**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재 교수님

이름 / 학번 : 김성일 / 20171612

개발 기간 : 2021.11.05 ~ 2021.11.27

1. **개발 목표**

현재 pintOS는 Virtual Memory가 구현되어있지 않습니다. 그렇기에, 이번 프로젝트에서는 Virtual Memory를 구현하고 Stack Growth와 Second Chance 알고리즘을 이용하여 페이지를 Disk로 Swap하는 기능을 추가하는 것을 목표로합니다.

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

Page Fault Handling을 위해 기존 Page Table에 필요한 정보를 추가하고, 그에 따른 Page Table을 개발합니다.

* 1. Disk Swap

물리 메모리에 공간이 부족하다면 Disk로 Swap을하여 페이지를 Swap파일로 내보냅니다. Swap 정책은 LRU Cache를 근사하는 Second Chance 알고리즘을 사용하며 Swap table을 통해 Swap disk가 현재 사용중인 슬롯과 빈 슬롯을 구분하여 관리하게 됩니다.

* 1. Stack Growth

Stack의 공간이 추가적으로 필요하게 되면 페이지를 추가적으로 할당하는 Stack Growth를 구현합니다. 현재 스택의 크기를 초과하는 주소에 접근이 발생했을 때, 유효한 스택 접근인지 Segmentation Fault인지 판별해야합니다. Stack은 최대 8MB 까지 확장할 수 있게 제한합니다.

* 1. **개발 내용**

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

말 그대로 메모리에 존재하지 않는 페이지에 접근하게 되었을 때 page fault가 발생합니다.

하지만 요구한 주소가 swap 되어 있거나, 확장 가능하다면 아래에서 구현 할 Disk Swap과 Stack Growth를 통해 handling하게 됩니다.

위의 두가지 상황이 아니라면 exit를 통해 종료하게 됩니다.

**2. Disk swap 발생 시 사용한 page replacement algorithm**

Second Chance 알고리즘을 사용할 예정입니다.

이 알고리즘은 FIFO를 기반으로하며 페이지를 선택합니다. 페이지가 선택되었을 때 그 페이지를 즉시 교체하지않고 refrence bit를 체크하여 bit가 0이라면 교체하고 1이라면 0으로 set 한 뒤 한 번 더 기회를 주어 나중에 선택 되었을 때 교체되도록합니다. 이렇게 한 번 더 기회를 주기 때문에 Second Chance 알고리즘으로 불리며 Circular queue를 이용하여 구현하는 것이 보통의 방법입니다.

이 알고리즘은 LRU-Approximation 알고리즘으로 알려져있습니다.

**3. Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법**

먼저 Stack이 확장할 수 있는 크기를 최대 8mb로 제한 할 것이기 때문에, boundary 체크를합니다.

정확한 체크는 다음과 같습니다.

**PHYS\_BASE – 8mb 와 PHYS\_BASE 사이에 fault address가 존재하는지 체크합니다.**

그 후 더 정확하게 확장 할 수 있는 접근인지 아래의 조건을 체크합니다.

**(1) esp - 32 가 fault address 보다 작은가?**

위 조건은 PUSHA 명령어 때문에 사용됩니다. 핀토스 매뉴얼 4.3.3 Stack Growth에도 언급되어 있듯이 PUSHA 명령어를 사용시 한 번에 32byte를 push하게 되는데 이는 제대로 된 stack access가 아니기 때문에 위 조건을 체크해야합니다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

Page Table과 Page Fault Handler 구현 시도 : 11/5 ~ 11/12

Disk Swap 구현 시도 : 11/13 ~ 11/20

Stack Growth 구현 : 11/21 ~ 11/26

보고서 작성 및 제출 : 11/27

Stack Growth를 제외한 다른 구현 요소는 구현을 하려하였지만 끝내 완전히 구현하지 못하여 시도로 그쳤습니다.

* 1. **개발 방법**

**B-1. Page Table, Page Fault Handler**

vm/page.h에 virtual memory를 위한 새로운 구조체와 필요한 함수들을 선언합니다.

이 때 필요한 여러 함수들을 vm/page.c에서 구현합니다.

