

# **Pintos Project 4: Virtual Memory**

담당 교수 : 김영재

이름 / 학번 : 정서영 / 20211589

개발 기간 : 25.11.20 ~ 25.12.21

## I. 개발 목표

본 프로젝트는 Pintos 운영체제에서 Virtual Memory Paging 시스템을 구현하는 것을 목표로 한다. 현재 Pintos는 프로그램을 메모리에 올릴 때 Virtual Page 단위로 나누지만 즉시 Frame을 할당 받아서 Physical Memory에 올리기 때문에, 실행할 수 있는 프로그램의 크기와 개수가 Physical Memory 크기에 의해 제한된다. 따라서 프로세스별 Supplemental Page Table을 구현하여 Demand Paging을 수행하고, Frame Pool에 여유 공간이 없을 때 Disk Swapping을 통해 Physical Memory 제약을 극복한다. 또한 Page Fault Handler를 수정하여 실질적인 Page Fault Handling을 수행하고, Stack Growth를 지원하여 효율적인 메모리 관리를 구현한다.

## II. 개발 범위 및 내용

### A. 개발 범위

#### 1. Page Table & Page Fault Handler

현재 Pintos는 Page Fault 발생 시 해당 프로그램을 즉시 종료시킨다. 하지만 Virtual Memory 시스템에서는 Page Fault가 정상적인 동작의 일부이므로, Page Fault Handler를 수정하여 Fault가 발생한 Page에 대해 Frame을 할당받고 필요한 데이터를 로드한 후 Faulting Instruction을 재실행할 수 있도록 해야 한다. 이를 위해 기존 Page Table에 보조 정보를 추가한 Supplemental Page Table을 구현하여 각 Page의 상태(메모리 로드 여부, 연결된 파일 등)를 추적하고, Page Fault 발생 시 적절한 처리를 수행할 수 있도록 한다. 이를 통해 프로그램이 실제로 필요한 Page만 메모리에 로드하는 Demand Paging이 가능해진다.

#### 2. Disk Swap

Physical Memory의 모든 Frame이 사용 중일 때 새로운 Frame이 필요하면, 현재 메모리에 있는 Page 중 하나를 Disk로 내보내고(Swap-out) 그 Frame을 재사용해야 한다. 이를 위해 Frame Table을 구현하여 모든 Frame의 사용 상태를 추적하고, Page Replacement Algorithm을 통해 Victim Page를 선정한다. Swap Table을 구축하여 Disk 내 Swap Space의 사용 가능한 슬롯을 관리하며, 필요 시 Swap-out된 Page를 다시 메모리로 가져오는(Swap-in) 작업을 수행한다. 이를 통해 Physical Memory 크기보다 큰 프로그램이나 더 많은 프로그램을 동시에 실행할 수 있게 된다.

#### 3. Stack Growth

프로그램 실행 초기에는 Stack을 최소 크기로 할당하고, 실행 중 필요에 따라 동적으로 확장할 수

있어야 한다. Page Fault가 발생했을 때 Faulting Address에 대한 PTE가 존재하지 않더라도, 해당 주소가 Stack의 확장 가능한 영역 내에 있다면 새로운 Stack Page를 할당하여 Stack을 성장시켜야 한다. 이를 위해 Faulting Address가 유효한 Stack 접근인지 판단하는 조건을 검사하고, 조건을 만족하면 Stack을 확장한다. 이를 통해 메모리를 효율적으로 사용하면서도 프로그램이 필요한 만큼의 Stack 공간을 확보할 수 있다.

## B. 개발 내용

### 1. Page Fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

Page Fault는 CPU가 특정 Virtual Address에 접근하려 할 때, 해당 Virtual Page가 Physical Frame에 매핑되어 있지 않을 때 발생하는 예외 상황이다. Virtual Memory 시스템에서는 크게 세 가지 경우에 Page Fault가 발생한다.

첫째, Demand Paging을 사용하는 경우이다. 프로그램을 처음 실행할 때 모든 Page를 메모리에 올리지 않고, 실제로 접근이 필요할 때 해당 Page만 메모리에 로드한다. 따라서 프로그램의 Code나 Data 영역에 처음 접근할 때 해당 Page가 아직 메모리에 로드되지 않았으므로 Page Fault가 발생한다.

둘째, Page가 Swap Disk로 내보내진 경우이다. Physical Memory가 부족하여 특정 Page를 Disk로 Swap Out했다가 나중에 그 Page에 다시 접근하려 하면 Page Fault가 발생한다. 이때 Page Fault Handler는 Swap Disk에서 해당 Page를 다시 메모리로 가져와야 한다.

셋째, Stack Growth가 필요한 경우이다. 프로그램이 Stack을 사용하다가 현재 할당된 Stack 공간을 초과하여 접근하면 Page Fault가 발생한다. 이때 접근이 유효한 Stack 확장이라면 새로운 Page를 할당하여 Stack을 확장해야 한다.

Page Fault가 발생하면 MMU가 Exception을 발생시키고, OS가 Page Fault Handler를 수행한다. Page Fault Handler는 다음과 같은 순서로 동작한다.

먼저, CR2 Register에서 Faulting Address를 읽어온다. 이 주소는 Page Fault를 일으킨 메모리 접근의 대상 주소이다. 그리고 Interrupt Frame의 error\_code를 확인하여 Fault의 원인을 파악한다. 다음으로, Faulting Address가 포함된 Virtual Page에 대한 PTE를 Supplemental Page Table에서 찾는다.

PTE가 존재한다면 이는 일반적인 Page Fault 상황이므로, PTE의 정보를 바탕으로 적절한 처리를

수행한다. PTE의 Type이 Binary File이면 File System에서 데이터를 읽어오고, Swap이면 Swap Disk에서 데이터를 읽어오며, Memory Mapped File이면 해당 파일에서 데이터를 읽어온다.

이때 Frame Pool에서 빈 Frame을 할당받아 데이터를 로드하고, Page Table을 업데이트하여 Virtual Page와 Physical Frame을 매핑한다. 만약 Frame Pool에 빈 Frame이 없다면 Page Replacement Algorithm을 실행하여 Victim Page를 선정하고, 해당 Page를 Swap Disk로 내보낸 후 그 Frame을 재사용한다. 모든 처리가 완료되면 CPU가 Faulting Instruction을 다시 실행한다. 이번에는 Page가 메모리에 존재하므로 정상적으로 실행된다.

만약 PTE가 존재하지 않는다면, Faulting Address가 Stack Growth 가능 영역인지 확인한다. 조건을 만족하면 새로운 Stack Page를 할당하고, 그렇지 않으면 이는 Invalid Memory Access이므로 프로세스를 종료시킨다.

## 2. Disk Swap 발생 시 사용한 page replacement algorithm에 대해 서술

Physical Memory의 모든 Frame이 사용 중일 때 새로운 Page를 로드하려면, 현재 메모리에 있는 Page 중 하나를 Victim으로 선정하여 Swap Disk로 내보내야 한다. 이때 사용하는 Page Replacement Algorithm은 시스템의 성능에 큰 영향을 미친다. 이상적인 Algorithm은 앞으로 가장 오랫동안 사용하지 않을 Page를 선택하는 것이지만, 미래를 예측할 수 없으므로 실제로는 과거의 참조 패턴을 바탕으로 예측한다. LRU Algorithm은 가장 오랫동안 참조되지 않은 Page를 선택하는 방식으로, Temporal Locality 원리에 기반한다. 하지만 정확한 LRU를 구현하려면 모든 Page의 참조 시간을 추적해야 하므로 오버헤드가 크다.

따라서 Pintos에서는 LRU를 근사한 Second Chance (Clock) Algorithm을 사용한다. 이 알고리즘은 하드웨어가 제공하는 Referenced Bit를 활용한다. CPU가 특정 Page에 접근하면 하드웨어가 자동으로 해당 Page의 Referenced Bit를 1로 설정한다. Second Chance Algorithm의 동작 방식은 다음과 같다. 모든 Frame을 원형 큐로 구성하고, Clock Hand라는 포인터가 큐를 순회한다. Victim을 찾을 때 Clock Hand가 가리키는 Frame의 Referenced Bit를 확인한다. Referenced Bit가 0이면 최근에 참조되지 않은 Page이므로 이를 Victim으로 선정한다. Referenced Bit가 1이면 최근에 참조된 Page이므로, Bit를 0으로 Clear하고 Clock Hand를 다음 Frame으로 이동시킨다. 이 과정을 Referenced Bit가 0인 Page를 찾을 때까지 반복한다.

이 알고리즘의 장점은 구현이 간단하고 오버헤드가 적으면서도, 최근에 자주 참조되는 Page는 보호하고 오래 참조되지 않은 Page를 우선적으로 선택한다는 점이다. 한 바퀴를 도는 동안 Referenced Bit가 1인 Page는 모두 0으로 Clear되므로, 다음 순회에서는 그동안 다시 참조되지

않은 Page가 Victim이 된다.

Victim Page를 선정한 후에는 해당 Page의 Dirty Bit를 확인한다. Dirty Bit가 1이면 Page의 내용이 수정되었으므로 Swap Disk에 Write해야 하고, 0이면 수정되지 않았으므로 단순히 Frame을 회수하면 된다. Swap Out이 완료되면 해당 Frame은 새로운 Page를 위해 재사용된다.

### 3. Stack Growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술

Page Fault가 발생했을 때, Supplemental Page Table에 해당 Virtual Page에 대한 PTE가 없다면, 이는 Invalid Memory Access이거나 Stack Growth가 필요한 상황이다. Stack Growth가 필요한지 판단하기 위해서는 여러 조건을 검사해야 한다.

첫째, Faulting Address가 User 주소 공간에 있어야 한다. User 프로그램이 Kernel 주소 공간에 접근하려는 것은 Invalid Access이므로, `is_user_vaddr()` 함수로 Faulting Address가 PHYS\_BASE 아래에 있는지 확인한다.

둘째, Stack의 최대 크기 제한을 확인한다. Pintos는 각 프로세스의 Stack 크기를 8MB로 제한한다. 초기 Stack은 User Virtual Address Space의 최상단에서 시작하여 낮은 주소 방향으로 성장한다. 따라서 ‘PHYS\_BASE - Faulting Address’의 값이 8MB를 초과하면 Stack Limit를 벗어난 것이므로 Invalid Access이다.

셋째, PUSHA 연산과 관련된 조건을 확인한다. 80x86 아키텍처의 PUSHA 명령어는 모든 레지스터를 Stack에 Push하는데, ESP(Stack Pointer)에서 최대 32 Bytes 떨어진 위치에 접근할 수 있다. 즉, PUSHA 명령어 실행 중에는 ESP-32 범위 내의 주소에 대해 Page Fault가 발생할 수 있다. 따라서 Faulting Address가 ESP에서 32 Bytes 이내에 있으면 유효한 Stack 접근으로 간주한다.

이러한 모든 조건을 만족하면 해당 Faulting Address는 유효한 Stack 확장 요청으로 판단하고, 새로운 Stack Page를 할당한다. 만약 위 조건 중 하나라도 만족하지 않으면 이는 Invalid Memory Access(Segmentation Fault)이므로 프로세스를 종료시킨다.

## III. 추진 일정 및 개발 방법

### A. 추진 일정

기간	추진 내용
----	-------

11/20 ~ 12/4	Pintos 소스 코드 분석 및 명세서 검토
12/5 ~ 12/6	Supplemental Page Table 설계 및 구현
12/7 ~ 12/8	Page Fault Handler 수정 및 기본 Demand Paging 구현
12/9 ~ 12/10	Frame Table 구현 및 Frame 할당 메커니즘 작성
12/11~12/12	Second Chance Algorithm 구현
12/13 ~ 12/14	Swap Table 및 Disk Swapping 기능 구현
12/15 ~ 12/16	Stack Growth 기능 구현
12/17 ~ 12/18	Memory Mapped Files 구현
12/19 ~ 12/21	전체 테스트 검증 및 문서 작성

## B. 개발 방법

### 1. Page Table

먼저, Supplemental Page Table을 구현하기 위해 vm/page.h와 vm/page.c 파일을 생성한다.

vm/page.h에 page\_table\_entry 구조체를 정의한다. 이 구조체는 virtual page 주소, physical frame 주소, 메모리 로드 여부, 현재 페이지 탑입, 원본 페이지 탑입, 연결된 파일, 파일 오프셋, 읽을 바이트 수, 0으로 채울 바이트 수, 쓰기 가능 여부, 스왑 슬롯 번호, 메모리 맵 ID 필드를 포함한다. page\_type enum을 정의하여 PAGE\_BINARY(바이너리 파일), PAGE\_SWAP(스왑 페이지), PAGE\_MMAP(메모리 맵 파일), PAGE\_STACK(스택 페이지)을 구분한다.

