**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재 교수님

이름 / 학번 : 정현정 / 20210428

개발 기간 : 12/19 ~ 12/22

1. **개발 목표**

현재 pintos 구조에 따르면 memory에 올릴 때, page 단위로 자르고 다른 작업 없이 frame을 할당받아 바로 memory에 올리는 방식이었다. 이러한 구조 때문에 pintos가 load할 수 있는 program의 크기와 개수가 physical memory의 사이즈에 의해 제한된다.

이번 프로젝트에서는 이러한 문제점을 해결하기 위해 page마다의 추가 정보를 갖고 있는 Supplemental Page Table을 구현한다. 이를 활용하여 page를 처음부터 memory에 모두 load하는 것이 아니라, program이 매 순간 필요한 page만 frame pool에서 할당받는 Lazy Loading을 구현할 것이다. 이 과정에서 필요한 disk swapping도 구현이 해야하며 이를 통해 physical memory size 제약을 해결할 수 있다.

1. **개발 범위 및 내용**
   1. **개발 범위**

**1. Page Table & Page Fault Handler**

개발 목표에서 언급한 바와 같이 현재 pintos에는 Demand Paging이 구현되어 있지 않다. program을 한번에 다 load하는 방식이 아닌, 매 순간 필요한 부분만 load하기 위해서는 Demand Paging을 구현해야 한다. 이를 구현하려면 기존의 Page Table 외에 추가적인 정보들이 필요하다. 단순히 VPN과 PFN 사이의 mapping 정보 뿐만 아니라, 해당 page가 memory에 load되어 있는 상태인지, write가 가능한지, disk에 있다면 swap parition의 어느 slot에 위치해 있는지 등에 관한 정보들이 필요한 것이다. 이러한 추가 정보들을 담는 테이블을 만들어야 하고, 이 테이블을 Supplemental Page Table이라고 부를 것이다.

Supplemental Page Table을 만든 후 Lazy Loading을 수행하도록 Page Fault Handler를 수정해야 한다. 기존의 Page Fault Handler는 Page Fault 발생 시 바로 program이 종료된다. 이번 프로젝트에서는 Demand Paging을 수행해야 하므로, Page Fault가 발생했을 때, disk에서 frame을 load해야 하는 상황 또는 스택을 늘려주어야 하는 상황을 고려해 Page Fault Handler를 수정해야 한다. 이러한 구현 및 수정을 통해 Demand Paging을 수행할 수 있으며 필요한 page만 선택해 메모리 공간을 할당하거나 해제할 수 있을 것이다.

**2. Disk Swap**

Demand Paging 구현에 고려해야 할 케이스 중 하나는 physical memory가 full인 상태이다. 만약 physical memory가 꽉 차있는 상태, 즉 Frame Pool에서 프로세스에게 할당할 frame이 더 이상 없는 상황을 고려해야 한다. 이러한 경우, Disk Swapping을 통해 특정 page를 disk의 swap partition(swap 영역으로 지정된 disk의 일부분)으로 내리고 메모리에 올리고자 하는 page를 메모리로 이동시키는 과정이 필요하다. 즉, Page Replacement가 필요하다. frame이 할당된 전체 page들 중 Clock Algorithm에 기반하여 evict할 page를 찾고 swap out/in을 수행하는 과정을 구현해야 한다.

**3. Stack Growth**

앞선 개발 범위 1번 항목에서 서술한 바와 같이, 이번 프로젝트에서 올바른 주소 접근으로 간주하고 커버해주어야 하는 Page Fault는 크게 두 가지이다. 접근하려고 한 page가 아직 load되지 않아 Lazy Loading이 필요한 경우와 stack 영역에 접근한 경우이다. 후자는 프로그램 실행 도중 Page Fault가 발생했을 때 fault address가 virtual address space 상에서 stack segment에 포함되는 경우를 말한다. 쉽게 말해서 CPU가 특정 program의 stack 영역에 접근하려고 하는데 Page Fault가 발생한 것이다. stack은 동적으로 사용되는 영역인데 Demand Paging을 사용하는 경우 모든 메모리 page를 미리 할당하거나 physical memory에 mapping하지 않기 때문에 Page Fault가 발생하는 셈이다.

이러한 문제를 해결하기 위해서 fault address가 stack segment를 확장했을 때 커버되는 영역 내에 있는지 확인하고, stack을 확장시키는 과정이 필요하다. pintos에서는 stack의 최대 size를 8MB로 제한하고 있다. 이 이상 stack을 확장시킬 수 없음에 주의해야 한다.

* 1. **개발 내용**

**1. Page Fault가 발생하는 이유와 이를 handling하는 전반적인 과정**

Page Fault는 user가 접근할 수 없는 영역, kernel 영역에 접근했거나 아직 physical frame을 할당받지 못한 virtual page에 접근하려고 할 때 발생한다. Page Fault Handler는 다음과 같은 순서로 동작한다.

**1)** Page Fault가 발생한 fault address를 받아온다. 만약 접근할 수 없는 영역에 접근을 시도한 경우라면 해당 프로세스를 exit한다.

**2)** 해당 virtual page의 유효성을 검사한다. 즉, 해당 page를 포함하는 프로그램을 실행 중이나 Demand Paing에 의해 아직 load되지 않은 것인지 확인한다. 이는 해당 page에 대한 Supplemental Page Table Entry가 존재하는지 확인하면 된다.

**2-1)** 만약 Supplemental Page Table에 entry가 존재한다면, 이는 단순히 해당 page에 대해 physical frame이 할당되지 않았음을 의미한다. disk에서 해당 page를 메모리로 load하고 새롭게 만들어진 frame을 virtual page와 mapping해 Page Table을 업데이트 해야 한다. 이후 다시 instruction을 재실행 하면 Page Fault를 해결할 수 있다.

**2-2)** 만약 entry가 존재하지 않는다면 stack segment를 확장시켰을 때 커버할 수 있는 영역인지 확인해야 한다. 해당 영역에 존재하는 fault address라면 stack growth를 수행하면 된다.

