

Pintos Project 4: Virtual Memory

담당 교수: 박성용

이름 / 학번: 20171643 / 박준혁

개발 기간: 11/15~12/03

I. 개발 목표

지금까지의 pintOS는 Program을 Memory에 올릴 때 Page 단위로 자르긴 했지만, 그 Page들을 그저 Memory 상에 별다른 작업 없이 바로 올렸다. 이는 곧 pintOS가 Load할 수 있는 Program의 개수가 Physical Memory의 사이즈에 의해 제한될 수 있다는 한계점을 내포하고, 즉, 불필요한 Memory 낭비를 의미한다.

본 Project 4 Phase에선 이를 ‘진정한 의미의 Paging’으로 변모시킨다. 이를 위해 Supplemental Page Table를 구현한다. pintOS에서 내장 형태로 제공하는 Page Table을 토대로, 몇 가지 주요 정보들을 추가해 Logical Context에 따로 또 Page Table을 두는 것이다. 이어 System은 이 Page Table을 이용해 Program을 Page 단위로 잘라 Loading한다. 이때 중요한 점은, 단순 Loading이 아니라, 현대 OS의 가장 큰 특징 중 하나인 **Lazy Loading**이 이뤄지게 한다는 것이다. Memory를 효율적으로 관리하기 위해, Program 수행 시 해당 Program의 모든 Data를 Memory에 한 번에 올리지 않고, 매 순간 필요에 따라 특정 Data만을 Memory에 올리는, **Demand Paging**을 적용하는 것이다. 이를 위해선 Page Fault Handler의 수정이 불가피하다. 이전 Phase까지의 pintOS는 Page Fault 발생 시 바로 해당 Program을 종료시켰다. 그러나, 본 Phase에선 이 Handler가 실질적인 Page Fault Handling, 즉, Fault Exception을 야기한 Page를 Disk에서 Memory로 옮긴 후, System이 해당 Faulting Instruction을 다시 실행시켜 Hit를 야기하는, 그러한 절차를 밟을 수 있도록 만들어주는 것이다.

허나, Page Fault의 Case는 이것만 있는 것은 아니다. Faulting Address가 Un-mapped Page가 아닌, Un-grown Stack Segment에서 기인했을 수 있다. 이 경우, Stack을 이전 할당 사이즈에서 확장시키는 작업이 필요하다. 역시나 마찬가지로 Page 단위로 말이다. 이러한 처리 역시 본 Phase의 요구사항이다. 정상적으로 Stack이 확장할 수 있도록 설계하자.

한편, 상기한 Page Fault 처리 루틴에서, 만약 새롭게 할당할 수 있는 Memory 영역이 없으면 어떡할 것인가? 달리 말해, Program에게 새롭게 제공할 Page(Frame)가 더 이상 없는 경우엔 어떤 작업이 필요할까? 그렇다. **Page Replacement**가 필요하다. 이를 위해 본 Phase에선 LRU(Least Recently Used) Policy와 Swapping을 기반한 Page Replacement도 구현한다. 이 과정에서 특정 Page에 대한 Access 여부, Dirty 여부의 판단이 중요할 것임은 자명하다.

위와 같은 다방면의 개발을 통해 pintOS가 Paging 기반 Lazy Loading을 성공적으로 제공할 수 있어야 한다. 우리는 이러한 과정에서 **운영체제의 핵심이라 할 수 있는 Memory Virtualization의 전반적인 이론을 꼭 넓게 이해할 수 있을 것이다**. 참고로, 본 Phase의 일부 Test Case(ex. page-merge-mm)는 mmap() System Call을 위시한 Memory Mapping의 구현도 요구하기에 Memory Mapping 기능도 추가적으로 구현하도록 한다.

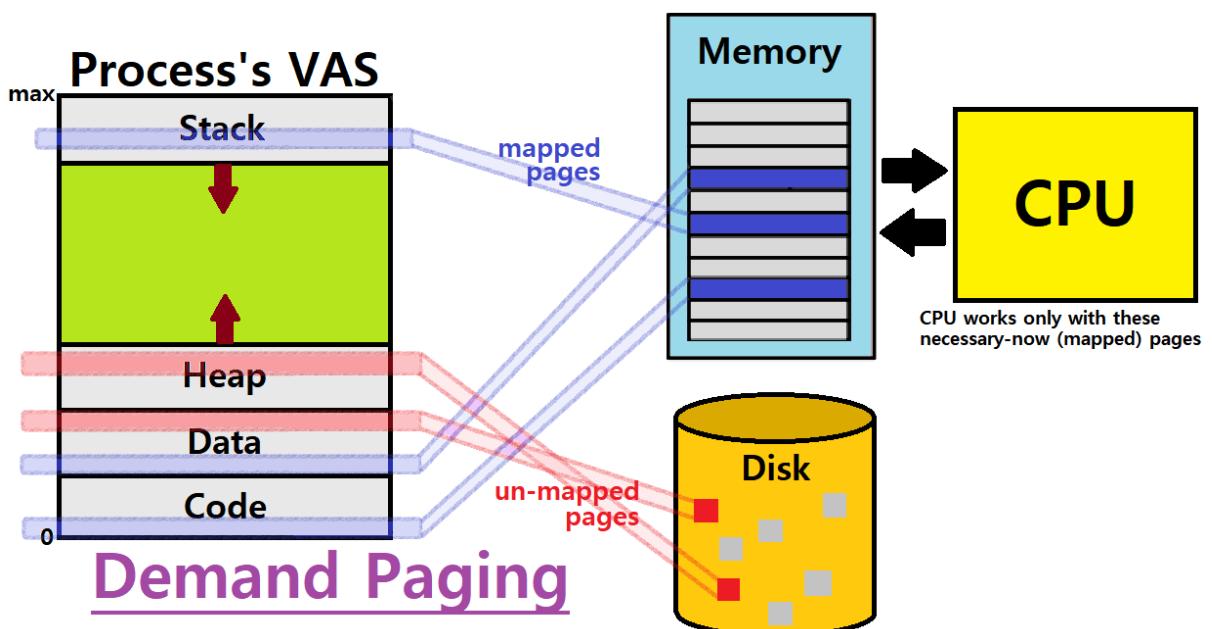
II. 개발 범위 및 내용

A. 개발 범위

1. Page Table & Page Fault Handler

위의 '개발 목표'에서 언급한 것처럼, 3번째 Phase까지의 pintOS에는 Demand Paging이 구현되어 있지 않아 불필요한 Memory 낭비가 발생하고 있다. System이 Process의 Virtual Address Space를 Physical Memory에 Load할 때, 그냥 해당 VAS를 통째로 Load할 경우, 그 내부엔 System이 접근하지 않고, 수행되지도 않는, 쓸모 없는 공간이 존재한다. 그래서 우린 Demand Paging을 구현해야 한다.

"Program을 한 번에 다 Load하지 말고, 매 순간 필요한 부분만 Load하자!"

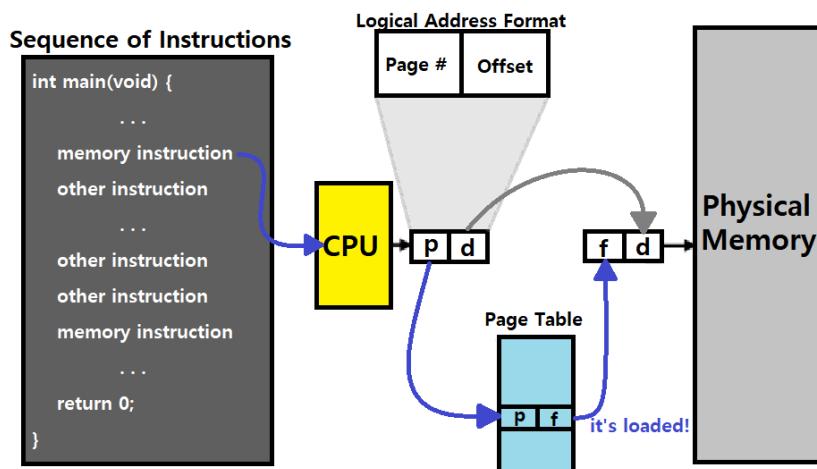


이를 위해선 pintOS에서 기본으로 제공하는 빈약한 정보의 Page Table에 'Demand Paging 시 필요한 정보들'을 더 마련해야 한다. 단순히 'VPN to PFN Translation 정보'뿐만 아니라, 해당 Page가 Memory 또는 Disk 중 어디에 존재하는지, 어떤 File에 Mapping되어 있는지, 변형(Write)은 가능한지 등의 다양한 정보를 추가하는 것이다. 후술할 다양한 Paging & Page Fault Handling 관련 기능들의 구현을 위해 말이다. pintOS Manual에선 이러한 Page Table을 'Supplemental Page Table'이라고 부르고 있다.

Supplemental Page Table이 마련된다면, 그 다음은 Page Fault Handler에 대한 수정이 필요하다. Lazy Loading, Demand Paging은 곧 다음과 같은 System Routine을 기반으로 동작하기 때문이다.

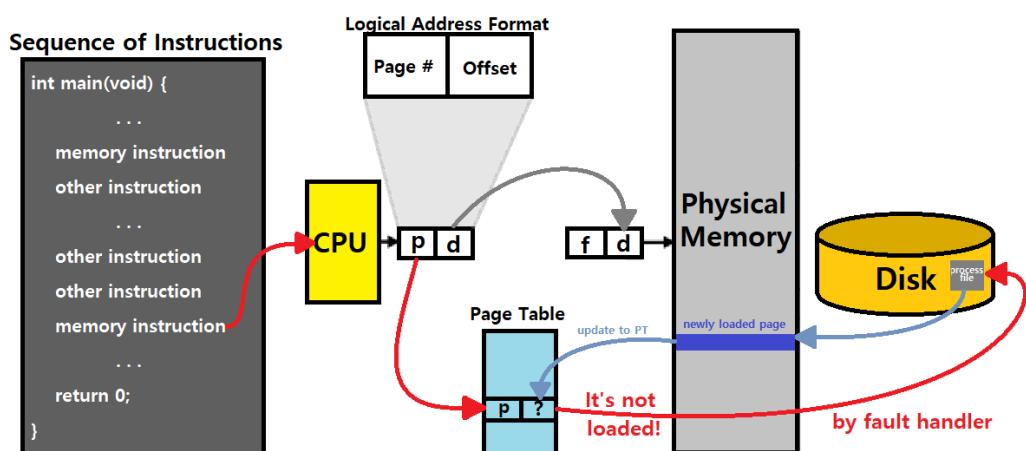
"Page Fault 발생 시, 해당 Faulting Address가 가리키는 Page를 Disk에서 Memory로 Load한다. 그 다음, 이어서 다시 Fault를 야기한 Instruction을 재-실행해 Hit시킨다."

어떤 Program이 실행 중이라 하자. OS에선 Demand Paging을 위해 Program 시작을 위한 최소한의 Page들만을 Memory에 Loading해놓은 상태이다. CPU는 이 Program의 Instruction Sequence를 순차적으로 읽어가며 수행하고 있다. 이때, 여러 명령 중 당연 Memory Access 명령이 있을 것이고, 이러한 명령을 수행하기 위해 Paging 기법에 따라 해당 Address에 대한 Page Table Mapping 정보를 확인한다. Translation 정보가 확인된다면, 그대로 그 정보를 따라가 대응되는 Frame에서 Data를 접근하면 된다.



Case 1) Page Found

하지만, 해당 Page가 Memory에 Mapping되어 있지 않다면 어떡할 것인가? 그때 바로 위와 같은 Fault Routine이 필요한 것이다. 그래야만 ‘필요할 때마다 Load하는’ Demand Paging이 구현되는 것이다.



Case 2) Page Fault

즉, 우리는 Page Fault Handler를 수정해야 하는 것이다. 기준의 Phase까진 우리는 Page Fault 시 해당 Program을 그냥 바로 종료시키도록 유도했었다. 그러나, 이젠 Lazy Loading의 실현을 위해, 우리는 상기한 Routine을 동작할 수 있도록 만들어주어야 한다. 우리가 System Programming과 Operating Systems 과목에서 학습했던 ‘진짜 Page Fault’

Exception Handling'을 구현해야 하는 것이다. 물론, 정말 Segmentation/Protection Fault에 해당하는 상황이라면 종료가 맞다.

이렇게 해서 (Supplemental) Page Table과 Page Fault Handler를 구축했다면, pintOS는 기본적인 Process(Thread) Loading을 무리 없이 수행할 수 있게 된다. 하지만, 아직 부족하다. 우리는 아래의 항목을 더 구현해야 한다.

2. Disk Swap

위와 같이 Supplemental Page Table과 Page Fault Handler를 구축하여 기본적인 Process Loading이 가능해졌다. 하지만, 이 상태에서 **만일 Physical Memory가 꽉 차면 어떡할 것인가?** User Page Pool에서 Process(Thread)에게 나눠줄 Free Frame이 더 이상 없는 상황 말이다. 이때는 상기한 것처럼 **Page Replacement**가 필요하다. 현재 Frame이 할당된 전체 Page들 중 특정 Algorithm을 기반해 Evict할 Page를 찾고, 해당 Page를 Disk의 Swap Space로 보내는 것이다. 이를 Swap Out이라 한다. 그리고, 만약 Swap Out된 Page를 다시 접근해야 하는 상황이 발생하면 해당 Page를 Swap Space로부터 Physical Memory로 다시 Swap In하는 기능도 필요하다. 우리는 이러한 일련의 과정을 구현해야 한다.

즉, Swapping을 기반한 Page Replacement을 구현해야 하는 것인데, 이때, Evicted Page의 선정에는 LRU(Least Recently Used) Policy를 적용하도록 하자. 물론, 우리가 강의 시간에 학습한 것처럼, '진정한 의미의 LRU 논리'는 비효율성을 전제하기 때문에 현실적으로 구현이 어렵다. 따라서 우린 (Supplemental) Page Table의 각 Entry가 가리키는 Page의 Accessed(Referenced) 여부와 Modified(Dirty) 여부를 특정 Bits로 나타낸 후, 이를 토대로 희생자를 결정하는 **Approximate LRU**를 구현토록 한다.

허나, 구현에 앞서 다음과 같은 고민은 필요하다.

