**Pintos Project 4: Virtual Memory**

담당 교수 : 박성용 교수님

이름 / 학번 : 20191559 / 강상원

개발 기간 : 2023. 11. 20. ~ 2023. 12. 6.

1. **개발 목표**

가상 메모리 기능을 구현한다. Page Fault에 대해 더 reliable하게 한다. Page Table을 만들고, page fault 처리 로직을 작성한다. pseudo-LRU 정책을 도입하여 디스크에서 paging, swap을 하도록 한다. 또한 Stack growth가 되게 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
      * Page Table & Page Fault Handler

Page Fault에 대해 더 reliable하게 한다: 부적절한 접근에 대해 디스크에 해당 페이지가 있는지 확인하는 로직을 추가하고 이를 불러오는 작업을 한다. 이는 Page Fault Handler에 의해 동작이 이루어지게 된다.

* + - Disk Swap

Physical memory가 부족한 상황일 때 디스크에서 page를 얻어야 한다.(swap-out) Disk Swap algorithm에 대해 앞서 언급한 pseudo-LRU 정책을 도입하게 된다.

* + - Stack Growth

스택에 접근하고자 하는 요청 중 페이지 폴트가 일어난다면 스택 growth를 통해 크기를 조절하고, 적절한 page를 할당해 주도록 한다.

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

커널 영역이라던가, 접근 불가한 영역에 접근 시도할 때 페이지 폴트가 발생하였다. 또한 swap out이 이루어져 물리 메모리에 존재하지 않고, 디스크에 저장된 페이지 접근에 대해서도 페이지 폴트가 발생한다. 이 때 page fault handler는 디스크에 있는 경우를 찾아내 해당 페이지를 검색해 낸다.

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

pseudo-LRU 정책 중 second chance algorithm을 사용한다. Access 여부를 체크하는 bit를 두고, 첫 번째는 봐주고, 두 번째 access시 bit 0을 확인하고 eviction을 진행한다. Load 후 수정되었는지도 확인하여, 만약 그런 경우 swap out하여 값을 갱신한다.

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

일단 페이지 폴트가 발생한 시점에서, 유효성을 확인하고 만약 가상 주소 entry를 찾을 수 없다면 stack growth를 담당하는 함수를 호출하여 stack을 늘려준다. 만약 더 할당할 수 없는 경우에는 예외를 발생시킨다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**
      * 11/20 ~ 11/23: Page Table & Page Fault Handler 개발
      * 11/24 ~ 11/28: Disk Swap 개발
      * 11/29 ~ 12/5: Stack Growth 개발
      * 12/6: 보고서 작성
   2. **개발 방법**
      * Page Table & Page Fault Handler

[vm/page.h]

struct *virtual\_page\_entr* {

unsigned long file\_offset; // Offset within the backing file.

unsigned long read\_bytes; // Number of bytes to read from the file.

unsigned long zero\_bytes; // Number of bytes to be zero-filled.

unsigned long swap\_index; // Swap slot index if the page is in swap space.

bool type; // Type of VM entry: 0: File / 1: Swap

bool is\_dirty; // True if the page has been modified since it was loaded.

bool is\_writable; // Indicates if the memory area is writable.

bool is\_in\_memory; // True if the page is loaded into physical memory.

void \*upage; // User virtual address of the page.

void \*vaddr; // Virtual address mapped by this entry.

struct *hash\_elem* elem; // Hash table element for thread's VM hash table.

struct *file* \*backing\_file; // File backing this VM entry, if any.

};

struct *virtual\_page\_entr* \*get\_virtual\_page\_entr\_by\_vaddr(void \**virtual\_address*); // Get a VM entry by its virtual address.

bool read\_file\_into\_memory(void \**kernel\_addr*, struct *virtual\_page\_entr* \**virtual\_page\_entr*); // Read a file into memory.

bool add\_virtual\_page\_entr(struct *hash* \**vm\_table*, struct *virtual\_page\_entr* \**virtual\_page\_entr*); // Add a VM entry to the VM hash table.

bool remove\_virtual\_page\_entr(struct *hash* \**vm\_table*, struct *virtual\_page\_entr* \**virtual\_page\_entr*); // Remove a VM entry from the VM hash table.

