**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재 교수님

이름 / 학번 : 김동현 / 20190345

개발 기간 : 2023.11.20 - 2023.12.8

1. **개발 목표**

코드, 데이터, 스택 영역을 물리메모리에 모두 올리는 것이 아닌, 가상 페이지를 만들어 물리 메모리와 매핑하여 관리하도록 한다. 필요할 때마다 page fault를 발생시켜 page fault handler가 메모리를 할당해주도록 한다. 디스크에서 물리 메모리로 데이터를 로드할 때 물리 메모리가 가득 찰 경우 second chance algorithm을 적용하여 swap을 실시한다. 또한 스택 크기를 초과하는 주소로 접근할 경우, 유효한 스택 접근에 대해 스택을 확장하도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
   2. Page Table & Page Fault Handler  
      기존의 pintos 프로그램은 valid address를 확인하는 과정에서 유효하지 않다면 page\_fualt( )에서 sys\_exit(-1)을 호출하여 종료하였다. page table과 page fault handler를 구현하여 demand paging 기법을 구현한다. 모든 데이터를 메모리에 로드하는 것이 아닌 page table로 관리하여 page를 필요할 때마다 로드하여 가상 페이지와 물리 페이지를 매핑하여 불필요한 물리 메모리 점유를 예방한다. page fualt가 발생할 경우 유효한 주소인지 확인 후, 유효한 주소라면 디스크에 존재하지만 메모리에 로드 되지 않은 경우이므로, 데이터를 메모리에 로드한 뒤, 가상 메모리와 매핑하도록 해 사용자가 해당 주소에 접근할 수 있도록 한다. 이러한 기법을 통해 모든 데이터를 메모리에 로드하지 않고, 필요한 페이지만 로드하여 효율적인 메모리 관리를 가능토록 한다.
   3. Disk Swap  
      물리 메모리를 할당하는 과정에서 물리 메모리가 가득 찰 경우 디스크와 swapping을 하여 물리 메모리 공간을 확보한다. second chance algorithm을 적용한 LRU 기법을 이용하여 물리메모리에서 swap out할 물리 페이지를 선정한다. swap 영역에 존재하는 페이지를 다시 접근하고자 할 경우 swap in하여 해당 물리 페이지를 다시 물리 메모리 영역에 로드한다. 이러한 기법을 통해 실제 물리 메모리의 크기보다 더 큰 가상 메모리를 구성하여 매핑할 수 있다.
   4. Stack Growth  
      스택을 확장 가능하도록 한다. 만약 현재 스택 포인터에서 휴리스틱하게 설정된 범위 내로 접근하도록 한다면, 스택을 확장하여 해당 영역에 접근할 수 있도록 한다.
   5. **개발 내용**
   6. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술  
      기존에는 모든 데이터를 메모리에 로드하였기 때문에 유효한 주소에 접근하고자 한다면 페이지 폴트가 발생하지 않고 가상 주소에 접근하여 물리 주소로 접근한다. 하지만 demand paging을 구현하게 되면서 모든 데이터를 메모리에 로드하지 않고, vm\_entry에 page entry를 생성하여 page table에 저장하게 된다. 그 결과 메모리에 아직 로드되지 않은 주소로 접근하고자 하면 페이지 폴트가 발생한다. 페이지 폴트가 발생한다면 페이지 폴트 핸들러가 호출되고, 해당 주소가 유효한 주소 접근인지, 해당 주소에 대한 가상 페이지가 존재하는지 확인한다. 만약 유효한 주소로의 접근이고 가상 페이지가 존재한다면 해당 주소를 매핑하는 페이지가 물리 메모리에 로드 되지 않은 상태이므로 페이지 타입에 따라 파일을 메모리에 로드한 뒤, 가상 페이지와 물리 페이지를 매핑한다. 이후 다시 해당 주소에 접근하고자 한다면, 유효한 주소가 메모리에 로드되어 있는 상황이므로 해당 주소에 정상적으로 접근할 수 있다.
   7. Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술  
      alloc\_page가 호출될 경우 물리메모리의 빈 영역에 메모리를 할당하여 반환한다. 만약 해당 함수가 NULL을 반환할 경우 물리 메모리가 가득찬 상태임을 의미한다. 이럴 경우 page replace algorithm을 사용하여 물리 메모리에 공간을 할당해준다. 페이지 교체 알고리즘은 LRU 알고리즘 중 second chance algorithm을 이용하였다. 해당 알고리즘은 물리 메모리를 차지하는 page를 리스트 형태로 관리하고 있으며, eviction 대상을 가리키는 clock hands가 선언되어 있다. 시계 바늘이 가리키고 있는 페이지의 access bit를 확인한다. 만약 해당 페이지가 최근에 접근하였다면 해당 비트가 1로 설정되어 있다. 해당 경우에는 한 번 더 기회를 주어 바로 쫓아내지 않고, access bit를 1로 설정한다. 만약 이미 기회를 받은 페이지라면 access bit가 0일 것이다. 그럴 경우 해당 페이지를 free 시켜 주어 물리 메모리 영역을 확보한다. 이때 쫓아내는 페이지의 dirty bit를 확인하여 데이터의 수정 여부를 확인한다. 만약 dirty bit가 1로 설정되어 있다면, 해당 페이지의 데이터가 수정되었음을 의미하므로 디스크에 해당 페이지를 write하여 동기화 시켜준 뒤, free 시켜준다. 만약 VM\_ANON의 경우 해당 페이지를 free 시키는 것이 아닌 스왑 영역으로 swap out 시켜준다. 이렇게 확보된 물리 메모리 영역에 기존에 실패했던 alloc\_page가 물리 메모리를 할당하여 반환하도록 한다.
   8. Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술  
      스택 범위를 넘어서는 주소에 접근하고자 한다면 스택 확장 여부를 확인하여 유효하다면 스택을 확장시킨다. 스택 확장 여부를 판단하는 방법은 다음과 같다. 우선 스택 확장이 스택의 최대 크기인 8MB를 넘지 않도록 확인한다. 혹은 접근하고자 하는 주소와 스택 포인트의 주소가 32바이트를 넘지 않도록 한다.
2. **추진 일정 및 개발 방법**
   1. **추진 일정**