그 후 userprog/process.c 에서 load시 page를 초기화하고 적절한 방법을 통해 설정해줍니다.

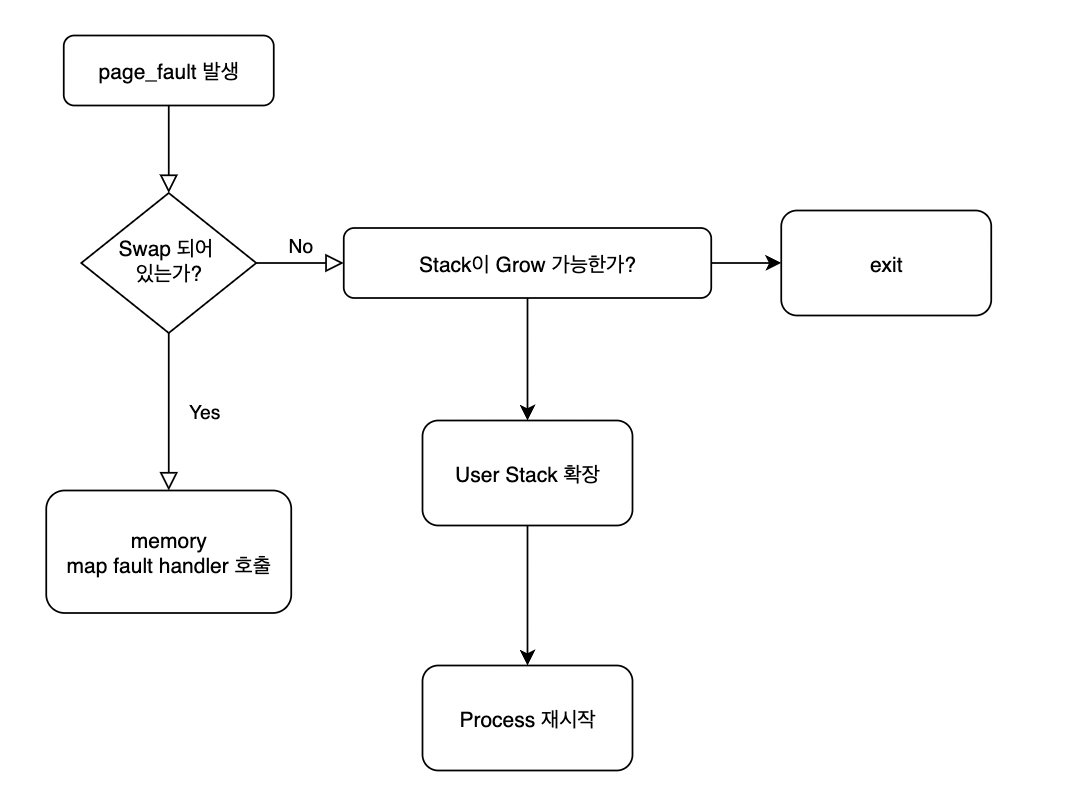
**B-2. Disk Swap**

앞서 II-B-2에서 설명한 LRU Approximate 알고리즘 중 하나인 Second Chance Algorithm을 구현하여 이것을 Swap policy로 사용합니다. Demand paging시 다시 메모리에 로드 되도록해야하며 이것을 vm/swap.h, vm/swap.h에서 선언하고 구현합니다. 그 후 userprog/process.c, userprog/syscall.c 에서 적절히 사용토록해야합니다.

**B-3. Stack Growth**

userprog/exception.c의 page\_fault 함수를 수정하여 Stack Growth에 해당하는 page fault를 처리하고 최대 8MB까지 확장 할 수 있는지와, 앞서 II-B-3 에서 언급한 Stack Growth 확장 조건을 체크하여 확장을 진행합니다.

1. **연구 결과**
   1. **Flow Chart**



* 1. **제작 내용**

**B-0. 사전 작업**

이전 프로젝트를 가져와 그대로 진행하게 되면, threads/synch.c 의 sema\_up 함수에서 thread\_yield()를 하는 부분이 Kernel Panic을 일으키게 됩니다.

