**Pintos Project 4: Virtual Memory**

담당 교수 : 박성용 교수님

이름 / 학번 : 전찬 / 20201635

개발 기간 : 2022.11.21 – 2022.12.05

1. **개발 목표**

이번 pintos 프로젝트 4 에서는 Virtual Memory와 관련된 동작을 구현해야 한다. 이를 위해 Page Table 자료구조를 바탕으로 VM 에서 PM(Physical Memory)로의 전환, 혹은 해당 VM이 valid한지의 평가를 수행해야 한다. 이후에는 page fault handling을 구현해야 하며, 이 과정에서 발생할 수 있는 page swap from / to disk 을 구현해야 한다. 마지막으로 기존 pintos에서 1 page size로 구현되어 있는 stack 영역에 대해, 해당 stack 영역의 size가 증가할 수 있도록 구현해야 한다.

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

Page Table을 통해, 각 process의 code, data, stack, heap 영역을 Page 단위로 분할할 수 있다. 더 나아가 Page Directory을 통해 Page Table 또한 page 단위로 나눌 수 있다. 이러한 방식은 memory space을 추상화하여 효율적으로 사용할 수 있도록 돕는다.

Page Fault Handler의 구현을 통해, page fault가 발생했을 때, memory swap, stack growth, exit 등의 privileged instructions 의 수행을 kernel에서 상황에 맞게 다룰 수 있도록 해 준다.

* 1. Disk Swap

Physical memory에서 FIFO, LRU 등의 알고리즘을 바탕으로 선택된 victim은 Physical memory에서 제거해 주어야 한다. 따라서 해당 victim을 disk 상에 저장해야 한다. 또한 해당 victim로 다시 memory access가 일어날 때 해당 victim을 memory에 load 할 수 있어야 한다. 따라서 Disk Swap 과정이 필요하다.

* 1. Stack Growth

stack 공간이 추가로 필요할 때, 해당 stack의 size을 키우는 형식으로 stack growth을 구현할 수 있다. 이를 통해 기존에는 제한된 stack size의 stack이 존재했던 pintos가 더 큰 stack size을 가질 수 있으며, 이를 토대로 더 많은 작업을 수행할 수 있다.

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

Page fault는 PM에 존재하지 않는 어떠한 page에 접근할 때 발생하게 된다. 이 때, 해당하는 주소가 valid한 경우, 해당 page을 PM에 load해 주어야 하며, 만약 PM에 empty frame이 존재하지 않는 경우, disk swap을 통해 empty frame을 만들어 주어야 한다. 이와 별개로, 해당 주소가 growable region인 경우, pintos 상의 user stack의 크기를 키우는 작업을 수행해 주어야 한다.

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

이번 pintos 프로젝트 4 에서는 disk swap에 pseudo-LRU인 second chance 알고리즘을 사용한다. LRU 방식은 최적의 방법이지만, 미래의 page access 까지 파악하고 있어야 하기 때문에 비현실적이다. 따라서 reference bit을 이용한 second chance 알고리즘은 LRU을 실제로 구현하는 대안이 될 수 있다. 이 알고리즘은 아래와 같다.

메모리에 존재하는 page에 대해, FIFO linked list 을 만들어 낸다. 해당 page의 reference bit이 0이라면 해당 page을 disk swap 하며 empty frame을 만들어 낸다. 해당 page의 reference bit이 1이라면 0으로 변경한 이후, 다음 page로 넘어간다. page fault 없이 어떠한 frame 상의 page에 접근하는 경우에는, reference bit을 1로 설정해 준다.

이러한 알고리즘을 통해 최근에 access되지 않은 PM에 존재하는 swap시킬 victim page을 선택해낼 수 있다.

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

stack 확장 여부는 page fault가 발생한 fault address에 대해, 해당 fault address가 PHYS\_BASE와 PHYS\_BASE – 8MB 사이에 있는지 파악해야 한다. 또한 fault address >= f->esp – 32 조건을 추가해 주어야 한다. 위 조건은 80x86의 PUSH, PUSHA instructions을 고려한 것이다. 80x86 상 PUSH는 4byte, PUSHA는 32byte의 push을 수행하며, 이는 각각 f->esp가 가리키고 있는 pointer 주소의 -4, 혹은 -32 값에서 page fault을 발생시킨다. 따라서 위 조건을 추가하여 stack에 한 번에 최대 push size을 제한하는 page fault handler을 구현해낼 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11.20 – 11.21 : userprog/make check 실행 오류 수정