2023.11.21 - 2023.11.24 : pintos.manual 정리

2023.11.25 - 2023.11.29 : paging 구현

2023.11.30 - 2023.12.2 : page fault handler 구현

2023.12.3 - 2023.12.5 : swapping 구현

2023.12.6 - 2023.12.6 : stack growth 구현

2023.12.7 - 2023.12.8 : 디버깅 및 보고서 작성

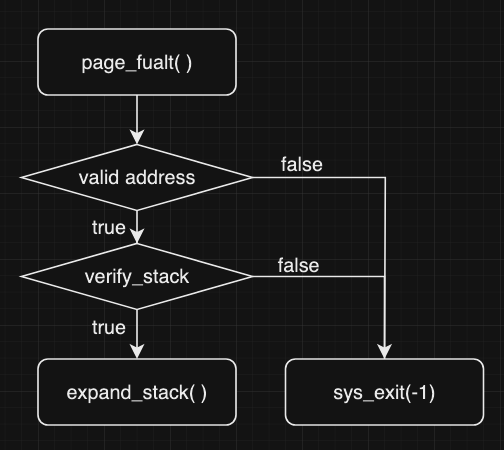
* 1. **개발 방법**
  + 수정해야하는 소스코드  
    1. userprog/exception.c  
     - page\_fault( ) : 기존에는 유효하지 않은 주소가 들어왔다면 바로 kill하였다면, page fault가 발생한 주소값의 유효성을 판단한 뒤, 해당 주소의 vm\_entry를 찾는다. 만약 vm\_entry가 존재한다면 handle\_mm\_fault( ) 함수를 호출하여 해당 데이터를 물리 메모리로 로드하고, 그렇지 않다면 스택을 확장할 수 있는지 판단하여 스택을 확장하도록 함.  
    2. userprog/process.c  
     - start\_process( ) : 생성된 스레드의 page table을 초기화 하는 함수를 호출하도록 함.  
     - process\_exit( ) : 프로세스가 종료되기 전, munmap\_all 함수를 호출하여 매핑된 파일을 저장하고 있는 리스트를 destroy시켜 해제한다. 또한 페이지 정보를 저장하는 hash table을 destroy하는 함수를 호출한다.  
     - load\_segment( ) : 모든 데이터를 물리 메모리에 올리도록 하는 기존 코드를 삭제하고, vm\_entry를 생성하여 필요한 정보로 초기화한다. 해당 페이지의 타입, 주소, 작성 가능 여부, 로드 여부, 파일 포인터, 오프셋, read &. zero byte로 구성된 vm\_entry를 스레드의 page\_table에 삽입한다. 한 페이지에 모든 정보를 매핑할 수 없을 경우를 대비하여 read\_bytes, zero\_bytes, ofs, upage를 업데이트 하도록 한다.  
     - setup\_stack( ) : 물리 메모리 주소값을 페이지를 가리키도록 변경하고 스택 영역에 대한 vm\_entry를 할당하고 초기화하여 page\_table에 삽입한다.  
    3. userprog/syscall.c  
     - syscall\_handler( ) : 새롭게 구현된 mmap, munmap 시스템 콜을 호출하도록 switch 문에 추가한다. 문자열을 인자로 사용하는 시스템 콜은 해당 인자를 check\_valid\_string을 통해 검사하도록 하고, 버퍼를 인자로 사용하는 시스템 콜은 해당 인자를 check\_valid\_buffer를 사용하여 검사하도록 한다.  
     - sys\_create( ) : filesys\_create 함수를 시스템 콜에서 사용하는 락으로 감싸준다.   
     - sys\_remove( ) : filesys\_remove 함수를 시스템 콜에서 사용하는 락으로 감싸준다.  
    4. threads/thread.c  
     - thread\_create( ) : thread\_yield( ) 함수가 USERPROG 영역에서만 실행되도록 감싸준다.  
     - init\_thread( ) : 스레드 구조체에 정의되어 있는 자료구조를 초기화한다.  
    5. threads/synch.c  
     - sema\_up( ) : thread\_yield( ) 함수가 USERPROG 영역에서만 실행되도록 감싸준다.  
    6. threads/init.c  
     - main( ) : 핀토스가 처음 실행될 때, 사용할 자료구조를 초기화하는데, lru\_list\_init과 swap\_init 함수를 호출하여 필요한 자료구조를 초기화 한다.
  + 수정하거나 추가해야 하는 자료구조  
    1. vm/frame.h  
     - struct lru\_list : 물리메모리에 올라와 있는 모든 물리 페이지를 관리하기 위한 리스트이다.  
     - struct lock lru\_list\_lock : 동기화를 위한 lock 자료구조이다.  
     - struct list\_elem \*lru\_clock : second chance algorithm 적용시 eviction 후보를 가리키는 list\_elem 자료구조이다.   
    2. vm/page.h  
     - Macro 정의 : 데이터를 어디서 저장하는지를 나타내는 메크로 함수로 바이너리 파일, 매핑된 파일, 스왑 영역으로 나뉜다.  