"한참 전에 Reference되고 나서 이후로 Reference되지 않고 있는데 여전히 Referenced Bit가 Set되어 있는 Page는 어떻게 처리할 것인가?"

알다시피 이런 상황에선 '**Second Chance (Clock) Algorithm**'을 적용하면 좋다. System에서 할당한 Frame들을 하나의 Queue에 넣어두고, 이를 쭉 훑으면서 Referenced Bit가 Unset인 Victim Page를 찾되, 한 번 Search할 때마다 Referenced Bit가 Set된 Page들을 다시 Unset하고 넘어가는 (즉, 재-기회를 부여하는) 것이다. 이러한 Queue Search를 Circular(Clock) Order로 수행하는 것이다. 이렇게 하면 상기 고민에 대한 상당한 개선을 도모할 수 있다고 알려진다.

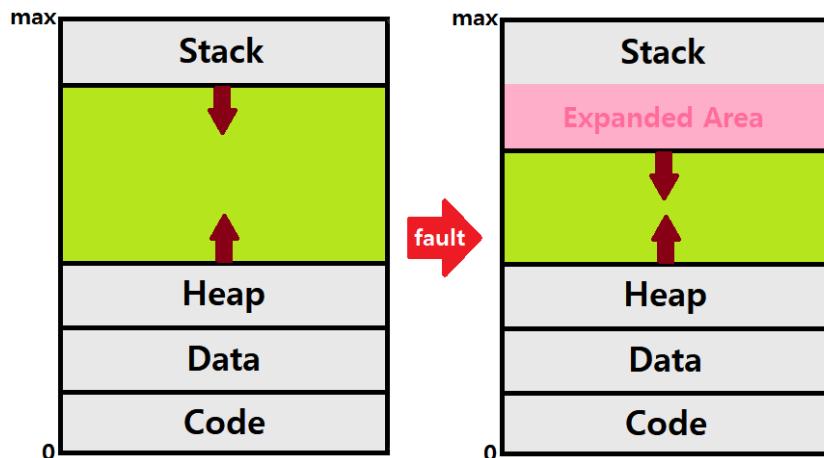
한편, 이러한 Page Replacement 과정에서 Swapping 시 'Block Device (쉽게 말해 HDD나 SSD와 같은 저장 장치, 즉, Disk)'에 대한 Sector-based I/O가 요구되고, 이는 pintOS의

'devices/block.h'에서 제공하는 기능들을 토대로 수행할 수 있다. 보다 자세한 Code-Level Detail은 이하 항목에서 서술한다.

3. Stack Growth

다시 1번 항목의 Page Fault 상황을 생각해보자. Page Fault Exception이 발생했는데, 해당 **Fault를 야기한 Address가 그 Process(Thread)의 Virtual Address Space 상에서 Stack Segment에 해당하는 위치**라고 하자. 쉽게 말해, CPU가 Program의 Stack을 접근하는데 Page Fault가 발생한 것이다.

이런 상황에선 Stack을 확장시켜야 한다. 사실 특별할 것은 없다. 1번과 2번 항목에서 알아본 Page Fault Handling Routine과 전혀 다르지 않다. 단지 그 '새롭게 부여하는 Frame'이 특정 Program File에 대한 것이 아니라 Stack Segment에 대한 것일 뿐이다. 즉, File Loading 과정이 빠진 Page Fault Handling인 것이다. 따라서, 당연하게도 이 상황에서 할당할 Physical Memory가 부족하다면 마찬가지로 Page Replacement가 발생할 것이다. 우리는 이러한 Stack Growth 기능도 구현해야 한다.

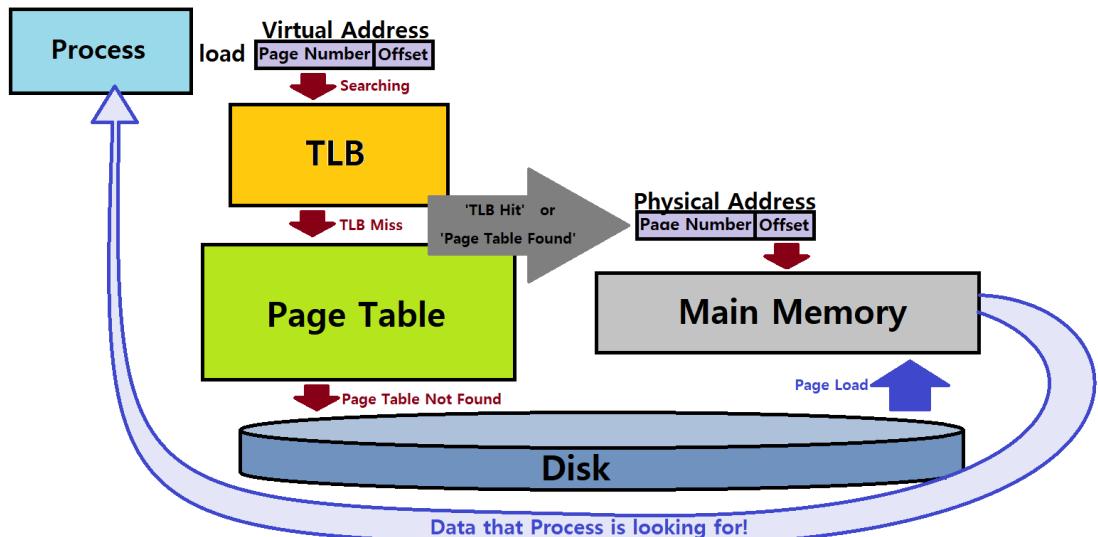


이때, Stack Growth가 필요한 상황인지 아는 방법은 무엇이 있을까? 당연히 기본적으로 Page Fault Handler에서 이를 매 Fault 상황 시마다 확인해야 할 것이다. 그리고 한 가지 상황이 더 존재한다. System Call Handler에 Argument를 넘길 때이다. System Call의 Arguments에는 분명 Memory Address도 있을 수 있고, 이들이 Memory Address를 접근하기 이전에, 만약 Stack Segment에서 기인한 Address이면 Stack을 확장해 Program 수행 과정에서의 문제가 발생하지 않도록 방지해야 한다.

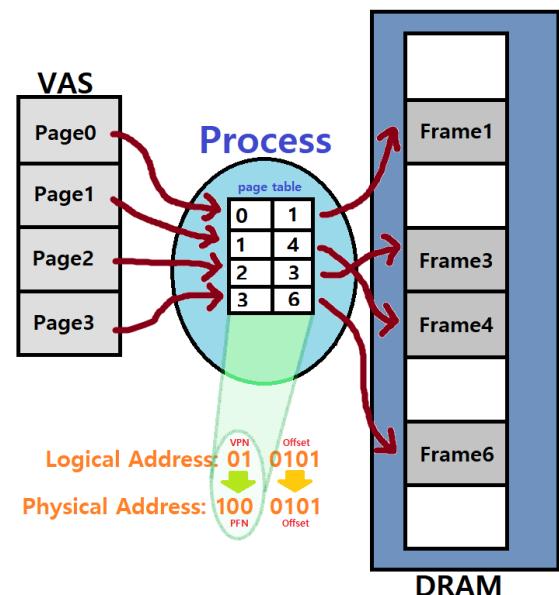
참고로, pintOS에선 Stack이 자랄 수 있는 Size에 Limit을 두길 권고하고 있고, 그 Size는 8MB이다. 따라서, 프로젝트 구현 간에 이 점을 유념해 구현도록 하자. 마찬가지로 자세한 Code-Level Detail은 후술한다.

B. 개발 내용

1. Page Fault가 발생하는 이유와 이를 Handling하는 전반적인 과정



위 그림을 보자. 현대 OS에서 CPU가 Process의 Memory Access Instruction을 수행할 때 어떠한 처리 과정이 이루어지는지를 자세하게 보여주고 있다. TLB(Translation-Lookaside Buffer)s 개념은 본 Project의 주제와 큰 관련이 없으므로 생략하고, Paging 개념을 떠올린 채 Page Table 부분을 보자. Memory Access 시엔 넘겨 받은 Virtual Address의 Format에서 VPN(Virtual Page Number)를 추출해 해당 Value를 Index로 하여 Page Table을 탐색한다. Index가 가리키는 Slot에 **'VPN to PFN Translation 정보'**가 있는지를 보는 것이다. 그러한 변환 정보가 있다면, PFN(Page Frame Number)을 그대로 추출하여 Physical Address를 도출해 동작을 수행한다. 우측 그림과 같이 말이다.



허나, **변환 정보가 없다면?** 다시 말해, 접근하고자 하는 Page에 대해 Mapping된 Physical Frame이 없다면, 그것이 바로 **Page Fault** 상황인 것이다. Process의 Virtual Address Space 상의 특정 Address가 포함된 Virtual Page에 대해, Page Table을 탐색했는데, Mapping된 Physical Frame이 없는 상황인 것이다. 이때, 일반적으로 Page Table이 설치되어 있는 MMU(Memory Management Unit)가 OS에게 Exception을 일으키고, OS는 이 Signal을 받아 Page Fault Handler라 하는 Code 부분을 수행하게 된다.

Page Fault Exception이 발생하면, OS의 Page Fault Handler는 다음의 순서로 동작한다.

- 1) 특정 Register(주로, CR2 Register)에 저장되어 있는 **Faulting Address**를 추출한다.
- 2) 해당 Address가 포함된 Virtual Page가 **Valid한지 확인**한다. 즉, 해당 Page에 대한 PTE(Page Table Entry)가 있는지를 보는 것이다.
 - A. 만약, PTE가 있다면, 이는 단순히 해당 Page에 대한 **Physical Frame**이 현재 **부재한 상황**, 즉, 일반적인 **Page Fault 상황**이다. Disk에서 해당하는 Data를 Memory로 올리고, 새롭게 만들어진 Frame을 해당 Page에 다시 Mapping해 Page Table을 업데이트한 후, 다시 명령을 재-실행하면 되는 상황이다.
 - B. 만약, PTE가 없다면? 이땐 **Faulting Address**가 **Stack Segment**가 확장했을 때 **Cover**할 수 있는 영역 내에 존재하는지를 확인한다. Grow-able Region에 해당하는지를 보는 것이다. 해당한다면? 상기 항목에서 언급한 것처럼 Stack Growth를 수행하면 된다.
 - i. **Grow-able Region**에 포함되지 않는다면? 이는 **Segmentation Fault**에 해당한다. 즉, Process(Thread)를 죽이고 Free해야 한다.



Page Fault Handler는 위와 같이 Case를 나누어 대처한다. A 또는 B Case에 해당하는 상황이라면 새로운 Frame을 마련해준 후, Fault를 야기한 Instruction을 다시 실행시켜 Program을 Resume 한다. 좌측 그림과 같이 말이다. 새로운 Frame을 마련할 땐, Physical Memory 공간이 여유가 있다면 여유 공간에서 Frame을 만들어 줄 수 있을 것이고, 그렇지 않다면 LRU Policy와 Swapping에 기반한 Page 교체를 시도할 것이다.

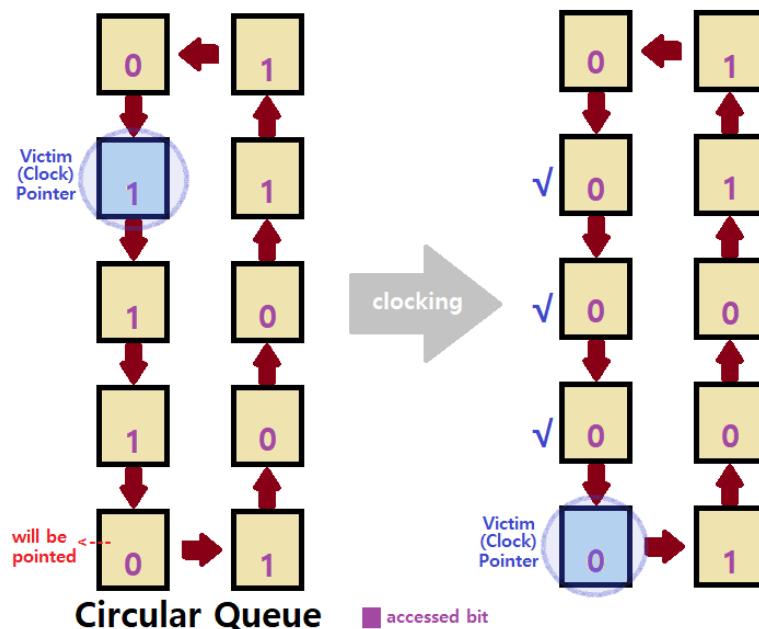
우리는 이러한 일련의 과정을 모두 구현한다. 이를 위해 Supplemental Page Table Entry에 단순히 VPN 정보뿐만 아니라 Memory Load 여부, 연결된 File, 그 File에서 어디까지 읽었는지 정보, 해당 Page가 Page Replacement의 대상이 될 경우 필요한 자료구조 등을 모조리 추가해야 할 것이다. 좀 더 자세한 Flow와 Code-Level Detail은 아래 항목에서 다룰 것이다.

2. Disk Swap 발생 시 사용한 Page Replacement Algorithm

앞선 항목에서 Disk Swap 시 우리는 Approximate LRU(Least Recently Used)인 '**Second Chance (Clock) Algorithm**'을 채택한다고 했다. pintOS Manual에서 추천하는 방식이기도 하고, OS 강의 시간에 열심히 학습한 개념이기도 하기 때문에 이를 선택했다. 이 Algorithm의 논리는 간단하다.

"Mapped Frame들의 Queue에서 Reference되지 않은 Page를 찾아서 Evict하되, 단순히 Reference 기반 LRU를 구현할 경우, 초기에만 Reference되고 이후엔 Reference되지 않는 상황이 발생할 수 있기에, 이를 방지하고자 매 Search 시 Referenced Page의 Reference Bit를 Unset하도록 한다. 그리고, 이러한 Search를 Clock Order로 수행해 가능한 Fair하게 Second Chance를 부여하도록 한다."

Second Chance (Clock) Algorithm



이 논리를 철저하게 지켜 구현하면 된다.

이렇게 해서 Evicting할 Victim Page를 선정하게 되면, 해당 Page의 유형을 분석해 아래와 같이 처리하도록 한다.