**3)** 그 어느 경우에도 해당하지 않는 fault address라면 이는 segment fault에 해당한다. 따라서 프로세스를 죽이고 free해야 한다.

**2. Disk Swap 발생 시 사용한 Page Replacement Algorithm**

앞선 개발 범위의 2번 항목에서 언급한 바와 같이 disk swap 시 Clock Algorithm을 통해 evict할 page를 찾아 Page Replacement를 수행한다. Clock Algorithm은 현재 clock이 가리키고 있는 페이지의 accessed bit가 1이면 0으로 바꾸고 다음 페이지로 이동한다. 이러한 동작을 반복하다 0인 page를 만나면 이 page를 victim으로 삼고 evict를 수행한다.

**3. Stack Growth 구현 시 stack 확장 여부를 판단 방법**

stack의 주소값을 확인하여 확장 여부를 판단한다. 기존의 4MB인 stack을 8MB로 확장시키는 것이므로, 현재 esp에서 4MB만큼을 뺀 것보다 크거나 같으면 stack growth를 수행한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

12/19 : 명세서 분석 및 개발 계획 작성

12/20 : page fault handler와 Demand Paging 구현

12/21 : disk swap과 stack growth 구현

12/22 : 테스트 케이스 점검 및 보고서 작성

* 1. **개발 방법**

**1. Page Table & Page Fault Handler**

수정하거나 추가한 자료구조와 함수는 다음과 같다. 기타 세부적인 내용들은 추후에 연구결과에서 다루도록 하겠다.

* vm/page.h, vm/page.c
  + struct vm\_entry
  + void vm\_init()
  + static unsigned vm\_hash\_func()
  + static bool vm\_less\_func()
  + bool insert\_vme()
  + bool delete\_vme()
  + struct vm\_entry\* find\_vme()
  + static void destructor()
  + void vm\_destory()
  + bool load\_file()
* threads/thread.h
  + struct thread에 관련 필드 추가 : supplemental\_table
* userprog/exception.c
  + static void page\_fault() 수정
* userprog/process.c
  + static void start\_process() 수정
  + process\_exit() 수정
  + static bool load\_segment() 수정
  + bool handle\_mm\_fault()
  + static bool setup\_stack() 수정
* userprog/syscall.c
  + static void syscall\_handler() 수정
  + struct vm\_entry\* check\_valid\_addr()로 기존의 void check\_valid\_addr() 대체
  + void check\_valid\_buffer()
  + void check\_valid\_string()

**2. Disk Swap**

* vm/page.h, vm/page.c
  + struct page
  + struct mmap\_file
  + void lru\_list\_init()
* vm/frame.c
  + struct page\* alloc\_page()
  + void free\_page()
  + void \_\_free\_page()
  + void add\_page\_to\_lru\_list()
  + void del\_page\_from\_lru\_list()
  + struct list\_elem\* get\_next\_lru\_clock()
  + void\* try\_to\_free\_pages()
* vm/swap.c
  + struct bitmap\* swap\_bitmap
  + struct block\* swap\_block
  + const size\_t SECTORS\_PER\_PAGE
  + void swap\_init()
  + size\_t swap\_out()
  + void swap\_in()
* threads/thread.h
  + struct thread에 관련 필드 추가 : memory\_mapped\_list, mapid
* threads/init.c
  + main() 수정
  + static void locate\_block\_devices() 수정
* threads/thread.c
  + static void init\_thread() 수정
* userprog/process.c
  + void munmap()
  + int mmap()

**3. Stack Growth**

* userprog/exception.c
  + static void page\_fault() 수정
* userprog/process.c
  + bool expand\_stack()

1. **연구 결과**
   1. **Flow Chart**

**1. page**

도표, 텍스트, 기술 도면, 평면도이(가) 표시된 사진

자동 생성된 설명

**2. swap disk**

**텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명**

**3. stack growth**

**텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명**

* 1. **제작 내용**

**1. page**

vm/page.h

#define VM\_BINARY 0 //프로세스 load할 때 생긴 page임을 entry가 나타냄

#define VM\_SWAP 1 //swap의 대상

#define VM\_MAPPED 2 //memory mapping시 만들어진 page

struct vm\_entry{

    uint8\_t type; //pte type

    void \*vaddr; //VPN

    bool writable; //true : write 가능

    bool is\_loaded; //true : physical memory에 loaded

    struct file \*file; //mapping된 file : 해당 페이지가 속한 file

    size\_t offset; //읽어야 할 file offset

    size\_t read\_bytes; //page에 쓰여져 있는 data 크기

    size\_t zero\_bytes; //0으로 채울 남은 page bytes

    //page가 mmap list에 속해 있는 경우

    struct list\_elem mmap\_elem; //mmap list element

    struct hash\_elem elem; //hash table element, hash table == supplement page table

    size\_t swap\_slot; //swap slot : 해당 page가 Swap type인 경우 어떤 slot에 위치해있는지 (== 인덱스)

    //pinned 상태인 페이지는 victim 후보에서 제외

    //pinned가 false이면 Accessed 비트를 검사해서 victim 선정 가능성을 평가

    bool pinned;

};

supplemental page table에서 각 page를 관리하기 위해서는 entry에 사용할 구조체가 필요하다. vm\_entry가 이를 담당한다. 그리고 이 entry들을 다룰 함수들을 구현해야 하는데, 이들은 다음과 같다.

vm/page.c

//해시테이블 초기화

void vm\_init(struct hash \*vm){

    hash\_init(vm, vm\_hash\_func, vm\_less\_func, NULL);

}

추후에 thread 구조체를 수정하여 supplemental\_table을 추가할 것이다. vm\_init()은 thread 구조체에 있는 supllemental\_table 해시 테이블을 초기화하는 역할을 수행한다. hash\_init()을 사용하며 여기에 내부적으로 사용될 두개의 함수는 다음과 같다.

