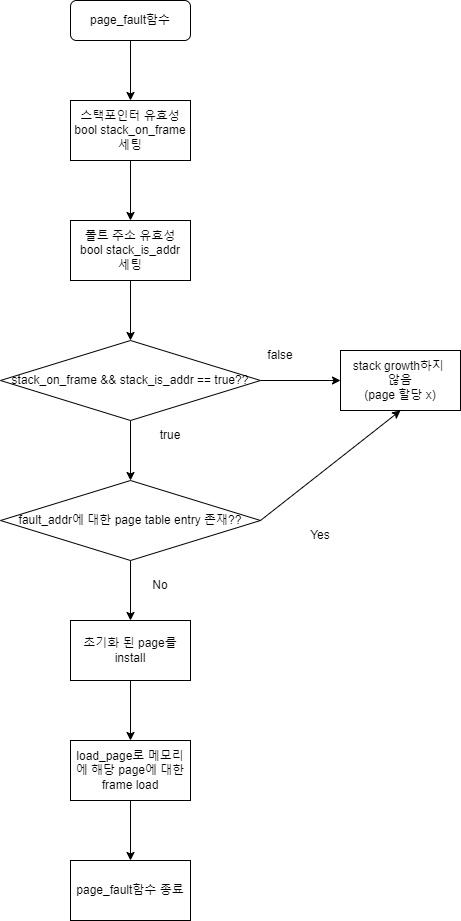
1.



2.



3.



* 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

1.

기존에 page fault를 핸들링 하기 이전에 page\_fault함수는 다음과 같이 작성이 되어 있었다.

static void

page\_fault (struct intr\_frame \*f)

{

bool not\_present; /\* True: not-present page, false: writing r/o page. \*/

bool write; /\* True: access was write, false: access was read. \*/

bool user; /\* True: access by user, false: access by kernel. \*/

void \*fault\_addr; /\* Fault address. \*/

/\* Obtain faulting address, the virtual address that was

accessed to cause the fault. It may point to code or to

data. It is not necessarily the address of the instruction

that caused the fault (that's f->eip).

See [IA32-v2a] "MOV--Move to/from Control Registers" and

[IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception

(#PF)". \*/

asm ("movl %%cr2, %0" : "=r" (fault\_addr));

/\* Turn interrupts back on (they were only off so that we could

be assured of reading CR2 before it changed). \*/

intr\_enable ();

/\* Count page faults. \*/

page\_fault\_cnt++;

/\* Determine cause. \*/

not\_present = (f->error\_code & PF\_P) == 0;

write = (f->error\_code & PF\_W) != 0;

user = (f->error\_code & PF\_U) != 0;

if (!user || is\_kernel\_vaddr(fault\_addr) || not\_present)

exit(-1);

/\* To implement virtual memory, delete the rest of the function

body, and replace it with code that brings in the page to

which fault\_addr refers. \*/

**printf ("Page fault at %p: %s error %s page in %s context.\n",**

**fault\_addr,**

**not\_present ? "not present" : "rights violation",**

**write ? "writing" : "reading",**

**user ? "user" : "kernel");**

**kill (f);**

}

빨간색 글씨를 살펴보면, 결국에 최종적으로 kill(f)를 함으로써 특별한 동작 없이 프로세스를 죽였다. page fault에 대한 handling없이 단순히 해당 인터럽트가 발생하면 프로세스를 죽이는 동작만 하고 있기에 이를 다음과 같이 수정하였다.

static void

page\_fault (struct intr\_frame \*f)

{

bool not\_present; /\* True: not-present page, false: writing r/o page. \*/

bool write; /\* True: access was write, false: access was read. \*/

bool user; /\* True: access by user, false: access by kernel. \*/

void \*fault\_addr; /\* Fault address. \*/

/\* Obtain faulting address, the virtual address that was

accessed to cause the fault. It may point to code or to

data. It is not necessarily the address of the instruction

that caused the fault (that's f->eip).

See [IA32-v2a] "MOV--Move to/from Control Registers" and

[IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception

(#PF)". \*/

asm ("movl %%cr2, %0" : "=r" (fault\_addr));

/\* Turn interrupts back on (they were only off so that we could

be assured of reading CR2 before it changed). \*/

intr\_enable ();

/\* Count page faults. \*/

page\_fault\_cnt++;

/\* Determine cause. \*/

not\_present = (f->error\_code & PF\_P) == 0;

write = (f->error\_code & PF\_W) != 0;

user = (f->error\_code & PF\_U) != 0;

#ifdef DEBUG

printf ("Page fault at %p: %s error %s page in %s context.\n",

fault\_addr,

not\_present ? "not present" : "rights violation",

write ? "writing" : "reading",

user ? "user" : "kernel");

#endif

#if VM

/\* Virtual memory handling.