     - struct vm\_entry : 페이지 영역 마다 할당되는 구조체로, 페이지의 정보를 담고 있으며, 프로그램이 로드될 때 생성되는 자료구조이다.  
     - struct mmap\_file : mmap과 munmap 시스템 콜을 위한 자료구조로 파일과 vm\_entry의 매핑 정보를 저장하고 있는 자료구조이다.   
     - struct page : 물리메모리와 가상 페이지를 매핑 정보는 저장하고 있는 자료구조이다.  
    3. vm/swap.h  
     - struct lock swap\_lock : swapping을 진행하는 동안 동기화를 위한 선언된 lock 자료구조이다.  
     - struct block \*swap\_block : swap 영역을 가리키는 포인터이다.   
     - uint8\_t \*swap\_array : swap 영역이 할당되어 있는지 비어있는지를 나타내는 배열 자료구조이다. 1이면 해당 영역이 할당되어 있음을, 0이면 비어있음을 의미한다.  
    4. thread/thread.h  
     - struct thread : 스레드가 실행하는 프로세스의 데이터를 매핑하는 vm\_entry를 해시 테이블 형태로 관리하는 해시 자료구조와 mmap\_file 구조체를 저장하기 위한 list 자료구조, 매핑된 mmap\_file의 개수를 저장하고 있는 cur\_mapid를 선언한다.
  + 수정하거나 추가해야 하는 함수  
    1. vm/frame.h  
     - 새롭게 추가된 함수의 프로토 타입을 정의한다.   
    2. vm/frame.c  
     - lru\_list\_init( ) : LRU 알고리즘과 관련된 자료구조를 초기화 한다.  
     - add\_page\_to\_lru\_list( ) : 인자로 들어온 페이지를 lru\_list에 삽입한다.  
     - del\_page\_from\_lru\_list( ) : 인자로 들어온 페이지를 lru\_list에서 제거한다.  
     - alloc\_page( ) : 물리메모리 영역에 페이지를 할당하는 함수로, 물리 메모리가 가득 차있다면 evict\_page\_lru\_policy 함수를 통해 물리 메모리에 공간을 확보한 뒤, 다시 페이지를 할당 받는다. 이후 해당 주소와 스레드 정보를 page 구조체에 할당하고 lru\_list에 추가한다. 할당된 page를 리턴한다.  
     - free\_page( ) : lru\_list를 순회하며 인자로 들어온 물리 주소에 해당하는 페이지를 탐색한다. 해당 페이지를 찾는다면 해당 페이지를 반환하고, lru\_list에서 제거한다.  
     - get\_lru\_clock( ) : second chance algorithm에서 사용한 eviction 후보가 될 페이지를 찾아서 반환한다.  
     - evict\_page\_lru\_policy( ) : seconde chance algorithm에 따라 쫓아낼 vm\_entry를 결정하고 쫓아내 물리 메모리 영역을 확보한다. access bit가 1이라면 해당 비트를 0으로 변경하여 기회를 한 번 더 주고, 그렇지 않다면 해당 vm\_entry를 쫓아낼 페이지로 정한다. 이후 해당 vm\_entry의 타입에 따라 VM\_ANON이라면 swap\_out을 이외의 타입에 대해서는 물리 메모리에서 제거하도록 한다. 이때 해당 데이터가 변경되었는지 여부를 dirty bit를 통해 확인하고, 변경이 발생했다면 디스크에 작성해준다. 이후 해당 vm\_entry의 로드 여부를 변경하고 해당 페이지를 반환한다.  
    3. vm/page.h  
     - 새롭게 추가된 함수의 프로토 타입을 정의한다.  
    4. vm/page.c  
     - init\_vm\_page\_table ( ) : vm\_entry를 저장할 해시 테이블을 초기화시킨다.  
     - vm\_page\_table\_hash\_func( ) : init\_vm\_page\_table함수에서 hash\_init 함수를 호출할 때 전달하는 함수로서 해시 테이블의 키 값을 vm\_entry의 vaddr로 설정한다.  
     - vm\_page\_table\_less\_func( ) : init\_vm\_page\_table 함수에서 hash\_init 함수를 호출할 때 전달하는 함수로서 두 해시 원소가 있을 때 두 해시값의 대소를 비교한 결과를 반환한다.  
     - insert\_vm\_page\_table( ) : 해시 테이블에 인자로 들어온 vm\_entry를 삽입한다. 만약 이미 존재한다면 false를, 유일한 값이라면 true를 리턴하도록 한다.  
     - delete\_vm\_page\_table( ) : 해시 테이블에서 인자로 들어온 vm\_entry를 제거한다. 만약 존재하지 않는다면 false를, 제거에 성공했다면 true를 리턴하도록 한다.  
