**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재

이름 / 학번 : 이도안 / 20201610

개발 기간 : ~2024-12-22

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

**이번 프로젝트에서는 가상 메모리의 대체적인 부분을 구현할 예정이다. 기존 핀토스에 있는것보다 supplement page table을 개발하고, 이를 통해 Page Fault에 대해 더 reliable하게 한다. 추가적으로 Second-chance 알고리즘을 구현하여 Swap을 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

Page fault에 대해서 exit 하지않고 스왑과 페이지 테이블 관리를 통해 reliable하게 구현한다. 적절한 접근인지 확인하는 로직 등을 추가하고 최종적으로 Page Fault Handler가 제 역할을 하게 한다.

* 1. Disk Swap

DRAM 메모리가 부족한 상황에서 디스크의 용량을 사용하는 것을 Swap이라고 하는데, 이를 LRU 알고리즘을 구현하여 해결한다.

* 1. Stack Growth

스택에 접근하는 중 스택이 꽉 차거나 Page fault가 일어날 때 스택 Growth를 구현하여 크기를 조절하고 적절한 할당으로 효율을 상향하도록 한다.

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

커널과 같은 접근 불가능한 영역에 접근을 요청하는 경우 page fault가 발생한다. 또는 swap out이 이루어져 현제 물리 메모리에 있는 것이 아니라 Disk에 존재하는 경우에도 페이지 폴트가 발생하기 때문에, 이 경우에서의 처리가 이 프로젝트의 핵심이 되겠다. 이 때 page fault handler가 작용하여 디스크의 페이지를 찾아 swap-in 한다.

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

수도-LRU 정책 중 second chance algorithm을 구현하였다. 각 페이지는 Recent access bit를 가지고, 첫 접근에서 이 bit을 0으로 설정한다. 다시 이 clock 알고리즘에 의해서 커서가 돌아오기 전에 access되어 bit를 다시 1로 돌려놓지 않으면 해당 페이지가 swap의 대상이 된다.

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

스택 확장은 페이지 폴트 예외 처리 시 스택 영역 접근 위반을 감지하여 이루어진다. 이때, 예외가 발생한 주소가 스택 영역의 경계에 가까운지 확인하고, 추가 스택 페이지를 할당하여 확장 여부를 결정한다.

**추진 일정 및 개발 방법**

* 1. **추진 일정**
* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

-12.6~12.19 : page, frame 개발

-12.20 : swap 개발

-12.21 : 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

**Page**

struct pt\_entry

{

void \*vaddr;

pt\_type type;

size\_t offset;

size\_t read\_bytes;

size\_t zero\_bytes;

bool writable;

bool is\_loaded;

struct hash\_elem elem;

struct file \*file;

size\_t swap\_slot;

};

void pt\_init (struct hash \*pt);

void pt\_destroy (struct hash \*pt);

bool pt\_insert\_entry (struct hash \*pt, struct pt\_entry \*pte);

bool pt\_delete\_entry (struct hash \*pt, struct pt\_entry \*pte);

struct pt\_entry \*pt\_find\_entry (void \*vaddr);

위 구조체는 virtual memory의 페이지 엔트리로서 필요한 모든 자료구조를 담도록 구현하였다. 가상 주소, 오프셋, 가상 메모리에 존재하는지 여부, 쓸 수 있는 공간인지, load 되었는지 등을 저장하고 있다.

Page table 핸들링을 위해서 5개의 함수를 선언하였다.

각각 page table을 초기화, 제거, 삽입, 삭제, 그리고 vaddr에 해당하는 pt entry를 찾는 함수이다. 이 함수는 page\_fault handling에 사용된다.

