**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재

이름 / 학번 : 박민준 / 20212020

개발 기간 : 12/5 ~ 12/22

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

1. Page Table Management

- 기존의 글로벌 페이지 테이블을 보완하여 프로세스별로 필요한 정보를 관리할 수 있는 보조 페이지 테이블을 구현한다. 이 테이블은 해시 테이블 또는 리스트와 같은 자료구조를 사용하며, 각 프로세스에 대해 동적으로 할당되고 해제된다.

- 이전까지는 페이지 폴트가 발생할 경우, 프로세스를 종료하였다. 따라서 Page fault handler를 통해 페이지 폴트가 발생했을 때 메모리 또는 디스크에서 데이터를 로드하고 페이지 테이블을 업데이트하며 명령 실행을 재개하도록 구현해야 한다. 이 때, 유효하지 않은 스택 확장을 확인하고 필요시 스택을 동적으로 확장해야 한다.

2. Paging to and from (swap) disk

- 물리 메모리가 부족할 경우 스왑 디스크를 사용해 메모리의 제약을 해결해야 한다. 이를 위해 대략적인 LRU 알고리즘을 구현해 페이지 교체 정책을 설정해야 한다. 이를 위해 스왑 디스크를 위한 관리 테이블(swap table)을 구축하고, 블록 장치 API인 block\_read, block\_write를 활용한다.

3. Stack Growth

- 페이지 폴트로 인해 새로운 스택 페이지가 필요한 경우, 유효한 주소인지를 확인한 후 페이지 단위로 스택을 확장하는 기능을 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

- 현재 pintos의 기본 페이지 테이블은 각 프로세스에 대해 제한된 정보를 제공하며, 보조적인 메모리 관리 기능이 부족하다. 페이지 폴트 발생 시 프로세스를 종료하지 않고 적절히 처리해 프로그램의 연속성을 보장하기 위해 보조 페이지 테이블 및 페이지 폴트 핸들러 구현이 필수적이다.

- 보조 페이지 테이블을 통해 메모리 로드 여부, 파일 매핑 정보, 페이지 교체 대상 여부 등 각 페이지의 상태를 효율적으로 관리할 수 있다. 또한 페이지 폴트 발생 시 디스크에서 데이터를 읽어오거나 스택 확장을 통해 문제를 해결함으로써 시스템의 안정성 및 효율성을 향상시키고 프로세스가 물리 메모리 한계를 넘어서서 실행될 수 있도록 지원한다.

* 1. Disk Swap

- 물리 메모리 부족 시 일부 페이지를 디스크로 옮기고 필요한 페이지를 다시 불러오는 swapping 기법이 필요하다. 페이지 교체 시 second chance algorithm을 통해 시스템 성능을 유지하면서 메모리를 최적화해야 한다.

- 이를 통해 물리 메모리 제한을 초과하는 프로그램의 실행이 가능해지며 빈번한 페이지 교체에도 시스템 성능 저하를 최소화한다. 또한 스왑 테이블 구축 및 스왑 슬롯 관리를 통해 데이터의 무결성을 보장한다.

* 1. Stack Growth

- 프로그램 실행 중 필요한 경우 스택 영역을 동적으로 확장하여 메모리 낭비를 줄이고 유연한 메모리 사용이 가능하도록 해야 한다. 스택 영역 확장은 프로그램 실행의 안정성을 보장하며 메모리 접근 오류를 방지하기 위해 필수적이다.

- 스택 영역 확장은 스택 크기를 초기에 고정하지 않고 필요한 만큼만 동적으로 할당하여 메모리 공간을 효율적으로 활용할 수 있다. 또한, 잘못된 스택 접근을 감지하고 처리하여 프로세스의 안정적인 실행을 보장해준다.