- 만약, Victim Page가 Swap Space에 기존에 존재했던 Page라면 그대로 다시 Swap Out한다. Swap된 이력이 있다는 것은 곧, Program 실행 시작 이후 Modify된 적이 있다는 의미이기 때문이다.
- 참고로, Stack Growth에서도 언급하겠지만, **Stack Segment**를 위한 Page (Frame)는 Program Execution에 필수적인 요소들이므로 System에서 아예 Evict하는 것은 굉장히 비효율이다. 지속적인 접근이 이뤄질 확률이 높기 때문이다. 그리고 무엇보다, 이들은 Evict 시, 만약 Swapping을 이

용치 않고 그냥 Free한다면 추후 해당 Data가 다시 필요한 경우 Data를 가져올 방법이 없다. 그냥 사라진다. 따라서 이들은 항상 Evict 시 Swapping을 통해 데이터를 유지할 수 있도록 한다. Dirty 여부와 상관없이 말이다.

- 이는 곧, **Victim Page가 Program의 Binary File의 일부분**이라면, 즉, 일반적인 상황이라면, 해당 Page가 Dirty한 경우엔 Mapped Frame을 Swap Space로 Swap Out해야 한다는 것이다. Dirty하지 않은 Page의 경우, Un-Accessed 이자 Not-Dirty이므로, 그대로 Frame을 해제해도 무방하다. 근사 LRU 논리 아래에서 말이다.
- **Victim Page가 mmap System Call을 통해 Memory Mapping된 영역에서 온** 경우는 어떠한가(개발 목표 항목에서 언급한 것처럼, 본 Project Test Case 중 Memory Mapping을 검사하는 Case가 존재)? 현재 System이 Frame이 부족한 상태이므로 Frame은 Free하되, Virtual Page 정보, 즉, PTE는 남겨야 한다. 언제 다시 해당 Mapped Page를 접근할지 모르지 않는가. 물론, 당연히 Page가 Dirty하다면 Mapped File에 Modified Data를 Write해야 할 것이다.

이와 같은 흐름으로 Disk Swap을 구현한다. 이를 위해 우리는 할당된 Frame들을 추적/관리하는 Queue 자료구조가 필요할 것이고, 이 Queue의 각 Element에는 Mapping된 Page의 PTE를 가리키는 포인터, 이 Element(Frame)를 자신의 VAS의 특정 Page에 Mapping해 사용하고 있는 Thread를 가리키는 포인터 등이 필요할 것이다. Thread에 대한 포인터가 필요한 이유는, 해당 Thread의 Page Table을 접근해 Referenced 및 Dirty 여부를 확인할 수 있어야 하기 때문이다. 역시나, 자세한 Detail은 이하 항목에서 확인하자.

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

다시 Page Fault 상황으로 돌아가보자. **Page Fault를 야기한 Faulting Address가 포함된 Virtual Page에 대해, PTE가 존재하지 않는 상황이 있다고 했다.** 즉, Invalid Reference 상황이고, 해당 Process를 최초에 Load할 때 그 Program의 Binary Image에 포함되어 있지 않은 Data일 것이다.

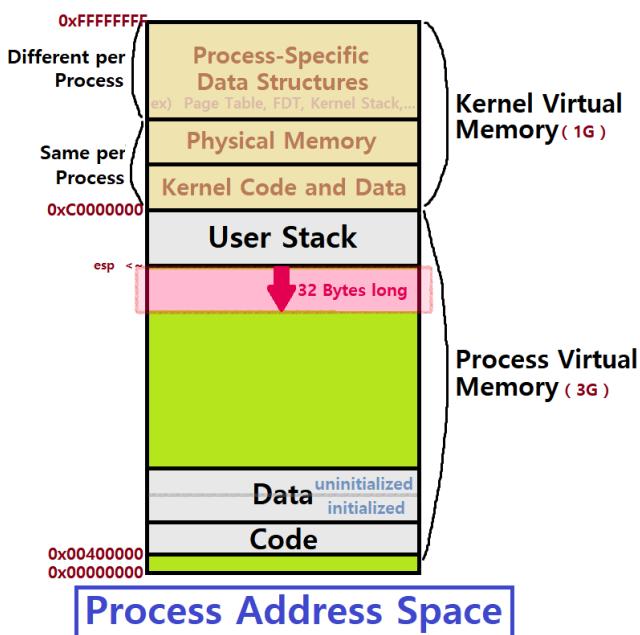
그렇다면, 우선, Faulting Address가 올바른지를 확인하자.

- Faulting Address가 VAS에서 **User가 접근할 수 있는 범위 안에 들어오는가?**
- Faulting Address가 **Stack의 Start Address로부터 8MB** 안에 들어오는가?
(pintOS가 권장하는 Stack Size의 Limit)

- pintOS의 기반인 **80x86 System**의 **Assembly PUSHA** 연산의 **ESP Pointer** 점검 조건을 충족하는가? (PUSHA 연산은 ESP Pointer에서 32Bytes 보다 더 멀리 떨어진 Virtual Address에 대해선 Page Fault를 일으키지 않는다. 이 경우엔 그냥 오류 상황이 됨)

이러한 조건을 충족하는지를 확인한다. 이 조건들을 모두 충족할 경우, 이는 **Fault**를 야기한 **Virtual Address**가 **Process**의 **Stack Segment**로부터 온 것임을 의미(Demand Paging에선 공간 효율을 위해 Stack을 초기에 크게 할당하지 않음)하고, 따라서 Stack을 위한 Page가 추가적으로 더 마련되어야 함을 나타낸다. 즉, Stack 확장을 해도 되는 상황인 것이다. 따라서, 이 경우엔 Stack을 위한 Page를 더 마련하고, 그에 대해 Physical Frame들을 할당하면 된다.

만약, 위 조건들 중 하나라도 충족하지 않는다면, 그것은 Fault를 야기한 Address가 미지의 영역을 가리키는, Program과 System에 중대한 오류를 범할 수 있는 상황으로, Protection Fault 상황에 해당한다. 따라서, 이 경우엔 Program을 종료시킨다.



이러한 Routine이 매 Page Fault마다 검사되어야 할 것이며, 매 System Call Argument, 그 중에서도 Pointer Variable이 넘겨지는 경우에도 검사되어야 할 것이다.

한편, 앞서 언급한 것처럼, Faulting Address에 대해 이미 PTE가 존재한다면 그것은 일반적인 Page Fault 상황이므로 1번 항목으로 처리하면 된다. Heap Segment 영역에 대해선 어떻게 할까? 다행히 pintOS에선 Heap Segment를 상정하지 않는다고 한다. 따라서, 이는 고민하지 않아도 된다.

III. 추진 일정 및 개발 방법

A. 추진 일정

개인적으로 이번 Project 4는 지난 Project Phase들과는 비교가 어려울 정도로 난이도가 높았다. 기본적으로 구현해야 하는 (위에서 설명한) 개념들의 이해가 상당한 노력을 요구하였고, 이를 실제 Code로 옮기는 과정 역시 굉장히 복잡했다. 따라서, 지난 Phase들보다 약 1주일 이상 더 여유를 두고 개발에 임했다. 자세한 과정은 다음과 같다.

1) Understanding the Requirements (11/15-11/20)

Project 안내 PDF와 pintOS Manual을 토대로 본 Phase에서 요구하는 Demand Paging, Supplemental Page Table, Page Table Handler를 비롯한 각 개발 사항의 이론적 개념을 이해한다. 이 중, 근래 강의 내용과 밀접한 관련을 지니는 Swapping, Clock Algorithm 등은 강의자료를 중점적으로 복습해 체득했다. 과거 Phase보다 약 이틀 정도 더 개념 이해에 시간을 투자해 Implementation 준비를 마쳤다.

2) Page Table & Page Fault Handler (11/20-11/25)

Supplemental Page Table과 Page Table Handler를 개발해 기초적인 Lazy Loading을 완성하는 단계이다. 상기 서술 내용대로 구축하였으며, Page Table Entry에 필요한 요소를 파악하는 과정이 가장 까다로웠다. PTE Format을 결정한 후, Table을 어떤 자료 구조로 구축할지, Handler에서는 어떤 Flow를 거쳐 Page Fault를 다룰 것인지 등을 고민하는 데에 많은 시간이 소요되었다. 이어, Process Loading 및 Termination 과정을 수정하는 작업 역시 상당한 난이도가 있었으나, 다 차례의 Troubleshooting을 통해 성공적으로 마칠 수 있었다.

3) Disk Swap (11/25-11/28)

Swapping, Second Chance (Clock) Algorithm을 기반한 Page Replacement를 개발하는 단계이다. 본 Project에서 가장 어려운 단계였다고 해도 과언이 아닌 부분으로, System에서 할당한 Frame들을 어떻게 관리할지, Frame Allocation 및 Freeing 과정에서 File 관련 I/O를 어떻게 수행할 것인지, 그리고 그 사이에서 **Synchronization**이 지켜지는지 등을 파악하는 데에 많은 시간과 노력이 요구되었다. Swapping에는 pintOS에서 제공하는 block.h를 이용하였고, Swapping을 위한 Bitmap 자료구조 구축 및 Block I/O 함수 사용 방식 역시 상당히 까다로웠다.

역시나 마찬가지로 다 차례의 Debugging과 Troubleshooting 과정을 거쳐 완수할 수 있었다. 약 4일 가량 이 단계에 매진했다.

4) Stack Growth (11/28-11/30)

Page Fault Handling의 또 다른 예시인 Stack Growth를 구현하는 단계이다. 기능 구현 자체는 어렵지 않았으나, **Stack Segment의 Page들이 Swapping 시엔 어떻게 처리되어어야 하는지를** 파악하기가 힘들었다. 초기 개발 과정에선 이러한 Page(Frame)들 역시 마찬가지로 다른 여타 Binary Image Page들의 Frame과 동일하게 취급하였는데, '개발 내용' 항목에서 언급하였듯, 실제로는 다르게 취급(이들은 Dirty 여부가 중요치 않고, Evict 시 반드시 Swapping해야 한다는 점)해야 했고, 이를 발견하는 과정이 이 단계의 핵심이라고 생각한다.

5) Memory Mapping (11/30-12/02)

상기 4)번 항목까지 개발했을 시 대부분의 Test Case를 통과하였으나, 일부 Test Case, 특히 '**page-merge-mm**'이 지속적으로 해결되지 않았는데, 이 원인이 Memory Mapping 미-개발에 있다고 자가 진단하였고, 이에 따라 **Memory Mapping 기능 추가**를 위한 mmap, munmap System Call의 구현이 개발 마지막 시기에 이뤄졌다.

역시나 4)번 항목과 마찬가지로 Memory Mapping으로 만들어진 Page들을 여타 다른 일반적인 Page들과 Replacement 과정에서 다르게 취급해야 함을 파악하는 과정이 까다로웠다. 이 단계의 완수 이후 한 가지 놀라웠던 점은, munmap System Call을 이용하면 Process Termination 시의 Frame Freeing 및 Data Update 작업이 굉장히 깔끔하게 이뤄질 수 있다는 점이었다. 이하 항목에서 자세히 서술하겠다.

모든 Test Case의 통과를 확인한 시점은 12/02일 전후였으며, 이후 사흘 정도 (12/02-12/04) 보고서 작성 및 코드 정리를 수행하였다.

B. 개발 방법

지금까지 설명한 개발 내용을 실제 Code로 구현하기 위해선 어떠한 작업이 필요한지를 항목 별로 소개하겠다. 어떤 Source Code File이 수정되고, 어떤 Data Structure가 사용되며, 어떤 Function이 추가되는지 등에 주목하자.

(1) Page Table 및 Page Fault Handler의 구현

pintOS Manual에선 본 Project 4에서 필요한 기능들을 각각 **page.h**, **frame.h**, **swap.h** 등의 **Header File**들을 작성해 구현하도록 권장하고 있다. 따라서 본인 역시 이 방식을 채택해 본 Project 개발에 임했다. 우선, (Supplemental) Page Table의 구현은 vm Directory 내에 page.h 파일을 만들어 수행했다.

Page Table을 구현하기 위해선 가장 먼저 어떤 Data Structure를 사용할지 정해야 한다. 본인은 Hash를 선택했다. 물론 Table이란 것이 당연 List나 다른 자료구조로도 구현이 가능하지만, Process 수행 시 항상 Page Table 접근 및 Search가 이뤄짐을 감안해, **Search 및 Insertion, Deletion 등이 효율적으로 이뤄질 수 있는 Hash가 가장 좋은 선택일 것**이라 판단해 Hash를 택했다.

이어, Hash의 Element, 즉, Page Table Entry의 Format을 정하는 작업이 필요하다. 본인은 '**pt_entry**'라는 이름의 구조체를 선언해 이를 수행했으며, 해당 구조체 내부엔 VPN을 가리키는 변수, Page의 Type(일반 Binary File을 나타내는지, Swapping의 대상인지, 또는 Memory Mapping의 대상인지를 명시. Page Replacement 시 중요)을 나타내는 변수, Page의 Memory Load 여부를 나타내는 Flag, Page에 Mapping된 File이 얼마큼 읽혔는지를 나타내는 변수 등을 Field로 하였다. 이들은 비단 Page Fault Handling 및 Process Loading뿐만 아니라, Page Replacement, Swapping, Memory Mapping 시에 중추적인 역할을 담당한다.

자료구조가 모두 결정되면, 이를 토대로 **Page(Hash) Table**의 초기화 및 각 종 연산을 담당하는 함수들(pt_create_entry, pt_insert_entry, pt_delete_entry 등)을 page.c 파일에 작성했다. 기본적으로 pintOS에서 Hash Library를 제공하기 때문에 이를 이용해 Compact하게 작성할 수 있다. 이를테면, 'pt_insert_entry'라는 이름의 함수는 새로운 PTE를 Supplemental Page Table에 삽입하는 기능을 수행하는데, 이는 간단히 'hash_insert'라는 기제공 함수를 호출해 구현한다. 참고로, Page Table은 각 Process마다 하나씩 존재해야 하므로 Page Table을 초기화하는 'pt_init' 함수는 'userprog/process.c'의 'start_process' 루틴에서 호출되어야 한다는 점을 기억하자. (그렇다면 당연히 Page Table Deallocation은 'process_exit'에서 호출되어야 할 것이고, 이보다 앞서 'threads/thread.h'에 각 Thread Structure에 Page Table을 정의(struct hash Type)해두는 것이 필요할 것임)

각 함수의 자세한 동작 방식은 아래 '제작 내용' 항목에서 소개하겠다.