//vaddr을 키값으로 해시값 반환

static unsigned vm\_hash\_func(const struct hash\_elem \*e, void \*aux){

    struct vm\_entry\* entry = hash\_entry(e, struct vm\_entry, elem);

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

}

//두 hash\_elem의 vaddr기준 정렬

static bool vm\_less\_func(const struct hash\_elem \*a, const struct hash\_elem \*b, void \*aux){

    struct vm\_entry\* entry\_a = hash\_entry(a, struct vm\_entry, elem);

    struct vm\_entry\* entry\_b = hash\_entry(b, struct vm\_entry, elem);

    return entry\_a->vaddr < entry\_b->vaddr;

}

vm\_hash\_func()은 entry의 vaddr값으로 해시값을 반환한다. vm\_less\_func()는 입력된 두 hash\_elem의 vaddr을 비교한다. hash\_entry()로 a, b에 해당하는 entry를 찾고, 이들의 vaddr값을 비교해서 같은 버켓 내에서의 위치를 결정한다.

//vm\_entry insert

bool insert\_vme(struct hash \*vm, struct vm\_entry \*vme){

    if(hash\_insert(vm, &(vme->elem)) == NULL) return true;

    return false;

}

//vm\_entry delete

bool delete\_vme(struct hash \*vm, struct vm\_entry \*vme){

    if(hash\_delete(vm, &(vme->elem)) != NULL) return true;

    return false;

}

insert\_vme() 함수는 먼저 해당 elem이 이미 supplemental\_table에 존재하는지 확인한다. 만약 이미 존재하면 이미 false를 반환하고, 존재하지 않으면 insert 후 true를 반환한다.

delete\_vme() 함수는 vme를 삭제하는 동작을 한다. hash\_delete()가 실패하면 false, 성공하면 true를 반환한다.

//해시테이블에서 vme 찾아서 반환

struct vm\_entry \*find\_vme(void \*vaddr){

    //pg\_round\_down()으로 vaddr의 페이지의 vaddr 얻음 -> 이 값을 기준으로 해시 테이블에서 찾음

    struct vm\_entry vme;

    vme.vaddr = pg\_round\_down(vaddr); //vaddr이 속한 페이지의 시작 주소

    //hash\_find() 함수로 hash\_elem 구조체 얻음

    struct hash\* table = &thread\_current()->supplemental\_table;

    struct hash\_elem\* e = NULL;

    e = hash\_find(table, &vme.elem);

    if(e != NULL) return hash\_entry(e, struct vm\_entry, elem);

    else return NULL;

}

find\_vme()는 해시 테이블에서 인자로 받은 vaddr에 해당하는 vme를 찾아 반환한다. pg\_round\_down() 함수를 사용해서 vaddr에 해당하는 페이지 번호를 얻는다. 이 페이지 번호를 갖고 해시값이 동일한 hash\_elem을 찾아 해당 vme를 반환한다.

static void destructor(struct hash\_elem\* e, void\* aux){

    struct vm\_entry\* entry = hash\_entry(e, struct vm\_entry, elem);

    //만약 load되어 있는 상태라면 page 삭제 필요

    if(entry->is\_loaded == true){

        //page 삭제

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

        //page table에서 not present 상태로 표시

        pagedir\_clear\_page(thread\_current()->pagedir, entry->vaddr);

    }

    free(entry);

}

//해시테이블에서 모든 vme 삭제

//프로세스 종료 시 사용

void vm\_destroy(struct hash \*vm){

    hash\_destroy(vm, destructor);

}

destructor()는 hash\_elem을 가지는 vme가 load된 상태이면 page를 삭제한 후 vme를 삭제한다. is\_loaded 필드가 true인 경우, page가 할당되어 있으므로 page를 free\_page를 이용해 삭제하고 pagedir\_clear\_page()를 통해 padedr에서 vaddr에 대응되는 page를 삭제한다.

vm\_destroy()는 destructor()를 사용하여 해시 테이블의 모든 vme를 삭제한다.

//디스크에서 메모리로 페이지 단위로 load

bool load\_file(void\* kaddr, struct vm\_entry\* vme){

    struct file\* file = vme->file;

    size\_t vme\_offset = vme->offset;

    size\_t vme\_read\_bytes = vme->read\_bytes;

    size\_t vme\_zero\_bytes = vme->zero\_bytes;

    size\_t read\_bytes = 0;

    if(vme\_read\_bytes > 0){

        lock\_acquire(&file\_access\_lock);

        file\_seek(file, vme\_offset);

        read\_bytes = file\_read(file, kaddr, vme\_read\_bytes);

        lock\_release(&file\_access\_lock);

        //남은 부분 0으로 패딩

        memset(kaddr + vme\_read\_bytes, 0, vme\_zero\_bytes);

    }

    return read\_bytes == vme\_read\_bytes;

}

load\_file()은 disk에 있는 page를 physical memory로 load한다. file\_seek()로 file의 offset 위치로 offset을 설정하고, file\_read()로 kaddr부터 read\_bytes만큼 physical page에 데이터를 쓴다. 남은 부분을 0으로 패딩하고 file\_read가 제대로 수행되었는지 여부를 bool 값으로 반환한다.

threads/thread.h

struct thread

  {

    /\* 생략 \*/

    struct hash supplemental\_table; //suplemental page tables

    struct list memory\_mapped\_list; //메모리에 매핑된 페이지 관리

    int mapid; //해당 thread에서 mapid가 어디까지 할당되었는지 기록

  };

위처럼 supplemental\_table을 thread 구조체에 추가한다. memory\_mapped\_list와 mapid는 추후에 다룰 예정이다.

userprog/exception.c

static void

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

{

   if(!user || fault\_addr == NULL || is\_kernel\_vaddr(fault\_addr)) exit(-1);

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

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

     which fault\_addr refers. \*/

   /\*

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

          fault\_addr,

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

          write ? "writing" : "reading",

          user ? "user" : "kernel");

  kill (f);

  \*/

   //fault\_addr 유효성 검사

   //페이지가 없는 경우 or write를 위한 주소인 경우만 디스크에서 로드

   //read only 페이지에 대한 접근을 막기 위함

   //유효하다면 vm\_entry 찾아서 handle\_mm\_fault() 호출 -> 디스크에서 메모리로 로드

   if(not\_present || write){

      struct vm\_entry\* entry = find\_vme(fault\_addr);

      if(entry){

         if(!handle\_mm\_fault(entry)){

            //load 실패한 경우

            entry->pinned = false;

            exit(-1);