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

- 페이지 폴트는 프로세스가 접근하려는 가상 주소가 현재 물리 메모리에 매핑되지 않은 경우 발생한다. 주로 메모리에 로드되지 않은 페이지에 대한 접근, 디스크로 스왑된 페이지에 대한 접근, 또는 잘못된 메모리 참조가 원인이 된다. 페이지 폴트가 발생하면 운영체제는 Faulting Address를 확인하고 페이지 테이블을 탐색하여 해당 주소가 유효한지 확인한다. 유효한 경우, 디스크에서 데이터를 읽어와 물리 메모리에 로드하고 페이지 테이블 정보를 갱신한 뒤, 중단된 명령어를 재실행하여 프로그램이 정상적으로 동작하도록 한다. 유효하지 않은 참조는 프로세스를 종료한다.

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

- 디스크 스왑은 물리 메모리가 부족한 상황에서 일부 페이지를 디스크로 옮겨 공간을 확보하는 과정이다. 이를 위해 Second Chance 알고리즘을 사용한다. 이 알고리즘은 원형 큐 형태로 페이지를 관리하며, 각 페이지의 참조 비트를 검사해 교체 대상을 결정한다. 참조 비트가 설정된 페이지는 다시 초기화하며 교체 대상에서 제외하고, 참조 비트가 없는 페이지를 찾아 디스크로 스왑 아웃한 후 새로운 페이지를 메모리에 로드한다. 이 방법은 LRU 알고리즘의 간소화된 버전으로 효율성과 성능을 적절히 균형 있게 제공한다.

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

- 스택 확장은 페이지 폴트가 스택 범위 내의 유효한 주소에서 발생했을 때 수행된다. Faulting Address가 스택 시작 주소로부터 권장 최대 크기 이내에 포함되고, 현재 esp에서 허용 범위 내에 있는지를 확인하여 유효성을 판단한다. 이러한 조건을 충족하면 스택에 새로운 페이지를 할당하고 물리 메모리에 매핑한다. 조건을 충족하지 않으면 잘못된 메모리 참조로 간주하여 프로세스를 종료한다. 이를 통해 메모리 효율성을 유지하며 안정적인 프로그램 실행을 보장할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

12/5 ~ 12/6 : page table and page fault handling 구현

12/7 : Paging to and from disk 구현

12/8 : Stack Growth 구현

12/21 : 코드 디버깅 및 수정

12/22 : 보고서 작성

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야 하는 소스코드

[userprog/exception.c]

- page\_fault(): 페이지 폴트가 발생하면 제어권이 해당 함수로 전달된다. 폴트 주소는 cr2 레지스터에서 가져오며 보조 페이지 테이블에서 폴트 주소에 해당하는 페이지 엔트리를 검색한다. 만약 페이지 엔트리가 있으면 mm\_fault\_handler() 함수를 호출하여 디스크에서 데이터를 로드하거나 swap in을 실행한다. 만약 페이지 엔트리가 없다면 주소가 스택 확장 가능한 범위인지 확인한 후 expand\_stack() 함수를 호출한다. 커널 주소에 접근하는 등 예외적인 경우에는 exit(-1)으로 프로세스를 곧바로 종료시킨다.

[userprog/process.c]

- start\_process(): 새 프로세스가 생성되면 보조 페이지 테이블을 초기화해야 한다. pt\_init() 함수를 이용해 보조 페이지 테이블을 초기화하는 코드를 추가해준다.

- setup\_stack(): 초기 스택 페이지를 설정할 때 보조 페이지 테이블에 해당 페이지를 추가해야 한다.

- load\_segment(): 파일에서 데이터를 로드할 때 페이지 단위로 보조 페이지 테이블에 엔트리를 추가해야 한다.

* + 수정하거나 추가해야 하는 자료구조

[vm/page.\*]

- struct pt\_entry: 각 프로세스가 독립적으로 페이지 상태를 관리하기 위한 자료구조를 추가해야 한다. 페이지 엔트리를 관리하기 위해 해시 테이블을 사용한다. vaddr은 가상 주소로, 페이지 단위로 정렬된 값이다. loaded 변수는 페이지가 물리 메모리에 로드되었는지 여부를 나타낸다. swap\_slot 변수는 스왑된 경우 스왑 슬롯의 번호를 나타낸다.