이렇게 Page Table을 사용할 준비를 마쳤으면, 이어 **Page Table 구축**을 시도하자. 기존의 pintOS에서 Page Loading을 수행하는 'userprog/process.c'의 'load_segment' 함수를 수정한다. 실행할 Program의 Binary File을 Page 단위로 쪼개, 각 Page에 대해 PTE를 마련하는 작업이 이뤄진다. 상술한 'pt_create_entry' 함수를 이용해 PTE를 할당하고, 할당 과정에서 Binary File의 어디까지 읽힌 것인지 등을 Field에 기록하도록 하자.

PTE가 마련되면, 그 Entry를 해당 Process의 Page Table에 Hash Insertion하도록 하자. 이어, 마찬가지로 Stack Segment에 대해서도 초기 할당이 이뤄질 때 대응되는 PTE를 마련해 Page Table 삽입을하도록 하자. 이는 process.c의 'setup_stack' 함수에서 이뤄진다. (앞서 언급한 바와 같이 이때 Stack Segment PTE의 경우엔 Page Type을 'SWAPPED'로 지정해준다. 참고로 **Page Type**에는 SWAPPED, BINARY(일반 Page), MAPPED(mmap을 통한 Page), 이렇게 총 3개를 미리 정의해두었다. 추후 더 자세히 설명할 것)

Process의 Page Table을 위와 같이 구축했다면, 이젠 **Page Fault Handler**의 수정이 필요하다. 이미 알다시피, Page Fault는 'userprog/exception.c'의 '**page_fault**' 함수에서 처리

하는데, 이곳에 상기 항목에서 설명한 Page Fault 처리 Flow를 작성하도록 한다. 대응 PTE 존재 여부를 통해 Stack Growth 상황과 일반 Page Fault 상황을 구분한다고 언급한 바 있는데, 이는 'vm/page.h'에 정의해둔 'pt_find_entry' 함수를 통해 수행할 수 있겠다. 자세한 Code 설명이 곧 이어질 것이다.

Stack Growth는 3번 항목에서 언급하도록 하고, 우선 '**일반적인 Page Fault Handling**' 상황을 보자. Project 안내 PDF를 따라 '**handle_mm_fault**' 함수를 process.c 파일에 구축한다. (process.c 파일에 구축하는 이유는 코드 주석에도 설명을 달아놓았지만, pintOS에서 미리 제공하고 있는 (pintOS 기본 내장 Page Table인 Page Directory에 Translation 정보를 기입하는) install_page 함수가 static으로서 process.c에 선언되어 있는데, 본 함수가 이 함수를 사용하기 때문임)

'handle_mm_fault' 함수는 Page의 Type에 따라 약간 달라지지만, 기본적으로 Frame Allocation을 통해 Given Page에 대해 Mapping을 수행하는 것은 변함없다. 이 함수가 매 Page Fault 때마다 호출되어 Demand Paging, Lazy Loading을 실현하게 된다. 자세한 디테일은 후술한다. 여기까지 개발함으로써 (1)번 단계는 마칠 수 있다. (사실, 좀 더 엄밀히 말하면, Page Fault Handler는 (2), (3)을 모두 구현해야 완성된다. 이는 Process Loading 및 Termination 시에도 마찬가지이다)

(2) Disk Swap 시의 Page Replacement Algorithm

Page Replacement은 vm Directory에 frame.h, swap.h 파일을 만들어 수행한다. frame.h는 System에서 할당한 Frame들을 추적/관리하기 위한 Frame Table의 구축을 담당한다. '개발 내용' 항목에서 소개한 바 있듯, Frame들은 추후 Eviction 시 Clock Algorithm이 적용되어야 하기 때문에 Queue 자료구조가 가장 적합하다. 따라서 pintOS에서 제공하는 **list**를 그대로 사용하자.

따라서 이를 토대로 **Frame Table 초기화 및 삽입/삭제/탐색** 함수들을 작성한다. 각 함수에 대한 소개는 이하 항목에서 진행하겠다. 참고로, Clock Algorithm은 'ft_clocking'이라는 함수가 수행하는데, 이 함수는 'frame_clock'이라는 전역 변수를 Frame Table에 대해 Circular Order로 돌리는 역할이다. (즉, Frame Table과 'frame_clock'은 System-Wide한 Data들이고, 이들은 복수의 Process가 동시 다발적으로 접근할 것이기에 Synchronization을 필요로 할 것이다. 이를 위해 Mutex Lock도 frame.h에 마련한다)

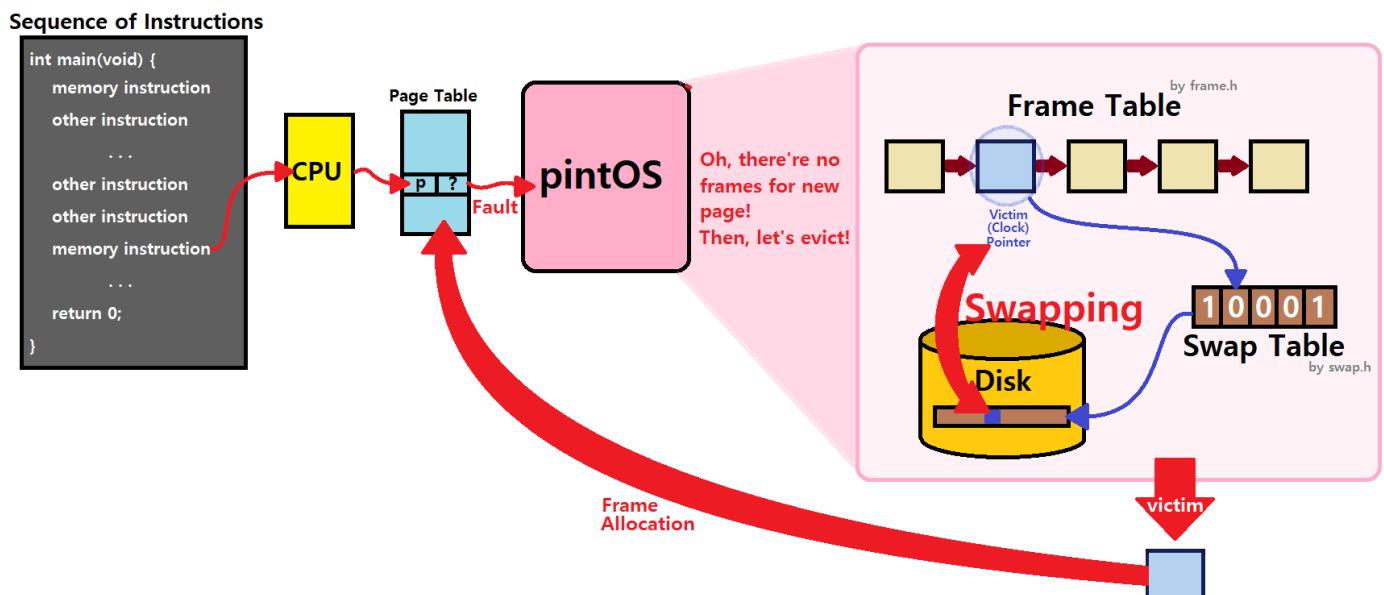
Eviction은 'ft_get_unaccessed_frame', 'pagedir_is_dirty(pintOS 제공)', 'ft_evict_frame' 함수들이 담당하며, 앞서 설명한 Clock Algorithm을 그대로 Code로 구현하는 과정이라 보면 된다. Victim Page의 Dirty 여부와 Type(BINARY/SWAPPED/MAPPED)에 따라 동작을 달리함을 기억하자. 자세한 동작은 후술하겠다.

이렇게 Frame Table이 완성되면, 이를 토대로 **동일 헤더 파일 내에 Frame Allocation / Deallocation 함수들을 작성하자.** 이들은 모두 기존 pintOS에서 ‘threads/palloc.h’가 제공하던 Physical Memory 할당/해제 함수들을 대체하는데, 각 함수의 이름은 ‘alloc_page’, ‘free_page’이다. ‘alloc_page’에선 Frame을 기준 ‘palloc_get_page’로 할당하되, Frame이 부족할 경우 Page Replacement을 수행하도록 설계되어 있고, ‘free_page’에선 이렇게 할당된 Frame들을 해제하는 기능이 구현되어 있다. 이하 항목에서 자세히 알아보자. (참고로 frame.h에선 이 두 함수와 ‘load_file_to_page’라는 Fault 시 File Loading에 쓰이는 함수, 이렇게 총 3개의 함수만이 Interface로서 외부로 공개된다. 나머지 Frame Table 관련 기본 연산 함수들은 모두 static 선언되어 가려져 있다. 이는 System-Wide Frame Table에 대한 보호 조치라고 볼 수 있다)

이제 **swap.h**를 보자. 이 헤더는 철저히 Swapping 기능만을 제공한다. Disk 내의 Swap Space를 OS 상에서 가상화한 ‘Swap Table’이 구축된다. Manual을 따라 **Bitmap**으로 구현하였으며, Bitmap의 각 Bit는 Swap Space의 Swap Slot이라 보면 된다. 즉, System에서 block.h를 이용해 Disk와 같은 Block Device와 소통해, Swapping을 위한 블록 공간을 부여 받고, 그 공간을 Bitmap의 특정 Bit로 연결시키는 것이다.

따라서 Swap Out을 담당하는 ‘swap_out’ 함수는 넘겨 받은 Frame을 특정 Swap Block에 Block Write를 수행하고, 그 Index에 대응하는 Bitmap Bit를 Setting할 것이다. ‘swap_in’은 반대로 동작할 것이다. (이들을 모두 설계한다. 아래 항목에서 확인 가능)

이런 식으로 간단하게 Swapping Routine들을 swap.h에 마련하도록 한다. 참고로, Disk의 Swap Space 역시 여러 Process에서 동시 다발적으로 접근할 수 있기에 File 접근이나 Frame Table 접근 시와 마찬가지로 Mutex Lock을 통해 보호하도록 하자.



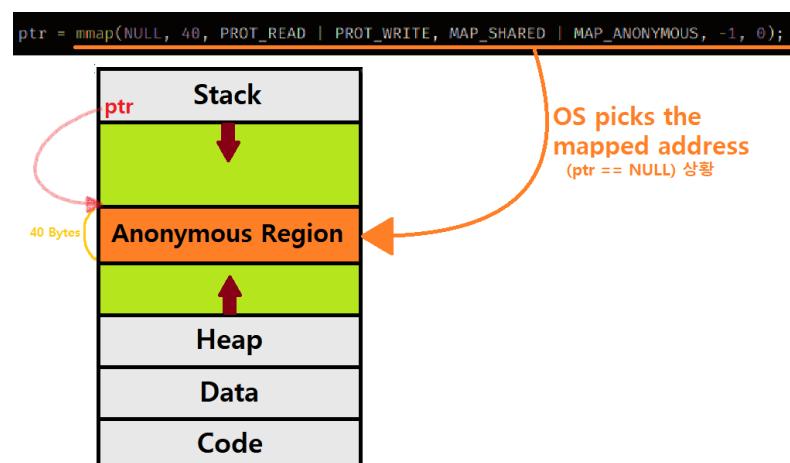
이렇게 Frame Table과 Swap Table의 구축이 완료되면 성공적인 Demand Paging을 제공하는 Frame Management가 가능해지고, 이를 토대로 앞선 Page Fault Handler와 Process Loading / Termination Routine을 수정해 pintOS를 보다 더 정답에 가깝게 이끌 수 있다. (참고로, Frame Table과 Swap Table은 모두 System-Wide Data Structure이므로 pintOS의 main 함수인 init.c에서 미리 초기화해야함)

(3) Stack Growth의 구현

Stack Growth는 따로 vm에 File을 만들 필요 없이, '**userprog/process.c**'에서 구현한다. '**expand_stack**'이라는 함수를 만들고, 해당 함수에서 pintOS 기본 제공 'is_user_vaddr' 함수와 PHYS_BASE 매크로 변수 등을 이용해 우선 Stack 확장 가능 여부부터 판단한다.

가능하다는 것이 확인되면, '**threads/vaddr.h**'에서 제공하는 '**pg_round_down**' 함수를 통해 적합한 VPN을 추출하고, 그 값을 이용해 새로운 Page를 할당한다. 이때, 새 Page에 대한 Frame Allocation은 (2)에서 완성한 'alloc_page'를 통해 수행한다. (여기뿐만 아니라, 이제 process.c의 Lazy Loading 수행 과정에선 모두 이 함수와 'free_page'를 사용함. 이렇게 각 항목의 개발 사항들이 유기적으로 연동되어 궁극적으로 **Lazy Loading**을 제공)

Frame Allocation이 이뤄지면, 해당 Frame과 Page를 'install_page'를 통해 매칭시키고, 이에 대한 PTE를 새롭게 할당 및 삽입함으로써 Stack Growth를 수행한다. 이 Stack Expansion 함수는 Page Fault Handler와 System Call Arguments Checking Macros에서 각각 호출해 사용하면 되겠다.