         }

         entry->pinned = false;

      }

      //stack growth : 4MB

      else if(fault\_addr >= f->esp - 32){

         if(!expand\_stack(fault\_addr)){

            exit(-1);

         }

      }

      else{

         exit(-1);

      }

   }

}

page\_fault() 함수는 page가 존재하지 않거나 write를 위한 주소인 경우 handle\_mm\_fault()를 호출해서 disk에 있는 file을 메모리로 load한다. 이를 위해 기존에 page\_fault() 동작을 수행하던 코드를 주석처리 해야 한다. 해당 vme가 존재하지 않으면 stack growth 영역에 해당하는지 확인한다. 만약 stack growth로 커버 가능한 영역이라면 exapand\_stack()을 호출해 page를 새롭게 할당한다. 두 가지 경우에 모두 해당되지 않는다면 segment fault이므로 exit(-1)로 비정상 종료를 알린다.

userprog/process.c

static void

start\_process (void \*file\_name\_)

{

  char \*file\_name = file\_name\_;

  struct intr\_frame if\_;

  bool success;

  //해시테이블 초기화

  vm\_init(&(thread\_current()->supplemental\_table));

우선 위와 같이 supplemental\_table을 초기화 하는 작업을 start\_process에 추가해주어야 한다.

void

process\_exit (void)

{

  struct thread \*cur = thread\_current ();

  uint32\_t \*pd;

  //모든 mmap\_list의 원소 삭제

  munmap(CLOSE\_ALL);

  //프로세스 종료 시 모든 vme 삭제

  vm\_destroy(&(cur->supplemental\_table));

또 프로세스 종료 시 모든 vme가 supplemental\_table에서 삭제될 수 있도록 vm\_destroy()를 process\_exit()에서 호출한다.

static bool

load\_segment (struct file \*file, off\_t ofs, uint8\_t \*upage,

              uint32\_t read\_bytes, uint32\_t zero\_bytes, bool writable)

{

  file\_seek (file, ofs);

  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;

      //기존의 physical page 할당 부분 삭제

      /\* Get a page of memory. \*/

      /\*

      uint8\_t \*kpage = palloc\_get\_page (PAL\_USER);

      if (kpage == NULL)

        return false;

      \*/

      /\* Load this page. \*/

      /\*

      if (file\_read (file, kpage, page\_read\_bytes) != (int) page\_read\_bytes)

        {

          palloc\_free\_page (kpage);

          return false;

        }

      memset (kpage + page\_read\_bytes, 0, page\_zero\_bytes);

      \*/

      /\* Add the page to the process's address space. \*/

      /\*

      if (!install\_page (upage, kpage, writable))

        {

          palloc\_free\_page (kpage);

          return false;

        }

      \*/

      //vm\_entry 생성하고 해시테이블에 추가

      struct vm\_entry\* entry = (struct vm\_entry\*)malloc(sizeof(struct vm\_entry));

      entry->type = VM\_BINARY;

      entry->vaddr = upage;

      entry->writable = writable;

      entry->is\_loaded = false;

      entry->file = file;

      entry->offset = ofs;

      entry->read\_bytes = page\_read\_bytes;

      entry->zero\_bytes = page\_zero\_bytes;

      entry->swap\_slot = 0;

      entry->pinned = false;

      if(insert\_vme(&thread\_current()->supplemental\_table, entry) == false){

        free(entry);

        return false;

      }

      /\* Advance. \*/

      read\_bytes -= page\_read\_bytes;

      zero\_bytes -= page\_zero\_bytes;

      ofs += page\_read\_bytes;

      upage += PGSIZE;

    }

  return true;

}

load\_segment()에서는 기존의 physical page를 할당하고 mapping하는 부분을 주석 처리한 후, vme를 생성하고 삽입하는 과정을 새롭게 추가한다. 새로운 vme를 malloc으로 할당하고 구조체의 필드들을 현재 상태에 맞게 초기화 한다. 그 후 insert\_vme()를 호출해 현재 thread의 supplemental\_table에 추가한다. while문의 정상 동작과 데이터가 중복되지 않고 page가 할당되도록 하기 위해, read\_bytes에서 읽은 만큼 빼고 ofs에는 읽은 만큼 더해준다. page 단위 정렬을 위해 upage는 PGSIZE만큼 증가한다.

//물리 페이지를 할당 + 디스크에서 메모리로 load + 페이지 테이블에 매핑

bool handle\_mm\_fault(struct vm\_entry\* vme){

  bool success = false;

  vme->pinned = true;

  //이미 load되어 있는 경우 false 반환

  if(vme->is\_loaded) return false;

  //물리 페이지 할당

  enum palloc\_flags flag = PAL\_USER; //유저 메모리풀에서 페이지 할당

  if(vme->type != VM\_SWAP && vme->read\_bytes == 0){

    flag |= PAL\_ZERO;

  }

  //oid\* page = palloc\_get\_page(flag); //page가 kaddr

  struct page\* page = alloc\_page(flag);

  if(page == NULL) return false;

  page->vme = vme;

  //vme type이 VM\_BINARY면 메모리에 로드

  if(vme->type == VM\_BINARY){

    success = load\_file(page->kaddr, vme);

    if(!success || !install\_page(vme->vaddr, page->kaddr, vme->writable)){

      //palloc\_free\_page(page);

      free\_page(page->kaddr);

      return false;

    }

  }

  else if(vme->type == VM\_MAPPED){

    success = load\_file(page, vme);

    if(!success || !install\_page(vme->vaddr, page->kaddr, vme->writable)){

      //palloc\_free\_page(page);

      free\_page(page->kaddr);

      return false;

    }

  }

  else if(vme->type == VM\_SWAP){

    if(!install\_page(vme->vaddr, page->kaddr, vme->writable)){

      free\_page(page->kaddr);

      return false;

    }

    swap\_in(vme->swap\_slot, page->kaddr);

    success = true;

  }

  vme->is\_loaded = true;

  return success;