Supplemental Page Table은 Hash 자료구조로 구현한다. swap\_init()으로 Hash Table을 초기화하고, page\_hash()와 page\_less()를 Hash 함수와 비교 함수로 사용한다. spt\_insert()로 PTE를 Hash Table에 삽입하고, spt\_find()로 특정 Virtual Address에 해당하는 PTE를 검색한다.

spt\_create\_page()로 새로운 PTE를 생성하고 초기화한다. spt\_remove()와 spt\_remove\_page()로 PTE를 제거하며, spt\_destory()로 전체 Supplemental Page Table을 해제한다. PTE 제거 시 cleanup\_pte\_resuources()를 호출하여 페이지 탑입에 따라 적절한 자원 해제를 수행한다.

## 2. Frame Table

다음으로, Frame Table을 구현하기 위해 vm/frame.h와 vm/frame.c 파일을 생성한다.

vm/frame.h에 frame\_table\_entry 구조체를 정의한다. 이 구조체는 Physical Frame 주소, Virtual Page 주소, 소유 스레드, 고정 여부, List 요소 필드를 포함한다.

Frame Table은 전역 리스트로 구현하며, Second Chance Algorithm을 위한 clock\_hand 포인터와 동기화를 위한 frame\_lock을 함께 관리한다. frame\_init()으로 Frame Table을 초기화하고, get\_frame()으로 Frame을 할당받으며, free\_frame()으로 Frame을 해제한다.

get\_frame() 함수는 palloc\_get\_page()로 Frame 할당을 시도하고, 실패하면 evict\_page()를 호출하여 Page Replacement을 수행한다. 할당받은 Frame에 대한 PTE를 생성하고 Frame Table에 추가한다.

evict\_page() 함수는 Second Chance Algorithm을 구현한다. clock\_hand를 따라 Frame Table을 순회하며 Accessed Bit가 0인 Frame을 찾는다. Accessed Bit가 1이면 0으로 설정하고 다음으로 이동한다. Victim을 찾으면 handle\_eviction()을 호출하여 실제 Eviction을 수행한다.

handle\_eviction() 함수는 페이지 타입에 따라 적절한 처리를 수행한다. PAGE\_BINARY는 Dirty Bit를 확인하여 수정되었으면 Swap Out하고, PAGE\_STACK은 항상 Swap Out하며, PAGE\_MMAP는 Dirty Bit를 확인하여 파일에 기록한다.

frame\_clear\_owner() 함수는 프로세스 종료 시 해당 스레드가 소유한 모든 Frame을 해제한다.

## 3. Swap Table

Swap Table을 구현하기 위해 vm/swap.h와 vm/swap.c 파일을 생성한다.

vm/swap.h에 swap\_table 구조체를 정의한다. 이 구조체는 swap\_block(Swap Disk의 block device), swap\_bitmap(슬롯 사용 여부 관리), swap\_lock(동기화용 락), swap\_size(전체 슬롯 개수) 필드를 포함한다.

Swap Table은 전역 자료구조로 하나만 존재하며, 모든 프로세스가 공유한다. swap\_init()으로 Swap Table을 초기화하고, swap\_out()으로 Frame을 Swap Disk에 기록하며, swap\_in()으로 Swap Disk에서 Frame으로 읽어온다. swap\_free()로 Swap 슬롯을 해제한다.

swap\_init() 함수는 block\_get\_role(BLOCK\_SWAP)로 Swap Disk를 얻고, block\_size()로 섹터 수를 구한 후 페이지 수를 계산한다. bitmap\_create()로 비트맵을 생성하여 각 슬롯의 사용 여부를

추적한다.

swap\_out() 함수는 bitmap\_scan\_and\_flip()으로 빈 슬롯을 찾아 예약하고, blcok\_write()로 페이지를 섹터 단위로 Swap Disk에 기록한다. PGSIZE를 BLOCK\_SECTOR\_SIZE로 나눈 만큼 반복하여 기록한다.

swap\_in() 함수는 block\_read()로 Swap Disk에서 페이지를 섹터 단위로 읽어온 후, bitmap\_set()으로 슬롯을 해제한다. Swap In 후 자동으로 슬롯을 해제하여 재사용 가능하게 한다.

#### 4. Stack Growth

스택 관리 기능을 구현하기 위해 vm/stack.h와 vm/stack.c 파일을 생성한다.

vm/stack.h에 스택 관련 상수와 함수 프로토타입을 정의한다. STACK\_MAX\_SIZE 매크로를 정의하여 스택의 최대 크기를 제한한다. is\_valid\_stack\_access() 함수로 페이지 폴트가 발생한 주소가 정당한 스택 접근인지 검사하고, grow\_stack() 함수로 스택을 동적으로 확장한다.

is\_valid\_stack\_access() 함수는 fault\_addr과 esp를 인자로 받아 세 가지 조건을 검증한다. 첫째, 사용자 주소 공간인지 확인한다. 둘째, 최대 스택 크기를 초과하지 않는지 검사한다. 셋째, 스택 포인터 기준으로 유효한 접근인지 판단한다.

grow\_stack() 함수는 upage를 인자로 받아 새로운 스택 페이지를 할당한다. 페이지 경계로 정렬하고, 중복 매핑을 방지하기 위해 SPT를 확인한다. 새 PTE를 생성하고 스택 페이지로 설정한다. 물리 메모리를 할당받고 페이지 테이블에 매핑한다. 각 단계에서 실패 시 할당받은 자원을 정리하고 false를 반환한다.

#### 5. Memory Mapped File

Memory Mapped File 기능을 구현하기 위해 vm/mmap.h와 vm/mmap.c 파일을 생성한다.

vm/mmap.h에 mmap\_entry 구조체를 정의한다. 이 구조체는 mapid(매핑 고유 ID), fd(파일 디스크립터), file(파일 포인터), addr(매핑 시작 주소), length(파일 길이), page\_count(페이지 개수), elem(리스트 요소) 필드를 포함한다. 각 프로세스는 여러 개의 Memory Mapped File을 가질 수 있으므로, thread 구조체의 mmap\_list에 mmap\_entry를 연결하여 관리한다.

mmap\_insert() 함수로 파일을 메모리에 매핑하고, mmap\_munmap() 함수로 매핑을 해제하며, mmap\_unmap\_all() 함수로 프로세스 종료 시 모든 매핑을 해제한다. check\_mmap\_overlap()

함수는 매핑 영역의 종복을 검사한다.

Memory Mapped File은 Lazy Loading 방식을 사용한다. mmap() 시스템 콜이 호출되면 즉시 파일을 메모리에 로드하지 않고, 각 Virtual Page에 대한 PTE만 생성하여 Supplemental Page Table에 등록한다. 실제 파일 내용은 해당 페이지에 접근할 때 Page Fault가 발생하면 그때 로드한다. 이를 통해 메모리 사용을 최적화하고 실제로 접근하는 부분만 메모리에 올릴 수 있다.

## 6. Demand Paging

Demand Paging을 구현하기 위해 threads/init.c, threads/thread.h, threads/thread.c, userprog/process.c 파일을 수정한다.

threads/init.c의 main() 함수에서 frame\_init()과 swap\_init()을 호출하여 시스템 초기화 시 Frame Table과 Swap Table을 준비한다.

threads/thread.h의 thread 구조체에 Supplemental Page Table, Memory Mapped File 리스트, 다음 mapid 필드를 추가한다. threads/thread.c의 init\_thread() 함수에서 SPT의 buckets 포인터를 NULL로, mmap\_list를 빈 리스트로, next\_mapid를 0으로 초기화한다.

userprog/process.c의 start\_process() 함수에서 spt\_init()을 호출하여 프로세스 시작 시 Supplemental Page Table을 초기화한다. process\_exit() 함수에서는 프로세스 종료 시 mmap\_unmap\_all()로 모든 Memory Mapped File을 정리하고, frame\_clear\_owner()로 해당 프로세스가 사용하던 모든 Frame을 해제하며, spt\_destroy()로 Supplemental Page Table을 정리한다.

load\_segment() 함수를 수정하여 Demand Paging을 구현한다. 기존 방식은 파일의 모든 내용을 즉시 메모리에 로드했으나, 이를 변경하여 각 페이지에 대한 PTE만 생성하고 Supplemental Page Table에 등록한다. 실제 데이터 로드는 해당 페이지에 처음 접근할 때 Page Fault Handler가 수행한다. 각 PTE는 파일 포인터, 파일 오프셋, 읽을 바이트 수, 0으로 채울 바이트 수 등의 정보를 저장한다.

## 7. Page Fault Handler

Page Fault Handler를 구현하기 위해 userprog/exception.c 파일을 수정한다.

page\_fault() 함수를 수정하여 실질적인 Page Fault 처리를 수행한다. Faulting Address가 사용자

주소 공간에 있고 not\_present 상황이면, 먼저 spt\_find()로 해당 페이지의 PTE를 찾는다.

PTE가 존재하면 handle\_mm\_fault()를 호출하여 Demand Paging을 수행한다. PTE가 존재하지 않으면 is\_valid\_stack\_access()로 유효한 Stack 접근인지 확인하고, 조건을 만족하면 grow\_stack()으로 Stack을 확장한다. 모든 처리가 실패하면 Invalid Memory Access로 판단하여 프로세스를 종료한다.

handle\_mm\_fault() 함수를 구현하여 페이지를 실제로 메모리에 로드한다. write 접근인데 writable이 false이면 권한 위반이므로 false를 반환한다. 이미 is\_loaded가 true이면 이미 로드된 이미 로드된 페이지이므로 true를 반환한다. get\_frame()으로 Physical Frame을 할당받고, PTE의 type에 따라 적절한 처리를 수행한다. PAGE\_BINARY와 PAGE\_MMAP는 파일에서 데이터를 읽고, PAGE\_SWAP은 Swap Disk에서 데이터를 읽으며, PAGE\_STACK은 0으로 초기화한다. pagedir\_set\_page()로 Page Table에 맵핑하고, PTE의 kpage와 is\_loaded를 갱신한다.

## 8. Memory Mapped File 관련 System Calls

Memory Mapped File을 지원하기 위해 userprog/syscall.h와 userprog/syscall.c 파일을 수정한다.

sys\_mmap() 함수는 파일을 메모리에 맵핑하는 시스템 콜이다. 파일 디스크립터와 맵핑할 시작 주소를 인자로 받아, 유효성 검사를 수행한 후 mmap\_insert()를 호출한다. fd가 0(STDIN)이나 1(STDOUT)이면 실패하고, addr이 NULL이거나 페이지 경계에 정렬되지 않았거나 사용자 주소 공간이 아니면 실패한다. file\_length()로 파일 크기를 확인하여 0이면 실패한다. check mmap\_overlap()으로 맵핑 영역이 기존 페이지와 겹치는지 확인한다. file\_reopen()으로 독립적인 파일 포인터를 얻고, mmap\_insert()를 호출하여 실제 맵핑을 수행한다. 성공하면 mapid를 반환하고, 실패하면 -1을 반환한다.

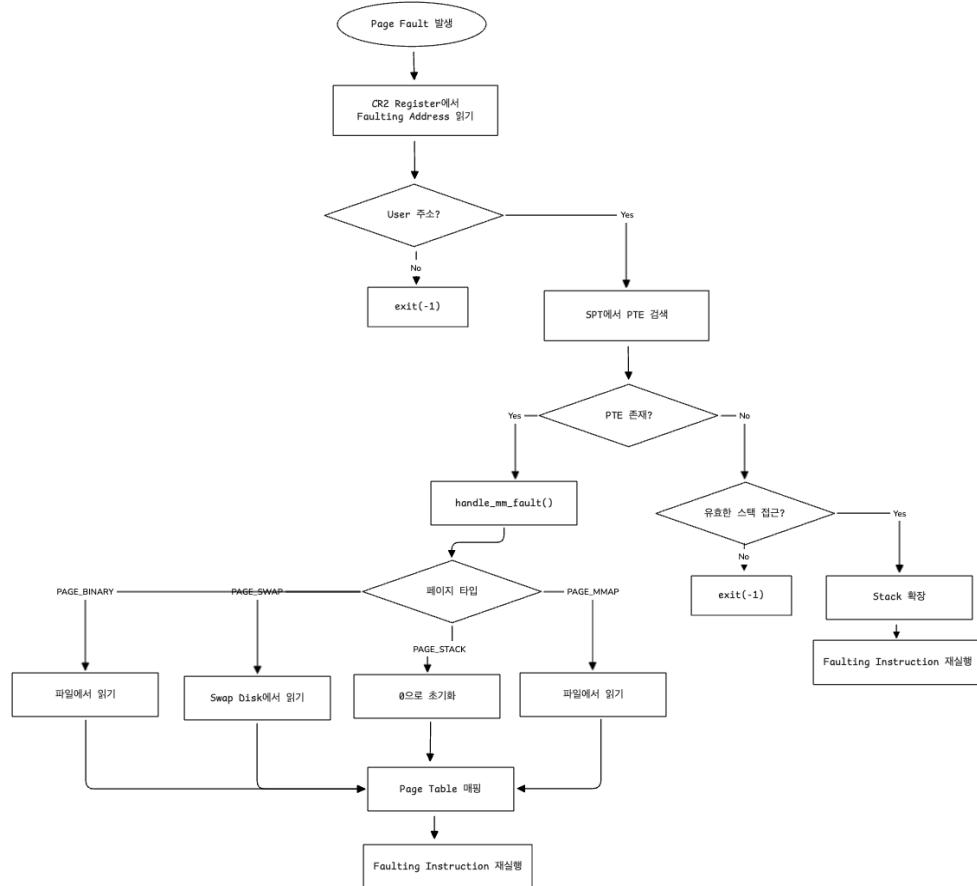
sys\_munmap() 함수는 맵핑을 해제하는 시스템 콜이다. mapid를 인자로 받아 mmap\_munmap()을 호출한다. 변경된 페이지는 파일에 Write Back되고, 모든 리소스가 정리된다.

