**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재 교수

이름 / 학번 : 최호진 / 20201654

개발 기간 : 2024.11.28. – 2024.12.21.

1. **개발 목표**

Virtual Memory를 구현한다. 기존의 PintOS에서 구현되어 있는 Page Fault 정책을 더욱 정교하게 바꾸어서 수업시간에 배운 Paging을 구현한다. 또한 Replacement 정책으로 LRU에 기반한 Clock Algorithm을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* Page Table & Page Fault Handler

기존 PintOS에 구현된 Paging은 Demand Paging이 구현되어 있지 않기 때문에 Memory 낭비가 발생했다. 기존의 구현된 Page fault를 수정해서, Faulting Address가 가리키는 Page를 메모리로 로드한 뒤 인스트럭션을 재실행하도록 수정한다. 이를 구현하기 위해 Page와 Page Frame과 관련된 함수를 구현한다.

* Disk Swap

Physical Memory의 공간이 부족하다면, Page Replacement를 해주어야 한다. 이때, LRU를 기반으로 한 Clock Algorithm (Second Chance Algorithm)을 구현하여, 알맞은 Page를 Evict 하고 교체를 해주기 위한 함수들을 구현한다.

* Stack Growth

Page fault가 발생한 위치가 프로세스의 스택 영역이라면, 스택을 확장해야한다. 이를 위한 함수를 구현한다.

* 1. **개발 내용**
  2. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

Page Table에 접근하고자 하는 Page에 대한 매핑 정보가 없을 때 (VPN 🡪 PFN) Page Fault가 발생한다. 발생 시, fault address가 유효한 address인지 확인한다. Page에 해당하는 PTE가 존재한다면, 일반적인 Page fault 상황으로 Memory로 Page를 옮긴 뒤 Instruction을 재실행한다. 만약 PTE가 존재하지 않는다면, Stack growth가 가능한지 확인하고, 수행한다. 그렇지 않다면 Segmentation Fault로 Exit (-1)로 종료한다.

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

수업 시간에 배운 Clock Algorithm(Second Chance Algorithm)을 사용하여 구현하였다. 구현 방식은 간단하게, Evict할 Page를 찾을 때, 포인터에게 처음으로 지목 당한다면 Reference bit을 0으로 하고, 두번째 지목 당하면 Evict 하는 방식으로 구현한다. 또한 Evict할 페이지의 유형에 따라 각각 다른 작업을 진행해준다.

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

페이지 폴트가 발생한 Address가 User Address인지, 또한 Start Address로부터 8MB (PintOS의 Stack Size 최대 크기) 안 범위인지 확인한다. 또한, PUSHA Instruction의 조건에 따라 esp에서 32 bytes보다 Address가 멀리 떨어져 있지는 않은지 확인한다. 모든 조건을 충족한 경우 Stack growth를 할 수 있는 address이므로, 스택을 확장한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 11/28 – 11/30: Page Fault Handler 관련 구조체 및 함수 구현
* 11/30 – 12/2: Disk Swap, Stack Growth 관련 함수 구현
* 12/2 – 12/5: mmap, unmap system call 구현

(시험 기간 관계로 중단)

* 12/19 – 12/21: 몇몇 오류 디버깅 및 보고서 작성
  1. **개발 방법**

PintOS 매뉴얼 및 KAIST PintOS 프로젝트 강의 자료를 참고하여서 구현하였다.

* + 1. Page Fault Handling

먼저, page fault 발생 시에 적절한 작업을 위해서는 vm 디렉토리 내에 page.c/h, frame.c/h, swap.c/h(Swapping에서 사용) 파일을 생성하고 필요한 자료구조 및 관련 함수를 구현한다.

먼저, Page Table을 구현하기 위해 PintOS에서 제공하는 자료구조 중 Hash를 선택했다. Search, Insert, Delete 작업을 많이 실행하기 때문에 가장 평균 시간복잡도가 작은 Hash 자료구조를 택했다. (추후에 설명하겠지만, 이로 인해 불필요한 디버깅이 필요했던 것 같기도 하다.)

또한 Page Table Eentry 구조체를 page.h에 선언해주어야 하는데, 해당 구조체 내에는 페이지의 타입과 파일에 대한 정보, Memory에 Load되었는지 확인하는 Flag 등이 포함되어있다. 그리고 연산을 위한 각종 함수들을 (init, insert, destroy, find, free 등) page.c 파일에서 구현한다. 이를 이용해서 page fault 시 적절한 작업을 진행한다.