[vm/swap.\*]

- struct bitmap \*swap\_bitmap: 각 비트는 스왑 슬롯을 나타낸다. 비트가 1이면 사용 중이고 0이면 사용 가능한 상태이다.

[threads/thread.h/thread 구조체]

- struct hash pt: 보조 페이지 테이블을 나타낸다. 각 프로세스는 자신의 보조 페이지 테이블을 가지며 페이지 폴트 처리 및 가상 메모리 관리에 사용된다. 페이지 테이블은 해시 자료구조로 구현되며 페이지 엔트리를 관리한다.

- struct list map\_list: 메모리 매핑 리스트로, 프로세스가 mmap()을 호출하여 매핑한 파일 정보들을 관리한다. 리스트의 각 항목은 struct mmap\_entry로 표현되며 매핑된 파일의 상태를 추적한다.

- unsigned int map\_list\_size: 메모리 매핑 리스트에서 매핑 ID를 할당하기 위한 값으로 사용된다. mmap() 호출 시 새로운 매핑에 고유한 ID를 부여한다. 초기값은 0이며, 매핑이 추가될 때마다 증가한다.

- struct file \*file: 프로세스의 실행 파일을 가리키는 포인터이다. 프로세스가 실행 중일 때 실행 파일의 정보를 유지하며 쓰기 접근 방지를 지원한다.

* + 수정하거나 추가해야 하는 함수

[vm/page.\*]

- 보조 페이지 테이블 관리를 위한 pt\_create, pt\_insert, pt\_find, pt\_delete, pt\_destroy 함수를 구현한다. 이 파일은 가상 메모리 시스템에서 페이지 상태를 추적하고 페이지 폴트 시 적절한 처리를 지원한다.

- pt\_init(): hash\_init()을 사용하여 각 스레드에 독립적인 테이블을 생성한다.

- pt\_create(): 페이지 엔트리를 생성하며 페이지 타입에 따라 메타 데이터를 설정한다.

- pt\_insert(): 보조 페이지 테이블에 페이지 엔트리를 삽입한다. 중복 삽입 방지를 위해 기존 항목을 확인한다.

- pt\_find(void \*vaddr): 주어진 가상 주소를 기준으로 페이지 엔트리를 검색한다.

- pt\_delete(): 페이지 테이블에서 특정 페이지 엔트리를 삭제한다. 스왑된 경우 스왑 슬롯을 해제한다.

- pt\_destroy(): 보조 페이지 테이블에 있는 모든 페이지 엔트리를 제거하고 메모리를 반환한다.

[vm/frame.\*]

- 물리 메모리 프레임 할당 및 해제 시 보조 페이지 테이블과 연계하여 적절한 매핑 및 해제를 수행하도록 frame\_alloc, frame\_free 함수를 구현한다. 부족한 메모리 상황에서 페이지를 스왑 아웃하는 frame\_evict 함수는 추가한다.

- frame\_alloc(): 새로운 물리 메모리 프레임을 할당한다. 메모리 부족 시 프레임을 스왑 아웃하여 공간을 확보한 후 새 프레임을 반환한다.

- frame\_free(): 특정 커널 가상 주소에 해당하는 물리 메모리 프레임을 해제한다. 프레임이 더 이상 사용되지 않을 경우, 이를 프레임 목록과 페이지 테이블에서 제거한다.

- frame\_evict(): 물리 메모리 부족 시 교체 대상 프레임을 선택하고 이를 스왑 공간으로 내보낸다. 스왑 아웃된 페이지는 페이지 테이블 상태를 업데이트한다.

- frame\_victim(): second chance 알고리즘을 기반으로 다음 교체 대상 프레임을 선택한다.