     - find\_vm\_entry( ) : 현재 스레드의 해시 테이블에서 인자로 들어온 주소를 포함하고 있는 vm\_entry를 찾는다. 만약 존재한다면 해당 vm\_entry의 포인터를, 존재하지 않는다면 NULL을 반환한다.  
     - destroy\_vm\_page\_table( ) : hash\_destroy 함수를 통해 해시 테이블을 destroy한다.  
     - vm\_page\_destroy( ) : destory\_vm\_page\_table의 hash\_destroy 함수에 전달되는 함수로서, 해당 vm\_entry가 로드되어 있다면 물리페이지에서 반환하도록 한뒤, vm\_entry 구조체 역시 반환한다.  
     - check\_valid\_string( ) : 인자로 들어온 문자열을 한바이트 씩 유저 영역을 참조하며, vm\_entry에 매핑되어 있는지 확인한다. 그렇지 않다면 sys\_exit(-1) 시스템 콜을 호출하여 프로그램을 종료한다.  
     - check\_valid\_buffer( ) : 인자로 들어온 버퍼를 한바이트 씩 유저 영역을 참조하며, vm\_entry에 매핑되어 있는지 확인한다. 만약 write 상황이라면 writeable 영역에 작성하는지 확인한다. 그렇지 않다면 sys\_exit(-1) 시스템 콜을 호출하여 프로그램을 종료한다.  
     - load\_file( ) : 디스크에 있는 데이터를 물리 메모리로 로드한다. 로드 하고 남은 영역에 대해서는 0으로 채워준다.  
    5. vm/swap.h  
     - 새롭게 추가된 함수의 프로토 타입을 정의한다.  
    6. vm/swap.c  
     - swap\_init( ) : swapping을 위한 자료구조를 초기화한다. swap\_block에 스왑 영역을 할당해주고, swap\_array에 스왑 영역을 관리하기 위한 자료구조를 할당해준다.  
     - swap\_in( ) : 스왑 영역에서 물리 메모리 영역으로 데이터를 가지고오기 위한 함수이다.   
     - swap\_out( ) : 물리 메모리에서 스왑 영역으로 데이터를 내보내기 위한 함수이다.  
    7. userprog/process.h  
     - 새롭게 추가된 함수의 프로토 타입을 정의한다.  
    8. userprog/process.c  
     - handle\_mm\_fault( ) : 유효한 주소에 대한 page fault가 발생될 경우 호출되는 함수이다. 인자로 들어온 vm\_entry가 유효하며 메모리가 로드되어 있는지 여부를 확인한다. page를 새롭게 할당한 뒤, vm\_entry의 타입에 따라 다음과 같이 동작한다. VM\_BIN과 VM\_FILE의 경우에는 load\_file 함수를 디스크로 부터 물리 메모리 영역으로 데이터를 로드한다. VM\_ANON은 스왑 영역에 데이터가 존재함을 의미하므로 swap\_in 함수를 통해 메모리를 물리메모리로 로드해준다. 이후 install\_page를 통해 가상 페이지와 물리 페이지를 매핑해준다.   
     - expand\_stack( ) : 스택을 확장할 경우 호출되는 함수이다. 물리 페이지와 vm\_entry를 할당받고 초기화 하여 스택을 확장한다.  
     - verify\_stack( ) : 스택이 확장될 수 있는지를 확인하는 함수이다. 스택의 확장 가능 최대 영역인 8MB 이내이거나 스택 포인터와 주소가 32바이트 이내라면 스택을 확장하도록 한다.  
    9. userprog/syscall.h  
     - 새롭게 추가된 함수의 프로토 타입을 정의한다.  
    10. userprog/syscall.c  
     - mmap( ) : 파일 디스크립터에 해당하는 파일과 page size 단위로 나뉜 가상 메모리를 매핑하는 시스템 콜이다. 파일 데이터를 PGSIZE로 나누어서 vm\_entry를 구성한 뒤, page table과 mmap\_file 리스트에 삽입한다.  
     - munmap( ) : 매핑된 파일과 가상 메모리를 destroy하는 시스템 콜이다. mmap\_list를 순회하면서 인자로 들어온 mapid를 가진 vm\_entry를 저장한 리스트를 찾는다. 이후 해당 리스트에 있는 모든 vm\_entry들을 반환하도록 한다.  
     - munmap\_all( ) : munmap이 인자로 들어온 mapid에 대해서만 destroy하도록 했다면, 해당 함수는 해당 프로세스에 있는 모든 mmap을 destroy 하도록 한다.  
     - vm\_file\_write( ) : 인자로 들어온 파일에 인자로 들어온 주소부터 인자로 들어온 오프셋 만큼 파일 쓰기를 수행한다.