// Page structure representing a physical frame.

struct *page* {

void\* user\_addr; // User virtual address of the page.

struct *list\_elem* lru; // List element for the LRU list.

struct *frame* \*frame; // Frame containing the page.

struct *thread* \*owner\_thread; // Thread that owns the page.

void\* kernel\_addr; // Kernel virtual address of the page.

struct *virtual\_page\_entr* \*vme; // VM entry corresponding to the page.

};

virtual\_page\_entr struct에 virtual memory에 필요한 자료구조를 구현하였다. virtual address를 포함하여, 수정된 적 있는지 기록하는 dirty bit, file인지 swap인지 판단하는 flag, 물리 메모리에 올라와 있는지 기록하는 flag 등이다.

이를 handle하기 위해 get\_virtual\_page\_entr\_by\_vaddr, read\_file\_into\_memory와 add\_virtual\_page\_entr, remove\_virtual\_page\_entr 함수를 구현하였다. 각 함수의 기능은 함수의 이름에서도 유추할 수 있고, 상단 주석으로도 보기쉽게 정리해 놓았다.

일례로 remove\_virtual\_page\_entr 함수는 다음과 같이 hash\_delete를 호출하여 전달받은 virtual page entry를 삭제하는 역할을 한다.

bool remove\_virtual\_page\_entr(struct *hash* \**vm\_table*, struct *virtual\_page\_entr* \**virtual\_page\_entr*) {

bool is\_removed = hash\_delete(*vm\_table*, &*virtual\_page\_entr*->elem) != NULL;

free(*virtual\_page\_entr*);

return is\_removed;

}

* + - Disk Swap

initialize\_swap으로 swap 영역 초기화, read\_from\_swap 함수로 swap 공간에서 page를 읽는 로직을 작성하였고, write\_to\_swap으로 swap 영역에 작성한다. read, write 할 때 critical section 보호를 위해 lock은 필수이다.

[vm/swap.h]

void handle\_block\_io(bool *is\_read*, size\_t *swap\_index*, void \**physical\_addr*); // handle block io

void initialize\_swap(); // initialize swap table

void iterate\_swap(size\_t *swap\_index*, void \**aux*, bool *rw*); // iterate swap table

void read\_from\_swap(size\_t *swap\_index*, void \**physical\_addr*) ; // read from swap table

size\_t write\_to\_swap(void \**physical\_addr*); // write to swap table

[vm/frame.h]

struct *page*\* page\_allocation(enum *palloc\_flags* *flags*); // Allocate a page of memory to be used as a user page

bool page\_emplace\_LRU(struct *page* \**new\_page*); // Add a page to the LRU list

bool page\_out\_LRU(struct *page* \**target\_page*); // Remove a page from the LRU list

void free\_and\_remove\_page(void \**page\_kernel\_addr*); // Free a page and remove it from the LRU list

struct *list\_elem*\* rotate\_lru\_pointer(); // Rotate the LRU clock pointer

void evict\_pages\_from\_lru(); // Evict pages from the LRU list until a page can be allocated

bool should\_evict(struct *page* \**page*); // Determine if a page should be evicted

void evict\_page(struct *page* \**page*); // Evict a page from physical memory

void advance\_lru\_clock(); // Advance the LRU clock pointer

void init\_LRU(void); // Initialize the LRU list and clock pointer

유저 페이지로 사용될 페이지를 할당하는 page\_allocation(), LRU second chance list에 페이지를 추가하는 page\_emplace\_list, 제거하는 page\_out\_LRU, 물리 메모리에서 페이지를 evict하는 evict\_page 함수와 같은 함수를 작성하였다. 각 함수의 역할은 주석으로 알기쉽게 작성되어 있으며 각 구현 세부 내용은 IV.B에 작성하겠다.