}

Page Fault 발생 시 핸들링을 위해 호출되는 함수이다. alloc\_page()를 사용해서 page를 할당하고 vme의 type에 따라 처리한다. VM\_BINARY, VM\_MAPPED의 경우 load\_file()과 install\_page()를 호출해 page table에 매핑 정보를 저장한다. VM\_SWAP의 경우 swap\_in()을 꼭 호출해야 한다. 또 새롭게 메모리에 load 되었으므로 is\_loaded 필드를 true로 바꿔준다.

static bool

setup\_stack (void \*\*esp)

{

  uint8\_t \*kpage;

  bool success = false;

  kpage = palloc\_get\_page (PAL\_USER | PAL\_ZERO);

  if (kpage != NULL)

    {

      success = install\_page (((uint8\_t \*) PHYS\_BASE) - PGSIZE, kpage, true);

      if (success){

        \*esp = PHYS\_BASE;

        //vm\_entry 생성하고 해시테이블에 추가

        struct vm\_entry\* entry = (struct vm\_entry\*)malloc(sizeof(struct vm\_entry));

        entry->type = VM\_BINARY;

        entry->vaddr = ((uint8\_t \*) PHYS\_BASE) - PGSIZE;

        entry->writable = true;

        entry->is\_loaded = true;

        entry->file = NULL;

        entry->offset = 0;

        entry->read\_bytes = 0;

        entry->zero\_bytes = 0;

        entry->swap\_slot = 0;

        if(insert\_vme(&thread\_current()->supplemental\_table, entry) == false){

          free(entry);

          return false;

        }

      }

      else{

        palloc\_free\_page (kpage);

      }

    }

  return success;

}

setup\_stack 함수도 vme를 통해서 supplemental\_table에서 관리되도록 수정해주어야 한다.

userprog/syscall.c

static void

syscall\_handler (struct intr\_frame \*f)

{

  //생략

  arg\_cnt[SYS\_MMAP] = 2;

  arg\_cnt[SYS\_MUNMAP] = 1;

  uint32\_t\* esp = (uint32\_t\*)f->esp;

  check\_valid\_addr(esp, esp);

//생략

  else if (\*esp == SYS\_EXEC) {

    char \*cmd\_line = (char\*)\*(esp + 1);

    check\_valid\_string(cmd\_line, esp);

    f->eax = exec(cmd\_line);

}

//생략

  else if (\*esp == SYS\_READ) {

    int fd = (int)\*(esp + 1);

    void\* buffer = (void\*)\*(esp + 2);

    unsigned size = (unsigned)\*(esp + 3);

    check\_valid\_buffer(buffer, size, esp);

    f->eax = read(fd, buffer, size);

  }

  else if (\*esp == SYS\_WRITE) {

    int fd = (int)\*(esp + 1);

    void\* buffer = (void\*)\*(esp + 2);

    unsigned size = (unsigned)\*(esp + 3);

    check\_valid\_string(buffer, esp);

    f->eax = write(fd, buffer, size);

}

//생략

  else if(\*esp == SYS\_OPEN){

    char \*file = (char\*)\*(esp + 1);

    check\_valid\_string(file, esp);

    f->eax = open(file);

}

//생략

  else if(\*esp == SYS\_MMAP){

    int fd = (int)\*(esp + 1);

    void\* addr = esp + 2;

    f->eax = mmap(fd, addr);

  }

  else if(\*esp == SYS\_MUNMAP){

    mapid\_t mapid = (mapid\_t)\*(esp + 1);

    munmap(mapid);

  }

  else {

    exit(-1);

  }

}

기존의 syscall\_handler()에서 주소의 유효성을 검사하는 부분을 수정한다. 맨 처음 수정된 check\_valid\_addr()을 호출하고, system call 종류에 맞게 check\_valid\_string()과 check\_valid\_buffer()를 호출해 string과 buffer의 유효성을 검사한다.

//vaddr에 대응되는 vm\_entry가 존재하는지 체크

struct vm\_entry\* check\_valid\_addr(void\* vaddr, void\* esp){

  if(vaddr == NULL || is\_kernel\_vaddr(vaddr)) exit(-1);

  struct vm\_entry\* entry = find\_vme(vaddr);

  if(entry){

    handle\_mm\_fault(entry);

    if(!(entry->is\_loaded)) exit(-1);

  }

  else if(vaddr >= esp - 32){

    if(!expand\_stack((void\*) vaddr)){

      exit(-1);

    }

  }

  return entry;

}

기존의 check\_valid\_addr()을 내부적으로 handle\_mm\_fault()가 호출되도록 수정한다. 이는 page가 할당되어 있지 않은 주소에 접근했을 경우 swap을 수행할 수 있도록 하기 위함이다. 즉, 기존의 check\_valid\_addr()에 vme 존재 여부를 확인하는 로직과 stack growth를 체크하는 로직을 추가한 것이다.

//buffer 주소 유효성 검사 : read syscall에서 사용

void check\_valid\_buffer(void\* buffer, unsigned size, void\* esp){

  for(void\* i = buffer; i < buffer + size; i++){

    struct vm\_entry\* entry = check\_valid\_addr(i, esp);

    //if(entry == NULL) exit(-1);

    if(!entry || !entry->writable) exit(-1);

  }

}

//syscall에서 사용할 파라미터의 문자열의 주소값이 유효한 vaddr인지 체크

void check\_valid\_string(const void\* str, void\* esp){

  if(check\_valid\_addr(str, esp) == NULL) exit(-1);

  while(\*(char\*)str != 0){

    str = (char\*)str + 1;

    check\_valid\_addr(str, esp);

  }

}

check\_valid\_buffer()는 buffer를 사용하는 system call에서 유효한 buffer를 갖는지 확인하고, check\_valid\_string()은 string에 대해 supplemental\_table에서 entry가 존재하는지 확인하는 함수이다.