1. **연구 결과**
   1. **Flow Chart**

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

A. Page Table & Page Fault Handler  
텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명  
B. Disk Swap  
텍스트, 도표, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명  
C. Stack Growth  


* 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. vm/frame.\*  
     - data structure : 페이지 교체 정책에 사용할 자료구조를 선언한다. 물리 페이지에 올라와 있는 page를 관리할 lru\_list를 리스트로 선언하고, 동시성 문제 해결을 위해 lock 자료구조 lru\_list\_lock을 선언한다. 또한 현재 쫓아낼 후보를 가리키기 위한 lru\_clock을 선언한다.  
     - header file : 새롭게 정의된 함수들의 프로토 타입을 선언한다.  
     - lru\_list\_init( ) : 페이지 교체 정책에 사용하기 위해 선언한 자료구조를 초기화한다. List\_init과 lock\_init을 사용하며, lru\_clock은 아직 가리킬 원소가 없으므로 NULL로 초기화 한다.  
     - add\_page\_to\_lru\_list( ) : 페이지를 lru\_list에 삽입한다. List 자료구조인 lru\_list에 삽입하기 위해 list\_push\_back을 사용한다.  
     - del\_page\_from\_lru\_list( ) : 페이지를 lru\_list에서 제거한다. 인자로 들어온 페이지를 list\_remove를 통해 lru\_list에서 제거한다.  
     - alloc\_page( ) : 물리 메모리에서 비어있는 물리페이지를 할당한다. palloc\_get\_page를 통해 free한 페이지 1개를 반환받는다. 이때 인자로 들어온 flags 인자로 주어 유저 풀에서 페이지를 얻도록 한다. 만약 NULL을 반환받는다면, 물리 메모리 영역이 가득 찬 상태를 의미한다. 이때에는 evict\_page\_lru\_policy를 호출하여 물리 메모리에서 second chance algorithm으로 선택된 특정 페이지가 물리메모리에서 쫓겨나도록 한다. 이후 다시 palloc\_get\_page를 통해 물리 페이지를 할당받는다. 페이지 구조체를 할당한 뒤, 해당 물리 주소에는 할당 받은 물리 페이지의 주소를 저장한다. .또한 현재 스레드를 가리키는 포인터를 thread\_current를 통해 얻어 저장 한다. 아직 vm\_entry와 매핑되지 않았으므로, NULL로 저장한다. add\_page\_to\_lru\_list를 호출하여 물리 메모리를 차지하고 있는 page를 lru\_list에 저장한다. 이후 할당한 페이지를 리턴한다.  
     - free\_page( ) : lru\_list를 순회하며 인자로 들어온 물리 메모리를 가르키는 페이지를 찾는다. 해당 페이지를 찾았다면, pagedir\_clear\_page를 통해 해당 페이지에 더이상 매핑하지 않음을 알린다. 또한 palloc\_free\_page를 통해 물리 메모리에 할당된 물리 페이지를 반환한다. del\_page\_from\_lru\_list를 통해 lru\_list에 저장된 페이지를 제거한다. 이후 page를 반환한다.   
     - evict\_page\_lru\_policy( ) : second chance algorithm을 적용하여 물리 메모리에서 한번의 기회를 받았지만 접근되지 않는 물리 페이지를 쫓아낸다. 무한 루프를 통해 access bit가 0인 페이지를 찾는다. get\_lru\_clock을 통해 쫓아낼 페이지 후보를 lru\_clock에 할당한다. 해당 후보의 page를 list\_entry를 통해 얻는다. 이후 pagedir\_is\_accessed를 통해 해당 페이지의 access bit를 확인한다. 만약 해당 비트가 true라면 한번의 기회를 더 주어 쫓아내지 않고, pagedir\_set\_accessed를 통해 access\_bit를 0으로 설정한다. 만약 false라면 기회를 이미 받은 상태이므로 해당 페이지를 쫓아내야 하므로 break를 통해 무한 루프를 빠져나온다. 이후 switch 문을 통해 쫓아낼 페이지의 타입을 확인한다. VM\_BIN이라면 pagedir\_is\_dirty를 통해 데이터의 수정이 발생했는지 확인한다. 만약 수정이 있었다면, 디스크로 쫓아내기 위해 해당 데이터를 업데이트 해야하므로 많은 시간이 필요하다. 이를 막기 위해 swap\_out을 통해 스왑 영역으로 복사하여 데이터를 업데이트하는 시간을 절약한다. 이때 해당 페이지의 타입을 VM\_ANON으로 바꿔주고, 스왑된 슬롯 인덱스 번호를 저장한다. VM\_FILE의 경우에도 pagedir\_is\_dirty를 통해 데이터의 업데이트가 발생했는지 확인한다. 만약 업데이트가 있다면 vm\_file\_write를 호출하여 파일에 업데이트 된 내용을 작성하여 동기화를 해준다. VM\_ANON은 swap\_out을 통해 스와핑을 해준다. 이후 쫓아낸 페이지는 더이상 물리페이지에 로드되어 있지 않으므로 is\_loaded를 false로 업데이트 한 뒤, free\_page를 호출하여 페이지를 반환한다.  
    2. vm/page.\*  
     - Macros : 데이터가 어디서 로드되는지 나타내는 매크로로 VM\_BIN은 바이너리 파일에서 로드됨을, VM\_FILE은 매핑된 파일에서 로드됨을, VM\_ANON은 스왑 영역에서 로드됨을 나타낸다.  