11.21 – 11.22 : stack growth 구현

11.23 – 12.03 : 구현 레포트 작성 및 disk swap 과정 구현 시도

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

기존의 pintos는 page fault handler가 수행되면 exit을 통해 해당 프로세스를 종료하는 형식으로 구현되어 있다. 이번 프로젝트 4는 Page fault을 handling하기 위해, 해당 page fault handler에 대해, valid한 경우 disk swap을, invalid하지만 growable region 영역인 경우 stack growth을 수행해야 한다. 따라서 disk swap을 수행할 수 있는 함수인 handle\_mm\_fault 을 구현해야 한다. 이후에는 stack growth을 구현하기 위한 코드를 추가해 주어야 한다.

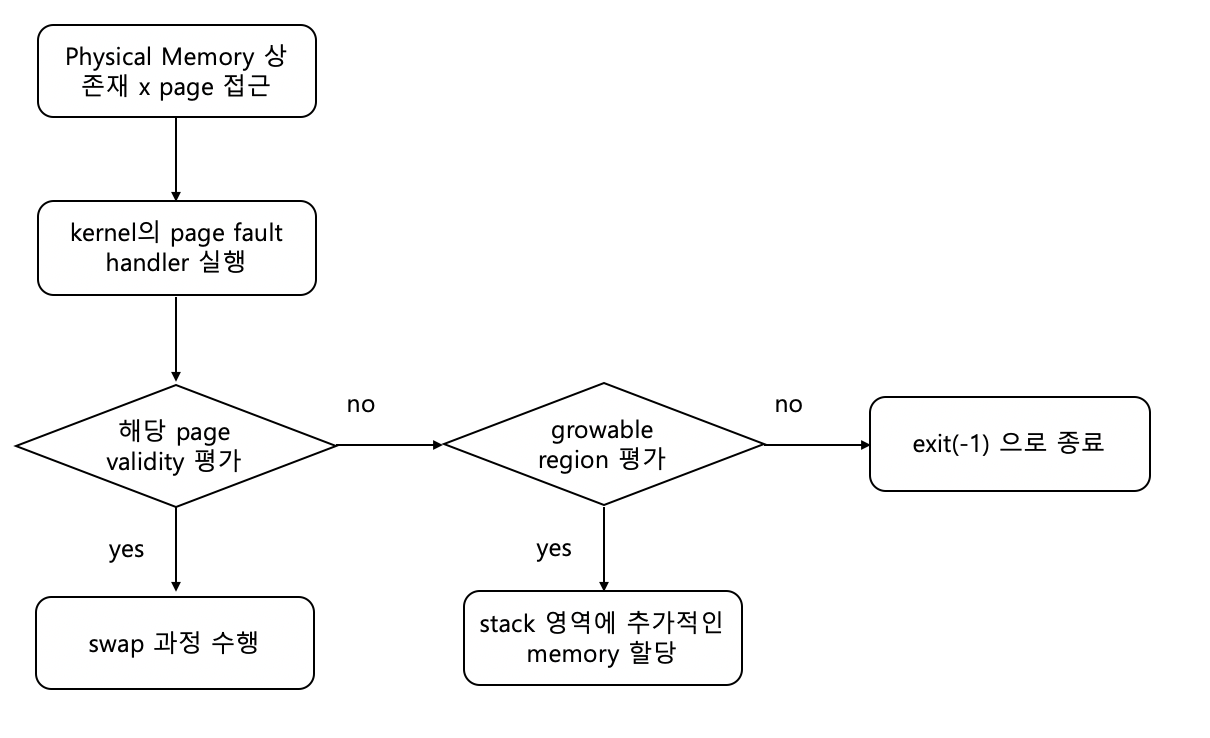
* Disk swap 발생 시 사용한 page replacement algorithm

Disk swap과 replacement algorithm을 구현하기 위해, 이전 page table entry format에는 존재하지 않는 reference bit 등을 추가로 설정해야 한다. 또한 현재 PM 상에 존재하는 page의 reference bit을 확인하기 위해 해당 page 들을 linked list로 연결하고 있어야 한다. 마지막으로 위 자료구조 등을 바탕으로 PM 상에 존재하는 page에 대해, victim을 선택하는 page replacement algorithm을 직접 구현해야 한다.