**Frame**

#ifndef VM\_FRAME\_H

#define VM\_FRAME\_H

#include <list.h>

#include <stdint.h>

#include "lib/kernel/hash.h"

#include "threads/palloc.h"

struct frame

{

  void \*kaddr;

  struct thread \*thread;

  struct pt\_entry \*pte;

  struct list\_elem frame\_elem;

};

extern struct list\_elem \*frame\_clock;

void frame\_init (void);

static void ft\_insert\_frame (struct frame \*frame);

static void ft\_delete\_frame (struct frame \*frame);

static struct frame \*ft\_find\_frame (void \*kaddr);

// struct frame\* frame\_find(void \*paddr);

// static struct frame \*ft\_get\_unaccessed\_frame (void);

static void ft\_second\_chance (void);

struct frame \*alloc\_page (enum palloc\_flags flags);

void free\_page (void \*kaddr);

bool frame\_load\_file (void \*kaddr, struct pt\_entry \*pte);

// struct frame\* frame\_alloc(enum palloc\_flags flags);

// void frame\_free(void \*paddr);

#endif

Second chance 알고리즘을 reliable하게 완성하기 위해 만들어진 frame.c의 구조체와 함수들이다.

Frame\_clock은 현재 화살표가 가리키고 있는 페이지이고, 대부분의 함수들의 기능과 맥락은 이름과 같아서 생략하겠다.

Page table을 추가로 구현하였기 때문에 그냥 palloc 함수를 사용하게 되면 문제가 생긴다. 따라서 page에 할당하는 함수 alloc\_page를 따로 구현하였다

**Swap**

#ifndef VM\_SWAP\_H

#define VM\_SWAP\_H

#include <bitmap.h>

#include "threads/vaddr.h"

#include "devices/block.h"

#include "threads/synch.h"

#include "vm/frame.h"

#include "vm/page.h"

#include "userprog/pagedir.h"

#include "threads/thread.h"

#include "filesys/filesys.h"

#include "filesys/file.h"

#define BLOCK\_MAX 8

void swap\_init(void);

void swap\_in(size\_t idx, void \*paddr);

size\_t swap\_out(void \*paddr);

void handle\_block(size\_t index, void\* kaddr, bool r\_w);

#endif

Swap.c 파일에서는 swap에 대한 구체적인 함수를 가진다. Swap\_init에서는 synchronization를 위한 swap\_lock을 초기화하고, 스왑 공간 여부를 확인하기 위한 swap\_bitmap을 초기화한다.

Swap\_in 에서는 disk에 있는 메모리를 physical memory로 swap해주는 작업을 하며, swap\_out에서는 반대의 작업을 한다. 이 함수들은 모두 process.c, exception.c 와 관련되어서 페이지 폴트 핸들링과 동시에 수행된다.

추가적으로 process.c에 stack\_growth 함수를 추가하여 stack을 확장하는 작업을 한다. 해당 함수는 인자로 받는 addr를 포함하도록 stack을 확장한다. 이는 handle\_mm\_fault 처럼 exception.c에서 stack과 관련된 access인지 확인하도록 한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성
  1. Page Table & Page fault handling
* 도표, 기술 도면, 종이접기, 패턴이(가) 표시된 사진

  자동 생성된 설명
  1. Disk swap
* 텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