* + - Stack Growth

페이지 폴트가 났을 때 유효한 주소였다면 expand\_stack 함수를 호출하여 처리하도록 하였다.

[userprog/process.h]

bool is\_valid\_stack\_expansion(void \**addr*);

struct *virtual\_page\_entry* \*create\_virtual\_page\_entry(void \**addr*);

struct *page* \*allocate\_and\_setup\_page(struct *virtual\_page\_entr* \**vme*);

bool add\_page\_to\_process\_vm(struct *virtual\_page\_entr* \**vme*, struct *page* \**stack\_page*);

bool expand\_stack(void \**addr*); // Expand the stack

static bool install\_page (void \**upage*, void \**kpage*, bool *writable*);

expand\_stack 함수에서는 virtual page entry를 생성하고, allocate하는 역할을 한다. is\_valid\_stack\_expansion()는 해당 주소가 할당 가능한 유효성을 체크한다. install\_page로 유저 가상 주소 UPAGE를 커널 주소 KPAGE로 바뀐 매핑을 페이지 테이블에 추가한다. Stack expansion에 관한 자세한 로직은 IV.B에 정리하였다.

1. **연구 결과**
   1. **Flow Chart**
      * Page Table & Page Fault Handler

텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명

* + - Disk Swap

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

* + - Stack Growth

텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
     + Page Table & Page Fault Handler

[vm/page.c]

struct *virtual\_page\_entr* \*get\_virtual\_page\_entr\_by\_vaddr(void \**virtual\_address*) {

struct *virtual\_page\_entr* search\_entry;

search\_entry.vaddr = pg\_round\_down(*virtual\_address*);

struct *hash\_elem* \*found\_entry = hash\_find(&thread\_current()->vm, &search\_entry.elem);

return found\_entry ? hash\_entry(found\_entry, struct virtual\_page\_entr, elem) : NULL;

}

bool read\_file\_into\_memory(void \**kernel\_addr*, struct *virtual\_page\_entr* \**virtual\_page\_entr*) {

*size\_t* bytes\_read = file\_read\_at(*virtual\_page\_entr*->backing\_file, *kernel\_addr*, *virtual\_page\_entr*->read\_bytes, *virtual\_page\_entr*->file\_offset);

if (bytes\_read == *virtual\_page\_entr*->read\_bytes) {

memset(*kernel\_addr* + *virtual\_page\_entr*->read\_bytes, 0, *virtual\_page\_entr*->zero\_bytes);

return true;

}

return false;

}

bool add\_virtual\_page\_entr(struct *hash* \**vm\_table*, struct *virtual\_page\_entr* \**virtual\_page\_entr*) {

return hash\_insert(*vm\_table*, &*virtual\_page\_entr*->elem) == NULL;

}

bool remove\_virtual\_page\_entr(struct *hash* \**vm\_table*, struct *virtual\_page\_entr* \**virtual\_page\_entr*) {

bool is\_removed = hash\_delete(*vm\_table*, &*virtual\_page\_entr*->elem) != NULL;

free(*virtual\_page\_entr*);

return is\_removed;

}

get\_virtual\_page\_entr\_by\_vaddr, read\_file\_into\_memory와 add\_virtual\_page\_entr, remove\_virtual\_page\_entr 함수 등을 구현하였다.

get\_virtual\_page\_entr\_by\_vaddr 함수는 pg\_round\_down으로 해당하는 페이지를 찾아 반환해 주고, read\_file\_into\_memory는 파일을 읽은 다음 그 바이트 수만큼 memset을 통해 할당, 결과를 bool 형식으로 반환한다. remove\_virtual\_page\_entr 함수는 hash\_delete를 호출하여 전달받은 virtual page entry를 삭제하는 역할, 반대로 add\_virtual\_page\_entr는 hash\_insert 함수를 호출하여 기능을 구현하였다.