* + 1. Disk swap

frame.c/h와 swap.c/h에서 구현한 구조체와 함수를 이용해서 Swapping을 구현한다. Frame에서는 frame table 구조체와 insert, delete, find등의 함수를 선언 및 구현하고, swap에서는 swap\_in, swap\_out., swap\_free와 같은 기능을 구현한다. 또한 이때 bitmap 자료구조(핀토스 매뉴얼 및 KAIST 강의자료 참고)를 이용한다.

* + 1. Stack Growth

process.c 함수에 stack\_growth 함수를 구현한다. page\_fault가 발생했을 때 stack\_growth가 가능한지 확인하고 수행한다.

1. **연구 결과**
   1. **Flow Chart**
   * Page Fault Handling

텍스트, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

* + Disk Swap

텍스트, 도표, 라인, 평행이(가) 표시된 사진

자동 생성된 설명

* + Stack Growth

도표, 텍스트, 종이접기이(가) 표시된 사진

자동 생성된 설명텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
* Page fault Handling, Disk Swap (Page Replacement)

*struct* PTE{

*void* \*vpn; */\* Virtual page number \*/*

   pte\_typ type; */\* Type of the page \*/*

*bool* writable; */\* True if writable \*/*

*struct* file \*file; */\* Reference to the file \*/*

*size\_t* offset; */\* Offset of the file \*/*

*size\_t* read\_bytes; */\* Amount of the data written in the page \*/*

*size\_t* swap\_slot; */\* Location of the swap slot \*/*

*bool* mem\_flag; */\* True if the page is in the memory \*/*

*struct* hash\_elem elem; */\* Hash element for page table \*/*

*struct* list\_elem mmap\_elem; */\* List element for mmap list \*/* };

Page Table Entry 구조체를 선언하다. 해당 구조체에는 virtual address, type, file과 관련한 정보 및 memory에 페이지가 위치하는지 확인하는 flag를 추가한다. 또한, page\_table을 hash로 구현하였기 때문에 hash element를 저장하는 hash\_elem을 구현하였다. mmap\_elem은 마지막에 설명하겠다.

각 프로세스(Thread)가 해당 구조체를 가져야 하기 때문에 thread.h의 thread 구조체에 hash 구조체의 page table을 추가한다.

#ifdef USERPROG

*/\* Owned by userprog/process.c. \*/*

*uint32\_t* \*pagedir; */\* Page directory. \*/*

*/\* File Descriptor Table\*/*

*struct* file\* fd\_table[128];

*/\*child\*/*

*struct* list children;

*struct* list\_elem child;

*struct* semaphore mem\_lock;

*struct* semaphore child\_lock;

*struct* semaphore load\_lock;

*struct* thread \*parent;

*struct* file \*file; */\*mapped file in this thread\*/*

*struct* hash page\_table;

*struct* list mmap\_list;

*int* flag;

#endif

그리고 page.c에는 entry를 관리하기 위한 함수들을 구현하였다.

*struct* PTE \*create\_pte(*void* \**vpn*, pte\_typ *type*, *bool* *writable*, *struct* file \**file*, \

*size\_t* *offset*, *size\_t* *read\_bytes*, *bool* *mem\_flag*) {

*struct* PTE \*pte = (*struct* PTE \*)malloc(sizeof(*struct* PTE));

  if(pte == NULL) return NULL;

  memset(pte, 0, sizeof(*struct* PTE));

  pte->vpn = vpn;

  pte->type = type;

  pte->writable = writable;

  pte->file = file;

  pte->offset = offset;

  pte->read\_bytes = read\_bytes;

  pte->mem\_flag = mem\_flag;

  return pte;

}

*struct* PTE \*page\_lookup(*void* \**vpn*) {

*struct* PTE p;

*struct* hash\_elem \*e;

  p.vpn = pg\_round\_down(vpn);

  e = hash\_find(&thread\_current()->page\_table, &p.elem);

  if (e == NULL) return NULL;

  else return hash\_entry(e, *struct* PTE, elem);

}

*bool* page\_insert\_entry(*struct* hash \**pt*, *struct* PTE \**pte*) {

*struct* hash\_elem \*e = &pte->elem;

  return hash\_insert(pt, e);

}

*bool* page\_delete\_entry(*struct* hash \**pt*, *struct* PTE \**pte*) {

*struct* hash\_elem \*e = &pte->elem;

*bool* success = hash\_delete(pt, e);

  if(!success) return false;

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

  swap\_free(pte->swap\_slot);

  free(pte);

  return success;

Page fault 발생 시 fault address로 page\_lookup 함수를 이용해 알맞은 entry를 찾아 할당한다. (exception.c)