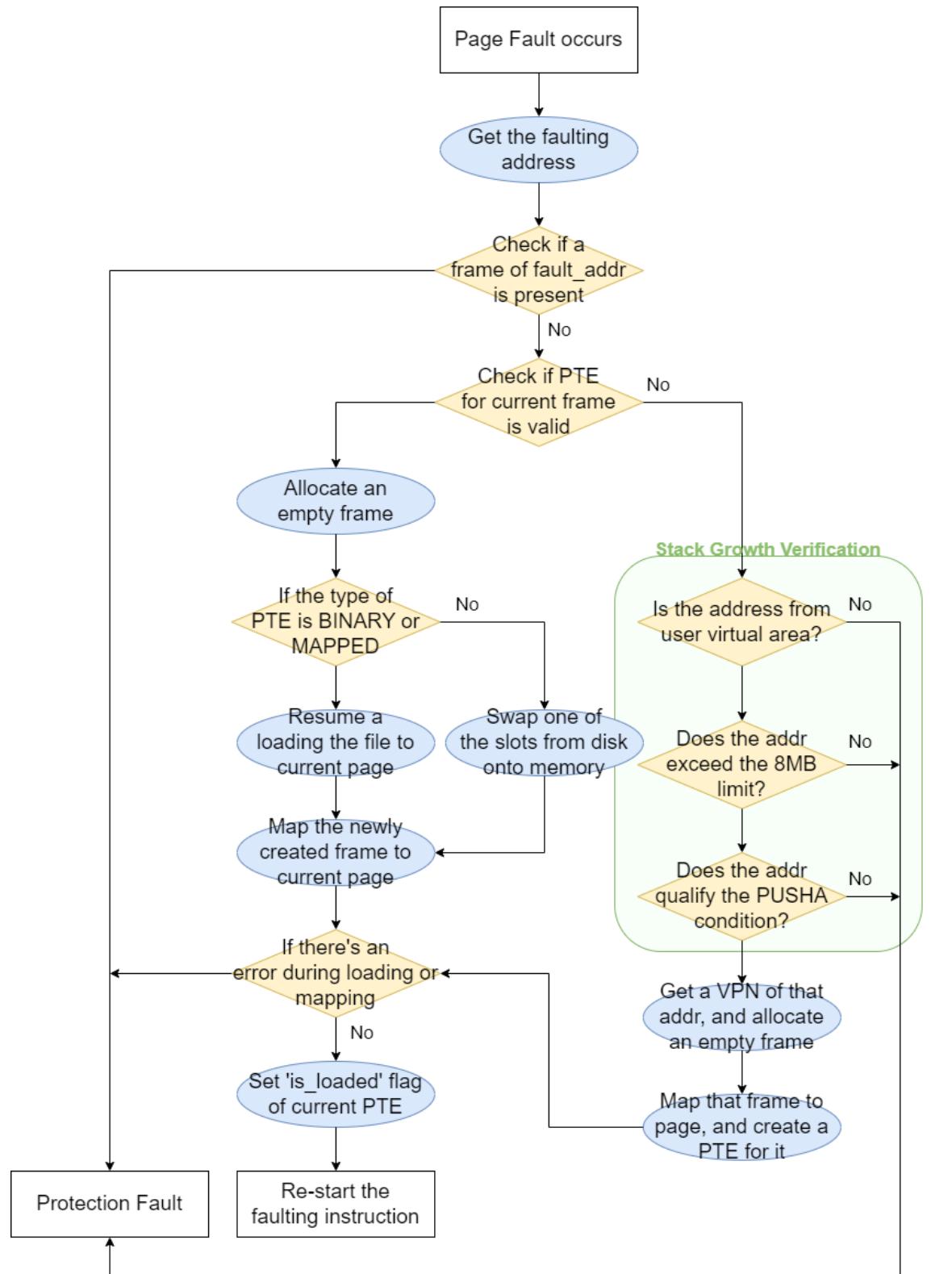


한편, 본 (3) 항목엔 해당하지 않지만, vm Directory에 **mmap.h**라는 파일을 만들어 본 Project를 위한 **Memory Mapping** 역시 구현한다. 이는 이하 '제작 내용' 항목에서 소개하겠다.

IV. 연구 결과

A. Flow Chart

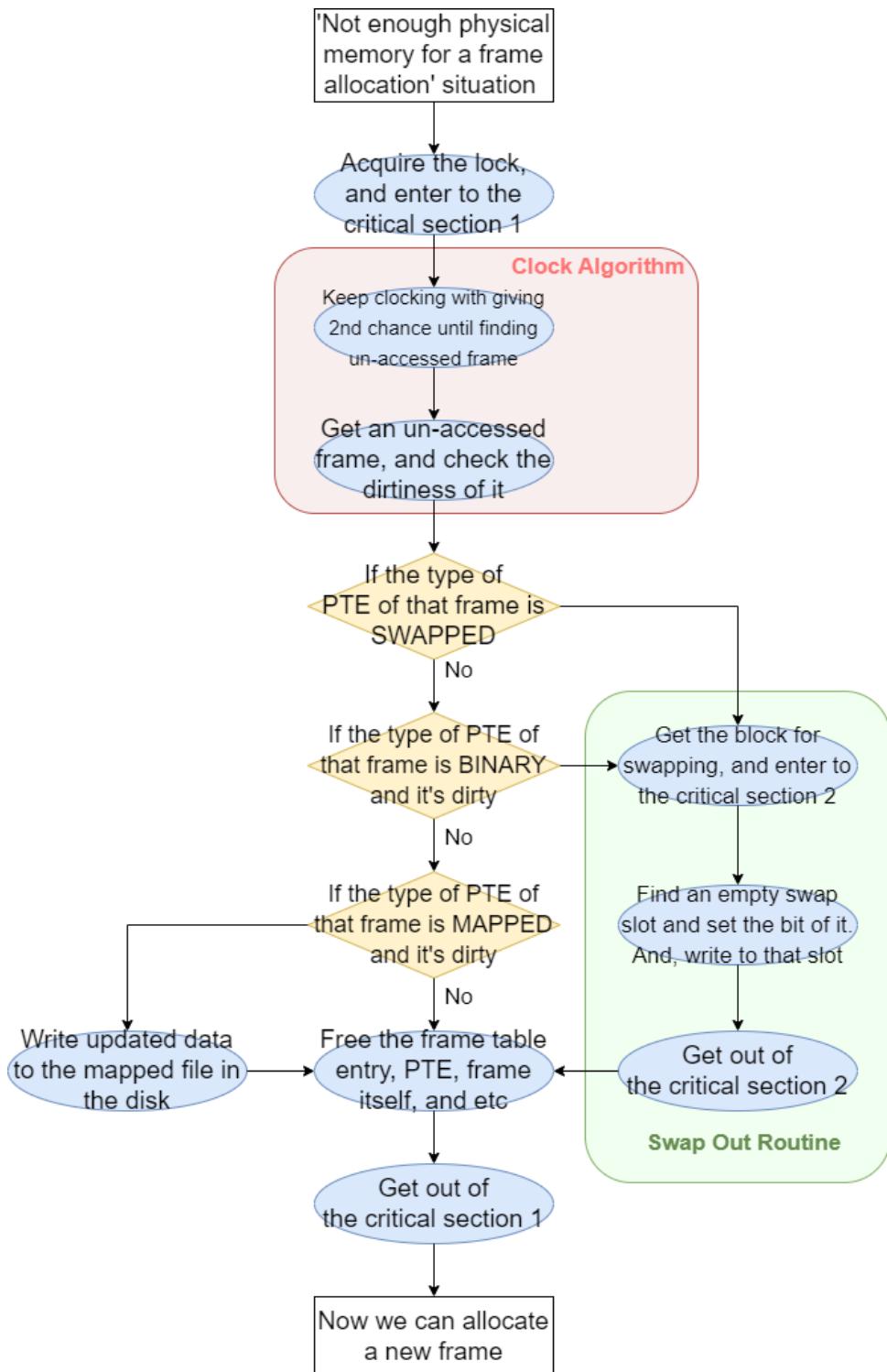
(1) Page Fault Handling



(3) Stack Growth 시의 Stack 확장 여부 판단 방법

(Stack Growth 확장 여부 판단 과정은 Page Fault Handling 과정의 한 분기 중 하나인 Stack Growth Flow로도 충분히 표현할 수 있기에, 두 항목을 하나의 Flow Chart로 합쳐서 위와 같이 표현하였다.)

(2) Disk Swap 발생 시의 Page Replacement Algorithm



(PTE의 Type에 대해선 이미 앞서 몇 차례 언급하였지만, 바로 아래 항목에서 한 번 더 자세히 설명할 것임)

B. 제작 내용

(1) Page Fault Handling

```
/* This header file is the core of the project 4 phase, that is, the core
of the lazy loading with paging. This header provides the supplemental page
table for the system, and you can see the details in little minutes.

Now, before looking those details, let me explain my implementation of
pintOS virtual memory concepts for this project 4.

As mentioned in the pintOS manual, the virtual address of pintOS just looks
like the figure below.

  31           12 11          0
+-----+-----+-----+
| Virtual Page Num | Offset |
+-----+-----+-----+-----+-----+-----+
```

And the format of physical address is very similar to this.

```
  31           12 11          0
+-----+-----+-----+
| Physical Frame Num | Offset |
+-----+-----+-----+-----+-----+
```

The virtual address is translated to the corresponding physical address via the page directory and the page table. These two are already provided by the basic pintOS, but the topic(problem) of this phase is how to apply the lazy loading concept with paging & replacement & memory mapping to this naive pintOS system, by following these steps below.

(1) First, we need to implement a supplemental page table which is the virtual(logical-side) page table of each thread.
-> 'vm/page.h'(this header)' takes in charge of this part, with a hash and the highly sophisticated page table entry format(structure).

(2) Second, we need to implement the lazy loading mechanisms by declaring a frame structure and the table(list) of those frames. In this step, we also have to consider a page(frame) replacement with swapping.
Maybe the hardest part of this project 4 phase.
-> 'vm/frame.h' and 'vm/swap.h' takes in charge of this part. The former provides a LRU(Least Recently Used)-based frame list, and the latter provides a swap table representation via bitmap and block structure.

(3) After those two consecutive steps, we can complete the lazy loading. However, this is not the end of this phase, cause we also need to implement a memory mapping concepts (some testcases use mmap() call).
-> This is easy part. 'vm/mmap.h' gives crucial memory mapping routines. 'syscall.h' in userprog directory will use this routines.

(4) Replace the previous naive loading mechanisms with these newly provided lazy loading mechanisms from (1) ~ (3), in 'process.h', 'exception.h', etc

You can see a detailed description of these steps in the attached report.
Now, let's get back to this 'page.h' file. Look at the below! */

이 Swapping되어야 함을 나타내는데, 여기엔 Stack Segment를 구성하는 Page, 그리고 BINARY Page에서 SWAPPED Page로 변화한 Page, 즉, 이전에 Victim이 된 경험이 있는 Page 등이 포함된다. MAPPED는 Memory Mapping 시 만들어진 Page를 의미한다.

```
/* Type of the page. */
typedef enum { BINARY, MAPPED, SWAPPED } pt_type;
```

이제, 본 Project에서 가장 중요한 자료구조라 할 수 있는 Supplemental Page Table의 PTE Format 정의 구조체를 보자 (다음 페이지의 코드). VPN, Page Type, Memory Load 여부, Write 가능 여부, Mapped File, File 내에서 얼마큼 읽었는지를 나타내는 Offset들 등 정말 다양한 Field가 존재한다. Swap Slot은 해당 Page가 Swapped 상태일 경우 어떤 Slot으로 Swap되었는지를 나타내는, 일종의 Index 변수이다.

(2) Disk Swap, Page Replacement Algorithm

우선, 앞서 말한 바와 같이, **Supplemental Page Table** 구축을 담당하는 **page.h**를 vm Directory에 만든다. 본인은 page.h의 시작 부분에 좌측 같은 Prologue 역할의 주석 부분을 만들어놓았다. 이 부분은 본 보고서에서 설명한 내용을 간략하게 압축한 것으로, 코드 이해에 조금 더 도움이 될 것이다. (영어 표현이 조금 어색할 수 있으며, 이러한 주석 설명은 지난 Phase들도 마찬가지였지만, 모든 수정 소스 파일에 명시되어 있음)

이어, 아래와 같이 각 Page의 Type을 나타내는 **Enum Type**과, Supplemental Page Table의 PTE Format을 나타내는 구조체를 정의한다. 우선, Enum Type부터 보자. **BINARY**는 Page가 자신의 주인 Process의 Binary Image File Loading 과정에서 왔음을 의미한다. **SWAPPED**는 기본적으로 이 Page가 Page Replacement 시 Dirty 여부와 상관 없

```

/* Structure below is a format of the entry of the 'supplemental page
   table' (From now on, I'm gonna call this as just 'page table').
   That is, each page table is implemented via this structure, for forming
   a hash table. The reason why a hash table is selected as data structure
   of page table is because it's simple and fast to search and modify. */
struct pt_entry
{
    /* Information about the page. */
    void *vaddr;           /* VPN(Virtual Page Number). */
    pt_type type;          /* Type of page indicated by this PTE. */
    bool writable;         /* Is it OK to write to this page? */
    bool is_loaded;        /* Is this page loaded onto physical memory? */

    /* Used for hash operations. */
    struct hash_elem elem; /* Hash element for each page table. */

    /* Variables about file mapped to this page. */
    struct file *file;     /* Pointer to the mapped file. */
    size_t offset;          /* Current file position of the file. */
    size_t read_bytes;      /* Number of bytes written on page. */
    size_t zero_bytes;      /* Number of rest of bytes of that page. */

    /* If this page is used for the memory mapping. */
    struct list_elem mm_elem; /* Iterator for the mmap list. */

    /* If this page is mapped to disk(swapping). */
    size_t swap_slot;       /* Index of the slot for swapping this. */
};


```

이제, 동일 Directory 내에 page.c 파일을 만들어, Page Table 관련 연산을 제공하는 함수들을 작성한다. 앞서 언급한 바와 같이 Supplemental Page Table을 **Hash로 구현**하기 때문에 본 함수들은 사실상 Hash Library 호출에 대한 Wrapper라고 볼 수 있다. 아래를 보자. PTE를 생성해 반환하는 간단한 함수이다.

```

/* Allocate and initialize the new entry of page(frame) table.
   And after the initialization, it returns the newly created entry.
   Yes, this procedure is used during the process-loading routine,
   which means, mainly in the 'load_segment' function in process.c. */
struct pt_entry *
pt_create_entry (void *vaddr, pt_type type, bool writable, bool is_loaded,
                 struct file *file, size_t offset, size_t read_bytes, size_t zero_bytes)
{
    /* Allocate the entry. */
    struct pt_entry *pte;
    if (!(pte = (struct pt_entry *)malloc(sizeof(struct pt_entry))))
        return NULL;

    /* Initialize the entry. */
    memset (pte, 0, sizeof(struct pt_entry));
    pte->type = type; /*******/ pte->file = file;
    pte->vaddr = vaddr; /*******/ pte->offset = offset;
    pte->writable = writable; /**/ pte->read_bytes = read_bytes;
    pte->is_loaded = is_loaded; /**/ pte->zero_bytes = zero_bytes;

    return pte;
}


```

memset을 사용해 PTE의 메모리 영역을 0으로 Setting하는 이유는, 해당 함수 호출자의 부정확한 Arguments Passing을 방지하고, 동시에 Field들을 초기화하기 위함이다.