또한 기존 check\_valid\_uaddr() 함수는 pagedir\_get\_page()로만 주소를 검증했으나, Demand Paging에서는 유효한 페이지가 아직 메모리에 로드되지 않았을 수 있다. 따라서 spt\_find()로 Supplemental Page Table도 확인하여, SPT에 등록된 페이지는 유효한 주소로 인정한다.

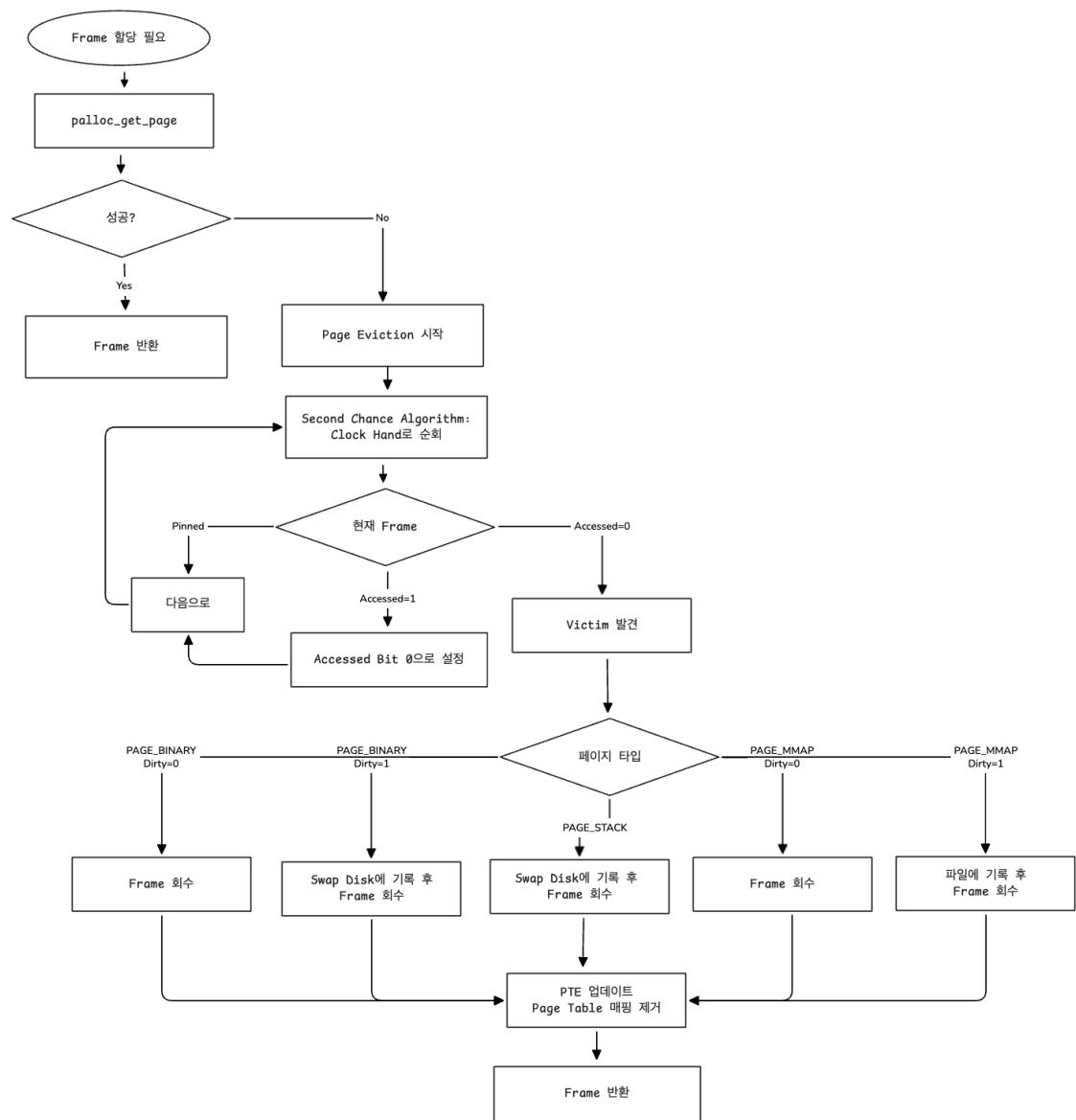
## IV. 연구 결과

### A. Flow Chart

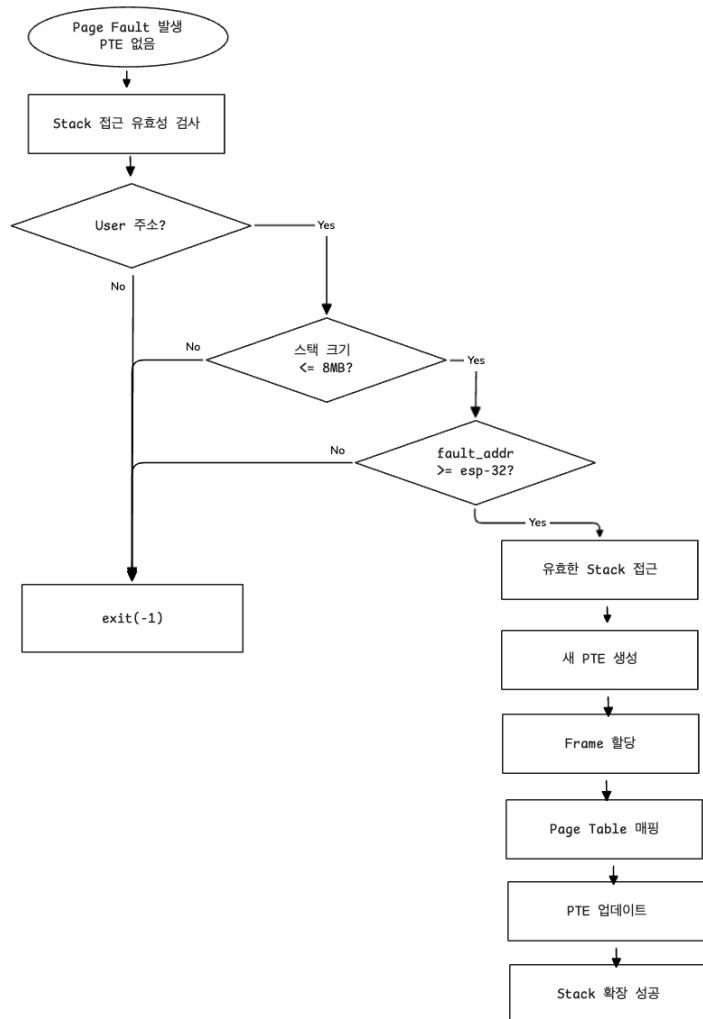
#### 1. Page Fault Handling 과정



## 2. page replacement algorithm 과정



### 3. Stack Growth - Stack 확장 여부 판단 과정



## B. 제작 내용

- [ Project 3 세마포어 우선순위 스케줄링 구현 수정 ]

먼저 Project 3에서 구현한 스케줄링 메커니즘의 세마포어 구현에 문제가 있음을 발견했다.

(As is)

```
void
sema_up (struct semaphore *sema)
{
    enum intr_level old_level;

    ASSERT (sema != NULL);

    old_level = intr_disable ();
    if (!list_empty (&sema->waiters)) {
        // waiters 리스트를 우선순위 순으로 정렬
        list_sort(&sema->waiters, thread_priority_compare, NULL);
        // 가장 높은 우선순위 스레드를 꺼내서 깨움
        thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                    struct thread, elem));
    }
    sema->value++;

    // 현재 스레드보다 높은 우선순위 스레드가 깨어났으면 양보할 수 있도록 함
    thread_yield();
    intr_set_level (old_level);
}
```

Project 3에서 sema\_up() 함수에서 waiters 리스트를 list\_sort()로 정렬한 후 가장 높은 우선순위를 가진 스레드를 깨우는 방식으로 구현했으나, userprog 테스트에서 “Kernel PANIC at ../../threads/vaddr.h:84 in vtop(): assertion ‘is\_kernel\_vaddr (vaddr)’ failed” 커널 패닉이 발생했다. 디버깅 출력을 추가한 결과, sema\_up()이 이미 인터럽트가 비활성화된 컨텍스트에서 호출되고 있었고, thread\_yield()를 인터럽트 복원 전에 호출하여 스케줄러가 비정상 동작하는 것을 확인했다. 이를 해결하기 위해 intr\_set\_level(old\_level)을 thread\_yield() 이전에 호출하고, old\_level이 INTR\_ON일 때만 조건부로 thread\_yield()를 호출하도록 수정했다.

그러나 인터럽트 순서를 수정한 후에도 “Kernel PANIC at ../../threads/thread.c:730 in schedule(): assertion ‘is\_thread (next)’ failed” 에러가 발생했다. 이는 스케줄러가 유효하지 않은 스레드 구조체를 참조하고 있다는 의미였다. 근본적인 문제는 list\_sort() 사용에 있었다. list\_sort()는 리스트 노드들의 포인터를 재배치하는데, struct thread의 elem 필드는 하나의 리스트에만 속할 수 있다. 정렬 과정에서 리스트 구조가 재배치되면서 스레드 구조체의 연결이 손상되고, 이로 인해 스케줄러가 유효하지 않은 스레드를 참조하게 되어 커널 패닉이 발생한 것이다. 즉, list\_sort()는 세마포어의 waiters 리스트처럼 스레드가 직접 참여하는 리스트에 사용할 수 없었다.

list\_sort()를 제거하고 삽입 시점에 우선순위를 유지하는 방식으로 변경했다. sema\_down()에서 list\_push\_back() 대신 list\_insert\_ordered()를 사용하여 스레드를 우선순위 순서에 맞게 삽입하도록 했다. 이렇게 하면 waiters 리스트가 항상 우선순위 순으로 정렬된 상태를 유지하므로 sema\_up()에서는 단순히 list\_pop\_front()로 가장 높은 우선순위의 스레드를 꺼내기만 하면 된다. 이 방식은 리스트 노드들을 재배치하지 않으므로 스레드 구조체의 연결을 손상시키지 않으며, 인터럽트 컨텍스트를 고려한 thread\_yield() 호출로 안정성도 보장된다.