*// valid한 주소인지 확인 + page fault가 발생했는데 page가 할당되어 있다면 잘못된 것이기 때문에 Exit*

   if (!is\_user\_vaddr(fault\_addr)) Exit(-1);

   if (!not\_present) Exit(-1);

*struct* PTE \*pte = page\_lookup(fault\_addr);

   if(pte == NULL && !stack\_growth(fault\_addr, f->esp)) Exit(-1);

   else if(pte != NULL && !handle\_mm\_fault(pte)) Exit(-1);

frame.c와 frame.h에서는 frame table을 관리한다. 이를 위한 구조체를 frame.h에 선언하고, 해당 구조체에는 Physical address와 소유한 Thread, mapping된 page table entry 등이 포함된다. Evict할 page를 선택하기 위해서 앞서 말했듯이 Clock algorithm을 사용했는데, pointer가 각 frame table을 순회하면서 evict할 frame을 선택한다.

*struct* frame{

*void* \*pfn; *// Physical Frame Number*

*struct* thread \*t; *// Thread that owns the frame*

*struct* PTE \*pte; *// Page Table Entry*

*struct* list\_elem elem; *// List element for frame list*

};

*// clock algorithm*

*// Cycle the frame table and find the frame to evict*

*static* *struct* list\_elem \*clock(*void*){

*// if the frame pointer is NULL and the frame table is not empty, set the frame pointer to the beginning*

    if(frame\_ptr == NULL && !list\_empty(&frame\_table)){

        frame\_ptr = list\_begin(&frame\_table);

        return frame\_ptr;

    }

*// if the frame pointer is at the end of the frame table, set it to the beginning*

    if(frame\_ptr == list\_end(&frame\_table) && !list\_empty(&frame\_table)){

        frame\_ptr = list\_begin(&frame\_table);

        return frame\_ptr;

    }

*// if the frame pointer is at the end of the frame table, set it to the beginning*

    frame\_ptr = list\_next(frame\_ptr);

    if(frame\_ptr == list\_end(&frame\_table))

        frame\_ptr = clock();

    return frame\_ptr;

}

*// evict frame using clock algorithm*

*void* evict\_frame (*void*){

*struct* frame \*f;

*// get victim frame*

    while (true){

        f = list\_entry(clock(), *struct* frame, elem);

*// if the frame is accessed, set the accessed bit to false*

        if(pagedir\_is\_accessed(f->t->pagedir, f->pte->vpn)){

            pagedir\_set\_accessed(f->t->pagedir, f->pte->vpn, 0);

        }

*// if the frame is not accessed, evict the frame*

        else {

*bool* dirty = pagedir\_is\_dirty(f->t->pagedir, f->pte->vpn);

*// if the frame is dirty*

            if(f->pte->type == MEMMAP && dirty){

*// if the frame is memmaped and dirty, then write*

*// data to the file, and evict*

                file\_write\_at (f->pte->file, f->pfn, f->pte->read\_bytes, \

                    f->pte->offset);

            }

*// if the frame is from the swap slot, swap out.*

            else if(f->pte->type == SWAP){

                f->pte->swap\_slot = swap\_out(f->pfn);

            }

*// swap out and change the type to SWAP*

            else if(f->pte->type == LOAD && dirty){

                f->pte->swap\_slot = swap\_out(f->pfn);

                f->pte->type = SWAP;

            }

*// free the frame*

            f->pte->mem\_flag = false;

            delete\_frame(f);

            pagedir\_clear\_page(f->t->pagedir, f->pte->vpn);

            palloc\_free\_page(f->pfn);

            free(f);

            return;

        }

    }

}

위 코드에서 볼 수 있듯이, 알맞은 frame을 찾을 때까지 반복해서 clock 함수를 호출하여, 찾는다. Clock 함수에서는 pinter가 frame table 마지막 원소에 도달하면 처음으로 돌아가고, 그 외에는 다음 원소를 가리키도록 움직여준다. 그 다음 포인터가 가리키는 원소가 accessed bit가 1이라면 0으로 바꾼 뒤 다시 clock 알고리즘을 수행해 다음 원소를 찾고, 그렇지 않다면 frame의 종류에 따라 알맞은 작업을 수행한다.

Entry의 type이 MEMMAP(Memory mapping된 페이지)이고 dirty page라면 write하고 evict한다. 만약 swap type이라면 swap out하고, load type에 dirty page라면 swap out후 type을 swap으로 바꿔준다. 그 다음 frame을 free해준다.