이 함수는 이후 설명할 다양한 부분에서 등장하니 잘 기억해두자.

```

/* Pop the given entry from the page table. (Simple hash delete) */
bool
pt_delete_entry (struct hash *pt, struct pt_entry *pte)
{
    if (!hash_delete (pt, &(pte->elem)))
        return false;

    /* If hash deletion is success, then deallocate all the data
       structures that are related to this page table entry.
       - Page indicated by this entry
       - Swap slot (if exists)
       - Page Table Entry itself
       These three things are now deallocated! */
    free_page (pagedir_get_page (thread_current ()->pagedir, pte->vaddr));
    swap_free (pte->swap_slot);
    free (pte);

    return true;
}

```

한편, PTE를 Page Table에서 제거하는 함수는 위와 같이 작성한다. PTE에 대한 Free는 곧 해당 Page가 Invalid해짐을 의미하고, 따라서 Frame Deallocation, 혹시 모를 Swap Space 점유 구간 Clearing 등이 함께 이뤄짐을 기억하자.

이 밖의 나머지 Page Table 관련 함수는 함께 제출한 코드에서 주석들을 통해 자세히 확인할 수 있다. 여기선, 해당 함수들의 간단함 및 보고서 분량 등을 이유로 생략하겠다.

이제 Supplemental Page Table을 사용할 준비가 되었으니, 이들을 각 Process가 가질 수 있도록 만들어주자. 'threads/thread.h'의 Thread Structure에 다음과 같이 Hash Table을 추가하자. 이에 대한 Initialization은 process.c의 'start_process'에서 'pt_init'을 호출함으로써 간단히 수행할 수 있을 것이며, 반대로 Process 종료 시엔 Hash Table이 해제되도록 만드는 것도 잊지 말자.

```

#ifndef USERPROG
/* Owned by userprog/process.c. */
uint32_t *pagedir;           /* Page directory. */
struct semaphore load_lock; /* Semaphore for i */
struct thread *parent;      /* This is a helper */

struct semaphore parent_lock; /* Semaphore for i */
struct semaphore child_lock; /* Sem for suspend */

struct list child_list;     /* List storing ch
struct list_elem child_elem; /* List element de

struct hash pt;             /* Supplemental Pa

struct file *fd[FD_MAX];   /* File Descriptor
struct file *file;          /* Mapped file of

struct list mm_list;        /* List of loaded-
unsigned mm_list_size;     /* Number of mmap

int exit_status;            /* Exit status of
#endif

static void
start_process (void *file_name_)
{
    char *file_name = file_name_;
    struct intr_frame if_;
    bool success;

    /* Initialize the page table of newly created process. */
    pt_init (&(thread_current ()->pt));
}

/* Free the current process's resources. */
void
process_exit (void)
{
    struct thread *cur = thread_current ();
    uint32_t *pd; unsigned i;

    /* If there're mmaped pages that are not freed yet,
       deallocate all of it, by calling munmap syscall. */
    for (i = 1; i < cur->mm_list_size; i++) munmap (i);

    /* Close the mapped file of this(current) thread. */
    file_close (cur->file);

    /* Deallocate the page table. */
    pt_destroy (&(cur->pt));
}

```

('process_exit'에서 munmap System Call을 통해 현재 종료 중인 Process에서 해제되지 않고 남아있는 Memory Mapping들을 모두 Free하는 루틴을 확인할 수 있는데, 이는 Memory Mapping 관련 Test Case도 모두 통과하기 위해 추가한 것이다.)

자, 여기서 잠시 멈추어, **Page Fault Handler**를 바로 확인하지 않고, (2)번 개발 항목인 **Page Replacement, Swapping**을 먼저 분석하자. 이유는, Page Fault Handler가 이 두 개념이 완성되고 나야 비로소 완성될 수 있기 때문이다. 먼저, vm Directory 안의 frame.h를 보자. 이전에 설명한 바와 같이, 이는 **Frame Table**의 관리를 담당한다. 아래는 Frame Table을 구성하는 Element에 대한 Structure, 즉, Frame Structure이다. Frame Table을 관리하기 위해선 필수적으로 알아야 하는 정보(이를 테면 PFN부터, 해당 Frame에 연결된 Page를 소유한 Thread가 누구인지 등)들을 넣었다.

```
/* Structure below has an information about the frame. You know, this is
not about the supplemental page table entry, this is about some metadata
related to the physical frame.

* ALERT: Note that the pintOS usually calls '(physical) frame' just
as 'page'. Thus, the interface of this header like 'alloc_page' uses
a word 'page', but in fact it's 'frame'. We should keep in mind it. */
struct frame
{
    void *kaddr;           /* Physical Address of this frame. */
    struct thread *thread; /* The thread who uses this frame. */
    struct pt_entry *pte;  /* Pointer to the mapped PTE for this. */
    struct list_elem frame_elem; /* Iterator for the page replacement. */
};
```

이어, **System-Wide Frame Table**과 **Frame Clock**을 아래와 같이 선언한다. Frame Table은 List 형태이며, Frame Clock은 '**Second Chance (Clock) Algorithm**'에서 Frame Table을 Circular Order로 순회하며 'Evict할 Victim Frame'을 선정하는 Pointer이다.

```
/* Frame table based on the list structure. */
struct list frame_list;

/* Global iterator that cycles the frame table.
That is, we use a clock algorithm to search for evicting. */
struct list_elem *frame_clock;
```

한편, System-Wide하다는 것은 곧 Shared Data Structure를 의미하므로, 이들에 대한 접근을 일렬화할 Mutex Lock도 아래와 같이 frame.c에 내부적으로 선언하자.

```
/* For the synchronization of accessing frame table. Note that when this
mutex lock is used is not the beginning and end of each frame table
subroutines, but the front and back of the call instruction of each
subroutines, because interfaces that this header provides use those frame
table subroutines in a nested fashion in some cases, which could cause a
subtle synchronization error that can be detected by 'make check' via
'lock_held_by_current_thread' assertion. Thus, remember this point.*/
struct lock frame_lock;
```

이어, Frame Table 관련 기본 연산을 제공하는 함수들을 설계한다. 대부분은 page.h의 함수들처럼 단순한 pintOS 제공 Library 호출 기능을 수행하지만, 아래와 같은 부분들은 주목할만하다.

```

/* Delete the list entry(frame) from the frame table. The target
   frame must be equal to the current global clock iterator. */
static void
ft_delete_frame (struct frame *frame)
{
    struct list_elem *entry, *ret;

    entry = &(frame->frame_elem);
    ret = list_remove (entry);

    /* If the deleted element is equal to the current global
       clock iterator, then update it to the next one(frame). */
    if (entry == frame_clock)
        frame_clock = ret;
}

/* Cycle(clock) the frame table, by making the current global
   iterator move to the next position (in a circular way). */
static struct list_elem *
ft_clocking (void)
{
    /* If the iterator reaches the end of the list, then get
       back to the front of the swap table (list). */
    if ((frame_clock == NULL) || (frame_clock == list_end (&frame_list)))
    {
        if (!list_empty (&frame_list))
            frame_clock = list_begin (&frame_list);

        return frame_clock;
    }

    /* If not, just move to the next. If the next one is the end
       of the table (list), then do this procedure once again. */
    frame_clock = list_next (frame_clock);
    if (frame_clock == list_end (&frame_list))
        frame_clock = ft_clocking ();

    return frame_clock;
}

/* Get the first unaccessed frame from the frame table, based on
   the LRU(Least Recently Used) policy. To implement this policy,
   we can use some useful functions defined in the 'pagedir.h',
   which provides routines to check accesses of given page(frame). */
static struct frame *
ft_get_unaccessed_frame (void)
{
    struct list_elem *g_iter; struct frame *entry;

    /* Find all the pages whose accessed bit is true, and set
       those bits as false. Keep doing this until we first find
       the page whose accessed bit is false. (in the frame table) */
    while (1)
    {
        g_iter = ft_clocking ();
        entry = list_entry(g_iter, struct frame, frame_elem);

        if (!(pagedir_is_accessed (entry->thread->pagedir, entry->pte->vaddr)))
            return entry;

        pagedir_set_accessed (entry->thread->pagedir, entry->pte->vaddr, 0);
    }
}

```

'ft_delete_frame' 함수는 단순히 List Deletion 함수를 호출해 사용하지만, Frame Table에서 삭제된 Entry가 현재 Frame Clock과 동일하면, Frame Clock 을 삭제된 Entry 다음의 Entry로 바꿔 주고 있음을 주목하자. Frame Table 위를 움직이는 **Global Clock Iterator**의 존재가 느껴지는 부분이다.

이어, 좌측을 보자. 이 함수는 Table에 대한 Clock Algorithm이 돌아가게 하는 **Clocking** 함수이다. Frame Clock이 Frame Table의 끝에 도달하면 앞으로 움직여주고, 그 외에 일반적인 상황에선 Frame Clock이 Next Entry를 가리키도록 만들면 된다. 다만, Next Entry가 List End에 해당하면, pintOS에서 제공하는 List 자료구조의 특성 (list_end는 Tail Node를 반환함. 즉, Data가 없음)을 고려해, 재귀적으로 Clocking을 한 번 더 수행해주자.

이제, **Frame Eviction** 기능을 알아보자. 우선, 좌측의 함수를 먼저 정의한다. Frame Table의 Entry(Frame)들을 쭉 훑어가며, Second Chance (Clock) Algorithm을 수행한다. Mapped Page의 Accessed Bit가 True인 Frame들은 모두 Un-accessed로 바꿔주고, 이러한 행위를 Un-accessed Frame을 마주칠 때까지 반복하는 것이다. Frame Table을 Clocking하면서 말이다. 이 함수는 아래와 같이 Eviction Main 함수에서 호출된다.

```

/* If there's a need for eviction of the frame, then search the unaccessed
frame from the frame table with clock algorithm. After find it, then
check the dirtiness of that frame and the type of the mapped PTE, and
perform the corresponding routine for the dirtiness and the type.
(Therefore, this routine uses an approximate LRU(Least Recently Used)
algorithm) */
static void
ft_evict_frame (void)
{
    struct frame *frame;
    bool is_dirty;

    /* Find an unaccessed frame and check the dirtiness of it. */
    frame = ft_get_unaccessed_frame ();
    is_dirty = pagedir_is_dirty (frame->thread->pagedir, frame->pte->vaddr);

    /* If the selected frame is from a memory-mapped file, then
       write data to that file if it's dirty, and evict it. */
    if (frame->pte->type == MAPPED && is_dirty)
        file_write_at (frame->pte->file, frame->kaddr,
                      frame->pte->read_bytes, frame->pte->offset);

    /* If it's from the swap space, then simply swap out. */
    else if (frame->pte->type == SWAPPED)
        frame->pte->swap_slot = swap_out (frame->kaddr);

    /* If it's from the dirty binary file, then swap out. */
    else if (frame->pte->type == BINARY && is_dirty)
    {
        frame->pte->swap_slot = swap_out (frame->kaddr);
        frame->pte->type = SWAPPED;
    }

    /* Frame eviction occurs here, by clearing it from
       the frame table, page directory, memory. */
    frame->pte->is_loaded = false;
    ft_delete_frame (frame);
    pagedir_clear_page (frame->thread->pagedir, frame->pte->vaddr);
    palloc_free_page (frame->kaddr);
    free (frame);
}

```

좌측의 함수는 앞선 '개발 내용' 파트에서 설명한 Frame Eviction을 수행하는 핵심 함수이다. 바로 위에서 정의한 함수를 호출해 Accessed Bit가 False인 Frame을 찾는다. 이어, 해당 Frame의 Mapped Page를 PTE를 통해 찾은 후, Page의 VPN을 이용해 pintOS 제공 pagedir_is_dirty 함수를 호출하여 해당 Page의 Dirty 여부를 판단한다.

Frame이 Dirty이며, 동시에, 연결된 Page의 Type이 BINARY이면, 해당 Frame은 Disk의 Swap Space로 Swap Out 시킨다. Type을 SWAPPED로 바꿔주며 말이다. 그리고, Frame의 Mapped Page Type이 SWAPPED이면 Dirty 여부 상관 없이 Swap Out 시킨다. 이 이유는 위에서 여러 차례 설명했으므로 생략한다. 참고로, Swap Out은 Swap Slot Index를 반환한다.

Frame이 Memory Mapping으로 마련된 Frame이면, Dirty할 경우에만 Mapped File로 Data를 Write해 Update한다. 주목할 점은, Memory Mapping으로 만들어진 Frame은 Swap Space에 보관해두지 않는다는 점인데, 이는 Aliasing을 막고, File을 최신으로 유지하기 위함이다. Swap Space에 둘 경우 Swap In 시 Inconsistency가 발생할 수 있기 때문에, 그냥 Evict해버리고, Page Fault 시 다시 Load하는 방식을 선택한다.