[vm/swap.\*]

- swap\_init(): 스왑 공간을 초기화하고, 스왑 블록 디바이스를 설정한다.

- swap\_out(): 페이지를 스왑 공간으로 내보낸다. 스왑 비트맵에서 사용 가능한 슬롯 검색 후 데이터를 저장한다.

- swap\_in(): 스왑 공간에서 데이터를 복원하여 물리 메모리에 로드한다.

- find\_swap\_index(): 비어 있는 스왑 슬롯을 검색한다.

- swap\_free(): 스왑 슬롯을 반환한다.

[vm/mmap.\*]

- mm\_map(): 주어진 파일 디스크립터와 주소를 기준으로 파일을 매핑하고 이를 관리하는 mmap\_entry를 생성한다. Mmap\_entry와 페이지 엔트리를 생성하여 보조 페이지 테이블에 추가한다.

- mm\_free(): 주어진 매핑 ID를 기반으로 매핑된 모든 페이지를 해제하고 파일과 동기화를 진행한다. 페이지 테이블에서 해당 페이지 엔트리를 제거한다.

- create\_page\_entries(): 파일 크기와 오프셋을 기준으로 페이지 단위로 페이지 테이블 엔트리를 생성한다. 파일 크기가 페이지 크기(PGSIZE)보다 작거나 정확히 나누어지지 않을 경우 나머지 바이트를 0으로 초기화한다.

- remove\_mapping(): 매핑된 페이지를 해제하고, dirty 상태일 경우 파일과 동기화를 진행한다.

[userprog/process.c]

- expand\_stack(void \*addr, void \*esp): 페이지 폴트 주소가 스택 확장 조건을 충족할 경우 새로운 페이지를 생성한다. addr가 현재 스택 포인터 esp로부터 허용된 거리 이내인지 확인한다. 유효한 경우 frame\_alloc() 함수를 통해 프레임을 할당하고, install\_page()로 가상 주소와 물리 메모리를 매핑한다.

- mm\_fault\_handler(struct pt\_entry \*pte): 디스크에서 페이지를 로드하거나 스왑된 페이지를 복구한다. 성공적으로 처리한 경우 페이지 테이블을 업데이트한다.

[userprog/syscall.c]

- syscall\_handler(): SYS\_MMAP, SYS\_MUNMAP을 통해 페이지 매핑을 보조 페이지 테이블과 연계한다. check\_user\_vaddr 함수를 이용해 사용자 주소를 보조 페이지 테이블과 연계하여 검증한다.

- mmap, munmap 함수: 파일 매핑 페이지를 보조 페이지 테이블에 추가하거나 삭제한다.

1. **연구 결과**
   1. **Flow Chart**

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

1. Page Table Management

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

2. Paging to and from (swap) disk

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

3. Stack Growth

텍스트, 스크린샷, 도표, 폰트이(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

**1. Page Table Management (Supplemental page table & page fault handling)**

[vm/page.\*]

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- 보조 페이지 테이블은 struct hash를 기반으로 구현하였다. 각 페이지 엔트리는 struct pt\_entry 구조체로 표현되며, 해시 테이블에서 가상 주소를 키로 사용하여 관리한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 주어진 가상 주소를 기반으로 새로운 페이지 엔트리를 생성한다. 페이지 타입, 파일 정보, 파일 오프셋, 읽기/초기화 바이트 수, 페이지 상태 등을 초기화한다. 반환된 struct pt\_entry는 페이지 테이블에 추가된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- 새로 생성된 페이지 엔트리를 보조 페이지 테이블에 추가한다. hash\_insert를 호출하여 페이지 엔트리를 테이블에 삽입한다. 반환값이 NULL이면 삽입에 성공한 것이며, 이미 동일한 키가 존재하면 실패한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- 특정 페이지 엔트리를 보조 페이지 테이블에서 제거하고 메모리를 해제한다. hash\_delete() 함수를 호출하여 테이블에서 페이지 엔트리를 삭제한다. 삭제 성공 후 frame\_free로 물리 메모리 프레임을 해제하며, free를 호출하여 페이지 엔트리 구조체 메모리를 반환한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 보조 페이지 테이블에서 특정 가상 주소에 해당하는 페이지 엔트리를 검색한다. 주어진 주소를 페이지 경계로 정렬한 후 hash\_find() 함수를 호출하여 해시 테이블을 검색한다. 검색 성공 시 struct pt\_entry 포인터를 반환하며 실패하면 NULL값을 반환한다.