다음 Disk swap에 대한 설명을 함께 진행하겠다. Swapping은 swap.h와 swap.c에서 이를 담당하는데, bitmap 자료구조를 이용해 관리한다. 또한 swap과 관련된 작업 시 lock을 걸어주어 synchronization을 고려한다. 대부분 PintOS 내부 라이브러리를 이용해 구현하였고, 구현이 간단하기 때문에 대표로 swap\_out 함수에 대해서 설명하고 이를 제외한 나머지 코드는 설명을 생략하겠다.

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

*// find empty slot in swap table, and set the bit to 1*

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

    lock\_acquire(&swap\_lock);

*size\_t* free\_index = bitmap\_scan\_and\_flip(swap\_bitmap, 0, 1, false);

    for(*size\_t* i = 0; i < 8; i++){

        block\_write(swap\_block, free\_index \* 8 + i, pfn + \

            i \* BLOCK\_SECTOR\_SIZE);

    }

    lock\_release(&swap\_lock);

    free\_index += 1;

    return

Swap out시 free index(빈 슬롯)를 찾고, block\_write 데이터를 스왑 공간으로 이동하는 작업을 한다.

마지막으로, Stack Growth에 대한 설명이다.

Page fault 발생 시 stack\_growth 함수를 호출하는데, 이 함수는 process.c에 구현이 되어있다.

*bool*

stack\_growth (*void* \**addr*, *void* \**esp*) {

*// check if the address is valid*

*// and check if the address is below PHYS\_BASE - 0x800000*

*// if address is below PHYS\_BASE - 0x800000, then it cannot be a stack growth*

  if (!is\_user\_vaddr(addr) || addr < PHYS\_BASE - 0x800000) return false;

*// check if the address is below PHYS\_BASE - 32 (qualify the PUSHA instruction)*

  if (addr < esp -32) return false;

*// get boundary*

*void* \*upage = pg\_round\_down(addr);

*struct* frame \*f = alloc\_page\_to\_frame(PAL\_USER | PAL\_ZERO);

  if(f == NULL) return false;

*bool* success = install\_page(upage, f->pfn, true);

  if(success){

    f->pte = create\_pte(upage, SWAP, true, NULL, 0, 0, true);

    if(f->pte == NULL) return false;

    page\_insert\_entry(&(thread\_current()->page\_table), f->pte);

  }

  else

    free\_frame(f->pfn);

  return success;

}

Fault address가 user address 내에 있는지, 그리고 stack의 사이즈 범위에 있는지 확인한다. 해당 조건을 통과하지 못하면 false를 반환하고, 통과했다면 마지막으로 PUSHA 인스트럭션 연산의 조건을 검사한다. esp 레지스터로부터 32byte 내에 fault address가 포함되는지 확인한다. 모든 조건이 만족한다면 stack을 growth한다. 이후 frame을 할당하고, page table entry를 만들어 진행한다.

* 추가 구현 및 디버깅

이렇게 다 구현하였을 때 page-merge-par, page-merge-stk, page-merge-mm 3가지 테스트 케이스를 통과하지 못했다. 이들을 통과하기 위해서는 mmap이 필요하다고 생각해서 mmap 구조체를 page.h에 구현하고 관련 system call (mmap, munmap)을 구현한다.

*struct* mmap\_file {

*unsigned* mapid; */\*mapping id\*/*

*struct* file \*file; */\*mapping file object\*/*

*struct* list\_elem elem; */\*list element for mmap list\*/*

*struct* list pte\_list; */\*list of page table entries\*/*

};

PTE 구조체 내에 mmap\_elem을 추가한다.

*struct* list\_elem mmap\_elem; */\* List element for mmap list \*/*

그리고 systemcall에 Mmap과 Munmap을 구현한다.

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

  if (*addr* == NULL || pg\_ofs(*addr*) != 0 || page\_lookup(*addr*)) return -1;

  if (*fd* < 3 || *fd* >= 128 || thread\_current()->fd\_table[*fd*] == NULL) return -1;

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

  if (mmap\_file == NULL) return -1;

  list\_init(&mmap\_file->pte\_list);

  list\_push\_back(&thread\_current()->mmap\_list, &mmap\_file->elem);

  mmap\_file->mapid = thread\_current()->max\_mapid;

  thread\_current()->max\_mapid += 1;

  lock\_acquire(&filesys\_lock);

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

  lock\_release(&filesys\_lock);

*//file size = 0 -> return -1*

*off\_t* file\_size = file\_length(mmap\_file->file);

  if (file\_size == 0) {

*// free(mmap\_file);*

    return -1;

  }

;

*off\_t* offset = 0;

*// read the file into memory*

  while (file\_size > 0) {

*size\_t* read\_bytes = file\_size < PGSIZE ? file\_size : PGSIZE;