     - data structure : 하나의 페이지를 관리하기 위한 vm\_entry 구조체를 선언한다. 해당 구조체에는 데이터의 타입, 가상 주소 페이지 번호, 쓸 수 있는지 여부, 로드되었는지 여부, 파일 포인터, 오프셋, 읽은 바이트 수, 0인 바이트 수, 어느 스왑 블록이 스왑되어 있는지 여부 등으로 구성되어 있다. mmap\_file 구조체는 매핑된 파일을 관리하기 위한 구조체이다. 매핑된 아이디, 어떤 파일을 매핑하고 있는지 나타내는 파일 포인터, 매핑된 vm\_entry를 관리하는 리스트 등으로 구성되어 있다. page는 물리 메모리의 주소와 vm\_entry의 매핑을 관리하는 구조체이다. 물리 메모리의 주소와, 매핑된 vm\_entry, 매핑된 페이지의 스레드 포인터 등으로 구성되어 있다.   
     - header file : 새롭게 정의된 함수들의 프로토 타입이 선언되어 있다.  
     - init\_vm\_page\_table( ) : 스레드의 페이지를 저장하는 해시 함수를 초기화 하기 위해 hash\_init 함수를 호출하고 있다. 첫번째 인자로 초기화할 해시 자료구조가, 두번째 인자로 해시의 value값을 계산하며, 세번째 인자로 해시 원소를 비교한다.  
     - insert\_vm\_page\_table( ) : 해시 테이블에 인자로 들어온 vm\_entry를 삽입한다. 이를 위해 hash\_insert 함수를 호출한다. 해당 함수는 새로운 원소를 해시 테이블에 넣는다. 해당 키 값에 원소가 존재하지 않으면, NULL을 리턴하고, 존재한다면 새로운 엔트리를 삽입하지 않고 원래 있던 엔트리를 반환한다. 따라서 원래 존재한다면 잘못된 입력이므로 false를 리턴한다.  
     - delete\_vm\_page\_table( ) : 해시 테이블에서 인자로 들어온 vm\_entry를 삭제한다. hash\_delete 함수를 사용하며, 해당 키 값에 원소가 존재하지 않으면 NULL을 반환하고, 존재한다면 해당 vm\_entry를 리턴한다. 따라서 반환값이 null이라면 false를 리턴하고, true라면 해당 엔트리를 반환한 뒤, true를 리턴한다.  
     - find\_vm\_entry( ) : 키 값이 될 vaddr를 임시로 할당한 vm\_entry\_elem 구조체를 지역 변수로 선언한다. vaddr 값에 인자로 들어온 vaddr을 pg\_round\_down 함수를 통해 VPN으로 변환한다. 이후 hash\_find 함수를 통해 해당 키 값을 가지고 있는 해시 값을 찾는다. 해당 값이 존재한다면 hash\_entry를 통해 vm\_entry 값을 반환하고, 없다면 NULL을 반환하도록 한다.  
     - destroy\_vm\_page\_table( ) : hash\_destroy 함수를 통해 해시 테이블을 destroy한다. 인자로 전달된 함수를 통해 해시 테이블에 있는 해시 값들을 모두 destory한 뒤, 테이블도 destory 한다.  
     - check\_valid\_string( ) : 문자열의 길이만큼 반복문을 순회한다. 만약 문자열의 주소가 유저 영역이 아니거나 vm\_entry에 매핑되어 있지 않다면 sys\_exit(-1)을 통해 종료한다.  
     - check\_valid\_buffer( ) : 버퍼의 길이만큼 반복문을 순회한다. 만약 버퍼의 주소가 유저 영역이 아니거나 vm\_entry에 매핑되어 있지 않다면 sys\_exit(-1)을 통해 종료한다. 또한 작성할 수 없는 영역에 작성을 하고자 한다면 sys\_exit(-1)을 통해 종료한다.  
     - load\_file( ) : 디스크에 있는 페이지를 물리메모리에 로드하는 함수이다. file\_read\_at 함수를 통해 디스크에 있는 파일을 두번째 인자의 물리 메모리 주소로 복사한다. 이후 남은 영역만큼은 0으로 초기화해준다.  
    3. vm/swap.\*  
     - data structure : swapping 과정에서 Concurrency를 보장하기 위한 Lock 자료구조 swap\_lock과 swap 구역을 가리키는 block 구조체 포인터 swap\_block, 스왑 구역에 데이터가 존재하는지 여부를 나타낼 포인터 swap\_array를 선언한다.  
     - header file : 새롭게 정의되는 함수 swap\_init, swap\_in, swap\_out의 프로토타입을 정의하고 있다.  
     - swap\_init( ) : lock\_init을 통해 전역변수로 선언된 swap\_lock을 초기화한다. 또한 block\_get\_role을 통해 swap 영역으로 사용할 block을 할당받아 swap\_block에 할당한다. 해당 스왑 영역은 PGSIZE 별로 나뉘어 블록을 형성하기 때문에 해당 블럭 개수만큼 배열을 동적 할당한다. 배열의 초기값을 모두 0으로 설정하여 모든 스왑 블록이 비어있음을 나타내도록 한다.  
     - swap\_in( ) : swap\_block이라는 shared memory에 접근하므로 lock을 통해 동시성 문제를 예방한다. 인자로 들어온 used\_index는 저장된 스왑 영역이 비어있는지를 확인한다. 만약 비어있다면 잘못된 케이스이므로 sys\_exit(-1)을 호출한다. 해당 영역에 데이터가 있다면 데이터를 스왑 영역에서 물리 페이지 영역으로 복사한다. 이를 위해 block\_read를 통해 스왑 영역의 데이터를 BLOCK\_SECTOR\_SIZE 단위로 물리 메모리에 복사한다. 이후 해당 스왑 영역은 빈 상태임을 나타내기 위해 swap\_array의 해당 인덱스를 0으로 초기화한다. 이후 락을 풀어준 뒤 함수를 종료한다.  
     - swap\_out( ) : swap\_block이라는 shared memory에 접근하므로 lock을 통해 동시성 문제를 예방한다. 이후 스왑 영역의 블럭들을 순회하며 빈 블럭을 찾는다. 만약 빈 블럭이 없다면 sys\_exit(-1)을 호출하여 프로그램을 종료한다. 빈 블럭이 있다면 block\_write를 통해 page를 BLOCK\_SECTOR\_SIZE 단위로 나누어 물리 메모리에서 스왑영역으로 복사한다. 이후 해당 스왑 영역이 존재함을 알리기 위해 swap\_array의 해당 인덱스를 1로 초기화 한다. 이후 락을 풀어준 뒤, 스왑 영역의 인덱스를 리턴한다.  
    4. userprog/exception.c  