[userprog/exception.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 페이지 폴트를 처리한다. cr2 레지스터에서 페이지 폴트가 발생한 주소를 가져온다. 보조 페이지 테이블에서 해당 페이지 엔트리를 찾고, 없으면 스택 확장을 시도한다.

[userprog/process.c]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 주어진 struct pt\_entry 정보를 바탕으로 페이지를 물리 메모리에 로드하고 가상 메모리와 매핑한다. 페이지 타입에 따라 적절한 조치를 취한다. BINARY 또는 MAPPED 타입인 경우 디스크에서 데이터를 물리 메모리로 읽어온다. SWAPPED 타입의 경우 스왑 영역에서 물리 메모리로 데이터를 복원한다. 마지막에 매핑 성공 여부를 반환한다.

**2. Paging to and from (swap) disk**

[vm/swap.\*]

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

- 스왑 작업 중 동시 접근을 방지하기 위한 락을 초기화한다. 이후 스왑 슬롯 상태를 추적하기 위해 비트맵을 생성한다. 비트맵의 크기는 스왑 공간의 페이지 개수에 따라 결정된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- block\_get\_role(BLOCK\_SWAP)으로 스왑 공간을 관리하는 블록 디바이스를 가져온다.

- find\_swap\_index() 함수를 호출하여 사용 가능한 스왑 슬롯을 비트맵에서 검색한다. 슬롯이 없을 경우 프로그램이 중단된다.

- swap\_data()를 호출하여 물리 메모리 페이지를 스왑 디스크에 기록한다.

- 스왑 데이터 저장이 성공하면 bitmap\_set으로 해당 스왑 슬롯을 사용 상태로 설정한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 먼저, 스왑 인덱스를 확인하여 index가 유효하지 않으면 프로그램을 중단한다. swap\_data() 함수를 호출하여 스왑 디스크의 데이터를 물리 메모리로 복원한다. 데이터 복원이 성공하면 bitmap\_set으로 해당 스왑 슬롯을 비사용 상태로 설정한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- 스왑 슬롯 하나는 8개의 섹터로 구성되어 있으며 페이지 데이터를 섹터 단위로 읽거나 쓴다. is\_read 플래그로 읽기(swap in) 또는 쓰기(swap out) 동작을 구분한다.

**3. Stack Growth**

[userprog/process.c]

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- 페이지 폴트가 발생한 주소가 스택 확장 조건을 만족할 경우 새로운 페이지를 할당하고 스택 크기를 확장한다. 먼저, is\_user\_vaddr(addr) 함수를 사용하여 가상 주소가 사용자 주소 공간에 있는지 확인한다. 주소가 커널 공간에 접근하거나 esp - 32보다 낮은 경우 스택 확장에 실패하게 된다.

- frame\_alloc(PAL\_USER | PAL\_ZERO)를 호출하여 새로운 물리 메모리 페이지를 할당한다. 성공 시 해당 페이지는 0으로 초기화된다.

- pt\_create() 함수를 호출하여 스택 페이지를 위한 페이지 엔트리를 생성한다.

- pg\_round\_down(addr)로 주소를 페이지 단위로 정렬하여 관리한다. 스택 페이지는 SWAPPED 타입으로 설정해준다.

- install\_page() 함수를 호출하여 가상 주소와 물리 메모리를 매핑한다. 매핑 실패 시 frame\_free 함수를 통해 물리 메모리 프레임을 해제하고 false를 반환한다.