이렇게 Frame Table 연산과 Clock Algorithm, Eviction이 구축되면, 다음과 같이 기존 pintOS의 Frame 할당/해제를 담당한 '**threads/palloc.h**'의 함수들을 대체할 새로운 **Frame Allocation/Deallocation** 함수들을 마련한다. (다음 페이지)

'alloc_page'는 기본적으로 'palloc_get_page'를 통해 Empty Frame의 PFN을 받는다. 다른 점은, 바로 Page Replacement을 위해 해당 PFN이 Invalid할 경우, 할당 가능한 Frame이 생길 때까지 Page Replacement with Swapping을 수행한다는 것이다. 앞서 정의한 'ft_evict_frame' 함수를 while 문 안에 넣고, Valid PFN이 얻어질 때까지 반복하는 것이다. 아래를 보자.

```

/* It creates a new physical frame and initializes some metadata about
it. That is, this function completely replaces 'palloc_get_page' func
used in the previous phase (especially in 'userprog/process.c' file).

The main job is surely the frame allocation, just like the previous one,
but the lazy loading concepts applied to here. That is, the frame will
be allocated, but there can be a frame from the swap space(a.k.a. page
replacement concepts), and it is managed by the supplemental page table
also (possibly).

* ALERT: Note that the pintOS usually calls '(physical) frame' just
as 'page'. Thus, a word 'page' in here is in fact a 'frame'. We should
keep in mind it when read the below codes. */
struct frame *
alloc_page (enum palloc_flags flags)
{
    struct frame *page; // page == frame //
    if (!(page = (struct frame *)malloc(sizeof(struct frame))))
        return NULL;

    memset(page, 0, sizeof(struct frame));
    page->thread = thread_current ();
    page->kaddr = palloc_get_page (flags);

    /* If there's no enough space for the allocation, then
       evict the specific frame from the frame table, and by
       this eviction, there will be a new physical frame. */
    while (page->kaddr == NULL)
    {
        lock_acquire (&frame_lock);
        ft_evict_frame ();
        lock_release (&frame_lock);
        page->kaddr = palloc_get_page (flags);
    }

    /* Insert the newly created frame into the frame table. */
    lock_acquire (&frame_lock);
    ft_insert_frame (page);
    lock_release (&frame_lock);

    return page;
}

```

```

/* It frees a frame indicated by the passed physical address. That is,
remove it from the frame table, from the page directory, and deallocate
it. During this procedure, there should be a mutual exclusion.

* ALERT: Note that the pintOS usually calls '(physical) frame' just
as 'page'. Thus, a word 'page' in here is in fact a 'frame'. We should
keep in mind it when read the below codes. */
void
free_page (void *kaddr)
{
    struct frame *page;

    lock_acquire (&frame_lock);

    if ((page = ft_find_frame (kaddr)) != NULL)
    {
        ft_delete_frame (page);
        pagedir_clear_page (page->thread->pagedir, page->pte->vaddr);
        palloc_free_page (page->kaddr);
        free (page);
    }

    lock_release (&frame_lock);
}

```

```

/* Load a file from the disk onto the physical memory. After loading,
the remaining part of the given frame will be set to zero. */
bool
load_file_to_page (void *kaddr, struct pt_entry *pte)
{
    bool success;

    /* Read(load) the file onto the memory. */
    size_t read_byte = pte->read_bytes;
    size_t temp = (size_t)file_read_at (pte->file,
                                       kaddr, pte->read_bytes, pte->offset);

    /* Set all the remaining bytes of that frame to zero,
       only if the file read operation was successful. */
    success = (read_byte == temp);
    if (success)
        memset (kaddr + pte->read_bytes, 0, pte->zero_bytes);

    return success;
}

```

이 세 함수는 frame.h의 Interface로서, Frame Allocation/Deallocation을 담당하고, load_file_to_page 함수는 Page Fault 시 특정 Page에 특정 File을 특정 부분을 Load하는 역할을 수행한다. 이제, Page Fault Handler를 구축할 준비가 거의 다 됐다.

하지만, 아직 Swapping을 확인하지 않았다. **Swapping은 vm에 swap.h 파일을 만들어 전달시킨다.** 즉, frame.h와 swap.h가 협업하여 Page Replacement를 구현하고 있는 것이라 보면 된다. frame.h는 Frame Table 파트를, swap.h는 Swap Table 파트를 담당한다.

```
/* This library provides data structures for the page replacement
concepts (a.k.a Swapping). We will use a LRU(Least Recently Used)
policy for this implementation.
And note that this 'swap' library will work with 'frame' header file
in the same directory. Various functions in both libraries will be
used for performing the second chance (clock) algorithm in this pintOS.

* You should first allocate a swap disk in the 'vm/build' directory
by 'pintos-mkdisk swap.dsk --swap-size=n' instruction.
-> swap.dsk will be automatically attached to the hdb1 while booting.

* Note that the difference between 'frame' and 'swap' is, functions
in the 'frame.h' manage the frame table by performing list operations.

In the contrast, functions in 'swap.h' manage the swap table, with
'actually' accessing to and working on the swap disk, by using the
evicted(on LRU) frame provided from the frame table of 'frame.h' */

/* Bitmap indicates the 'freed or allocated' info.
That is, it represents the status of the swap table. */
struct bitmap *swap_bitmap;
#define OFS_ZERO 0      /* Starting offset of the block structure. */
#define OFS_MAX 8       /* Maximum offset of the block structure. */
```

System-Wide Data Structure인 Swap Bitmap을 선언하고 있음을 주목하자. 그리고 이러한 swap.h는 다음과 같은 Interface들을 제공해 앞서 확인한 frame.h에서 요긴하게 사용하는 것이다. 이들은 모두 pintOS에서 미리 제공하는 Block Device I/O 함수들과 Bitmap Data Structure로 구현할 수 있는데, **직관적이고 간단하기 때문에 내부 코드는 생략한다.**

```
/* These only four functions are interfaces of this header.
swap_init should be called in the beginning of the system,
and the other three will be used in the frame management.
(That is, main user(customer) of this header is 'frame.c') */
void swap_init (void);
void swap_in (size_t index, void *kaddr);
size_t swap_out (void *kaddr);
void swap_free (size_t index);
```

자, 이제 모든 준비를 마쳤다. **Frame Table과 Swap Table이 마련되었으니, 이들을 통해 Page Fault Handler를 수정해, 나아가 Demand Paging, Lazy Loading을 실현할 수 있을 것이다.** 이를 위해 가장 먼저 해야 할 일은, System-Wide Data Structure인 Frame Table과 Swap Table을 pintOS에 설치하는 것이다. 이는 'threads/init.c의 pintOS Main 함수'에서 아래와 같이 수행하자.

```
/* Initialize the frame table of the system. */
ft_init ();
/* Initialize the swap table of the system. */
swap_init ();
```

자, 이어 'userprog/process.c'로 이동하자. 이곳엔 Demand Paging을 관장할 핵심 함수들이 즐비하다. 먼저, 'load' 함수를 보자.

```

/* Open and map an executable file. Note that this file system access must
be protected by mutex lock. We should keep in mind that a synchronization
is the most important thing in the lazy loading implementation. */
lock_acquire (&access_lock);

file = filesys_open (file_name);
if (file == NULL)
{
    lock_release (&access_lock);
    printf ("load: %s: open failed\n", file_name);
    goto done;
}
t->file = file;

lock_release (&access_lock);

```

좌측은 load 함수 내부이다. 기존과 다르게, Program의 Binary File을 열 때 Synchronization을 적용했다. 아직 언급하지는 않았는데, Demand Paging에선 **File System 접근에 대한 Mutual Exclusion**이 굉장히 중요하다. 왜냐? 복수의 Process가 동시에 다발적으로 Page

Fault를 일으키고, File System을 접근해 Load하고 Update하고 그럴 것이기 때문이다. 따라서, File 접근이 반드시 동기화되어야 한다. 따라서, 먼저 이 부분의 파일 오픈 작업을 동기화시킨다. 그리고, Thread Structure의 file Field에 이 Binary File을 기록해둔다. 이러한 동기화 작업은 'userprog/syscall.c'의 다양한 File 관련 System Call들에도 적용되어야 할 것이다. (이를 위해 syscall.h의 File System Lock인 '**access_lock**'을 extern으로 가져온다)

이어 이 load함수는 'load_segment' 함수를 호출해 File Image에서 Data를 뽑아 **Lazy Loading을 수행한다**. 바로 Program의 File Image를 모조리 Load하는 것이 아니라, Program에 대한 PTE만을 만들어 Supplemental Page Table을 구축해놓고, 실질적인 Loading은 Page Fault Handler에게 맡기는 것이다. (즉, 일부만 Load될 것임)

```

while (read_bytes > 0 || zero_bytes > 0)
{
    /* Calculate how to fill this page.
       We will read PAGE_READ_BYTES bytes from FILE
       and zero the final PAGE_ZERO_BYTES bytes. */
    size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
    size_t page_zero_bytes = PGSIZE - page_read_bytes;

    /* Create a page table entry for this, and push to the
       page table. Note that this is not a loading, this is
       just constructing the page table only. (Lazy Loading) */
    pte = pt_create_entry (upage, BINARY, writable, false,
                          file, ofs, page_read_bytes, page_zero_bytes);
    if(pte == NULL) return false;

    pt_insert_entry (&(thread_current ()->pt), pte);

    /* Advance. Note that the offset is updated. */
    read_bytes -= page_read_bytes;
    zero_bytes -= page_zero_bytes;
    ofs += page_read_bytes;
    upage += PGSIZE;
}

```

그리고, 'setup_stack' 함수도 아래와 같이 변형한다 (다음 페이지). 이 함수는 알다시피 Stack Segment를 위한 Page를 마련하는 함수인데, 기존과 다르게 이 역시 **PTE만을 구축** 한다. 참고로, PTE의 Type을 SWAPPED로 설정하고 있음에 주목하자. (이 이유는 이미 앞서 여러 차례 설명한 바 있음)

```

static bool
setup_stack (void **esp)
{
    struct frame *kpage;
    bool success = false;

    /* We should note that the function 'alloc_page' below provides a frame
       allocation based on the lazy loading concepts. You can see more details
       about it in 'vm/frame.h' file, and in here, it's enough to know that
       this function returns a newly created page-sized frame. */
    kpage = alloc_page (PAL_USER | PAL_ZERO);
    if (kpage != NULL)
    {
        /* Record this new frame to the 'read(non-supplemental)' page table.
           And then, set the esp pointer value as recommended. */
        success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage->kaddr, true);
        if (success)
        {
            *esp = PHYS_BASE;

            /* After installing pages for the stack segment, then
               create a page table entry for these pages and push
               it to the supplemental page table of current thread. */
            kpage->pte = pt_create_entry (((uint8_t *)PHYS_BASE) - PGSIZE,
                                           SWAPPED, true, true, NULL, 0, 0, 0);
            if (kpage->pte == NULL) return false;

            pt_insert_entry (&(thread_current ()->pt), kpage->pte);
        }
        else
            free_page (kpage->kaddr);
    }

    return success;
}

```

굉장히 간단하다. 우리가 주목할 부분은 kpage 변수에 'alloc_page' 함수의 반환 값을 대입하는 부분이다. 앞서 정의한 'alloc_page' 함수는 Frame Allocation을 수행하되, Frame Table과 Swap Table을 이용해 Frame이 부족할 경우 Page Replacement을 수행해 Frame을 마련하는 함수임을 우리는 알고 있다. User Stack을 위한 최소한의 Page를 미리 할당해놓는 것이다.

```

/* If the frame of faulting address is present, then
   it's an abnormal accessing situation, so terminate! */
if (!not_present)
    exit (-1);

/* If not, get the PTE of faulting address, and analyze. */
pte = pt_find_entry (fault_addr);
/* (1) If it's not a valid reference, then check if it's in
   a growable region. If yes, expand the user stack. */
if (!pte)
{
    if (!expand_stack (fault_addr, f->esp))
        exit (-1);
}
/* (2) If it's a valid reference, then get an available frame.
   'handle_mm_fault' function in process.c will do this. */
else
{
    if (!handle_mm_fault (pte))
        exit (-1);
}

/* As you can see a flow above, any unexpected condition will
   be treated as a 'segmentation fault' situation. (exit(-1))
   And, if there's no segmentation/protection fault, that is,
   if it is the (1) stack-growth situation or (2) page fault
   situation, then the system will restart the process. */

```

자, 이제 Supplemental Page Table에 Process Loading을 위한 준비를 마쳤고, 동시에 User Stack도 구현해놓았다. 이제 할 일은 Page Fault Handler의 수정이다. Demand Paging을 구현하려는 것이다. 따라서 'userprog/exception.c'를 보자.

Page Fault가 발생하면 Faulting Address가 소속된 Virtual Page에 대해 PTE가 존재하는지를 'pt_find_entry' 함수로 확인한다. **PTE가 있다면**, 이는 '**load_segment**'에서 미리 마련된 Page일 것이고, 이는 곧 '일반적인 Page Fault 상황'을 의미한다. 반대로 PTE가 없다면, 이는 Stack Segment를 확장해야 하는 상황일 수 있다.

Stack Growth는 다음 항목에서 서술하고, '일반적인 Page Fault'를 확인하자. Project 안내 PDF에 따라 'handle_mm_fault' 함수를 호출한다. 이 함수의 Return Value가 false이면 Protection Fault를 일으킬 것이다. (Stack Growth도 마찬가지로, 실패하면 SegFault임)

'handle_mm_fault' 함수는 process.c에 정의되어 있다. 아래와 같이 말이다.

```
(See the previous description in the comment of 'stack-growth routine')

'page_fault()' function in exception.c calls this function if the
faulting address is from the valid reference, which means the type
of the fault is not a segmentation or protection fault.

Meanwhile, in case you are curious about that why this 'page fault
handling' routine is declared in 'process.h' not in 'exception.h',
I brought the reason, that is, it's because of 'install_page' func.
The basic pintOS system want 'install_page' func remains as static,
so I choose to declare this function here. Not a big reason. */

bool
handle_mm_fault (struct pt_entry *pte)
{
    struct frame *kpage;
    bool success = false;

    /* Allocate a new physical frame and map to the passed PTE.
       This frame possibly replaces the original virtual page.*/
    kpage = alloc_page (PAL_USER);
    kpage->pte = pte;

    /* What is a type of the virtual page of the faulting address?
       --> here are two cases by type of page, just like below.
       (1) If it's the binary file or the mmapped file, then simply
           load related data from the same file in the disk-side.*/
    if (pte->type == BINARY || pte->type == MAPPED)
    {
        if (load_file_to_page (kpage->kaddr, pte))
            success = install_page (pte->vaddr, kpage->kaddr, pte->writable);
    }

    /* (2) If it's the page that are from the swap space but not in
       the memory right now, just swapping in that frame. Note that in
       both cases we just install the newly created frame into system. */
    else if (pte->type == SWAPPED)
    {
        swap_in (pte->swap_slot, kpage->kaddr);
        success = install_page (pte->vaddr, kpage->kaddr, pte->writable);
    }

    /* If installation(frame-to-page mapping in 'real'
       page table) is done, then set this page as 'loaded'.
       If installation failed, then free that newly created frame. */
    if (success)
        pte->is_loaded = true;
    else
        free_page (kpage->kaddr);

    return success;
/* If it reaches here with success == true, then the loading is successful.
   Thus, 'page_fault()' will return properly, and the system will execute
   the faulting instruction once again. (fault exception handling) */
}
```

System이 재-실행하게 될 것이다.

이렇게, (1) Page Table & Page Fault Handling과 (2) Disk Swap & Page Replacement Algorithm에 대한 코드 설명을 마친다.