**2. swap disk**

vm/page.h

struct mmap\_file{

    int mapid;

    struct list\_elem elem;

    struct vm\_entry\* vme;

};

struct page{

    void\* kaddr; //페이지의 물리주소

    struct vm\_entry \*vme; //물리 페이지가 매핑된 가상 주소의 vm\_entry 포인터

    struct thread \*thread; //해당 물리 페이지를 사용 중인 스레드의 포인터

    struct list\_elem lru; //list 연결을 위한 필드

};

struct list lru\_list;       // page 구조체의 리스트

struct lock lru\_list\_lock;

void\* lru\_clock;

disk swap을 위해 필요한 구조체를 page.h에 추가한다. 그 아래의 변수들은 Clock Algorithm 구현을 위해 선언한 변수들이다.

vm/page.c

//lru\_list, lru\_list\_lock, lru\_clock 초기화

void lru\_list\_init(void){

    list\_init(&lru\_list);

    lock\_init(&lru\_list\_lock);

    lru\_clock = NULL;

}

page.h에 선언되어 있는 lru\_list, lru\_list\_lock, lru\_clock를 초기화한다.

vm/frame.c

//LRU 알고리즘에 따라 페이지 할당 + LRU 리스트에 page 구조체 삽입

struct page\* alloc\_page(enum palloc\_flags flags){

    void\* kaddr = palloc\_get\_page(flags);

    if(kaddr == NULL){

        kaddr = try\_to\_free\_pages(flags);

    }

    struct page\* page = (struct page\*)malloc(sizeof(struct page));

    page->kaddr = kaddr;

    page->thread = thread\_current();

    page->vme = NULL;

    add\_page\_to\_lru\_list(page);

    return page;

}

alloc\_page() 함수는 LRU 알고리즘 중 하나인 Clock Algorithm에 따라 page를 할당한다. palloc\_get\_page()가 정상적으로 이루어지면 kaddr에는 새로운 frame이 할당된다. 새로운 page를 malloc으로 할당하고 이 필드들을 초기화한다. 이때 새롭게 할당된 frame의 주소값을 사용한다. 마지막으로 lru\_list에 추가한다. 만약 palloc\_get\_page가 정상적으로 종료되지 않았다면 try\_to\_free\_pages()를 호출하여서 swap을 수행한다.

//물리 주소 kaddr에 해당하는 page 구조체를 lru 리스트에서 검색

//매치되는 것 찾으면 \_\_free\_page() 호출

void free\_page(void\* kaddr){

    struct list\_elem\* elem;

    struct page\* p = NULL;

    for(elem = list\_begin(&lru\_list) ; elem != list\_end(&lru\_list) ; elem = list\_next(elem)){

        p = list\_entry(elem, struct page, lru);

        if(p->kaddr == kaddr){

            //lru\_clock이 삭제할 elem을 갖고 있음

            if(lru\_clock != NULL && lru\_clock == elem){

                if(list\_size(&lru\_list) == 1) lru\_clock = NULL;

                else{

                    if(list\_end(&lru\_list) == lru\_clock){

                        lru\_clock = list\_front(&lru\_list);

                    }

                    else{

                        lru\_clock = list\_next(lru\_clock);

                    }

                }

            }

            \_\_free\_page(p);

            break;

        }

    }

}

//LRU 리스트에서 page 제거하고 메모리 공간 해제

void \_\_free\_page(struct page\* page){

    del\_page\_from\_lru\_list(page);

    palloc\_free\_page(page->kaddr);

    free(page);

}

free\_page()는 kaddr에 해당하는 page 구조체를 삭제한다. lru\_list를 순회하면서 kaddr에 대응되는 page를 찾고, 찾은 page가 clock에 해당할 경우 clock을 다음 page로 이동시킨다. Clock Algorithm은 circular linked list를 전제로 하므로 이를 list\_front(), list\_next()를 통해 구현한다. 이후 \_\_free\_page()를 호출하는데 \_\_free\_page()는 lru\_list에서 page를 삭제하고 메모리 공간을 free하는 동작을 수행한다.

//LRU list의 끝에 유저 페이지 삽입

void add\_page\_to\_lru\_list(struct page\* page){

    lock\_acquire(&lru\_list\_lock);

    list\_push\_back(&lru\_list, &(page->lru));

    lock\_release(&lru\_list\_lock);

}

//LRU list에 유저 페이지 제거

void del\_page\_from\_lru\_list(struct page\* page){

    struct list\_elem\* elem;

    struct page\* delete\_page = NULL;

    lock\_acquire(&lru\_list\_lock);

    list\_remove(&page->lru);

    lock\_release(&lru\_list\_lock);

}

add\_page\_to\_lru\_list()와 del\_page\_from\_lru\_list()는 각각 lru\_list에서 page를 삽입하고 제거하는 동작을 수행한다.

//clock 알고리즘에서 lru 리스트의 다음 노드 위치를 반환

//victim이 될 페이지를 선택하는 것

struct list\_elem\* get\_next\_lru\_clock(void){

    lock\_acquire(&lru\_list\_lock);

    if(lru\_clock == NULL){

        //lru list가 비어있는 경우

        if(list\_empty(&lru\_list)){

            lock\_release(&lru\_list\_lock);

            return lru\_clock;

        }

        //lru list가 비어있지 않은 경우

        else{

            lru\_clock = list\_front(&lru\_list); //list의 맨 앞으로 일단 설정

        }

    }

    //lru\_clock이 NULL이 아니지만 list가 비어있는 경우 -> NULL로 설정하고 반환

    if(list\_empty(&lru\_list)){

        lru\_clock = NULL;

        lock\_release(&lru\_list\_lock);

        return lru\_clock;

    }

    //list의 맨 앞부터 돌면서 victim 찾기

    while(true){

        struct page\* p = list\_entry((struct list\_elem\*) lru\_clock, struct page, lru);

        //victim 후보에서 제외되는 page인 경우 -> 검사 X

        //pinned가 false인 것만 검사

        if(!p->vme->pinned){

            //accessed bit가 1이면 -> 0으로 바꾸고 second chance

            if(pagedir\_is\_accessed(p->thread->pagedir, p->vme->vaddr)){

                pagedir\_set\_accessed(p->thread->pagedir, p->vme->vaddr, false);

            }

            //accessed bit가 0이면 -> victim

            else break;

        }

        //circular list로 순회하기

        if(lru\_clock == list\_back(&lru\_list)){

            lru\_clock = list\_front(&lru\_list);

        }

        else lru\_clock = list\_next(lru\_clock);

    }

    lock\_release(&lru\_list\_lock);

    return lru\_clock;

}

get\_next\_lru\_clock()은 Clock Algorithm에 따라 lru\_list를 탐색하는 작업을 수행한다. lru\_clock을 기준으로 page의 accessed bit를 검사한다. 이때 pagedir\_is\_accessed()를 사용한다. 만약 1이면 0으로 바꾸고 다음으로 넘어가며, 0이면 victim page로 선정한다.