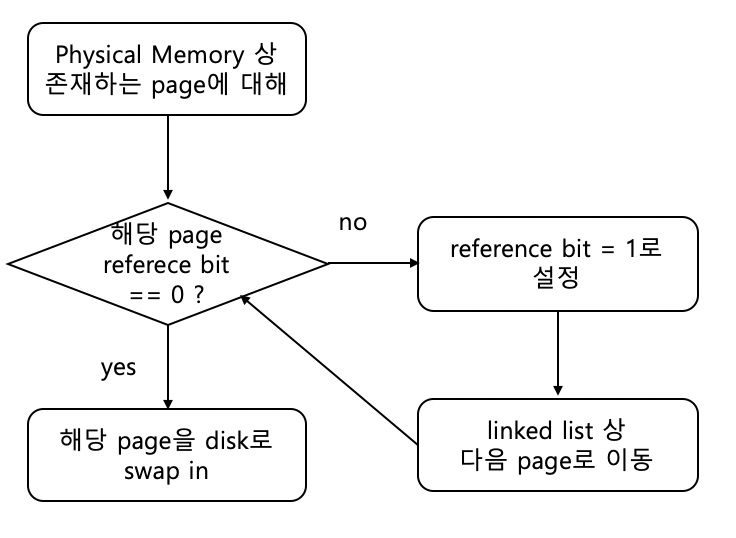
* Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법

page fault handler에서 위에서 설명한 조건에 해당하는 경우, stack growth을 수행해야 한다. 이를 위해 실제 PM에서 empty frame을 얻어와야 하며, 해당 empty frame을 추가된 stack에 연결시켜야 한다. 이는 pagedir 관련 함수를 통해 구현해낼 수 있다.

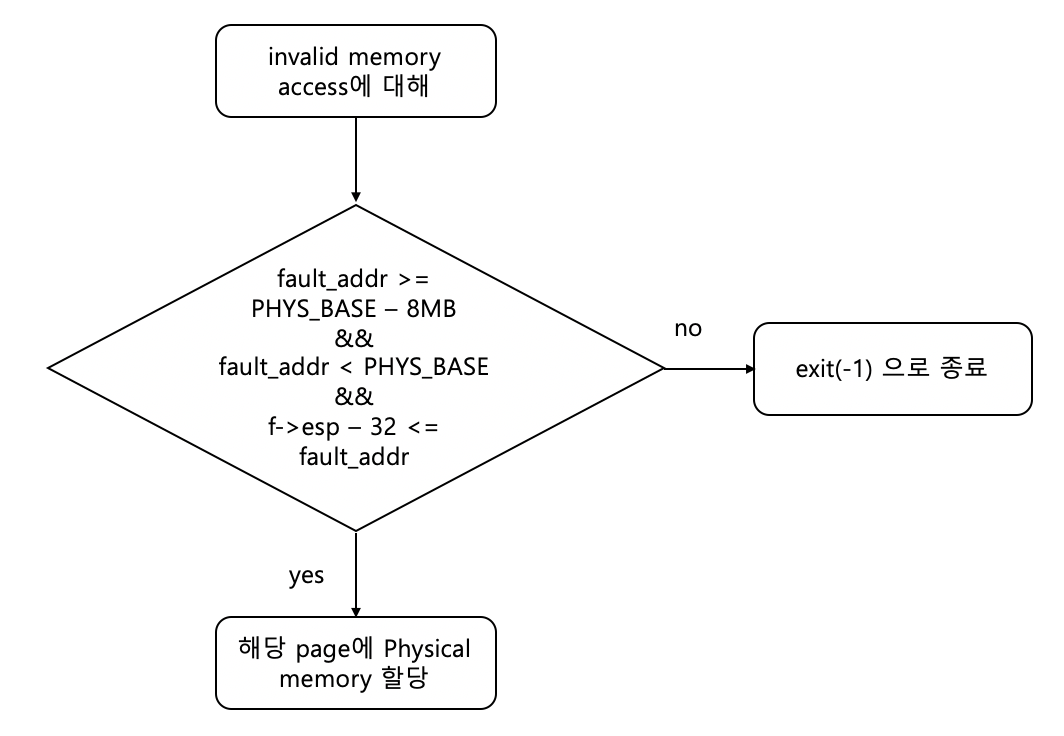
1. **연구 결과**
   1. **Flow Chart**



<Page fault handler의 flowchart>



<replacement algorithm의 flowchart>



<stack growth의 flowchart>

* 1. **제작 내용**

첫 번째로, 기존 프로젝트 1~3을 거친 pintos에 대해 userprog/make check을 실행하면 모든 test에 대해 오류가 발생한다. 이는 이전에 구현했던 threads/synch.c의 sema\_up 수행에서 thread\_yield을 수행하며 생기는 커널 패닉 문제이다. 따라서 thread\_yield 부분을 제거해 준다. 이는 아래와 같다.