- pt\_insert로 새로운 페이지 엔트리를 보조 페이지 테이블에 삽입하고 true를 반환한다.

[vm/frame.\*]

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

- frame\_alloc() 함수는 새로운 frame 구조체를 할당하고 초기화한다. frame 구조체는 물리 메모리 프레임에 대한 정보를 저장한다.

- palloc\_get\_page(flags)를 사용하여 물리 메모리에서 페이지를 할당한다. 할당된 페이지는 frame->kaddr에 저장된다.

- 만약 palloc\_get\_page()가 NULL을 반환하여 메모리 할당에 실패할 경우 frame\_evict() 함수를 호출하여 페이지를 스왑 아웃하고 다시 할당을 시도한다.

- 할당된 프레임을 frame\_list에 추가하여 관리한다. frame\_lock을 사용해 메모리 할당 및 해제를 안전하게 처리한다.

- 마지막에 할당된 프레임을 반환한다. 이 프레임은 이후에 사용되며 페이지 폴트 처리에서 가상 메모리와 물리 메모리 간의 매핑에 사용된다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

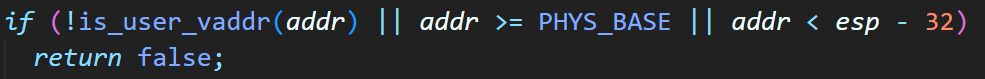
- frame\_free() 함수는 물리 메모리 페이지 kaddr을 해제하는 역할을 한다.

- 먼저 frame\_list를 순회하여 kaddr에 해당하는 프레임을 찾는다. 찾은 프레임을 frame\_remove() 함수를 호출하여 리스트에서 제거하고 해당 페이지의 페이지 테이블 엔트리를 pagedir\_clear\_page()로 삭제한다.

- 이후 물리 메모리 페이지를 palloc\_free\_page()로 해제하고 frame 구조체를 free() 하여 메모리 리소스를 반환하다. 해제 작업 시 frame\_lock을 사용하여 프레임 해제 작업이 다른 스레드에서 변경되지 않도록 보호한다.

* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

[userprog/process.c/expand\_stack 함수]



- 스택 포인터(esp)보다 충분히 낮은 주소에서 페이지 폴트가 발생할 때도 잘못된 스택 확장이 이루어지는 경우가 있었다. 이는 스택 확장의 유효성 검사 로직에서 경계 조건이 제대로 설정되지 않아 발생하였다. 따라서 expand\_stack() 함수에서 다음과 같은 유효성 검사를 추가하여 문제를 해결하였다.

[vm/swap.\*]

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- 메모리 부족 상태에서 스왑 공간이 모두 사용되면 프로그램이 중단되어 PANIC이 발생하는 문제가 있었다. 이는 find\_swap\_index 함수에서 빈 스왑 슬롯을 찾을 수 없을 때 즉시 프로그램을 중단하도록 설정되어 있었기 때문이다. 따라서 BITMAP\_ERROR를 반환하여 스왑 슬롯 부족을 호출자에게 알리도록 if문을 추가하였다.

[vm/mmap.\*]

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- mmap으로 매핑된 파일이 프로그램 종료 시 디스크에 올바르게 기록되지 않는 문제가 있었는데, 이는 remove\_mapping 함수에서 더티 페이지 처리 로직이 누락되어 발생하였다. 이를 위해 remove\_mapping 함수에서 더티 플래그를 확인한 뒤 mmap 데이터를 처리하도록 코드를 수정하였다.

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

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

- cspro에서 make check 수행 시 수행 시간이 매우 오래 걸려 정확한 수행 결과를 확인할 수 없었다. 따라서 vscode를 이용해 local 환경에서 작업한 후 make check 수행 결과를 확인하였고, 이를 캡쳐하여 첨부하였다.

- 채점 대상 테스트 케이스 16개를 포함한 113개 test 모두 pass 하였다.