     - page\_fault( ) : page fault가 발생할 경우 호출되는 함수이다. 폴트가 난 주소를 검사하여 적절한 대처를 해야한다. 만약 read-only영역(not\_present==false)에 write(write == true)를 하거나, 유저(user == true)간 커널 메모리에 접근(is\_kernel\_vaddr(fault\_addr) == true)한다면 유효하지 않은 접근이므로 sys\_exit(-1)을 호출한다. 이외에는 유효한 접근이다. 해당 주소에 대한 vm\_entry를 find\_vm\_entry 함수를 이용해서 찾는다. 만약 vm\_entry가 존재한다면 해당 주소는 유효한 주소이지만 아직 물리메모리에 로드 되지 않은 경우 이므로, handle\_mm\_fault 함수를 호출하여 물리 메모리에 로드해준다. vm\_entry가 존재하지 않는다면, 스택을 확장할 수 있는지 verify\_stack으로 확인한 뒤, 확장 가능하다면 expand\_stack을 통해 스택 영역을 확장한다. 만약 확장하지 못한다면 sys\_exit(-1)을 호출한다. vm\_entry도 존재하지 않고, 스택을 확장시킬 수 없다면 sys\_exit(-1)을 호출한다.  
    5. userprog/process.\*  
     - header file : 새롭게 구현한 함수 handle\_mm\_fault, expand\_stack, verify\_stack 함수의 프로토 타입을 정의해준다.  
     - start\_process( ) : init\_vm\_page\_table을 호출하여 프로세스에서 할당된 vm\_entry를 저장하기 위한 해시 테이블을 초기화한다.  
     - process\_exit( ) : munmap\_all 함수를 호출하여 프로세스를 종료하기 전 매핑되어 있는 mmap\_file을 모두 반환해준다. destroy\_vm\_page\_table 함수를 호출하여 page table에 할당된 vm\_entry들을 모두 반환해준다. 이를 통해 메모리 누수를 예방한다.  
     - load\_segment( ) : 기존 모든 코드, 데이터 영역을 물리 메모리에 로드하는 부분을 주석 처리한다. 이후 해당 페이지에 대한 vm\_entry를 할당하여 페이지 테이블에 삽입해 준다.   
     - setup\_stack( ) : 기존 물리메모리 주소를 받는 kpage를 페이지 구조체로 변경한다. 이후 vm\_entry를 할당하여 초기화한 뒤, 페이지 테이블에 삽입한다.  
     - handle\_mm\_fault( ) : page fault가 발생할 때, 호출되는 함수이다. 인자로 들어온 vm\_entry가 유효하며, 데이터가 물리 메모리에 로드되어 있지 않은지 확인한다. page 자료구조를 할당한 뒤, vm\_entry를 매핑해준다. 이후 인자로 들어온 vm\_entry의 타입을 확인하여 타입에 맞는 동작을 수행하도록 한다. VM\_BIN, VM\_FILE은 디스크에 데이터가 존재하는 상황이므로, load\_file을 통해 할당받은 물리메모리에 데이터를 로드하도록 한다. 로드에 실패하면 할당 받은 페이지를 반환하고 false를 리턴하도록 한다. 반면 VM\_ANON인 경우 데이터가 swap 영역에 있으므로 swap\_in 함수를 통해 스왑 영역에서 물리 메모리 영역으로 데이터를 가지고 온다. 물리 메모리에 데이터를 로드한 뒤, install\_page 함수를 통해 물리 페이지와 가상 페이지를 매핑한 뒤, 로드가 완료되었음을 알리기 위해 vm\_entry의 is\_loaded를 true로 설정한다. 만약 매핑에 실패했다면, 할당받은 물리 페이지를 반환하고 false를 리턴한다.   
     - expand\_stack( ) : 스택을 실제로 확장하는 함수로 확장할 스택 영역을 alloc\_page를 통해 물리메모리에 확보한다. 또한 확장된 스택 영역의 정보를 담을 vm\_entry도 할당한다. 만약 두 자료구조가 정상적으로 할당되지 않았다면, false를 반환한다. 이후 vm\_entry의 값을 초기화해준 뒤, page 구조체와 매핑해주고 page table에 삽입한다. 만약 해당 가상 메모리가 이미 존재한다면, 해당 자료구조들을 반환해주고, 그렇지 않다면 install\_page를 통해 물리 페이지와 가상 페이지를 매핑해준다.  
     - verify\_stack( ) : 스택을 확장할 수 있는지 판단한다. 기준은 휴리스틱을 적용하였으며, 현재 접근하고자 하는 주소에서 스택 포인터를 뺀 값이 32바이트 보다 크다면, false를 반환한다. 또한 스택의 최대 영역이 8MB이므로, 메모리의 가장 base 값에서 현재 스택포인터의 차이가 8MB보다 크다면 false를 리턴하도록 한다. 이외의 경우에는 true를 리턴하여 스택을 확장할 수 있음을 알려준다.  
    6. userprog/syscall.\*  
     - header file : 새롭게 구현한 함수 mmap, munmap, munmap\_all, vm\_file\_write 함수의 프로토 타입을 정의해준다.  
     - syscall\_handler( ) : 새롭게 추가된 시스템 콜 mmap과 munmap을 switch 문에 삽입한다. 이때 인자가 커널 영역이 아닌지 check\_user\_address 함수를 호출하여 확인한다. 이후 시스템 콜 함수를 호출한 뒤, 반환값이 있는 munmap 함수는 f->eax에 반환값을 저장한다. 또한 인자에 문자열이 포함된 SYS\_EXEC, SYS\_CREATE, SYS\_REMOVE, SYS\_OPEN, SYS\_FILESIZE, SYS\_MMAP에 대해서는 check\_valid\_string 함수를 호출하여 문자열이 유효한 주소에 매핑되어 있는지 확인하도록 한다. 인자에 버퍼가 포함된 SYS\_WRITE, SYS\_EXIT은 check\_valid\_buffer 함수를 호출하여 버퍼가 유효한 주소에 매핑되어 있는지, 올바른 영역에 쓰기를 하고 있는지 확인한다.  
     - sys\_create( ) : 테스트 중 page-merge-par, page-merge-mm을 수행하는 과정에서 filesys\_create함수가 실행되는 도중 해당 파일에 접근하는 다른 시스템 콜이 호출되어 파일 생성에 실패하는 경우가 발생하였다. 이를 위해 lock을 잡아주는 lock\_acquire함수와 lock을 반환하는 lock\_release 함수로 filesys\_create함수를 감싸주었다.  