  자동 생성된 설명
  1. Stack\_growth

도표, 텍스트, 라인, 기술 도면이(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

Frame.c

* bool load\_file\_to\_page(void \*kaddr, struct pt\_entry \*pte) {//완
* size\_t bytes\_read;
* bytes\_read = file\_read\_at(pte->file, kaddr, pte->read\_bytes, pte->offset);
* if (bytes\_read != pte->read\_bytes) {
* return false; // 파일 읽기 실패
* }
* if(pte->zero\_bytes > 0){
* memset(kaddr + pte->read\_bytes, 0, pte->zero\_bytes);
* }
* return true; // 성공
* }
* static struct frame \*//완
* ft\_find\_frame(void \*kaddr) {
* struct frame \*frm;
* struct list\_elem \*elem;
* for (elem = list\_begin(&frame\_list); elem != list\_end(&frame\_list); elem = list\_next(elem)) {
* frm = list\_entry(elem, struct frame, frame\_elem);
* if (frm->kaddr == kaddr) {
* return frm; // 일치하는 frame을 찾으면 반환
* }
* }
* return NULL; // 찾지 못하면 NULL 반환
* }
* void //완
* frame\_init (void)
* {
* frame\_clock = NULL;
* list\_init (&frame\_list);
* lock\_init (&frame\_lock);
* }
* struct frame \*//완
* alloc\_page (enum palloc\_flags flags)
* {
* uint8\_t \*p\_page = palloc\_get\_page(flags);
* // memset(p\_page, 0, sizeof(struct frame));//있어야하나
* while(p\_page == NULL){
* ft\_second\_chance ();
* p\_page = palloc\_get\_page(flags);
* }
* struct frame \*frm = malloc(sizeof(struct frame));
* frm->thread = thread\_current();
* frm->kaddr = p\_page;
* ft\_insert\_frame (frm);
* return frm;
* }
* void
* free\_page (void \*kaddr)//완
* {
* struct frame \*frm = NULL;
* frm = ft\_find\_frame(kaddr);
* if(frm!=NULL){
* if(frm->pte != NULL) pagedir\_clear\_page (frm->thread->pagedir, frm->pte->vaddr);
* ft\_delete\_frame(frm);
* palloc\_free\_page(frm->kaddr);
* free(frm);
* }
* }
* static void//완 add\_to\_lrulist
* ft\_insert\_frame (struct frame \*frame)
* {
* lock\_acquire(&frame\_lock);
* list\_push\_back (&frame\_list, &(frame->frame\_elem));
* lock\_release(&frame\_lock);
* }
* static void//완 delete\_from\_lrulist
* ft\_delete\_frame (struct frame \*frm)
* {
* // lock\_acquire(&frame\_lock);
* if(frame\_clock == &frm->frame\_elem)
* frame\_clock = list\_next(frame\_clock);
* list\_remove(&frm->frame\_elem);
* // lock\_release(&frame\_lock);
* }
* struct list\_elem \*frame\_next(){//완
* //원형순회를 위한 next 알고리즘
* struct list\_elem \*nxt;
* if(list\_empty(&frame\_list)) return frame\_clock;
* if(frame\_clock == NULL || frame\_clock == list\_end(&frame\_list)){
* nxt = list\_begin(&frame\_list);
* }
* else nxt = list\_next(frame\_clock);
* if(nxt == list\_end(&frame\_list)) nxt = list\_begin(&frame\_list);
* return nxt;
* }
* static void
* ft\_second\_chance (void)//완
* {
* frame\_clock = frame\_next();
* lock\_acquire (&frame\_lock);
* if (!frame\_clock)
* {
* lock\_release (&frame\_lock);
* return;
* }
* struct frame \*victim = list\_entry (frame\_clock, struct frame, frame\_elem);
* while(victim->pte != NULL && pagedir\_is\_accessed (victim->thread->pagedir, victim->pte->vaddr)){
* if(victim->pte!=NULL) pagedir\_set\_accessed (victim->thread->pagedir, victim->pte->vaddr, 0);
* frame\_clock = frame\_next();
* if(!frame\_clock){
* lock\_release (&frame\_lock);
* return;
* }
* victim = list\_entry (frame\_clock, struct frame, frame\_elem);
* }
* if(victim->pte!=NULL && victim->pte->type == BINARY){
* if (pagedir\_is\_dirty (victim->thread->pagedir, victim->pte->vaddr))
* {
* victim->pte->swap\_slot = swap\_out (victim->kaddr);
* victim->pte->type = SWAPPED;
* }
* }
* else if(victim->pte!=NULL && victim->pte->type == SWAPPED)
* {
* victim->pte->swap\_slot = swap\_out (victim->kaddr);
* }
* if(victim->pte) victim->pte->is\_loaded = false;
* if(victim->pte) pagedir\_clear\_page (victim->thread->pagedir, victim->pte->vaddr);
* ft\_delete\_frame (victim);
* palloc\_free\_page (victim->kaddr);
* free (victim);
* lock\_release (&frame\_lock);
* return;
* }