*struct* PTE \*pte = create\_pte(*addr*, MEMMAP, true, mmap\_file->file, offset,\

                      read\_bytes, false);

    if (pte == NULL) return 0;

    page\_insert\_entry(&thread\_current()->page\_table, pte);

    list\_push\_back(&mmap\_file->pte\_list, &pte->mmap\_elem);

*// update variables*

    file\_size -= PGSIZE;

    offset += PGSIZE;

*addr* += PGSIZE;

  }

  return mmap\_file->mapid;

}

*void* Munmap(*unsigned* *mapid*) {

*struct* mmap\_file \*mmap\_file = NULL;

*struct* list\_elem \*e;

*//find the mmap\_file with the given mapid*

  for (e = list\_begin(&thread\_current()->mmap\_list); e != list\_end(&thread\_current()->mmap\_list); e = list\_next(e)) {

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

    if (mmap\_file->mapid == *mapid*) break;

  }

  if (mmap\_file == NULL) return;

*struct* list\_elem \*e2;

  for (e2 = list\_begin(&mmap\_file->pte\_list); e2 != list\_end(&mmap\_file->pte\_list);) {

*struct* PTE \*pte = list\_entry(e2, *struct* PTE, mmap\_elem);

    if (pte->mem\_flag && pagedir\_is\_dirty(thread\_current()->pagedir, pte->vpn)) {

      lock\_acquire(&filesys\_lock);

*size\_t* read\_byte = pte->read\_bytes;

      if ((*size\_t*)file\_write\_at(pte->file, pte->vpn, pte->read\_bytes, pte->offset) != read\_byte) {

        NOT\_REACHED();

      }

      lock\_release(&filesys\_lock);

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

    }

    pte->mem\_flag = false;

    e2 = list\_remove(e2);

    page\_delete\_entry(&thread\_current()->page\_table, pte);

  }

  list\_remove(&mmap\_file->elem);

  free(mmap\_file);

  return;

}

해당 시스템 콜의 구현은 KAIST 강의자료를 참고해서 구현하였다. 해당 구현은 이번 보고서에서 요구하는 것은 아니기 때문에 간단하게 설명하자면, 특정 조건들 (user address내에 addr가 위치하고, 할당된 PTE가 있는지 등)을 확인하고 충족하면 memory mapping을 수행하는 mmap과 free 작업을 하는 munmap이 있다.

위의 3가지 테스트 케이스를 통과하기 위해 구현했는데, 로컬 서버에서 테스트했을 때에는 통과했지만, 실제 cspro에서 확인했을 때에는 계속해서 3가지 케이스를 통과를 하지 못했다. 디버깅을 위해 결과를 확인해보니 file을 open할 때 계속해서 fail이 뜨는 상황이 발생했다. 그래서 구글링을 통해 확인해보니, 비슷한 경우가 발생하는 사람들이 있었는데 file open이 너무 빨리 일어나서 create전에 open을 해버려 문제가 생긴다고 하였다. 이를 위해 file과 관련된 system call에 모두 lock을 추가해주고, process.c에서도 load 함수에 lock을 추가하였다.

(Open syscall 예시)

*int* Open (*const* *char* \**file*) {

  check\_user\_vaddr(*file* - 8, *file*);

  if(*file* == NULL) Exit(-1);

  lock\_acquire(&filesys\_lock);

*struct* file \*f = filesys\_open(*file*);

  if(f == NULL){

    lock\_release(&filesys\_lock);

    return TID\_ERROR;

  }

  else {

    if(strcmp(thread\_current()->name, *file*) == NULL) file\_deny\_write(f);

    for(*int* i = 3; i < 128; i++){

      if(thread\_current()->fd\_table[i] == NULL) {

        thread\_current()->fd\_table[i] = f;

        lock\_release(&filesys\_lock);

        return i;

      }

    }

  }

  lock\_release(&filesys\_lock);

  return TID\_ERROR;

}

이렇게 lock을 구현하였는데도 같은 문제가 해결되지 않아서, spin lock을 걸어주기로 하였다. Filesys\_open 앞에 for문으로 약 천만번을 반복 (0.1초 대기)하니 이 문제가 해결되어 이번 프로젝트에서 요구하는 테스트케이스를 모두 통과할 수 있었다.

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

텍스트, 폰트, 스크린샷, 흑백이(가) 표시된 사진

자동 생성된 설명

다만 syn-read, syn-write가 통과하지 않았지만 이번 프로젝트에서 요구하지 않는 문제이고, 시간이 부족하여 완성은 하지 않고 남겨놓기로 했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명