(To be)

```
void
sema_down (struct semaphore *sema)
{
    enum intr_level old_level;

    ASSERT (sema != NULL);
    ASSERT (!intr_context ());

    old_level = intr_disable ();
    while (sema->value == 0)
    {
        list_insert_ordered(&sema->waiters, &thread_current()->elem,
                            ||| thread_priority_compare, NULL);
        thread_block ();
    }
    sema->value--;
    intr_set_level (old_level);
}
```

```
void
sema_up (struct semaphore *sema)
{
    enum intr_level old_level;

    ASSERT (sema != NULL);

    old_level = intr_disable ();
    if (!list_empty (&sema->waiters))
        thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                    struct thread, elem));
    sema->value++;
    intr_set_level (old_level);

    if (old_level == INTR_ON) {
        thread_yield();
    }
}
```

## 1. Page Table 관련 구현 (vm/page.h, vm/page.c)

vm/page.h에 page\_table\_entry 구조체를 정의했다. 이 구조체는 각 Virtual Page에 대한 상세 정보를 저장한다.

```
struct page_table_entry {
    void *upage;           // Virtual Page 주소
    void *kpage;           // Physical Frame 주소

    bool is_loaded;        // 메모리 로드 여부

    enum page_type type;   // 현재 페이지 타입
    enum page_type original_type; // 원본 페이지 타입

    struct hash_elem elem; // Hash Table 요소

    struct file *file;     // 연결된 파일
    off_t file_offset;    // 파일 오프셋
    uint32_t read_bytes;  // 읽을 바이트 수
    uint32_t zero_bytes;  // 0으로 채울 바이트 수
    bool writable;         // 쓰기 가능 여부

    size_t swap_slot;     // SWAP 슬롯 번호

    mapid_t mapid;        // 메모리 맵 ID
};
```

page\_type enum을 정의하여 네 가지 페이지 타입을 구분한다. PAGE\_BINARY는 실행 파일에서 로드된 페이지, PAGE\_SWAP은 스왑 디스크에 저장된 페이지, PAGE\_MMAP는 메모리 매핑된 파일 페이지, PAGE\_STACK은 스택 페이지를 나타낸다. original\_type 필드는 페이지가 스왑 아웃되기 전의 원래 타입을 저장하여, 스왑 인 후 원래 타입으로 복원할 수 있도록 한다.

```
enum page_type {
    PAGE_BINARY, // 바이너리 파일
    PAGE_SWAP,   // SWAP 페이지
    PAGE_MMAP,   // 메모리 맵 파일
    PAGE_STACK   // 스택 페이지
};
```

spt\_init() 함수를 구현하여 Hash Table을 초기화한다. hash\_init()을 호출하며 page\_hash()를 해시 함수로, page\_less()를 비교 함수로 지정한다. page\_hash() 함수는 PTE의 upage 주소를 기반으로 해시 값을 계산한다. hash\_bytes()를 사용하여 upage의 바이트 값을 해시한다. page\_less() 함수는 두 PTE의 upage 주소를 비교하여 Hash Table 내에서 순서를 결정한다.

```

void spt_init(struct hash *spt) {
    hash_init(spt, page_hash, page_less, NULL);
}

unsigned page_hash(const struct hash_elem *e, void *aux UNUSED) {
    const struct page_table_entry *pte = hash_entry(e, struct page_table_entry, elem);
    return hash_bytes(&pte->upage, sizeof(pte->upage));
}

bool page_less(const struct hash_elem *a, const struct hash_elem *b, void *aux UNUSED) {
    const struct page_table_entry *pte_a = hash_entry(a, struct page_table_entry, elem);
    const struct page_table_entry *pte_b = hash_entry(b, struct page_table_entry, elem);
    return pte_a->upage < pte_b->upage;
}

```

spt\_insert() 함수는 새로운 PTE를 Hash Table에 삽입한다. hash\_insert()를 호출하며, 이미 같은 upage를 가진 PTE가 존재하면 false를 반환한다.

```

bool spt_insert(struct hash *spt, struct page_table_entry *pte) {
    struct hash_elem *old_elem = hash_insert(spt, &pte->elem);
    return old_elem == NULL;
}

```

spt\_create\_page() 함수는 새로운 PTE를 생성하고 초기화한다. malloc()으로 메모리를 할당하고 memset()으로 0으로 초기화한 후, 기본값을 설정한다. upage는 pg\_round\_down()으로 페이지 경계에 맞춘다. 기본적으로 스택 페이지로 설정하며, writable을 true로 설정한다. spt\_insert()를 호출하여 Hash Table에 삽입하고, 실패하면 메모리를 해제하고 NULL을 반환한다.

```

struct page_table_entry *spt_create_page(struct hash *spt, void *upage) {
    struct page_table_entry *pte = malloc(sizeof(struct page_table_entry));
    if (pte == NULL) {
        return NULL;
    }

    memset(pte, 0, sizeof(*pte));
    pte->upage = pg_round_down(upage);
    pte->kpage = NULL;
    pte->is_loaded = false;
    pte->type = PAGE_STACK;
    pte->original_type = PAGE_STACK;
    pte->writable = true;
    pte->file = NULL;
    pte->file_offset = 0;
    pte->read_bytes = 0;
    pte->zero_bytes = 0;
    pte->swap_slot = 0;
    pte->mapid = -1;

    if(!spt_insert(spt, pte)) {
        free(pte);
        return NULL;
    }
    return pte;
}

```

spt\_find() 함수는 특정 Virtual Address에 해당하는 PTE를 검색한다. 임시 PTE 구조체를 생성하여 upage를 설정하고, hash\_find()로 Hash Table에서 검색한다. 찾으면 hash\_entry()로 PTE 포인터를 반환하고, 없으면 NULL을 반환한다.

```
struct page_table_entry *spt_find(struct hash *spt, void *upage) {
    struct page_table_entry pte_temp;
    struct hash_elem *e;

    pte_temp.upage = pg_round_down(upage);
    e = hash_find(spt, &pte_temp.elem);

    return e != NULL ? hash_entry(e, struct page_table_entry, elem) : NULL;
}
```

cleanup\_pte\_resources() 함수는 PTE가 보유한 리소스를 정리한다. 페이지 타입에 따라 적절한 자원 해제를 수행한다. PAGE\_BINARY인 경우 file을 file\_close()로 닫는다. PAGE\_SWAP인 경우 swap\_free()로 스왑 슬롯을 해제한다. PAGE\_MMAP인 경우 mmap\_munmap()에서 별도로 처리한다. Frame은 이후 Frame Table에서 관리되므로 여기서 해제하지 않는다.

```
static void cleanup_pte_resources(struct page_table_entry *pte) {
    if (pte == NULL) {
        return;
    }

    switch (pte->type) {
        case PAGE_BINARY:
            if (pte->file != NULL) {
                lock_acquire(&filesys_lock);
                file_close(pte->file);
                lock_release(&filesys_lock);
                pte->file = NULL;
            }
            break;

        case PAGE_SWAP:
            if (pte->swap_slot != 0) {
                swap_free(pte->swap_slot);
                pte->swap_slot = 0;
            }
            break;

        case PAGE_MMAP:
            break;

        case PAGE_STACK:
            break;

        default:
            break;
    }
}
```

초기 구현에서 PTE 제거 시 Frame을 즉시 해제하려 했으나, Frame은 Frame Table에서 별도로 관리되므로 cleanup\_pte\_resources()에서는 Frame을 해제하지 않도록 수정했다. Frame 해제는 후술될 frame\_clear\_owner()가 담당한다.

또한 페이지가 스왑 아웃된 후 탑입이 PAGE\_SWAP으로 변경되면 원래 탑입 정보를 잃어버리는 문제가 있었다. 이를 해결하기 위해 original\_type 필드를 추가하여 원본 탑입을 보존하고, 스왑 인 후 원래 탑입으로 복원할 수 있도록 했다.

spt\_remove() 함수는 PTE를 Hash Table에서 제거하고 메모리를 해제한다. hash\_delete()로 Hash Table에서 제거하고, pagedir\_clear\_page()로 Page Table의 매팅을 제거한다. cleanup\_pte\_resources()로 PTE의 리소스를 정리하고, free()로 PTE 메모리를 해제한다.

```
void spt_remove_page(struct hash *spt, void *upage) {
    struct page_table_entry *pte = spt_find(spt, upage);
    if (pte != NULL) {
        spt_remove(spt, pte);
    }
}
```

spt\_destroy() 함수는 전체 Supplemental Page Table을 해제한다. hash\_destroy()를 호출하며 spt\_destroy\_func()를 콜백으로 전달한다.

spt\_destroy\_func()은 각 PTE에 대해 cleanup\_pte\_resources()를 호출하고 메모리를 해제한다.

```
static void spt_destroy_func(struct hash_elem *e, void *aux UNUSED) {
    struct page_table_entry *pte = hash_entry(e, struct page_table_entry, elem);
    cleanup_pte_resources(pte);
    free(pte);
}

void spt_destroy(struct hash *spt) {
    hash_destroy(spt, spt_destroy_func);
}
```

## 2. Frame Table 관련 구현 (vm/frame.h, vm/frame.c)

vm/frame.h에 frame\_table\_entry 구조체를 정의했다. frame은 실제 Physical Frame의 주소를 저장하고, upage는 이 Frame에 매팅된 Virtual Page 주소를 저장한다. owner는 이 Frame을 소유한 스레드를 가리키며, pinned는 Frame이 Eviction 대상에서 제외되어야 하는지를 나타낸다.

```
struct frame_table_entry {
    void *frame;           // Physical Frame 주소
    void *upage;          // Virtual Page 주소
    struct thread *owner; // 소유 스레드
    bool pinned;          // 고정 여부

    struct list_elem elem; // List 요소
};
```

vm/frame.c에 전역 변수들을 선언했다. frame\_table은 모든 PTE를 관리하는 리스트이고, clock\_hand는 Second Chance Algorithm을 위한 포인터이며, frame\_lock은 Frame Table 접근 시 동기화를 위한 락이다.

```
static struct list frame_table;
static struct list_elem *clock_hand;
static struct lock frame_lock;
```

frame\_init() 함수를 구현하여 Frame Table을 초기화한다. list\_init()으로 frame\_table을 초기화하고, clock\_hand를 NULL로, frame\_lock을 lock\_init()으로 초기화한다.

```
void frame_init (void) {
    list_init(&frame_table);
    clock_hand = NULL;
    lock_init(&frame_lock);
}
```

get\_frame() 함수를 구현하여 Frame을 할당받는다. 먼저 palloc\_get\_page()로 Frame 할당을 시도한다. 실패하면 frame\_lock을 획득한 후 evict\_page()를 호출하여 페이지 교체를 수행한다. 할당받은 프레임을 얻지 못하면 NULL을 반환한다. 할당받은 Frame에 대한 FTE를 malloc()으로 생성하고, 필드를 초기화한 후 frame\_table에 추가한다.

```

void *get_frame (enum palloc_flags flags, void *upage) {
    void *frame = palloc_get_page(PAL_USER | flags);
    if (frame == NULL) {
        lock_acquire(&frame_lock);
        frame = evict_page();
        lock_release(&frame_lock);
        if (frame == NULL) {
            return NULL;
        }
    }

    struct frame_table_entry *fte = malloc(sizeof(struct frame_table_entry));
    if (fte == NULL) {
        palloc_free_page(frame);
        return NULL;
    }

    fte->frame = frame;
    fte->upage = upage;
    fte->owner = thread_current();
    fte->pinned = false;

    lock_acquire(&frame_lock);
    list_push_back(&frame_table, &fte->elem);
    lock_release(&frame_lock);

    return frame;
}

```

`cleanup_invalid_frames()` 함수를 구현하여 무효한 Frame들을 정리한다. Frame Table을 순회하면서 owner가 NULL이거나 pagedir가 NULL인 FTE를 찾아 제거한다. 제거 전 `update_clock_hand_if_needed()`를 호출하여 clock\_hand가 제거될 요소를 가리키고 있으면 갱신한다.

```

static void cleanup_invalid_frames(void) {
    struct list_elem *e = list_begin(&frame_table);

    while (e != list_end(&frame_table)) {
        struct frame_table_entry *fte = list_entry(e, struct frame_table_entry, elem);
        struct list_elem *next = list_next(e);

        if (fte->owner == NULL || fte->owner->pagedir == NULL) {
            update_clock_hand_if_needed(e);
            list_remove(e);
            palloc_free_page(fte->frame);
            free(fte);
        }

        e = next;
    }
}

```

```
static void update_clock_hand_if_needed(struct list_elem *removed_elem) {
    if (clock_hand == removed_elem) {
        clock_hand = list_next(clock_hand);
        if (clock_hand == list_end(&frame_table)) {
            clock_hand = list_begin(&frame_table);
        }
    }
}
```

evict\_page() 함수를 구현하여 Second Chance Algorithm을 수행한다.