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

자동 생성된 설명

struct list\_elem \*frame\_clock;

struct lock frame\_lock;

struct list frame\_list;

**Frame\_clock을 사용하여 포인터를 나타내었다. Frame\_list는 현재 LRU를 계산하기 위한 프레임 리스트이다. Frame\_lock은 위에 서술한 바와 같이 동기화를 맞춰주기 위해서 선언하였다.**

**LRU Clock이 위 강의자료의 사진과 같이 list를 따라 회전하면서 second chance algorithm을 만족시키도록 구현하였다. 순회를 위한 알고리즘은 frame\_next에 구현되어 있고, 순회를 하며 victim을 찾고, victim을 swap\_out 하는 알고리즘은 ft\_second\_chance에 구현되어 있다.**

**Page.c**

void

pt\_init (struct hash \*pt)//완

{

  hash\_init (pt, pt\_hash, pt\_cmp, NULL);

}

void

pt\_destroy (struct hash \*pt)//완

{

  lock\_acquire(&frame\_lock);

  hash\_destroy (pt, pt\_destroy\_func);

  lock\_release(&frame\_lock);

}

bool //0

pt\_insert\_entry (struct hash \*pt, struct pt\_entry \*pte)

{

  lock\_acquire(&frame\_lock);

  bool success=hash\_insert(pt, &(pte->elem));

  lock\_release(&frame\_lock);

  return success;

}

bool //완

pt\_delete\_entry (struct hash \*pt, struct pt\_entry \*pte)

{

  lock\_acquire(&frame\_lock);

  bool success=hash\_delete(pt, &(pte->elem));

  lock\_release(&frame\_lock);

  if (!success) return false;

  free\_page (pagedir\_get\_page (thread\_current ()->pagedir, pte->vaddr));

  free (pte);

  return true;

}

struct pt\_entry \*

pt\_find\_entry (void \*vaddr)//완

{

  struct pt\_entry tmp;

  struct pt\_entry \*ve = NULL;

  struct hash\_elem \*elem;

  tmp.vaddr = pg\_round\_down(vaddr); //malloc?

  elem = hash\_find(&thread\_current()->pt, &tmp.elem);

  if(elem) ve = hash\_entry(elem, struct pt\_entry, elem);

  return ve;//NULL처리?

}

static unsigned//완

pt\_hash (const struct hash\_elem \*h\_elem, void \*aux UNUSED)

{

  struct pt\_entry \*pte = hash\_entry(h\_elem, struct pt\_entry, elem);

  return hash\_int ((int)(pte->vaddr));

}

static bool//완

pt\_cmp (const struct hash\_elem \*A,

  const struct hash\_elem \*B, void \*aux UNUSED)

{

  struct pt\_entry \*a = hash\_entry(A, struct pt\_entry, elem);

  struct pt\_entry \*b = hash\_entry(B, struct pt\_entry, elem);

  return ((a->vaddr) < (b->vaddr));

}

void page\_delete(struct pt\_entry \*pte){//완

    void \*paddr = pagedir\_get\_page(thread\_current()->pagedir, pte->vaddr);

    free\_page(paddr); //=frame\_free

}

static void pt\_destroy\_func(struct hash\_elem \*e, void \*aux UNUSED){//완

    struct pt\_entry \*ve = hash\_entry(e, struct pt\_entry, elem);

    if(ve->is\_loaded){

      page\_delete(ve);

    }

    free(ve);

}

 struct pt\_entry \*pte = pt\_find\_entry (fault\_addr);

  if (pte){

    if (!handle\_mm\_fault (pte)) sys\_exit (-1);

  }

  else{

   if (!stack\_growth (fault\_addr, f->esp))   sys\_exit (-1);

  }

   return;

   struct pt\_entry \*ve = pt\_find\_entry(fault\_addr);

   if(ve!= NULL){

      if(handle\_mm\_fault(ve) == NULL) sys\_exit(-1);