\* First, bring in the page to which fault\_addr refers. \*/

struct thread \*curr = thread\_current(); /\* Current thread. \*/

void\* fault\_page = (void\*) pg\_round\_down(fault\_addr);

if(fault\_addr < 0x8048100){

exit(-1);

}

//printf("fault addr : %p\n", fault\_addr);

if (!not\_present) {

goto PAGE\_FAULT\_VIOLATED\_ACCESS;

}

/\* (4.3.3) Obtain the current value of the user program's stack pointer.

\* If the page fault is from user mode, we can obtain from intr\_frame `f`,

\* but we cannot from kernel mode. We've stored the current esp

\* at the beginning of system call into the thread for this case. \*/

void\* esp = user ? f->esp : curr->current\_esp;

bool stack\_on\_frame = false, stack\_is\_addr = false;

if(fault\_addr == f->esp-32 || fault\_addr == f->esp-4 || esp <= fault\_addr)

stack\_on\_frame = true;

if(fault\_addr < PHYS\_BASE && PHYS\_BASE <= fault\_addr + MAX\_STACK\_SIZE)

stack\_is\_addr = true;

//stack\_on\_frame = (esp <= fault\_addr || fault\_addr == f->esp - 4 || fault\_addr == f->esp - 32);

//stack\_is\_addr = (PHYS\_BASE - MAX\_STACK\_SIZE <= fault\_addr && fault\_addr < PHYS\_BASE);

if (stack\_on\_frame && stack\_is\_addr) {

pt\_entry \*pte = page\_table\_lookup(curr->page\_table, fault\_page);

if(pte == NULL)

page\_table\_install(curr->page\_table, fault\_page, NULL, 0, NULL, 0, 0, 0, false, ZERO\_PAGE);

}

if(!load\_page(curr->page\_table, curr->pagedir, fault\_page) ) {

goto PAGE\_FAULT\_VIOLATED\_ACCESS;

}

// success

return;

PAGE\_FAULT\_VIOLATED\_ACCESS:

#endif

if(!user) {

f->eip = (void \*) f->eax;

f->eax = 0xffffffff;

return;

}

/\* Page fault can't be handled - kill the process \*/

printf ("Page fault at %p: %s error %s page in %s context.\n",

fault\_addr,

not\_present ? "not present" : "rights violation",

write ? "writing" : "reading",

user ? "user" : "kernel");

kill (f);

}

유효하지 않은 접근 등 오류가 발생할 시에만 PAGE\_FAULT\_VIOLATED\_ACCESS로 이동하여 경고 메시지를 출력한 뒤 프로세스를 kill을 통해 죽였고, 그 이전에 새로운 page를 할당 받게 되는 load\_page과정, 조건에 따라 stack을 growth하는 과정을 담은 코드를 추가하였다.

2.

실제로 디스크 to 메모리 swap in, 메모리 to 디스크 swap out되는 함수는 swap.c에 다음과 같이 구현하였다.

void swap\_in(uint32\_t swap\_index, void\* page)

{

for (int i = 0; i < SECTORS\_PER\_PAGE; ++i) {

block\_read(swap\_block,

swap\_index \* SECTORS\_PER\_PAGE + i,

page + (BLOCK\_SECTOR\_SIZE \* i)

);

}

bitmap\_set(swap\_available, swap\_index, true);

}

uint32\_t swap\_out (void \*page)

{

size\_t swap\_index = bitmap\_scan (swap\_available, 0, 1, true);

for (int i = 0; i < SECTORS\_PER\_PAGE; ++i) {

block\_write(swap\_block,

swap\_index \* SECTORS\_PER\_PAGE + i,

page + (BLOCK\_SECTOR\_SIZE \* i)

);

}

bitmap\_set(swap\_available, swap\_index, false);

return swap\_index;

}

swap in을 하는 경우 swap\_index에 대해 block을 block\_read로 읽어와 bitmap에 대해 true로 세팅하였고 swap out을 하는 경우 역시 비슷하게 swap\_index에 대해 block\_write를 하고 bitmap에 대해 false로 세팅하였다. swap\_in함수는 page.c의 load\_page함수에서

else if(pte->status == SWAP\_PAGE){

if(pte == NULL || fpage == NULL)

return false;

swap\_in(pte->swap\_idx, fpage);

