**Pintos Project 4: Virtual Memory**

담당 교수 : 박성용

이름 / 학번 : 20181687 / 정지석

개발 기간 : 11/17~11/26

1. **개발 목표**

지금까지 구현한 pintos 환경에서는 page fault가 일어날 경우 프로그램이 바로 종료가 되었다. 이번 프로젝트에서는 page fault handler를 구현하여 위와 같은 상황이 발생했을 때 프로그램이 좀더 안정적으로 실행될 수 있도록 한다.

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

프로세스 실행 중에 page fault가 발생하면 kernel에서 page fault handler를 호출하여 page fault를 handling한다.

만약 memory reference가 valid하다면 frame을 받아서 page를 저장해준 뒤, frame으로부터 data를 가져온다. 그 후 physical address에 virtual address를 할당해준다.

만약 memory reference가 invalid하다면 해당 process를 종료하고 모든 자원들을 free한다.

위의 과정을 통해 page fault가 발생하더라도 프로그램이 바로 종료되지 않고 계속 실행될 수 있다.

* 1. Disk Swap

Disk swap을 구현할 경우 process에 할당해 줄 physical memory가 부족할 경우 disk로 swap out이 발생한다. 이때 어떤 page를 swap할지 결정해야 하는데, 이는 page replacement algorithm을 통해 결정한다. Swap table을 작성하여 현재 사용하는 slot과 빈 slot을 관리한다.

* 1. Stack Growth

이전까지는 program의 크기가 제한되어 stack 영역이 page의 범위를 벗어나 두 개 이상의 page에 걸치게 되면 error가 발생했으나, 이번 프로젝트를 통해 stack 영역보다 크기가 클 경우 page를 할당하여 stack이 grow할 수 있게 하였다.

* 1. **개발 내용**

1. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

찾으려는 page가 memory 안에 존재하지 않을 경우(invalid) page fault가 발생한다. Page fault가 발생하면 해당 address를 찾아내서 valid한 접근인지 아닌지를 판단한다. 만약 valid 하다면, 해당 frame으로부터 data를 가져오고 page를 저장한 뒤, physical memory에 virtual address를 할당해 준다.

Invalid 할 경우에는 growable 한지를 판단하여 growable 할 경우에는 stack 공간을 확장한 뒤 process를 재개하고, 아닐 경우에는 process를 종료하고 모든 resource 들을 free 해준다.

2. Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술

LRU와 유사한 second chance policy를 사용하여 page replacement algorithm을 구현한다. Page가 memory에 처음 load 되었을 때에는 reference bit을 1로 설정한다. 그 후 page가 page fault handler에 의해 replace 될 때 reference bit이 0이라면 page를 replace한다. 만약 1이라면, reference bit을 0으로 바꿔주고 해당 page에게 기회를 한번 더 준다(second chance).

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

Page fault가 발생했을 때, 해당 주소가 stack access인 경우에만 stack growth라고 가정한다. Push 연산인 경우의 4byte 이거나 pusha 연산일 때 32byte 보다 작은 경우 일 때 stack growth라고 판단한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11/17~11/25: page fault handler/stack growth 구현

11/26: 보고서 작성

* 1. **개발 방법**

vm 폴더에 page.c와 page.h 파일을 새로 생성해 주어야 한다.

* + 수정해야하는 소스코드

Makefile.build

새로 생성한 page.c 파일을 컴파일하기 위해 makefile 파일을 수정해 주어야 한다.

Userprog/exception.c

Page fault handling을 위해 page\_fault() 함수 부분을 수정해 주어야 한다.

Threads/synch.c

Sema\_up() 함수에서 오류가 일어나는 부분을 수정해 주어야 한다.

* + 수정하거나 추가해야 하는 자료구조

vm/page.h

virtual address page들을 관리하기 위해 vm\_entry라는 이름의 struct를 생성해주어야 한다.

* + 수정하거나 추가해야 하는 함수

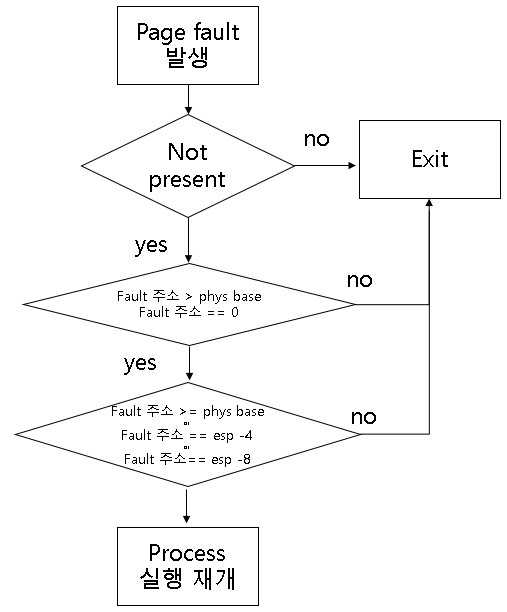
Vm/page.c

Vm\_init(), vm\_hash\_func(), vm\_less\_func()

Virtual page들을 관리하기 위한 함수들을 작성해주어야 한다.