   }

   else if(!stack\_growth(fault\_addr,f->esp)) sys\_exit(-1);

bool

stack\_growth (void \*addr, void \*esp)//완

{

  void \*upage;

  struct frame \*kpage;

  bool success = false;

  //에러체크

  if (!is\_user\_vaddr (addr)) return false;

  if (addr < (PHYS\_BASE - 0x8000000)) return false;//wait-killed

  if (addr < (esp - 32)) return false;//pt\_bad\_read

  //get upage

  upage = pg\_round\_down (addr);

  kpage = alloc\_page (PAL\_USER | PAL\_ZERO);

  if(kpage != NULL)

    {

      success = install\_page (upage, kpage->kaddr, true);

      if(!success){

        free\_page (kpage->kaddr);

        return success;

      }

      kpage->pte = (struct pt\_entry \*)malloc(sizeof(struct pt\_entry));

      if(kpage->pte == NULL) return false;

      kpage->pte->type = SWAPPED;

      kpage->pte->vaddr = upage;

      kpage->pte->read\_bytes = 0;

      kpage->pte->zero\_bytes = 0;

      kpage->pte->file = NULL;

      kpage->pte->offset = 0;

      kpage->pte->is\_loaded = true;

      kpage->pte->writable = true;

      success = !pt\_insert\_entry (&(thread\_current ()->pt), kpage->pte);

    }

  return success;

}

pt\_find\_entry(void \*vaddr): 주어진 가상 주소 vaddr에 해당하는 페이지 테이블 엔트리(pte)를 현재 스레드의 페이지 테이블에서 찾아 반환한다. 페이지 테이블에 해당 엔트리가 없으면 NULL을 반환한다.

pt\_hash(const struct hash\_elem \*h\_elem, void \*aux UNUSED): 페이지 테이블 해시 함수이다. 페이지 테이블 엔트리의 가상 주소를 기반으로 해시 값을 계산한다.

pt\_cmp(const struct hash\_elem \*A, const struct hash\_elem \*B, void \*aux UNUSED): 페이지 테이블 엔트리를 비교하는 함수이다. 두 엔트리의 가상 주소를 비교하여 정렬 순서를 결정한다.

page\_delete(struct pt\_entry \*pte): 주어진 페이지 테이블 엔트리에 해당하는 물리 페이지를 해제한다. 페이지 디렉토리에서 해당 페이지를 제거하고, 물리 페이지 메모리를 해제합니다. (해당 페이지를 더 이상 사용하지 않을 때 호출)

pt\_destroy\_func(struct hash\_elem \*e, void \*aux UNUSED): 페이지 테이블을 소멸시키는 데 사용되는 함수이다. 해시 테이블의 각 엔트리에 대해 page\_delete()를 호출하여 페이지를 해제하고 페이지 테이블 엔트리 자체도 메모리에서 해제한다.

stack\_growth(void \*addr, void \*esp): 스택 영역이 확장되어야 할 때 호출되는 함수이다. 주어진 주소 addr이 스택 영역 내에 있고, 확장 조건이 충족되면 새로운 페이지를 할당하고 페이지 테이블에 추가하여 스택을 확장한다.

bool//완

handle\_mm\_fault (struct pt\_entry \*pte)

{

  struct frame \*frm = alloc\_page(PAL\_USER);

  bool success = false;

  if(frm == NULL)

    return false; // alloc\_page 실패 시 false 반환

  frm->pte = pte;

  if (pte->type == BINARY) {

    if (!load\_file\_to\_page(frm->kaddr, pte)) {

      free\_page(frm->kaddr);

      free(frm); // load\_file\_to\_page 실패 시 frm 해제

      return false;

    }

  } else if (pte->type == SWAPPED) {

      swap\_in(pte->swap\_slot, frm->kaddr);

  }

   success = install\_page(pte->vaddr, frm->kaddr, pte->writable);

   if (!success) {

      free\_page(frm->kaddr);

      free(frm);

      return false;

    }

    pte->is\_loaded=true;

    return success;

}

handle\_mm\_fault 함수는 페이지 폴트 발생 시 해당 pte에 따라 필요한 메모리 페이지를 할당하고 load하는 함수이다. pte가 파일에서 로드해야 하는 페이지(BINARY)라면 파일을 읽어오고, 스왑 영역에 있는 페이지(SWAPPED)라면 스왑 인을 수행한다. 최종적으로 페이지를 가상 주소에 매핑하고 성공 여부를 반환하는 역할을 한다.