}

다음과 같이 페이지 테이블 엔트리의 상태가 SWAP\_PAGE인 경우 호출하였으며 swap\_out함수는 frame.c의 frame\_allocate함수에서

if(frame\_page == NULL){

ft\_entry \*f\_evicted = pick\_frame\_to\_evict(thread\_current()->pagedir);

is\_dirty = false;

pagedir\_clear\_page(f\_evicted->t->pagedir, f\_evicted->user\_page);

if(pagedir\_is\_dirty(f\_evicted->t->pagedir, f\_evicted->user\_page) == true)

is\_dirty = true;

if(pagedir\_is\_dirty(f\_evicted->t->pagedir, f\_evicted->kernel\_page) == true)

is\_dirty = true;

pt\_entry \*pte = page\_table\_lookup(f\_evicted->t->supt, f\_evicted->user\_page);

pte->swap\_idx = swap\_out(f\_evicted->kernel\_page);

pte->status = SWAP\_PAGE;

pte->kernel\_page = NULL;

if(is\_dirty == true);

pte->dirty\_bit = true;

frame\_do\_free(f\_evicted->kernel\_page, true); //쫓아낸건 free

frame\_page = palloc\_get\_page (PAL\_USER | flags);

}

다음과 같이 evict를 하고자 하는 프레임을 pick\_frame\_to\_evict함수로 select하여 이를 인자로 넘겨 swap\_out함수를 호출하는 식으로 구현하였다. evict하고자 하는 frame을 선택하는 pick\_frame\_to\_evict함수는 다음과 같이 선언하였다.

struct frame\_table\_entry\* pick\_frame\_to\_evict(uint32\_t \*pagedir){

size\_t n = hash\_size(&frame\_map);

for(int i = 0; i <= 2 \* n; ++i){

ft\_entry \*e = clock\_frame\_next();

//pinned면 스왑하면 안되지!

if(e->pinned)

continue;

//clock알고리즘은 access되지 않은 페이지를 스왑 대상으로 선택한다

//보고 있는 페이지가 access된거라면 access bit을 false로 바꾸고 그냥 넘어감

else if(pagedir\_is\_accessed(pagedir, e->user\_page)){

pagedir\_set\_accessed(pagedir, e->user\_page, false);

continue;

}

return e;

}

}

clock lru 알고리즘이기 때문에 clock\_frame\_next함수를 통해 clock방향으로 순회하면서 accessed bit를 보고 스왑을 할 수 있는 frame에 걸리게 된다면 이를 continue하지 않고 해당 frame entry를 리턴하였다.

3.

stack growth가 되는 동작은 exception.c에 구현이 되어 있다. page\_fault함수 안에 stack growth에 대한 동작은 다음과 같이 구현을 하였다.

bool stack\_on\_frame = false, stack\_is\_addr = false;

if(fault\_addr == f->esp-32 || fault\_addr == f->esp-4 || esp <= fault\_addr)

stack\_on\_frame = true;

if(fault\_addr < PHYS\_BASE && PHYS\_BASE <= fault\_addr + MAX\_STACK\_SIZE)

stack\_is\_addr = true;

//stack\_on\_frame = (esp <= fault\_addr || fault\_addr == f->esp - 4 || fault\_addr == f->esp - 32);

//stack\_is\_addr = (PHYS\_BASE - MAX\_STACK\_SIZE <= fault\_addr && fault\_addr < PHYS\_BASE);

if (stack\_on\_frame && stack\_is\_addr) {

pt\_entry \*pte = page\_table\_lookup(curr->page\_table, fault\_page);

if(pte == NULL)

page\_table\_install(curr->page\_table, fault\_page, NULL, 0, NULL, 0, 0, 0, false, ZERO\_PAGE);

}

stack\_on\_frame은 현재 스텍 포인터가 스택 프레임 위에 올라와 있는 주소인지를 판별하는 bool형 변수이며, stack\_is\_address는 페이지 폴트가 발생한 fault\_addr이 세그먼트 영역 등 이상한 영역에 있는 것이 아닌 user memory영역에 있는지를 판별하는 변수이다. 이 두 변수가 모두 true를 나타내면 비로소 0으로 초기화가 된 페이지 테이블 엔트리를 하나 생성하여 stack을 growth시키는 과정을 진행한다.

**시험 및 평가 내용**

* (채점 대상 테스트 케이스에 해당하는) make check 수행 결과를 캡처하여 첨부
* 텍스트이(가) 표시된 사진

  자동 생성된 설명