먼저 cleanup\_invalid\_frames()로 무효한 Frame들을 정리한다. frame\_table이 비어있으면 NULL을 반환한다. clock\_hand가 NULL이거나 list\_end()를 가리키면 list\_begin()으로 초기화한다.

```
static void *evict_page (void) {
    struct frame_table_entry *victim = NULL;

    if (list_empty(&frame_table)) {
        return NULL;
    }

    cleanup_invalid_frames();

    if (list_empty(&frame_table)) {
        return NULL;
    }

    if (clock_hand == NULL || clock_hand == list_end(&frame_table))
        clock_hand = list_begin(&frame_table);
```

while 루프로 Victim을 찾는다. clcok\_hand가 가리키는 FTE의 owner와 pagedir의 유휴성을 확인하고, 무효하면 다음으로 이동한다. Pinned가 true이면 Eviction 대상에서 제외하고 다음으로 이동한다. spt\_find()로 PTE를 찾는다. PTE가 없거나 is\_loaded가 false이면 데이터 일관성이 깨진 상태이므로 간단히 정리하고 Frame을 반환한다.

```

while (true) {
    struct frame_table_entry *fte = list_entry(clock_hand, struct frame_table_entry, elem);

    if (fte->owner == NULL || fte->owner->pagedir == NULL) {
        clock_hand = list_next(clock_hand);
        if (clock_hand == list_end(&frame_table))
            | clock_hand = list_begin(&frame_table);
        continue;
    }

    if (fte->pinned) {
        clock_hand = list_next(clock_hand);
        if (clock_hand == list_end(&frame_table))
            | clock_hand = list_begin(&frame_table);
        continue;
    }

    struct page_table_entry *pte = spt_find(fte->owner->spt, fte->upage);

    if (pte == NULL || !pte->is_loaded) {
        victim = fte;
        clock_hand = list_next(clock_hand);
        if (clock_hand == list_end(&frame_table))
            | clock_hand = list_begin(&frame_table);

        list_remove(&victim->elem);
        void *frame = victim->frame;
        pagedir_clear_page(victim->owner->pagedir, victim->upage);
        free(victim);
        return frame;
    }
}

```

pagedir\_is\_accessed()로 Accessed Bit를 확인한다.

Accessed Bit가 1이면 pagedir\_set\_accessed()로 0으로 설정하고 clock\_hand를 다음으로 이동한다. Accessed Bit가 0이면 Victim으로 선택하고 clock\_hand를 다음으로 이동한 후 break한다.

```

// Second chance algorithm
if (pagedir_is_accessed(fte->owner->pagedir, fte->upage)) {
    pagedir_set_accessed(fte->owner->pagedir, fte->upage, false);
    clock_hand = list_next(clock_hand);
    if (clock_hand == list_end(&frame_table)) {
        | clock_hand = list_begin(&frame_table);
    }
} else {
    victim = fte;
    clock_hand = list_next(clock_hand);
    if (clock_hand == list_end(&frame_table)) {
        | clock_hand = list_begin(&frame_table);
    }
    break;
}

void *result = handle_eviction(victim);
return result;
}

```

handle\_eviction() 함수를 구현하여 실제 Eviction을 수행한다. Victim을 frame\_table에서 제거하고, spt\_find()로 PTE를 찾는다. PTE가 없으면 pagedir\_clear\_page()로 매팅을 제거하고 Frame을 반환한다. pte->type이 PAGE\_SWAP이거나 is\_loaded가 false이면 이미 Swap되었거나 로드되지 않은 상태이므로 매팅만 제거하고 Frame을 반환한다.

pagedir\_is\_dirty()로 Dirty Bit를 확인하고, pte->is\_loaded를 false로 설정한다.

```
static void *handle_eviction(struct frame_table_entry *victim) {
    struct page_table_entry *pte;
    void *frame = victim->frame;
    void *upage = victim->upage;
    struct thread *owner = victim->owner;

    // frame_table에서 제거
    list_remove(&victim->elem);

    // PTE 찾기
    pte = spt_find(&owner->spt, upage);
    if (pte == NULL) {
        pagedir_clear_page(owner->pagedir, upage);
        free(victim);
        return frame;
    }

    if (pte->type == PAGE_SWAP) {
        pagedir_clear_page(owner->pagedir, upage);
        pte->is_loaded = false;
        pte->kpage = NULL;
        free(victim);
        return frame;
    }

    if (!pte->is_loaded) {
        pagedir_clear_page(owner->pagedir, upage);
        pte->is_loaded = false;
        pte->kpage = NULL;
        free(victim);
        return frame;
    }

    bool dirty = pagedir_is_dirty(owner->pagedir, upage);
    pte->is_loaded = false;

    size_t swap_slot = 0;
    bool need_swap = false;
```

이후 페이지 탑입에 따라 처리한다.

PAGE\_BINARY인 경우 writable이거나 dirty이면 swap\_out()으로 Swap Disk에 기록한다. swap\_out() 실패 시 is\_loaded를 다시 true로 설정하고 victim을 frame\_table에 다시 추가한 후 NULL을 반환한다. writable이 아니고 dirty도 아니면 Swap 없이 Frame만 회수한다.

PAGE\_STACK인 경우 항상 swap\_out()으로 Swap Disk에 기록한다. PAGE\_MMAP인 경우 dirty이면 file\_write\_at()으로 파일에 기록한다. 이때 fileys\_lock을 획득하여 동기화를 보장한다.

```
switch (pte->type) {
    case PAGE_BINARY:
        if (pte->writable || dirty) {
            swap_slot = swap_out(frame);
            if (swap_slot == BITMAP_ERROR) {
                pte->is_loaded = true;
                list_push_back(&frame_table, &victim->elem);
                return NULL;
            }
            need_swap = true;
        } else {
        }
        break;

    case PAGE_STACK:
    {
        swap_slot = swap_out(frame);
        if (swap_slot == BITMAP_ERROR) {
            pte->is_loaded = true;
            list_push_back(&frame_table, &victim->elem);
            return NULL;
        }
        need_swap = true;
    }
    break;

    case PAGE_MMAP:
        if (dirty) {
            lock_acquire(&fileys_lock);
            file_write_at(pte->file, frame, pte->read_bytes, pte->file_offset);
            lock_release(&fileys_lock);
        }
        break;

    default:
        pagedir_clear_page(owner->pagedir, upage);
        pte->kpage = NULL;
        free(victim);
        return frame;
}
```

이후 pagedir\_clear\_page()로 Page Directory에서 맵핑을 제거하고 pte->kpage를 NULL로 설정한다. need\_swap이 true면 pte->swap\_slot을 설정하고 pte->original\_type에 현재 탑입을 저장한 후, pte->type을 PAGE\_SWAP으로 변경한다. victim을 free()하고 Frame을 반환한다.

```

// page directory에서 맵핑 제거
pagedir_clear_page(owner->pagedir, upage);
pte->kpage = NULL;

if (need_swap) {
    pte->swap_slot = swap_slot;
    pte->original_type = pte->type;
    pte->type = PAGE_SWAP;
}

free(victim);

return frame;
}

```

free\_frame() 함수를 구현하여 Frame을 해제한다. frame\_lock을 획득하고 frame\_table을 순회하여 해당 Frame을 가진 FTE를 찾는다. 찾으면 update\_clock\_hand\_if\_nedded()를 호출하고 list\_remove()로 제거한 후 free()한다. frame\_lock을 해제하고, 찾았으면 palloc\_free\_page()로 실제 메모리를 해제한다.

```

void free_frame (void *frame) {
    if (frame == NULL) {
        return;
    }

    lock_acquire(&frame_lock);

    struct list_elem *e;
    bool found = false;

    for (e = list_begin(&frame_table); e != list_end(&frame_table); e = list_next(e)) {
        struct frame_table_entry *fte = list_entry(e, struct frame_table_entry, elem);
        if (fte->frame == frame) {
            update_clock_hand_if_needed(e);

            list_remove(e);
            found = true;
            free(fte);
            break;
        }
    }

    lock_release(&frame_lock);

    if (found) {
        palloc_free_page(frame);
    }
}

```

frame\_clear\_owner() 함수를 구현하여 프로세스 종료 시 해당 스레드가 소요한 모든 Frame을 해제한다. frame\_lock을 획득하고 frame\_table을 순회하면서 owner가 해당 스레드인 PTE를 찾는다. Kernel 주소는 스킁한다. pagedir가 유효하면 pagedir\_clear\_page()로 맵핑을 제거하고, update\_clock\_hand\_if\_needed()를 호출한 후 list\_remove()로 제거하고 free()한다.

palloc\_free\_page()로 실제 메모리를 해제한다.

```
void frame_clear_owner(struct thread *t) {
    lock_acquire(&frame_lock);

    struct list_elem *e = list_begin(&frame_table);

    while (e != list_end(&frame_table)) {
        struct frame_table_entry *fte = list_entry(e, struct frame_table_entry, elem);
        struct list_elem *next = list_next(e);

        if (fte->owner == t) {
            if (is_kernel_vaddr(fte->upage)) {
                e = next;
                continue;
            }

            void *frame = fte->frame;
            void *upage = fte->upage;

            if (t->pagedir != NULL) {
                pagedir_clear_page(t->pagedir, upage);
            }

            update_clock_hand_if_needed(e);

            list_remove(e);
            free(fte);

            palloc_free_page(frame);
        }

        e = next;
    }

    lock_release(&frame_lock);
}
```

### 3. Swap Table 관련 구현 (vm/swap.h, vm/swap.c)

vm/swap.h에 swap\_table 구조체를 정의했다.

```
struct swap_table {
    struct block *swap_block;
    struct bitmap *swap_bitmap;
    struct lock swap_lock;
    size_t swap_size;
};
```

swap\_block은 devices/block.h의 block 구조체로 Swap Disk를 나타낸다. swap\_bitmap은 각 슬롯이 사용 중인지 여부를 비트로 관리한다. swap\_lock은 Swap Table 접근 시 동기화를 보장한다. swap\_size는 Swap Disk가 수용할 수 있는 페이지 개수이다.

vm/swap.c에 SECTORS\_PER\_PAGE 매크로를 정의했다. 하나의 페이지는 여러 섹터로 구성되므로 PGSIZE를 BLOCK\_SECTOR\_SIZE로 나눈 값을 사용한다.

전역 변수로 swap\_table을 선언했다. 이는 시스템 전체에서 하나만 존재하며 모든 프로세스가 공유한다.

```
#define SECTORS_PER_PAGE (PGSIZE / BLOCK_SECTOR_SIZE)

static struct swap_table swap_table;
```

swap\_init() 함수를 구현하여 Swap Table을 초기화한다.

```
void swap_init(void) {
    swap_table.swap_block = block_get_role(BLOCK_SWAP);
    block_sector_t swap_sectors = block_size(swap_table.swap_block);
    swap_table.swap_size = swap_sectors / SECTORS_PER_PAGE;
    swap_table.swap_bitmap = bitmap_create(swap_table.swap_size);
    bitmap_set_all(swap_table.swap_bitmap, false);
    lock_init(&swap_table.swap_lock);
}
```

block\_get\_role(BLOCK\_SWAP)로 Swap Disk를 얻는다. block\_size()로 Swap Disk의 전체 섹터 수를 구하고, SECTORS\_PER\_PAGE로 나누어 swap\_size를 계산한다. bitmap\_create()로 swap\_bitmap을 생성한다. bitmap\_set\_all()로 모든 비트를 false로 초기화하여 모든 슬롯을 사용 가능 상태로 만든다. lock\_init()으로 swap\_lock을 초기화한다.

swap\_out() 함수를 구현하여 Frame을 Swap Disk에 기록한다. ASSERT로 frame이 NULL이 아니고 페이지 경계에 정렬되어 있는지 확인한다. swap\_bitmap이나 swap\_block이 NULL이면 BITMAP\_ERROR를 반환한다. swap\_lock을 획득하고 bitmap\_scan\_and\_flip()으로 빈 슬롯을 찾는다. 슬롯 0을 예약해서 사용하도록 구현해서 1부터 스캔을 시작한다. 슬롯 번호에 SECTORS\_PER\_PAGE를 곱하여 시작 섹터를 계산한다. for 루프로 SECTORS\_PER\_PAGE만큼