이를 수정하기 위해, 아래와 같이 thread\_yield()를 제거하였습니다.

B-0-1. threads/synch.c의 sema\_up 함수 수정

void sema\_up(struct semaphore \*sema)

{

enum intr\_level tmp;

struct list\_elem \*elem\_iter, \*max\_elem;

struct thread \*thread\_iter, \*max\_thread;

ASSERT(sema != NULL);

tmp = intr\_disable();

if (list\_empty(&sema->waiters) == 0)

{

elem\_iter = list\_begin(&sema->waiters);

max\_elem = elem\_iter;

max\_thread = list\_entry(elem\_iter, struct thread, elem);

elem\_iter = list\_next(elem\_iter);

while (elem\_iter != list\_end(&sema->waiters))

{

thread\_iter = list\_entry(elem\_iter, struct thread, elem);

if (thread\_iter->priority > max\_thread->priority)

{

max\_thread = thread\_iter;

max\_elem = elem\_iter;

}

elem\_iter = list\_next(elem\_iter);

}

list\_remove(max\_elem);

thread\_unblock(max\_thread);

}

sema->value++;

intr\_set\_level(tmp);

//thread\_yield();

}

프로젝트3에서 동기화를 위해 추가하였던 thread\_yield() 함수가 Kernel panic을 발생시키는 이유였기 때문에, 위와 같이 thread\_yield() 함수를 제거하였습니다.

**B-1. Stack Growth 구현**

이전 프로젝트를 진행하면서, userprog/exception.c의 page\_fault 함수에서 여러 bit(flag)를 통해 exit 처리하였던 코드를 Stack 확장가능한 조건을 체크하기위해 ( 즉, page fault handling을 위해 ) 모두 삭제하였습니다.

not\_present가 true이고, Stack이 확장 가능한 조건일 때 Stack을 확장하도록 하였습니다.

B-1-1. userprog/exception.c의 page\_fault 함수 수정

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;

if (not\_present)

{

// EIGHT\_MB == 8 \* 1024 \* 1024

if ((PHYS\_BASE - EIGHT\_MB <= fault\_addr && fault\_addr < PHYS\_BASE) &&

(f->esp - 32 <= fault\_addr))

{

uint32\_t \*pd = thread\_current()->pagedir;

const void \*upage = pg\_round\_down(fault\_addr);

pagedir\_get\_page(pd, upage);

pagedir\_set\_page(pd, upage, palloc\_get\_page(PAL\_USER), 1);

return;

}

}

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);

}

위와 같이 not\_present bit를 체크하여 true라면 Stack은 최대 8MB까지 확장가능하므로 fault\_addr이 PHYS\_BASE – 8MB와 PHYS\_BASE 사이에 있는지를 먼저 체크합니다.

( EIGHT\_MB 는 8\*1024\*1024 의 값을 갖는 int형 변수입니다. )

그리고 그 후 더 정확하게 확장 할 수 있는지 아래의 조건을 추가로 체크합니다.

esp - 32 가 fault\_addr 보다 작은가?

위 조건은 PUSHA 명령어가 32 byte를 한꺼번에 push하기 때문에 제대로 된 stack access인지 체크하기 위해 사용합니다.

이 모든 조건을 만족하면 현재 쓰레드의 pg\_round\_down을 통해 fault\_addr에 근접한 주소를 UPAGE로 사용하여 pagedir\_get\_page, pagedir\_set\_page 함수를 순차적으로 호출하여 스택을 확장합니다.

**B-2. Page Table & Page Fault Handler**

B-1의 exception.c를 수정하여 not\_present일 때 stack이 확장 가능한지를 체크하여

Stack Growth 부분의 page fault handling을 하였습니다.

Disk Swap은 구현하지 못하였기 때문에 하지 못하였습니다.

**B-3. Disk Swap**

구현하지 못하였습니다.

* 1. **시험 및 평가 내용**
* 텍스트이(가) 표시된 사진

  자동 생성된 설명

위와 같이 아래의 5개의 테스트를 제외하고는 모두 통과하였습니다.

**page-linear**

**page-merge-seq**

**page-merge-par**

**page-merge-stk**

**page-merge-mm**