     - sys\_remove( ) : sys\_create와 같은 이유로 filesys\_remove를 lock으로 보호해주었다.  
     - mmap( ) : 해당 시스템 콜이 호출되면 우선 인자로 들어온 값이 올바른지 확인한다. 표준 입출력에 해당하는 파일 디스크립터 0, 1번은 매핑될 수 없다. 또한 매핑하고자 하는 주소값이 0이거나 오프셋이 PGSIZE 단위가 아니라면 매핑될 수 없다. 파일 디스크립터에 파일 포인터가 할당되지 않는 경우도 매핑할 수 없다. 이 경우들에 대해서는 -1을 리턴하도록 한다. 이후 mmap\_file을 동적 할당하며, 동적 할당 성공 여부를 확인한다. 해당 구조체에 mapid를 할당하여 구분을 위한 고유 번호를 할당한다. 이외에도 파일을 다시 오픈하여 파일의 첫번째 offset부터 매핑할 수 있도록 file\_reopen 함수를 통해 얻은 파일 포인터를 저장한다. 또한 해당 파일에 매핑되는 vm\_entry 저장하기 위한 리스트를 초기화하도록 list\_init을 호출한다. 스레드의 모든 mmap\_file을 관리하는 리스트에 할당받은 mmap\_file을 삽입한다. 이후 파일의 길이를 file\_length 함수를 통해 구한 뒤, PGSIZE로 나누어 vm\_entry에 할당하여 mmap\_file의 vm\_entry\_list에 삽입하도록 한다. 이후 작성한 read\_bytes, zero\_bytes, ofs, 주소를 다시 계산하여 반복문 실행 여부를 결정한다. vm\_entry를 할당 받은 뒤, 타입은 파일과 매핑되었으므로 VM\_FILE로 지정해주어야 한다. 또한 생성된 vm\_entry를 insert\_vm\_page\_table을 통해 페이지 테이블에 삽입해주어야 한다. 파일 전체에 대한 매핑이 완료되었다면 해당 mmap\_file의 mapid를 반환한다.  
     - munmap( ) : 현재 스레드를 thread\_current를 통해 찾는다. 이후 반복문을 통해 mmap\_list를 순회하며 mmap\_file에 저장된 mapid와 인자로 들어온 mapid가 같은 인스턴스를 찾는다. 만약 해당 mmap\_file을 찾았다면 또 다시 반복문을 순회하며 mmap\_file에 저장되어 있는 vm\_entry\_list를 순회하며 모든 vm\_entry를 반환한다. 이때 vm\_entry가 NULL이거나 로드되어 있지 않다면, 굳이 반환할 필요가 없다. 그렇지 않다면, pagedir\_get\_page를 통해 스레드의 페이지 디렉토리에 저장되어 있는 vm\_entry의 물리 주소를 얻어온다. pagedir\_is\_dirty를 통해 해당 페이지의 dirty bit가 1인지 확인한다. 만약 1이라면 해당 메모리에 수정이 발생한 상황이므로, file\_write\_at 함수를 호출하여 해당 파일에 데이터를 작성하여 동기화 시켜준다. 이후 free\_page를 통해 해당 물리 메모리 영역을 반환한다. 모든 mmap\_file을 반환하였다면, 파일을 file\_close를 통해 닫아주고, list\_remove를 통해 해당 mmap\_file을 리스트에서 제거해준다. 또한 해당 엔트리를 반환한다.  
     - munmap\_all( ) : 기존 munmap은 인자로 들어온 mapid와 일치하는 mmap\_file에 대해서 반환하였다면, 이 함수는 현재 스레드에 있는 모든 mmap\_file을 반환하도록 한다.  
     - vm\_file\_write( ) : file\_write\_at 함수를 통해 인자로 들어온 파일에 인자로 들어온 물리 메모리 영역을 오프셋만큼 작성해주는 함수이다. lock을 통해 concurrency를 보장해준다.  
    7. threads/thread.\*  
     - struct thread : 해당 구조체에 VM을 위한 자료구조를 추가한다. vm\_entry를 관리하기 위한 vm\_page\_table을 hash 자료구조로 선언하고, mmap된 vm\_entry를 관리할 mmap\_list를 list 자료구조로 선언한다. 이후 현재까지 매핑된 mmap의 개수를 나타내는 cur\_mapid를 정수형으로 선언한다.  
     - thread\_create( ) : 스레드가 생성되고 난 뒤, 현재 실행중인 스레드와 생성된 스레드의 우선순위를 비교하여 thread\_yield( )를 호출한다. 이때 USERPROG 영역에서만 해당 함수가 호출되도록 #ifndef USERPROG, #endif 로 감싸준다.  
     - init\_thread( ) : 스레드가 생성되고 난 뒤, 스레드 구조체에 선언된 자료구조를 초기화하도록 한다. cur\_mapid는 아직 매핑된 mmap이 없으므로 0으로 초기화한다. mmap\_list 역시 list\_init 함수를 사용하여 리스트를 초기화한다.  
    8. threads/synch.c  
     - sema\_up( ) : thread\_yield( ) 함수가 USERPROG 영역에서만 호출될 수 있도록 #ifndef USERPROG, #endif로 감싸준다.   
    9. thread/init.c  
     - main( ) : 메인 함수에서는 핀토스가 시작하면서 사용하게 될 자료구조 등을 초기화 하는 함수를 호출한다. Virtual Memory 활용을 위해서 lru\_list\_init, swap\_init 함수를 호출한다. 이를 통해 LRU 알고리즘과 swapping을 사용하기 위해 필요한 자료구조를 초기화 한다.
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

첫 테스트를 진행하는 데에 있어 thread\_yield( ) 에서 kernel panic이 발생하였다. 이는 userprog 영역이 아닌 부분에서 해당 함수가 호출되어서 발생한 함수이다. 이를 위해 userprog 영역을 #ifndef USERPROG #endif로 감싸주어 userprog영역에서만 함수가 실행되도록 하여 해결했다.

* 1. **시험 및 평가 내용**
* 텍스트, 스크린샷, 폰트, 흑백이(가) 표시된 사진

  자동 생성된 설명