//thread\_yield();

//intr\_yield\_on\_return();

<threads/synch.c의 sema\_up 함수 변경>

위 수정을 통해 프로젝트 1~2에서 구현했던 userprog / syscall 등이 정상적으로 작동할 수 있다.

이후에는 page fault handler의 stack growth을 구현해 주었다. 위에서 설명한 stack 확장 여부를 판단하는 방법에 따라 구현한 stack growth는 아래와 같다.

if(fault\_addr >= PHYS\_BASE - 8\*1024\*1024 && 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), true);

return;

}

<userprog/exception.c 의 page fault 수정 부분>

위 코드는 현재 프로세스의 pagedir을 pd에 저장하며, fault\_addr에 해당하는 page인 upage에 대해, 해당 page을 실제 PM에 set 해주는 과정을 구현한 것이다. 이때 pagedir\_get\_page, pagedir\_set\_page 코드 내부에서 ASSERT을 통해 error check을 수행하기 때문에, 별다른 handling이 필요하지는 않다. 이를 통해 stack growth을 구현해낼 수 있다.

이후에는 disk swap 과정을 구현하기 위해 생각한 부분이다. 하지만 구현을 성공하지는 못했다.

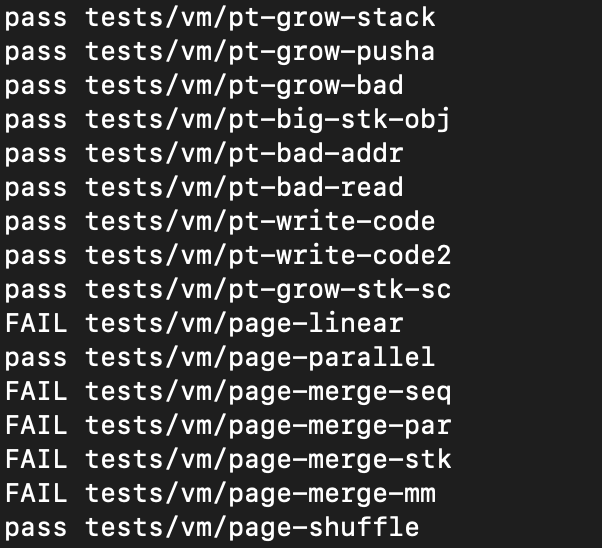
첫 번째로, 해당 page table의 각 page가 frame에 load 되어 있을 때, 해당 page들 사이에서 circular linked list을 구현해야 한다. 하지만 pintos는 start, end가 존재하는 linked list만 지원하기 때문에, linked list을 바탕으로 frame에 존재하는 page을 저장하는 형태를 구현하려 했다.

두 번째로, page fault가 발생했을 때, 해당 page가 첫 번째로 memory load 되는 것인지, 혹은 이미 load 되었다가 disk swap으로 인해 현재는 disk 상에 존재하는지를 파악해야 할 것 같다. 이를 구현하기 위해서도 추가적인 자료구조가 필요할 것 같으며, 각 thread의 page table에 설정해 초기 값으로 첫 load 이전에 0, 이후에는 1로 설정하는 형태로 구현해야 할 것 같다.

또한 Block swap 을 통해 disk 상에 해당 page을 저장하고, 그 정보를 갖고 있어야 한다. 이는 이미 구현되어 있는 BLOCK\_SWAP 관련 파일의 함수를 사용하면 될 것 같다.

위 생각들을 바탕으로 구현해 보았지만 pintos가 작동하지 않았다. pure demand paging으로 처음에는 memory가 빈 상태에서, 점점 채워나가는 형태로 구현을 시도했다. 하지만 이 과정에서 계속 에러가 발생해 더 진행할 수 없었다.

* 1. **시험 및 평가 내용**



<vm test 결과 캡쳐>