1. **연구 결과**
   1. **Flow Chart**

Stack growth



* 1. **제작 내용**

**\*Makefile.build**

Vm 폴더에 page.c, page.h 파일을 생성하였기 때문에 makefile.build 파일에 아래의 내용을 추가하여 새로 생성한 파일들이 컴파일 될 수 있도록 해준다.

vm\_SRC = vm/page.c

**\*vm/page.h**

struct vm\_entry

{

    uint8\_t type;

    void \*vaddr;

    bool writable;

    bool is\_loaded;

    struct file \*file;

    struct list\_elem mmap\_elem;

    size\_t offset;

    size\_t read\_bytes;

    size\_t zero\_bytes;

    struct hash\_elem elem;

};

새로 생성한 page.h 파일에 virtual memory entry들을 관리해주기 위한 struct를 작성해 주어야한다.

**\*vm/page.c**

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

{

    ASSERT(vm != NULL);

    hash\_init(vm, vm\_hash\_func, vm\_less\_func, 0);

}

Hash\_init() 함수를 사용하여 hash table을 초기화하는 역할을 해준다.

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

{

    ASSERT(e != NULL);

*return* hash\_int(hash\_entry(e, struct vm\_entry, elem)->vaddr);

}

Vm\_entry의 vaddr 값을 받은 뒤, hash\_init() 함수를 사용하여 hash 값을 return하는 역할을 하는 함수이다.

static bool vm\_less\_func(const struct hash\_elem \**x*, const struct hash\_elem \**y*)

{

    ASSERT(x != NULL);

    ASSERT(y != NULL);

    int X = hash\_entry(x, struct vm\_entry, elem)->vaddr;

    int Y = hash\_entry(y, struct vm\_entry, elem)->vaddr;

*return* X < Y;

}

입력으로 들어온 두 hash\_elem의 vaddr 값을 비교한다.

**\* threads/synch.c**

void

sema\_up (struct semaphore \**sema*)

{

  enum intr\_level old\_level;

  ASSERT (sema != NULL);

  old\_level = intr\_disable ();

*if* (!list\_empty (&sema->waiters))

  {

    list\_sort(&sema->waiters, priority\_compare, 0);

    thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters),

                                struct thread, elem));

  }

  sema->value++;

*//ready\_running\_priority();*

  intr\_set\_level (old\_level);

}

Sema\_up() 함수에서 thread yield가 발생할 시에 지속적으로 error가 발생하여 thread yield를 실행하는 ready\_running\_priority() 함수를 주석 처리해주었다.

**\*userprog/exception.c**

static void

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

{

  bool not\_present; */\* True: not-present page, false: writing r/o page. \*/*

  bool write; */\* True: access was write, false: access was read. \*/*

  bool user; */\* True: access by user, false: access by kernel. \*/*

  void \*fault\_addr; */\* Fault address. \*/*

*/\* Obtain faulting address, the virtual address that was*

*accessed to cause the fault.  It may point to code or to*

*data.  It is not necessarily the address of the instruction*

*that caused the fault (that's f->eip).*

*See [IA32-v2a] "MOV--Move to/from Control Registers" and*

*[IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception*

*(#PF)". \*/*

  asm ("movl %%cr2, %0" : "=r" (fault\_addr));

*/\* Turn interrupts back on (they were only off so that we could*

*be assured of reading CR2 before it changed). \*/*

  intr\_enable ();

*/\* Count page faults. \*/*

  page\_fault\_cnt++;

*/\* Determine cause. \*/*

  not\_present = (f->error\_code & PF\_P) == 0;

  write = (f->error\_code & PF\_W) != 0;

  user = (f->error\_code & PF\_U) != 0;

  void \*user\_page = pg\_round\_down(fault\_addr);

  void \*kernel\_page = palloc\_get\_page(PAL\_USER);

*if* (!not\_present)

      exit(-1);

*if* (fault\_addr <= 0)

      exit(-1);

*if* (fault\_addr > PHYS\_BASE)

      exit(-1);

*if* (!(f->esp <= fault\_addr || fault\_addr == f->esp - 4 || fault\_addr == f->esp - 32))

      exit(-1);

   pagedir\_get\_page(thread\_current()->pagedir, user\_page);

   pagedir\_set\_page(thread\_current()->pagedir, user\_page, kernel\_page, true);

*return*;

}

Page\_fault() 함수에 stack growth를 해주는 부분을 추가한다. Page가 memory에 present하지 않을 경우 그대로 exit한다.

그런 후 fault address가 0보다 큰지, physical base 아래에 있는지 여부를 확인하고 그렇지 않을 경우 그대로 exit한다.

마지막으로 후 fault address를 esp와 비교하여 stack growth인지를 확인하고, 맞다면 stack을 늘려준다.

Pagedir\_get\_page() 함수와 Pagedir\_set\_page() 함수를 사용하여 virtual address인 page를 통해서 실제 physical address를 찾아낸다.

* 1. **시험 및 평가 내용**

Stack growth 관련 test case는 모두 잘 작동이 되는 것을 확인할 수 있다.