반복하며 block\_write()로 각 섹터를 Swap Disk에 기록한다. frame 주소에 섹터 크기를 곱한 오프셋을 더해 각 섹터의 데이터를 전달한다. swap\_lock을 해제하고 슬롯 번호를 반환한다.

```
size_t swap_out(void *frame) {
    ASSERT(frame != NULL);
    ASSERT(pg_ofs(frame) == 0);

    if (swap_table.swap_bitmap == NULL || swap_table.swap_block == NULL) {
        return BITMAP_ERROR;
    }

    lock_acquire(&swap_table.swap_lock);

    // 빈 swap 슬롯 찾기, 0 대신 1부터 스캔 시작 (슬롯 0을 예약)
    size_t slot = bitmap_scan_and_flip(swap_table.swap_bitmap, 1, 1, false);

    block_sector_t sector = slot * SECTORS_PER_PAGE;
    // 페이지를 섹터 단위로 swap 디스크에 쓰기
    for (int i = 0; i < SECTORS_PER_PAGE; i++) {
        block_write(swap_table.swap_block,
                    sector + i,
                    frame + (i * BLOCK_SECTOR_SIZE));
    }

    lock_release(&swap_table.swap_lock);
    return slot;
}
```

swap\_in() 함수를 구현하여 Swap Disk에서 Frame으로 데이터를 읽어온다. ASSERT로 frame, 페이지 정렬, 슬롯 범위를 확인한다. swap\_lock을 획득하고, bitmap\_test()로 슬롯이 사용 중인지 확인한다. 사용 중이 아니라면 오류이므로 락을 해제하고 종료한다. for 루프로 block\_read()를 반복 호출하여 각 섹터를 Frame으로 읽어온다. bitmap\_set()으로 슬롯을 false로 설정하여 해제하여 swap in 후 자동으로 슬롯을 해제하여 다시 사용할 수 있도록 하고 swap\_lock을 해제한다.

```

void swap_in(size_t slot, void *frame) {
    ASSERT(frame != NULL);
    ASSERT(pg_ofs(frame) == 0);
    ASSERT(slot < swap_table.swap_size);

    lock_acquire(&swap_table.swap_lock);

    // 슬롯이 사용 중인지 확인
    if (!bitmap_test(swap_table.swap_bitmap, slot)) {
        lock_release(&swap_table.swap_lock);
        return;
    }

    block_sector_t sector = slot * SECTORS_PER_PAGE;

    // swap 디스크에서 페이지 읽기
    for (int i = 0; i < SECTORS_PER_PAGE; i++) {
        block_read(swap_table.swap_block,
                   sector + i,
                   frame + (i * BLOCK_SECTOR_SIZE));
    }

    bitmap_set(swap_table.swap_bitmap, slot, false);

    lock_release(&swap_table.swap_lock);
}

```

`swap_free()` 함수를 구현하여 Swap 슬롯을 해제한다. `ASSERT`로 슬롯 범위를 확인한다. `swap_lock`을 획득하고 `bitmap_test()`로 슬롯이 사용 중인지 확인한다. `bitmap_set()`으로 슬롯을 `false`로 설정하고 `swap_lock`을 해제한다.

```

void swap_free(size_t slot) {
    ASSERT(slot < swap_table.swap_size);
    lock_acquire(&swap_table.swap_lock);
    if (!bitmap_test(swap_table.swap_bitmap, slot)) {
        lock_release(&swap_table.swap_lock);
        return;
    }
    bitmap_set(swap_table.swap_bitmap, slot, false);
    lock_release(&swap_table.swap_lock);
}

```

#### 4. Stack Growth 관련 구현 (vm/stack.h, vm/stack.c)

vm/stack.h에 스택 관련 상수와 함수 프로토타입을 정의했다.

STACK\_MAX\_SIZE를 8MB로 정의하여 각 프로세스의 스택 크기를 제한한다.

```
#define STACK_MAX_SIZE (8 * 1024 * 1024) // 8MB

bool is_valid_stack_access(void *fault_addr, void *esp);
bool grow_stack(void *upage);
```

vm/stack.c에 is\_valid\_stack\_access() 함수를 구현했다.

첫 번째 조건으로 is\_user\_vaddr()로 fault\_addr가 PHYS\_BASE 아래에 있는지 확인한다. User 프로그램이 Kernel 주소 공간에 접근하려는 것은 Invalid Access이므로 false를 반환한다.

두 번째 조건으로 스택의 최대 크기를 검사한다. PHYS\_BASE - pg\_round\_down(fault\_addr)는 스택이 성장한 전체 크기를 나타낸다. 스택은 PHYS\_BASE에서 시작하여 낮은 주소로 성장하므로, 이 거리가 STACK\_MAX\_SIZE(8MB)를 초과하면 스택 크기 제한을 위반한 것이다.

세 번째 조건으로 fault\_addr가 esp에서 32 bytes 이내에 있는지 확인한다. PUSHA 명령어 실행 중에는 esp에 최대 32 bytes 떨어진 위치에 접근할 수 있으므로, 이는 유효한 스택 접근으로 판단해 true를 반환한다.

하나라도 만족하지 않으면 false를 반환하여 Invalid Memory Access임을 나타낸다.

```
bool is_valid_stack_access(void *fault_addr, void *esp) {
    // 1. 사용자 주소 공간인지 확인
    if (!is_user_vaddr(fault_addr)) {
        return false;
    }

    // 2. 최대 스택 크기 제한 확인 (8MB)
    if (PHYS_BASE - pg_round_down(fault_addr) > STACK_MAX_SIZE) {
        return false;
    }

    // 3. 스택 포인터 기준 유효성 검사 (PUSHA 명령어는 esp - 32까지 접근 가능)
    if ((char *)fault_addr >= (char *)esp - 32) {
        return true;
    }

    return false;
}
```

`grow_stack()` 함수를 구현했다. 먼저 `pg_round_down()`으로 `upage`를 페이지 경계에 맞춘다.

`spt_find()`로 해당 `upage`에 대한 PTE가 이미 존재하는지 확인한다. 존재하면 중복 매핑이므로 `false`를 반환한다. `spt_create_page()`로 새로운 PTE를 생성하고 Supplemental Page Table에 삽입한다. 실패하면 `NULL`을 반환하므로 `false`를 반환한다. 이 함수는 내부적으로 `malloc()`과 `spt_insert()`를 수행한다.

PTE의 `type`과 `original_type`을 `PAGE_STACK`으로 설정한다. 이는 스택 페이지임을 나타내며, 나중에 Swap Out될 때도 원래 타입을 유지할 수 있도록 한다. `writable`을 `true`로 설정하여 스택에 쓰기가 가능하도록 한다.

`get_frame()`으로 Physical Frame을 할당받는다. `PAL_USER`는 사용자 페이지임을 나타내고, `PAL_ZERO`는 Frame을 0으로 초기화하라는 플래그이다. 스택 페이지는 초기값이 0이어야 하므로 `PAL_ZERO`를 사용한다. 실패하면 `spt_remove_page()`로 이미 생성한 PTE를 제거하고 `false`를 반환한다.

`pagedir_set_page()`로 Page Table에 `upage`와 `frame`을 맵핑한다. 실패하면 `free_frame()`으로 Frame을 해제하고 `spt_remove_page()`로 PTE를 제거한 후 `false`를 반환한다.

모든 작업이 성공하면 `pte->kpage`를 `frame`으로 설정하고, `is_loaded`를 `true`로 변경한다. 이는 페이지가 메모리에 로드되었음을 나타낸다. `true`를 반환하여 스택 확장이 성공했음을 알린다.

```

bool grow_stack(void *upage) {
    struct thread *t = thread_current();

    upage = pg_round_down(upage);

    if (spt_find(&t->spt, upage) != NULL) {
        return false;
    }

    struct page_table_entry *pte = spt_create_page(&t->spt, upage);
    if (pte == NULL) {
        return false;
    }

    pte->type = PAGE_STACK;
    pte->original_type = PAGE_STACK;
    pte->writable = true;

    void *frame = get_frame(PAL_USER | PAL_ZERO, upage);
    if (frame == NULL) {
        spt_remove_page(&t->spt, upage);
        return false;
    }

    if (!pagedir_set_page(t->pagedir, upage, frame, true)) {
        free_frame(frame);
        spt_remove_page(&t->spt, upage);
        return false;
    }

    pte->kpage = frame;
    pte->is_loaded = true;

    return true;
}

```

## 5. Memory Mapped File 관련 구현 (vm/mmap.h, vm/mmap.c)

vm/mmap.h에 mmap\_entry 구조체를 정의했다.

```

struct mmap_entry {
    mapid_t mapid;
    int fd;
    struct file *file;
    void *addr;
    size_t length;
    size_t page_count;
    struct list_elem elem;
};

```

mapid는 매핑을 식별하는 고유 ID이고 fd는 원본 파일 디스크립터, file은 file\_reopen()으로 얻은 독립적인 파일 포인터이다. addr은 매핑 시작 Virtual Address, length는 파일 길이, page\_count는 매핑에 필요한 페이지 개수이다.

mmap\_insert() 함수를 구현하여 파일을 메모리에 매핑한다.

먼저 파일 길이로 필요한 페이지 개수를 계산한다. pg\_ofs()로 주소가 페이지 경계에 정렬되어 있는지 확인하고, is\_user\_vaddr()로 사용자 주소 공간인지 검증한다. 모든 페이지 범위를 순회하며 유효성을 검사한다. spt\_find()로 이미 SPT에 등록된 페이지가 있는지 검사하고, pagedir\_get\_page()로 Page Table에 매핑이 있는지 확인한다. Stack 영역과 겹치는지 검사한다. 하나라도 조건을 위반하면 file\_close()로 파일을 닫고 -1을 반환한다.

mmap\_entry를 malloc()으로 할당하고 필드를 초기화한다. mapid는 thread의 next\_mapid를 사용하고 증가시킨다. 각 페이지에 대해 PTE를 생성하여 SPT에 등록한다. 페이지별로 offset, read\_bytes, zero\_bytes를 계산한다. 마지막 페이지는 파일 끝까지만 읽고 나머지는 0으로 채운다. PTE의 type과 original\_type을 PAGE\_MMAP로 설정하고, file 포인터와 file\_offset, read\_bytes, zero\_bytes를 저장한다. mapid도 PTE에 저장하여 나중에 munmap 시 해당 매핑의 페이지를 찾을 수 있도록 한다. spt\_insert()로 PTE를 SPT에 등록하며, 실패하면 mmap\_cleanup\_on\_fail()로 이미 생성한 PTE들을 정리한다.

모든 PTE 생성이 완료되면 mmap\_entry를 thread의 mmap\_list에 추가하고 mapid를 반환한다.

```

struct mmap_entry *me = malloc(sizeof(struct mmap_entry));
if (me == NULL) {
    lock_acquire(&file_reopen);
    file_close(file_reopen);
    lock_release(&file_reopen);
    return -1;
}

me->mapid = cur->next_mapid++;
me->fd = fd;
me->file = file_reopen;
me->addr = addr;
me->length = file_length;
me->page_count = page_count;

// 각 페이지에 대한 PTE 생성
for (size_t i = 0; i < page_count; i++) {
    void *upage = addr + (i * PGSIZE);
    off_t offset = i * PGSIZE;
    size_t read_bytes = (offset + PGSIZE < file_length) ? PGSIZE : (file_length - offset);
    size_t zero_bytes = PGSIZE - read_bytes;

    struct page_table_entry *pte = malloc(sizeof(struct page_table_entry));
    if (pte == NULL) {
        mmap_cleanup_on_fail(i, addr, file_reopen);
        free(me);
        return -1;
    }

    memset(pte, 0, sizeof(*pte));
    pte->upage = upage;
    pte->pkgage = NULL;
    pte->is_loaded = false;
    pte->type = PAGE_MMAP;
    pte->original_type = PAGE_MMAP;
    pte->file = file_reopen;
    pte->file_offset = offset;
    pte->read_bytes = read_bytes;
    pte->zero_bytes = zero_bytes;
    pte->writable = writable;
    pte->mapid = me->mapid;
    pte->swap_slot = 0;

    if (spt_insert(&cur->spt, pte)) {
        free(pte);
        mmap_cleanup_on_fail(i, addr, file_reopen);
        free(me);
        return -1;
    }
}

list_push_back(&cur->mmap_list, &me->elem);
return me->mapid;
}

```

mmap\_munmap() 함수를 구현하여 매팅을 해제한다.

mmap\_find\_entry()로 해당 mapid를 가진 mmap\_entry를 찾는다. 찾지 못하면 바로 반환한다.

각 페이지를 순회하며 spt\_find()로 PTE를 찾는다. PTE가 존재하고 is\_loaded가 true이면 해당 페이지가 메모리에 로드되어 있는 것이다. pagedir\_is\_dirty()로 Dirty Bit를 확인하여 페이지가 수정되었으면 file\_reopen을 획득하고 file\_write\_at()으로 파일에 기록한다. 이때 read\_bytes만큼만 기록하여 zero\_bytes 영역은 기록하지 않는다.

pagedir\_clear\_page()로 Page Table 매팅을 제거하고, free\_frame()으로 Physical Frame을 해제한다. hash\_delete()로 SPT에서 PTE를 제거하고 free()로 메모리를 해제한다.