//clock 알고리즘을 사용해서 여유 메모리를 확보하고

//여유 페이지의 커널 가상 주소 반환

void\* try\_to\_free\_pages(enum palloc\_flags flags){

    //victim page 선정

    struct list\_elem\* elem = get\_next\_lru\_clock();

    struct page\* victim\_page = list\_entry(elem, struct page, lru);

    void\* kaddr = victim\_page->kaddr;

    void\* uaddr = victim\_page->vme->vaddr;

    //만약 VM\_BINARY이면 -> dirsty bit 체크 -> 1이면 swap out

    if(victim\_page->vme->type == VM\_BINARY){

        if(pagedir\_is\_dirty(victim\_page->thread->pagedir, uaddr)){

            victim\_page->vme->type = VM\_SWAP;

            victim\_page->vme->swap\_slot = swap\_out(kaddr);

        }

    }

    //만약 VM\_SWAP이면 -> 항상 swap out

    else if(victim\_page->vme->type == VM\_SWAP){

        victim\_page->vme->swap\_slot = swap\_out(kaddr);

    }

    //is\_loaded 수정 + pagedir에서 삭제 + 페이지 삭제

    victim\_page->vme->is\_loaded = false;

    pagedir\_clear\_page(victim\_page->thread->pagedir, uaddr);

    free\_page(kaddr);

    //새로운 주소 할당

    return palloc\_get\_page(flags);

}

try\_to\_free\_pages()는 physical page가 부족할 때 disk swap을 수행하는 함수이다. 이 함수는 내부적으로 get\_next\_lru\_clock()을 호출해 victim page를 선정한다. 만약 victim page의 type이 VM\_BINARY이면 pagedir\_is\_dirty()를 사용하여 dirty bit를 체크한다. 1로 set되어 있다면 swap\_out을 수행하고 해당 vme의 type을 VM\_SWAP으로 바꾼다. 만약 타입이 VM\_SWAP이라면 항상 swap\_out을 수행한다. swap\_out을 수행하면 새로운 page 주소를 할당받고 그 값을 반환한다.

vm/swap.c

//1이면 사용 중, 0이면 비어있음을 의미

//disk에 있는 swap partiion을 추적하는 메모리 상의 bitmap

struct bitmap\* swap\_bitmap;

struct block\* swap\_partition; //disk 상의 swap partition(swap block)을 의미

size\_t SECTORS\_PER\_PAGE = PGSIZE / BLOCK\_SECTOR\_SIZE; //한 page에 몇개의 sector가 들어가는지 == 8개

//disk 상의 스왑 영역 초기화

void swap\_init(void){

    //블록 장치 중에서 BLOCK\_SWAP 역할을 가진 장치를 검색하고 반환

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

    //disk에 위치한 swap partition에 총 몇개의 page가 들어갈 수 있는지

    //그 개수만큼 bitmap 생성

    swap\_bitmap = bitmap\_create(block\_size(swap\_partition)/SECTORS\_PER\_PAGE);

    lock\_init(&swap\_lock);

}

swap.c에는 swap table을 이루는 두 개의 구조체와 SECTORS\_PER\_PAGE가 선언되어 있다. swap\_init()은 disk 상의 swap 영역을 초기화하는 역할을 담당한다.

//kaddr 주소가 가리키는 페이지를 disk 상의 swap partition 빈 slot에 기록

//해당 slot 번호 반환

size\_t swap\_out(void\* kaddr){

    lock\_acquire(&swap\_lock);

    int slot\_index = bitmap\_scan\_and\_flip(swap\_bitmap, 0, 1, false);

    int sector\_index = slot\_index \* SECTORS\_PER\_PAGE;

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

        block\_write(swap\_partition, sector\_index + i, kaddr);

        kaddr += BLOCK\_SECTOR\_SIZE;

    }

    lock\_release(&swap\_lock);

    return slot\_index;

}

swap\_out() 함수는 kaddr 주소가 가리키는 page를 swap partition에 기록한다. 그리고 해당 swap slot의 번호를 반호나한다. 이때 slot\_index에 SECTORS\_PER\_PAGE만큼을 곱해 sector\_index를 구하는 부분을 주의해야 한다.

//used\_index의 slot에 저장된 데이터를 kaddr로 복사

void swap\_in(size\_t used\_index, void\* kaddr){

    lock\_acquire(&swap\_lock);

    int sector\_index = used\_index \* SECTORS\_PER\_PAGE;

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

        block\_read(swap\_partition, sector\_index + i, kaddr);

        kaddr += BLOCK\_SECTOR\_SIZE;

    }

    bitmap\_set(swap\_bitmap, used\_index, false);

    lock\_release(&swap\_lock);

}

used\_index의 swap slot에 저장된 데이터를 kaddr로 복사하는 동작을 수행한다. bitmap의 used\_index에 해당하는 부분을 false로 set하여 지속적으로 slot을 추적할 수 있도록 해야 한다.

userprog/syscall.c

//파일 매핑 제거

void munmap(mapid\_t mapid){

  struct thread\* current\_thread = thread\_current();

  struct list\* mm\_list = &current\_thread->memory\_mapped\_list;

  struct list\_elem\* elem;

  for(elem = list\_begin(mm\_list); elem != list\_end(mm\_list); elem = list\_next(elem)){

    struct mmap\_file\* mm = list\_entry(elem, struct mmap\_file, elem);

    if(mm->mapid == mapid || mapid == CLOSE\_ALL){

      if(mm->vme->is\_loaded){

        lock\_acquire(&file\_access\_lock);