* + - Disk Swap

read\_from\_swap 함수로 swap 공간에서 page를 읽는 로직을 작성하였고, write\_to\_swap으로 swap 영역에 작성한다. read, write 할 때 critical section 보호를 위해 lock은 필수이다.

[vm/swap.c]

struct *lock* lock\_swp;

struct *bitmap* \*map;

void read\_from\_swap(*size\_t* *swap\_index*, void \**physical\_addr*) {

lock\_acquire(&lock\_swp);

if (bitmap\_test(map, *swap\_index*)) {

handle\_block\_io(true, *swap\_index*, *physical\_addr*);

bitmap\_flip(map, *swap\_index*);

}

lock\_release(&lock\_swp);

}

*size\_t* write\_to\_swap(void \**physical\_addr*) {

lock\_acquire(&lock\_swp);

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

if (swap\_index != BITMAP\_ERROR) handle\_block\_io(false, swap\_index, *physical\_addr*);

lock\_release(&lock\_swp);

return swap\_index;

}

[vm/frame.c]

struct *list\_elem*\* rotate\_lru\_pointer() {

// Early exit if the list is empty

if (list\_empty(&lru\_list)) return NULL;

struct *list\_elem* \*next\_elem;

// If lru\_clock is NULL, start from the beginning of the list

if (!lru\_clock) {

next\_elem = list\_begin(&lru\_list);

} else {

// Otherwise, get the next element in the list

next\_elem = list\_next(&lru\_clock->lru);

if (next\_elem == list\_end(&lru\_list)) next\_elem = list\_begin(&lru\_list);

}

// Update lru\_clock to the next element

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

return next\_elem;

}

void handle\_dirty\_page(struct *page* \**lru\_page*, bool *dirty*) {

if (*lru\_page*->vme->type || *dirty*) {

*lru\_page*->vme->type = true;

*lru\_page*->vme->swap\_index = write\_to\_swap(*lru\_page*->kernel\_addr);

}

}

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

자동 생성된 설명rotate\_lru\_pointer를 사용해 lru clock이 list를 따라 회전하면서 second chance algorithm을 동작시키도록 작성하였다. 이에 대한 흐름도는 강의자료를 참조하였다.

수정된 경우: dirty bit에 대해 핸들링 해서 값을 갱신해주는 로직을 작성하였다.

* + - Stack Growth

[userprog/exception.c]

if (!page\_entr) {

// If the fault address is within stack heuristic range, try expanding the stack

if (fault\_addr >= *f*->esp - 32) load\_success = expand\_stack(fault\_addr);

else EXIT(-1);

} else {

// Handle memory fault for existing page entry

load\_success = memory\_fault\_handler(page\_entr) || page\_entr->is\_in\_memory;

}

exception.c에서 기존 로직과 다르게, 페이지 폴트가 났을 때 유효한 주소였다면 expand\_stack 함수를 호출하여 처리한다. 해당 함수의 내용은 다음과 같다.

[userprog/process.c]

bool expand\_stack(void \**addr*) {

void \*rounded\_addr = pg\_round\_down(*addr*);

if (!is\_valid\_stack\_expansion(rounded\_addr)) return false;

struct *virtual\_page\_entry* \*vme = create\_virtual\_page\_entry(rounded\_addr);

if (!vme) return false;

struct *page* \*stack\_page = allocate\_and\_setup\_page(vme);

if (!stack\_page) {

free(vme);

return false;

}

return add\_page\_to\_process\_vm(vme, stack\_page);

}

process.c에 선언한 expand\_stack 함수에서, stack expansion이 valid한지 체크한 뒤에, virtual page entry를 생성하고, allocate한다. 주소는 rounding 방식으로 가장 가까운 페이지 바운더리 위치를 가지고 온다.

* 1. **시험 및 평가 내용**

page\_merge\_mm을 제외한 모든 테스트를 통과하였다.

텍스트, 전자제품, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명