모든 페이지 정리가 완료되면 file\_reopen을 획득하고 file\_close()로 파일을 닫는다.

list\_remove()로 mmap\_entry를 mmap\_list에서 제거하고 free()한다.

```

void mmap_munmap(struct thread *t, mapid_t mapping) {
    struct mmap_entry *me = mmap_find_entry(t, mapping);

    if (me == NULL)
        return;

    // 각 페이지 정리
    for (size_t i = 0; i < me->page_count; i++) {
        void *upage = me->addr + (i * PGSIZE);
        struct page_table_entry *pte = spt_find(&t->spt, upage);

        if (pte == NULL)
            continue;

        // dirty 페이지를 파일에 write back
        if (pte->is_loaded && pte->kpage != NULL) {
            if (pagedir_is_dirty(t->pagedir, upage)) {
                lock_acquire(&filesys_lock);
                file_write_at(pte->file, pte->kpage, pte->read_bytes, pte->file_offset);
                lock_release(&filesys_lock);
            }
            pagedir_clear_page(t->pagedir, upage);
            free_frame(pte->kpage);
        }

        hash_delete(&t->spt, &pte->elem);

        free(pte);
    }

    lock_acquire(&filesys_lock);
    file_close(me->file);
    lock_release(&filesys_lock);

    list_remove(&me->elem);
    free(me);
}

```

mmap\_unmap\_all() 함수를 구현하여 프로세스 종료 시 모든 매핑을 해제한다. mmap\_list가 빌 때까지 list\_front()로 첫 번째 mmap\_entry를 가져와 mmap\_munmap()을 호출한다. 이 방식으로 모든 매핑을 순차적으로 정리한다.

```

void mmap_unmap_all(struct thread *t) {
    while (!list_empty(&t->mmap_list)) {
        struct list_elem *e = list_front(&t->mmap_list);
        struct mmap_entry *me = list_entry(e, struct mmap_entry, elem);
        mmap_munmap(t, me->mapid);
    }
}

```

mmap\_find\_entry() 함수는 mmap\_list를 순회하며 mapid가 일치하는 mmap\_entry를 찾아

반환한다. 없으면 NULL을 반환한다.

```
static struct mmap_entry *mmap_find_entry(struct thread *t, mapid_t mapping) {
    struct list_elem *e;
    for (e = list_begin(&t->mmap_list); e != list_end(&t->mmap_list); e = list_next(e)) {
        struct mmap_entry *temp = list_entry(e, struct mmap_entry, elem);
        if (temp->mapid == mapping) {
            return temp;
        }
    }
    return NULL;
}
```

mmap\_cleanup\_on\_fail() 함수는 mmap\_insert() 중 실패 시 이미 생성한 PTE들을 정리한다. count만큼 페이지를 순회하며 spt\_find()로 PTE를 찾고, spt\_remove()로 제거한다. fileys\_lock을 획득하고 file\_close()로 파일을 닫는다.

```
static void mmap_cleanup_on_fail(size_t count, void *addr, struct file *file) {
    struct thread *cur = thread_current();

    for (size_t i = 0; i < count; i++) {
        void *upage = addr + (i * PGSIZE);
        struct page_table_entry *pte = spt_find(&cur->spt, upage);

        if (pte != NULL) {
            spt_remove(&cur->spt, pte);
        }
    }

    lock_acquire(&fileys_lock);
    file_close(file);
    lock_release(&fileys_lock);
}
```

check\_mmap\_overlap() 함수를 구현하여 매팅 영역의 중복을 검사한다. addr부터 addr+length까지 페이지 단위로 순회하며 spt\_find()로 SPT에 등록된 페이지가 있는지, pagedir\_get\_page()로 매팅이 있는지, Stack 영역과 겹치는지 확인한다. 하나라도 조건을 만족하면 true를 반환하여 중복을 알린다.

```

bool check_mmap_overlap(void *addr, off_t length) {
    struct thread *t = thread_current();
    void *end_addr = addr + length;

    for (void *page_addr = addr; page_addr < end_addr; page_addr += PGSIZE) {
        if (spt_find(&t->spt, page_addr) != NULL) {
            return true;
        }

        if (pagedir_get_page(t->pagedir, page_addr) != NULL) {
            return true;
        }

        if (page_addr >= (void *)(PHYS_BASE - STACK_MAX_SIZE) && page_addr < PHYS_BASE) {
            return true;
        }
    }

    return false;
}

```

## 6. Demand Paging (threads/init.c, threads/thread.h, threads/thread.c, userprog/process.c)

먼저, threads/init.c 파일에 있는 pintos의 main 함수에서 frame\_init() 함수와 swap\_init() 함수를 호출하여 Frame Table과 Swap Table을 초기화하도록 했다.

```

#ifndef VM
    frame_init ();
    swap_init ();
#endif

    printf ("Boot complete.\n");

```

threads/thread.h에 있는 thread 구조체에 Virtual Memory 관련 필드를 추가했다. spt는 해당 프로세스의 Supplemental Page Table로, 모든 Virtual Page의 정보를 Hash Table로 관리한다. mmap\_list는 프로세스가 생성한 모든 Memory Mapped File을 추적한다. next\_mapid는 새로운 맵핑에 할당할 고유 ID를 생성한다. threads/thread.c의 init\_thread()에서 이 필드들을 초기화했다.

```

#define VM
    struct hash spt;
    struct list mmap_list;
    mapid_t next_mapid;
#endif

```

```

#define VM
    t->spt.buckets = NULL;
    list_init(&t->mmap_list);
    t->next_mapid = 0;
#endif

```

userprog/process.c에서 start\_process() 함수에서 spt\_init()을 호출하여 프로세스 시작 시 Supplemental Page Table을 초기화한다.

```

static void
start_process (void *file_name)
{
    char *file_name = file_name;
    struct intr_frame if_;
    bool success;
    struct thread *cur = thread_current();

#define VM
    spt_init(&cur->spt);
#endif

```

process\_exit()함수에서 프로세스 종료 시 Virtual Memory 관련 자원을 정리하도록 추가했다.

먼저 mmap\_unmap\_all()로 Memory Mapped File의 변경 사항을 파일에 Write Back하고 파일을 닫는다. 그다음 frame\_clear\_owner()로 해당 프로세스가 사용하던 모든 Physical Frame을 해제하고 Page Table 매팅도 제거한다. 마지막으로 spt\_destroy()로 Supplemental Page Table의 메타데이터를 정리한다.

```

#define VM
    // MMAP 정리 (파일 write back 및 파일 닫기)
    mmap_unmap_all(cur);

    // Frame 정리 (물리 메모리 해제)
    frame_clear_owner(cur);

    // SPT 정리 (메타데이터만 정리, frame은 이미 해제됨)
    spt_destroy(&cur->spt);
#endif

```

`load_segment()` 함수를 수정하여 Demand Paging을 구현했다. 기존 코드는 `file_read()`로 파일 내용을 즉시 메모리에 로드했으나, 이를 제거하고 각 페이지에 대한 PTE만 생성하도록 변경했다. 각 페이지에 대해 PTE를 `malloc()`으로 할당하고 `memset()`으로 0으로 초기화한다. `upage`는 Virtual Page 주소, `kpage`는 NULL로 설정하여 아직 Physical Frame이 할당되지 않았음을 나타낸다. `is_loaded`는 `false`로 설정한다. `type`과 `original_type`은 `PAGE_BINARY`로 설정하여 실행 파일에서 로드된 페이지임을 표시한다.

`file_reopen()`으로 독립적인 파일 포인터를 얻는다. 이는 동일한 파일을 여러 프로세스가 동시에 사용할 때 `file_seek()` 위치가 섞이지 않도록 보장한다. `filesys_lock`을 획득하여 동기화를 보장한다. `file_offset`은 파일 내 시작 위치, `read_bytes`는 파일에서 읽을 바이트 수, `zero_bytes`는 0으로 채울 바이트 수를 저장한다. `writable`은 페이지의 쓰기 권한을 나타낸다.

`spt_insert()`로 PTE를 Supplemental Page Table에 등록한다. 실패하면 할당받은 자원을 정리하고 `false`를 반환한다. 성공하면 다음 페이지로 이동한다. 이 방식으로 프로그램 로드 시 메모리 사용량을 최소화하고, 실제로 필요한 페이지만 Page Fault 시 로드한다.

```

load_segment (struct file *file, off_t ofs, uint8_t *upage,
             uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
    ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
    ASSERT (pg_ofs (upage) == 0);
    ASSERT (ofs % PGSIZE == 0);

    while (read_bytes > 0 || zero_bytes > 0)
    {
        /* Calculate how to fill this page.
         * We will read PAGE_READ_BYTES bytes from FILE
         * and zero the final PAGE_ZERO_BYTES bytes. */
        size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
        size_t page_zero_bytes = PGSIZE - page_read_bytes;

        struct page_table_entry *pte = malloc(sizeof(struct page_table_entry));
        if (pte == NULL)
            return false;

        memset(pte, 0, sizeof(*pte));
        pte->upage = upage;
        pte->kpage = NULL;
        pte->is_loaded = false;
        pte->type = PAGE_BINARY;
        pte->original_type = PAGE_BINARY;

        lock_acquire(&filesys_lock);
        pte->file = file_reopen(file);
        lock_release(&filesys_lock);
        if (pte->file == NULL) {
            free(pte);
            return false;
        }

        pte->file_offset = ofs;
        pte->read_bytes = page_read_bytes;
        pte->zero_bytes = page_zero_bytes;
        pte->writable = writable;

        pte->swap_slot = 0;

        if (!spt_insert(&thread_current()->spt, pte)) {
            lock_acquire(&filesys_lock);
            file_close(pte->file);
            lock_release(&filesys_lock);
            free(pte);
            return false;
        }

        /* Advance. */
        read_bytes -= page_read_bytes;
        zero_bytes -= page_zero_bytes;
        upage += PGSIZE;
        ofs += page_read_bytes;
    }
    return true;
}

```

초기 구현에서 threads/thread.c의 init\_thread() 함수 내에서 spt\_init() 함수를 호출하여 Supplemental Page Table을 초기화했다. 그러나 프로그램 실행 시 assertion ‘t->status == THREAD\_RUNNING’ failed 메시지와 함께 커널 패닉이 발생했다. 원인 확인 결과 hash\_init 내부의 malloc() lock\_acquire를 호출하고, 이 함수가 내부적으로 thread\_current를 호출하면서 문제가 발생했다. init\_thread가 실행되는 시점에서 thread의 상태가 THREAD\_BLOCKED이므로 thread\_current의 assertion이 실패한 것이었다. 이를 해결하기 위해 Supplemental Page Table 초기화를 userprog/process.c의 start\_process 함수로 이동시켰다. start\_process는 thread가 실제로 실행되어 상태가 THREAD\_RUNNING으로 변경된 후 호출되므로 문제가 해결되었다.

```
printf("[DEBUG] init_thread: before VM init\n");

#ifndef VM
printf("[DEBUG] init_thread: calling hash_init\n");
spt_init(&t->spt);
printf("[DEBUG] init_thread: hash_init done\n");
#endif
```

```
Kernel command line: -q -f extract run args-none
[DEBUG] init_thread: before VM init
[DEBUG] init_thread: calling hash_init
Kernel PANIC at ../../threads/thread.c:311 in thread_current(): assertion `t->status == THREAD_RUNNING' failed.
Call stack: 0xc002a387.
```

## 7. Page Fault Handler (userprog/exception.c)

userprog/exception.c의 page\_fault() 함수를 수정하여 Page Fault 처리를 구현했다.

is\_user\_vaddr()로 fault\_addr가 사용자 주소 공간에 있는지 확인한다. pg\_round\_down()으로 fault\_page를 페이지 경계로 정렬한다. not\_present가 true이면 페이지가 메모리에 없는 경우이다. spt\_find()로 해당 Virtual Page의 PTE를 Supplemental Page Table에서 찾는다.

PTE가 존재하면 이미 등록된 페이지이므로 handle\_mm\_fault()를 호출하여 Demand Paging을 수행한다. 성공하면 return하여 Faulting Instruction을 재실행한다.

PTE가 존재하지 않으면 Stack Growth가 필요한 상황일 수 있다. user가 true인지 확인하고, is\_valid\_stack\_access()로 유효한 Stack 접근인지 확인한다. 조건을 만족하면 grow\_stack()으로 Stack을 확장하고 성공하면 return한다.

모든 처리가 실패하면 Invalid Memory Access이다. user가 false인 경우는 exit(-1)로 종료한다.

```

// 사용자 주소 공간의 페이지 플트 처리
if (is_user_vaddr(fault_addr)) {
    void *fault_page = pg_round_down(fault_addr);
    struct thread *t = thread_current();

    if (not_present) {
        // Not-present 페이지 플트
        struct page_table_entry *pte = spt_find(&t->spt, fault_page);

        if (pte != NULL) {
            // 이미 매핑된 페이지: demand paging
            if (handle_mm_fault(pte, write)) {
                return;
            }
        } else {
            // 매핑되지 않은 페이지: 스택 확장 시도
            if (user && is_valid_stack_access(fault_addr, f->esp)) {
                if (grow_stack(fault_page)) {
                    return;
                }
            }
        }
    }

    // 커널 모드에서의 페이지 플트
    if (!user) {
        exit(-1);
    }
}

```

handle\_mm\_fault() 함수를 구현하여 페이지를 실제로 메모리에 로드한다.