        //dirty page 처리를 위해서 vaddr에 write

        file\_write\_at(mm->vme->file, mm->vme->vaddr, mm->vme->read\_bytes, mm->vme->offset);

        lock\_release(&file\_access\_lock);

        //palloc\_free\_page(pagedir\_get\_page(current\_thread->pagedir, mm->vme->vaddr));

        free\_page(pagedir\_get\_page(current\_thread->pagedir, mm->vme->vaddr));

        pagedir\_clear\_page(current\_thread->pagedir, mm->vme->vaddr);

      }

      delete\_vme(&current\_thread->supplemental\_table, mm->vme);

      list\_remove(elem);

      free(mm->vme);

      free(mm);

    }

  }

}

munmap()은 mapid를 인자로 받아 해당 mapid와 연결된 mmap\_file을 mmap\_list에서 삭제한다. 만약 CLOSE\_ALL에 해당한다면 모든 mmap\_file을 삭제하도록 구현한다. mapid는 1부터 가질 수 있으므로 CLOSE\_ALL은 0에 해당한다.

//디스크에서 메모리로 파일 데이터 load

int mmap(int fd, void \*addr){

  //fd : 매핑할 파일

  //addr : 매핑을 시작할 가상 주소

  //addr 유효성 검사

  if(addr == NULL || is\_kernel\_vaddr(addr) || pg\_ofs(addr) != 0) return -1;

  //fd에 해당하는 파일 가져오기

  check\_valid\_fd(fd);

  lock\_acquire(&file\_access\_lock);

  struct file\* original\_file = thread\_current()->fd\_table[fd];

  if(original\_file == NULL){

    lock\_release(&file\_access\_lock);

    return -1;

  }

  size\_t original\_file\_len = file\_length(original\_file);

  lock\_release(&file\_access\_lock);

  if(original\_file\_len == 0) return -1;

  //fd에 해당하는 파일 열기

  lock\_acquire(&file\_access\_lock);

  struct file\* new\_file = file\_reopen(original\_file);

  lock\_release(&file\_access\_lock);

  if(!new\_file) return -1;

  //vme, mmap\_file 생성하고 초기화

  lock\_acquire(&file\_access\_lock);

  size\_t read\_bytes = file\_length(new\_file);

  lock\_release(&file\_access\_lock);

  thread\_current()->mapid++;

  mapid\_t mapid = thread\_current()->mapid;

  size\_t offset = 0;

  lock\_acquire(&file\_access\_lock);

  file\_seek(new\_file, offset);

  lock\_release(&file\_access\_lock);

  bool flag = true;

  while(read\_bytes > 0){

    //vme 생성

    size\_t vme\_read\_bytes;

    if(read\_bytes < PGSIZE) vme\_read\_bytes = read\_bytes;

    else vme\_read\_bytes = PGSIZE;

    size\_t vme\_zero\_bytes = PGSIZE - vme\_read\_bytes;

    struct vm\_entry\* vme = (struct vm\_entry\*)malloc(sizeof(struct vm\_entry));

    if(!vme){

      flag = false;

      //file\_close(new\_file);

      break;

    }

    vme->file = new\_file;

    vme->offset = offset;

    vme->vaddr = addr;

    vme->swap\_slot = 0;

    vme->read\_bytes = vme\_read\_bytes;

    vme->zero\_bytes = vme\_zero\_bytes;

    vme->is\_loaded = false;

    vme->type = VM\_MAPPED;

    vme->writable = true;

    //memory\_mapped\_list에 새로운 mmap\_file 추가

    struct mmap\_file\* new\_mm = (struct mmap\_file\*)malloc(sizeof(struct mmap\_file));

    if(!new\_mm){

      flag = false;

      break;

    }

    new\_mm->vme = vme;

    new\_mm->mapid = mapid;

    list\_push\_back(&(thread\_current()->memory\_mapped\_list), &(new\_mm->elem));

    //vme 삽입

    if(!insert\_vme(&thread\_current()->supplemental\_table, vme)){

      free(vme);

      list\_remove(new\_mm);

      free(new\_mm);

      flag = false;

      break;

    }

    read\_bytes -= vme\_read\_bytes;

    offset += vme\_read\_bytes;

    addr += PGSIZE; //페이지 크기 단위로 주소 align

  }

  if(!flag) return -1;

  else return mapid;

}

mmap()은 기본적으로 프로세스의 가상 주소 공간에 file을 mapping하는 동작을 수행하는 함수이다. fd값을 사용해 파일을 찾고 유효성 검사를 수행한다. file\_reopen()을 통해 inode를 할당받고 file을 생성한다. 이후 mapid를 할당하고 mmap\_file의 필드를 초기화한다. file을 PGSIZE 단위로 나누어 vme를 생성하고 저장한다. 모든 과정이 끝나면 할당받은 mapid를 반환한다.

**3. stack growth**

userprog/process.c

bool expand\_stack(void\* kaddr){

  void\* vaddr = pg\_round\_down(kaddr);

  //범위 체크 : 최대 8MB

  if((size\_t)(PHYS\_BASE - vaddr) > (1 << 23)){

    return false;

  }

  //vme 생성

  struct vm\_entry\* vme = malloc(sizeof(struct vm\_entry));

  if(!vme) return false;

  vme->vaddr = vaddr;

  vme->writable = true;

  vme->type = VM\_SWAP;

  vme->is\_loaded = true;

  vme->pinned = true; //확장 완료 전까지 swap out되지 않도록 설정

  //새로운 page 할당 + install

  struct page\* page = alloc\_page(PAL\_USER);

  if(!page){

    free(vme);

    return false;

  }

  page->vme = vme;

  if(!install\_page(vaddr, page->kaddr, vme->writable)){

    free(vme);

    free\_page(page->kaddr);

    return false;

  }

  vme->pinned = false;

  return insert\_vme(&thread\_current()->supplemental\_table, vme);

}

expand\_stack()은 stack을 확장하는 함수이다. supplemental\_table에서 관리될 수 있도록 vme를 생성하고 page를 할당한다. VM\_SWAP으로 type을 설정해서 swap in/out에 항상 사용되도록 한다.

* 1. **시험 및 평가 내용**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명