**Swap.c**

void handle\_block(size\_t index, void\* kaddr, bool r\_w){

  if(r\_w==0)

    for (size\_t i = 0; i < BLOCK\_MAX; i++)//4KB = 8\*512

      block\_read (swap\_block, index \* BLOCK\_MAX + i, kaddr + BLOCK\_SECTOR\_SIZE \* i);

  else

    for (size\_t i = 0; i < BLOCK\_MAX; i++)//4KB = 8\*512

      block\_write (swap\_block, index \* BLOCK\_MAX + i, kaddr + BLOCK\_SECTOR\_SIZE \* i);

}

void

swap\_init(void)

{

  swap\_bitmap = bitmap\_create (PGSIZE);

  lock\_init (&swap\_lock);

  if (swap\_bitmap == NULL) PANIC ("swap\_init");

}

void

swap\_in (size\_t index, void \*kaddr)

{

  if (index == 0) {

    PANIC ("swap\_in");

  }

    lock\_acquire (&swap\_lock);

    swap\_block = block\_get\_role (BLOCK\_SWAP);

    --index;

    bitmap\_set\_multiple (swap\_bitmap, index, 1, false);

    handle\_block(index, kaddr,0);

    lock\_release (&swap\_lock);

}

size\_t

swap\_out (void \*kaddr)

{

  size\_t index\_to\_swap;

  lock\_acquire (&swap\_lock);

  index\_to\_swap = bitmap\_scan\_and\_flip (swap\_bitmap, 0, 1, false);

  swap\_block = block\_get\_role (BLOCK\_SWAP);

  handle\_block(index\_to\_swap, kaddr,1);

  lock\_release (&swap\_lock);

  return (index\_to\_swap + 1);

}

handle\_block(size\_t index, void\* kaddr, bool r\_w): 주어진 인덱스와 주소를 사용하여 스왑 블록 장치에서 데이터를 읽거나 쓰는 함수이다. r\_w 값이 0이면 읽기 작업을, 1이면 쓰기 작업을 수행한다.

swap\_init(void): 스왑 시스템을 초기화하는 함수이다. 스왑 공간 관리를 위한 비트맵을 생성하고 락을 초기화한다.

swap\_in(size\_t index, void \*kaddr): 스왑 공간에서 주어진 인덱스에 해당하는 데이터를 메모리 주소 kaddr로 읽어오는 함수이다. 읽기 전에 비트맵에서 해당 슬롯을 사용 가능하도록 변경한다.

swap\_out(void \*kaddr): 주어진 메모리 주소 kaddr의 데이터를 스왑 공간의 빈 슬롯에 저장하는 함수이다. 스왑 공간의 빈 슬롯을 찾아 사용 중으로 표시하고, 데이터를 저장한 후 스왑 공간 슬롯 인덱스를 반환한다.

**구현이 어려웠던 부분**

페이지 테이블을 만들어서 이전 프로젝트 수행 능력을 갖추기부터, 모든 구현이 어려웠다. 특히 113/113 fail을 밤 새 보며 힘든 나날들을 보냈다. 초반부의 경우 lock을 두번 걸리게 되는 상황이 많이 나왔다. 중~후반부에서는 알 수 없는 pagedir fault가 계속 발생하였다. 현재도 이 버그에 대해서 진짜 원인을 파악하지 못하였으며, 발생할 때마다 코드를 거의 갈아엎었다.많은 소스를 참고하면서, 지속적인 테스트를 반복한 결과 현재 결과물까지 만들 수 있었다.

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

  자동 생성된 설명
* Page-merge-mm을 제외한 모든 케이스에 통과하였다.