먼저 write 접근인데 pte->writable이 false이면 권한 위반이므로 false를 반환한다. is\_loaded가 이미 true이면 페이지가 메모리에 있으므로 true를 반환한다.

get\_frame()으로 Physical Frame을 할당받는다. PAL\_USER 플래그로 사용자 페이지임을 표시하고, upage를 전달하여 Frame Table Entry에 저장한다. 실패하면 false를 반환한다.

PTE의 type에 따라 switch문으로 분기하여 적절한 처리를 수행한다.

PAGE\_BINARY는 실행 파일에서 로드된 페이지이므로, file\_seek()로 파일 위치를 설정하고 file\_read()로 read\_bytes만큼 읽어온다. bytes\_read가 read\_bytes와 다르면 파일 읽기에 실패한 것이므로 success를 false로 설정한다. 성공하면 memset()으로 나머지 zero\_bytes를 0으로 채운다.

PAGE\_SWAP은 Swap Disk에서 가져와야 하는 페이지이다. swap\_slot이 0이면 오류이므로 false를 설정한다. swap\_in()으로 Swap Disk에서 frame으로 데이터를 읽어온다. swap\_slot을 0으로 초기화하여 해제하고, type을 original\_type으로 복원한다. 이는 Swap Out 전의 원래 타입(PAGE\_BINARY, PAGE\_STACK 등)을 나타낸다.

PAGE\_STACK은 Stack 페이지이므로, memset()으로 전체를 0으로 초기화한다.

PAGE\_MMAP는 Memory Mapped File 페이지이므로, PAGE\_BINARY와 동일하게 file\_seek()와 file\_read()로 파일에서 데이터를 읽어오고 zero\_bytes를 0으로 채운다.

데이터 로드가 실패하면 free\_frame()으로 Frame을 해제하고 false를 반환한다. 성공하면 pagedir\_set\_page()로 Page Table에 매팅한다. 실패하면 Frame을 해제하고 false를 반환한다.

모든 작업이 성공하면 pte->kpage를 frame으로 설정하고, is\_loaded를 true로 변경한다. true를 반환하여 Page Fault 처리가 성공했음을 알린다.

```
static bool
handle_mm_fault (struct page_table_entry *pte, bool write)
{
    if (write && !pte->writable) {
        return false;
    }

    if (pte->is_loaded) {
        return true;
    }

    void *frame = get_frame(PAL_USER, pte->upage);
    if (frame == NULL) {
        return false;
    }

    bool success = true;
    switch (pte->type) {
        case PAGE_BINARY:
            file_seek(pte->file, pte->file_offset);
            int bytes_read = file_read(pte->file, frame, pte->read_bytes);
            if (bytes_read != (int)pte->read_bytes) {
                success = false;
            } else {
                memset(frame + pte->read_bytes, 0, pte->zero_bytes);
            }
            break;

        case PAGE_SWAP:
            if (pte->swap_slot == 0) {
                success = false;
                break;
            }
            swap_in(pte->swap_slot, frame);
            pte->swap_slot = 0;
            pte->type = pte->original_type;
            break;
    }
}
```

```

    case PAGE_STACK:
        memset(frame, 0, PGSIZE);
        break;

    case PAGE_MMAP:
        file_seek(pte->file, pte->file_offset);
        bytes_read = file_read(pte->file, frame, pte->read_bytes);
        if (bytes_read != (int)pte->read_bytes) {
            success = false;
        } else {
            memset(frame + pte->read_bytes, 0, pte->zero_bytes);
        }
        break;

    default:
        success = false;
        break;
    }

    if (!success) {
        free_frame(frame);
        return false;
    }

    if (!pagedir_set_page(thread_current->pagedir, pte->upage, frame, pte->writable)) {
        free_frame(frame);
        return false;
    }

    pte->kpage = frame;
    pte->is_loaded = true;
}

return true;
}

```

## 8. Memory Mapped File 관련 System Calls (userprog/syscall.h, userprog/syscall.c)

userprog/syscall.h에 sys\_mmap(), sys\_munmap() 함수 프로토타입을 선언했다.

```

mapid_t sys_mmap(int fd, void *addr);
void sys_munmap(mapid_t mapping);

```

userprog/syscall.c에서 syscall\_handler() 함수에 SYS\_MMAP과 SYS\_MUNMAP 케이스를 추가했다. check\_valid\_uaddr()로 각각 인자의 유효성을 검증한다. SYS\_MMAP는 반환값을 f->eax에 저장하고, SYS\_MUNMAP은 반환값이 없다.

```
case SYS_MMAP:
    check_valid_uaddr(esp + 1);
    check_valid_uaddr(esp + 2);
    f->eax = sys_mmap(*(esp + 1), (void *) *(esp + 2));
    break;

case SYS_MUNMAP:
    check_valid_uaddr(esp + 1);
    sys_munmap(*esp + 1);
    break;
```

다음과 같이 sys\_mmap() 함수를 구현했다.

먼저 인자의 유효성을 검사한다. fd가 0이거나 1이면 매핑할 수 없으므로 -1을 반환한다. addr이 NULL이거나 pg\_ofs(addr)가 0이 아니면 페이지 경계에 정렬되지 않은 것이므로 -1을 반환한다. is\_user\_vaddr()로 사용자 주소 공간인지 확인한다.

get\_file()로 fd에 해당하는 파일 포인터를 가져온다. NULL이면 유효하지 않은 fd이므로 -1을 반환한다. filesystem\_lock을 획득하고 file\_length()로 파일 크기를 얻는다. file\_len이 0이면 빈 파일이므로 매핑할 수 없어 -1을 반환한다.

check mmap\_overlap()으로 addr부터 addr+file\_len까지의 영역이 기존 페이지와 겹치는지 확인한다. 겹치며 -1을 반환한다. 이는 중복 매핑을 방지한다.

filesystem\_lock을 획득하고 file\_reopen()으로 독립적인 파일 포인터를 얻는다. 이는 동일한 파일을 여러 번 매핑하거나 여러 프로세스가 동시에 사용할 때 file\_seek() 위치가 섞이지 않도록 보장한다. reopened\_file이 NULL이면 실패하므로 -1을 반환한다.

Memory Mapped File에서 writable은 true로 설정한다. mmap\_insert()를 호출하여 실제 매핑을 수행하고, 반환된 mapid를 반환한다. 실패하면 mmap\_insert()가 -1을 반환한다.

```

mapid_t sys_mmap(int fd, void *addr) {
    if (fd == 0 || fd == 1 || addr == 0 || pg_ofs(addr) != 0 || !is_user_vaddr(addr)) {
        return -1;
    }

    struct file *file = get_file(fd);
    if (file == NULL) {
        return -1;
    }

    lock_acquire(&filesys_lock);
    off_t file_len = file_length(file);
    lock_release(&filesys_lock);
    if (file_len == 0) {
        return -1;
    }

    if (check mmap_overlap(addr, file_len)) {
        return -1;
    }

    lock_acquire(&filesys_lock);
    struct file *reopened_file = file_reopen(file);
    lock_release(&filesys_lock);
    if (reopened_file == NULL) {
        return -1;
    }

    bool writable = true;
    mapid_t mapid = mmap_insert(reopened_file, fd, addr, file_len, writable);
    return mapid;
}

```

sys\_munmap() 함수를 구현했다. 단순히 mmap\_munmap()을 호출하여 매핑을 해제한다. mmap\_munmap() 내부에서 변경된 페이지를 파일에 Write Back하고 Frame을 해제하며 PTE를 정리하고 파일을 닫는 모든 작업을 수행한다. mapping이 유효하지 않으면 mmap\_munmap()이 아무 작업도 수행하지 않고 반환한다.

```

void sys_munmap(mapid_t mapping) {
    mmap_munmap(thread_current(), mapping);
}

```

check\_valid\_uaddr() 함수를 수정하여 Virtual Memory 시스템에 맞게 주소 검증을 수행하도록 했다. 먼저 uaddr이 NULL이거나 is\_user\_vaddr()로 확인했을 때 사용자 주소 공간이 아니면 exit(-1)로 종료한다. pagedir\_get\_page()로 Page Table에서 해당 페이지의 매핑을 확인한다. NULL이 아니면 이미 메모리에 로드된 페이지이므로 유효하다고 판단하여 return한다. Page Table에 없으면 spt\_find()로 Supplemental Page Table에서 PTE를 찾는다. PTE가 존재하면 해당 페이지는 유효하지만 아직 로드되지 않은 페이지이므로 유효하다고 판단하여 return한다.

이는 Demand Paging을 지원하기 위한 설정이다. Page Table에도 SPT에도 없으면 유효하지 않은 주소이므로 exit(-1)로 프로세스를 종료한다.

```
void
check_valid_uaddr (const void *uaddr)
{
    if (uaddr == NULL || !is_user_vaddr (uaddr)) {
        | exit (-1);
    }

    struct thread *t = thread_current();
    void *page = pg_round_down(uaddr);

    // 이미 로드된 페이지
    if (pagedir_get_page(t->pagedir, page) != NULL) {
        | return;
    }

    // SPT에 있는 페이지
    struct page_table_entry *pte = spt_find(&t->spt, page);
    if (pte != NULL) {
        | return;
    }

    exit (-1);
}
```

추가적으로 preload\_buffer\_write() 함수를 구현하여 buffer의 모든 페이지를 미리 로드한다. buffer가 NULL이거나 size가 0이면 아무 작업도 수행하지 않고 return한다. buffer의 시작 페이지부터 끝 페이지까지 PGSIZE 단위로 순회하고 각 페이지에 대해 spt\_find()로 PTE를 찾는다. PTE가 존재하고 writable이 false면 해당 페이지는 읽기 전용이다. write 시스템 콜에서 읽기 전용 페이지에 쓰기를 시도하면 안되므로 exit(-1)로 종료한다. volatile unit8\_t dummy = \*(unit8\_t\*) upage로 각 페이지에 읽기 접근을 수행한다. 페이지가 아직 로드되지 않았으면 Page Fault가 발생하고 Page Fault Handler가 페이지를 메모리에 로드한다. volatile 키워드는 컴파일러가 이 읽기를 최적화로 제거하지 못하도록 방지한다. (void)dummy는 사용하지 않는 변수 경고를 제거하려고 추가했다. read()와 write() 함수에서 preload\_buffer\_write()를 호출하도록 해서 유저 모드에서 버퍼의 모든 페이지를 미리 로드하도록 수정했다.

```
static void preload_buffer_write(void *buffer, unsigned size) {
    if (buffer == NULL || size == 0) {
        return;
    }

    struct thread *t = thread_current();

    for (void *upage = pg_round_down(buffer);
         upage <= pg_round_down(buffer + size - 1);
         upage += PGSIZE) {

        // SPT 확인
        struct page_table_entry *pte = spt_find(&t->spt, upage);

        // writable이 아니면 exit
        if (pte != NULL && !pte->writable) {
            exit(-1);
        }

        // 읽기만으로 페이지 로드
        volatile uint8_t dummy = *(uint8_t*)upage;
        (void)dummy;
    }
}
```

## C. 시험 및 평가 내용

다음은 src/vm에서 ‘make check’를 실행한 결과로, 이번 프로젝트의 평가 기준에 해당하는 모든 테스트 케이스에 대해 pass한 것을 확인할 수 있다.

```
pass tests/vm/pt-grow-stack
pass tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
pass tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
pass tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
pass tests/vm/page-merge-stk
pass tests/vm/page-merge-mm
pass tests/vm/page-shuffle
pass tests/vm/mmap-read
pass tests/vm/mmap-close
pass tests/vm/mmap-unmap
pass tests/vm/mmap-overlap
pass tests/vm/mmap-twice
pass tests/vm/mmap-write
pass tests/vm/mmap-exit
pass tests/vm/mmap-shuffle
pass tests/vm/mmap-bad-fd
pass tests/vm/mmap-clean
pass tests/vm/mmap-inherit
pass tests/vm/mmap-misalign
pass tests/vm/mmap-null
pass tests/vm/mmap-over-code
pass tests/vm/mmap-over-data
pass tests/vm/mmap-over-stk
pass tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 113 tests passed.
make[1]: Leaving directory '/sogang/under/cse20211589/pintos/src/vm/build'
cse20211589@cspro5:~/pintos/src/vm$
```