우선, 'alloc_page' 함수를 이용해 Frame Allocation을 수행한다. 이어, Fault를 일으킨 Address가 포함된 Page의 Type을 확인한다. Page Type이 BINARY이면 해당 Page의 Mapped File에서 Data를 Load해 Frame으로 옮기면 될 것이다. 이어, 그 Frame을 pintOS 제공 'install_page' 함수를 통해 '실제 Page Directory'에 연결 관계를 기입하도록 한다. Page가 Memory Mapping으로 생성된 경우에도 같은 행위를 해주자.

Page가 SWAPPED인 경우에는, Swap Space에서 Frame을 Swap In 해오면 된다. 마찬가지로 Data가 Frame에 마련되면, Install하자.

이렇게 해서 Demand Paging이 완료되면, 해당 Page의 PTE에 '이 Page가 Memory에 Load되었음'을 기록해준다.

만약 이 과정에서 오류가 발생 했으면 Frame을 해제하고, False 를 반환해 Protection Fault를 일으키도록 하자. 오류가 없이 수행 되었으면, 'page_fault' 함수로 돌 아간 후, Faulting Instruction을

(3) Stack Growth

'setup_stack' 함수를 통해, 우리는 Process 최초 실행 시 Process Execution을 위해 필수적인 User Stack을 초기 Page 할당으로 마련했음을 확인했다. 그리고, Page Fault 발생 시 Faulting Address에 대한 PTE가 없으면 Stack Expansion을 고려하라고 했다. 그래서 위에 첨부한 'page_fault' 함수를 보면 'expand_stack'이란 함수 호출 부분을 볼 수 있고, 그 함수는 아래와 같이 정의되어 있다. (userprog/process.c에 정의)

```
bool
expand_stack (void *addr, void *esp)
{
    void *upage;
    struct frame *kpage;
    bool success = false;

    /* Is it OK to expand the stack in this case? That is, check if the
       faulting address can be within the 8MB range from the current stack
       pointer address, and whether it's from the user-virtual area, and
       qualify the PUSHA condition also. All these 3 checks should be passed. */
    if (!is_user_vaddr (addr)) return false;
    if (addr < (PHYS_BASE - MAX_STACK_SIZE)) return false;
    if (addr < (esp - 32)) return false;

    /* Get the nearest page boundary, to 'upage'. */
    upage = pg_round_down (addr);

    /* If the previous checking was successful, then expand
       the stack just like the way we set up the stack, except
       for the setting routine of the esp pointer. */
    kpage = alloc_page (PAL_USER | PAL_ZERO);
    if (kpage != NULL)
    {
        success = install_page (upage, kpage->kaddr, true);
        if (success)
        {
            kpage->pte = pt_create_entry (upage, SWAPPED, true, true,
                NULL, 0, 0, 0);
            if (kpage->pte == NULL) return false;

            pt_insert_entry (&(thread_current ()->pt), kpage->pte);
        }
        else
            free_page (kpage->kaddr);
    }

    return success;
}
```

함수 초반에는, 넘겨 받은 Faulting Address가 **User Virtual Address Space** 내에 있는지를 확인한다. 당연히 그 범위 내에 있어야 할 것이며, 그렇지 않다면 이 역시 Protection Fault이다. 이어, 해당 **Address**가 **User Stack**의 **Size Limit** 내에 있는지를 확인한다. 더 이상 Stack을 확장하면 안 되는 상황을 방지하려는 것이다.

마지막으로는 **80x86 System** 상의 **Assembly PUSHA** 연산 조건을 검사한다. Stack Pointer로부터 32Bytes 이내에 Faulting Address가 포함되어야 한다는 조건이다. 이렇게 총 3개의 조건을 모두 만족해야 Stack Expansion이 가능하다. 하나라도 만족하지 않는다면 Error를 표출하고 Program을 즉이도록 한다.

이후, Frame을 할당하고, 대응되는 Virtual Page와 Frame을 Mapping하고, 대응되는 PTE를 만드는, 일련의 과정이 위 코드처럼 이뤄진다. 어렵지 않게 이해할 수 있다.

이러한 Stack Expansion은 Page Fault 상황 외에도, 앞선 항목들에서 몇 차례 언급한 바와 같이, **System Call Argument Passing** 상황에서도 고려되어야 한다. Project 1에서 설계한 것처럼, pintOS System Call은 Argument를 Stack Pushing 방식으로 전달되는데, 이때, Pushing을 위한 공간이 부족한 상황이 있을 수 있기 때문이다. 따라서, 아래와 같이 이전 Phase에서 정의했던 Macro를 수정한다.

```
/* Macro functions for forming an argument that will be passed to syscall.
 - ARG_ADDR: returns the address of given passed-by-syscall arguments.
 - ARG: returns the value of given pointer, by casting of 'type'.      */
#define ARG_ADDR(k) ((uint8_t*)esp + 4*k)
#define ARG(k, type) *(type*)(ARG_ADDR(k))

/* Macro functions for checking passing arguments of syscall.
 - POINTER_CHECK: checks if an argument is in the user address space
   with pre-provided 'is_user_vaddr' function. And, check NULL also!
 - STACK_CHECK: checks if there's no mapping page for the current virtual
   address, then try to expand the user stack if it's possible! That means
   this macro should be called for each argument passing of syscall.
 - USER_ADDR_CHECK: checks all the parameters that a system call needs
   with consequently calling 'POINTER_CHECK' macro above!                  */
#define POINTER_CHECK(vaddr) if (vaddr == NULL || is_user_vaddr (vaddr) == false) exit (-1);
#define STACK_CHECK(vaddr, esp) if(!pt_find_entry (vaddr)) { if (!expand_stack (vaddr, esp)) exit (-1); }
#define USER_ADDR_CHECK(param_num, esp) for(int i=1;i<param_num;i++){POINTER_CHECK(ARG_ADDR(i)); STACK_CHECK(ARG_ADDR(i), esp); }
```

STACK_CHECK라는 매크로 함수를 만들어, 매 Argument Passing 시, 해당 Argument의 Address에 대해 PTE가 있는지를 확인하고, PTE가 없다면 Stack Expansion을 수행하는 것이다. 그리고 이 함수가 USER_ADDR_CHECK 매크로 내부에서 호출되고, 이 매크로는 아래와 같은 위치에서 사용된다. (Project 1에서 소개한 바 있음)

```
case SYS_FILESIZE: /* Obtain a file's size. */
    USER_ADDR_CHECK(1, esp);
    f->eax = filesize (ARG(1, int));
    break;

case SYS_READ: /* Read from a file. */
    USER_ADDR_CHECK(3, esp);
    f->eax = read (ARG(1, int), ARG(2, void *), ARG(3, unsigned));
    break;

case SYS_WRITE: /* Write to a file. */
    USER_ADDR_CHECK(3, esp);
    f->eax = write (ARG(1, int), ARG(2, void *), ARG(3, unsigned));
    break;
```

매 Argument Passing마다 Stack Expansion이 고려되고 있음을 알 수 있다. 이렇게 해서 Stack Growth까지 구현을 마치게 된다. 이로써, **Demand Paging, Lazy Loading**이 완성되었다.

그런데, 여기서 'make check'를 돌려보면, 'page-merge-mm' Case를 지속적으로 통과하지 못하는 상황이 발생한다. 원인을 분석해보면, 해당 Test Case가 mmap() System Call을 호출해서 그렇다(라고 본인은 판단하였음). 따라서, 마지막으로 **Memory Mapping**도 구현하자. vm Directory에 mmap.h 파일을 만들어 이 일을 전담시킨다.

아래는 mmap.h 파일의 모습이다.

```
/* This header file includes crucial data structures for the memory
   mapping routines of this pintOS system, and implements interfaces
   such as mmap() and munmap(). */

/* Mapping Identifier type. Note that 'mapid_t' is declared both in
   here and 'userprog/syscall.h'. It doesn't matter cause it's just uint. */
typedef unsigned mapid_t;

/* Data structures for the implementation of the mmap() system call.
   Note that the call for mmap()/munmap() takes place in 'syscall.c' file
   in the userprog directory. In here, the structure declaration and the
   implementation of the memory mapping procedures take place. */
struct mm_entry
{
    /* Mapping Identifier. */
    mapid_t mapid;

    /* Pointer for the mapped file. */
    struct file *file;

    /* List of corresponding pages about mapping. */
    struct list pte_list;

    /* Iterator for the mmap data list. */
    struct list_elem elem;
};

/* MMAP_ERROR: indicates that an error occurs in the 'mmap' syscall */
#define MMAP_ERROR -1

/* These two functions work as mmap() and munmap() respectively. */
mapid_t mm_mapping (int fd, void *addr);
void mm_freeing (mapid_t mapid);

/* Macros for raising a readability of codes.
   - VALIDATION: Checks whether there're a corresponding page offset,
     the NULLity, is_from_kernel_area?, is_already_paged?. */
#define VALIDATION(addr) (pg_ofs(addr)!=0||!addr||!is_user_vaddr(addr)||pt_find_entry(addr))
```

위의 'mm_entry' 구조체를 보자. 각 Process는 개별적으로 Memory Mapping을 수행하고, 매 Memory Mapping 시 해당 Mapping 정보를 기억하고 있어야 한다. 그래야만 mmap System Call로 생성한 Page들을 추적하고 변형/해제할 수 있을 것이기 때문이다.

따라서 'mapid_t'라는 Memory Mapping Entry의 Identifier Type을 마련하고, 그러한 Type의 Identifier Field를 Entry에 만들어 놓자. 이어, Mapped File에 대한 Pointer를 마련하고, Memory Mapping으로 만들어진 영역에 대한 Page들을 PTE의 형태로 'pte_list'에 추적/관리하자.

그리고, **mm_mapping** 함수는 mmap System Call을, **mm_freeing**은 munmap System Call을 각각 수행하는 Routine이 된다. 아래를 보자. 아래는 mm_mapping 함수이다. mmap System Call을 구현한 함수이다.

```
/* Memory mapping routine: map a file indicated by the passed descriptor
   onto the physical memory. In the project phase 4 'vm', the pintOS
   system uses this system call while testing, so we should implement this.

   Note that the mmap() system call in 'userprog/syscall.h' calls this
   procedure to perform 'real memory mapping'. */
mapid_t
mm_mapping (int fd, void *addr)
{
    struct mm_entry *mme; struct pt_entry *pte;
    size_t ofs = 0; off_t file_len;
    if (VALIDATION(addr)) return MMAP_ERROR;
```

```

if (!(mme = (struct mm_entry *)malloc(sizeof(struct mm_entry))))
    return MMAP_ERROR;

/* Initialize the newly created mmap entry. During
   initialization, file access should be protected. */
mme->mapid = (thread_current ()->mm_list_size)++;

list_init (&mme->pte_list);
list_push_back (&(thread_current ()->mm_list), &(mme->elem));

lock_acquire (&access_lock);
mme->file = file_reopen (mm_get_file (fd));
lock_release (&access_lock);

/* Now, read the mapped file, and load to some pages. */
file_len = file_length (mme->file);
while (file_len > 0)
{
    /* Calculate how to fill this page.
       We will read PAGE_READ_BYTES bytes from FILE
       and zero the final PAGE_ZERO_BYTES bytes. */
    size_t page_read_bytes = file_len < PGSIZE ? file_len : PGSIZE;
    size_t page_zero_bytes = PGSIZE - page_read_bytes;

    /* Create a page table entry for this, and push to the
       page table. Note that this is not a loading, this is
       just constructing the page table only. (Lazy Loading) */
    pte = pt_create_entry (addr, MAPPED, true, false,
                          mme->file, ofs, page_read_bytes, page_zero_bytes);
    if(!pte) return false;

    pt_insert_entry (&(thread_current ()->pt), pte);

    /* Push the page(entry) to the mapped-page list. */
    list_push_back (&(mme->pte_list), &(pte->mm_elem));

    /* Advance. Note that the offset is updated. */
    addr += PGSIZE;
    ofs += PGSIZE;
    file_len -= PGSIZE;
}
}

return mme->mapid;
}

```

VALIDATION 매크로는 이전 페이지의 mmap.h에서 확인할 수 있는데, 이는 Memory Mapping 수행 이전에 검사해야 할 조건들을 명시해둔 것이다. Passed Address가 User Virtual Area 내에 있는지, 해당 Address에 대해 이미 기-할당된 PTE가 있는지(즉, 이미 사용되고 있는 영역을 침범하지는 않는지 보는 것) 등, Manual에 명시된 Memory Mapping 조건들을 한꺼번에 검사한다.

이어, 검사를 통과하면, Memory Mapping을 수행한다. Memory Mapping Entry는 각 Process의 Memory Mapping 추적 List 'mm_list'의 끝에 삽입한다. (이를 위해 미리 threads/thread.h의 Thread Structure에 이를 추가해야 함)

이어, 해당 Memory Mapping에 대한 PTE List를 초기화하고, Mapping 대상 File을 받아서 file_reopen을 수행한다. 앞서 언급했듯, File System에 대한 접근은 Mutual Exclusion이 보장되어야 하기에, 'userprog/syscall.h'에서 사용하는 **File System Mutex Lock**인 'access_

lock'을 extern으로 끌어와 사용한다. 'mm_get_file'은 넘겨 받은 File Descriptor가 가리키는 File Pointer를 반환하는 Static 함수이다. 참고로 Reopen을 하는 이유는, File Descriptor가 있다는 것은 이미 해당 File이 Open되어 있음을 의미하기 때문이다.

File Open이 완료되면, File을 Page 단위로 쭉 읽어가며 PTE 생성 및 삽입을 수행한다. 일전의 Loading Mechanism과 전혀 다르지 않다. 이때, Memory Mapping Entry의 'pte_list'에도 해당 PTE를 삽입해야 함을 주목하자.

이렇게 하면 Memory Mapping이 완성된다. 자, 이제 Memory Mapping의 해제 루틴도 확인해보자. 아래를 보자.

```
/* Freeing mmap routine: free the mmap entry and info from the thread,
 by deleting it from the mmap list of that thread, and by deleting
 all the pages mapped to that mmapped region. In this second deletion,
 the 'real' is that freeing is applied only if the page is dirty.

 Note that the munmap() system call in 'userprog/syscall.h' calls this
 procedure to perform 'real memory map freeing'. */
void
mm_freeing (mapid_t mapid)
{
    struct list_elem *entry; struct pt_entry *pte;
    struct mm_entry *mme;

    /* Get the corresponding mmap entry of mapid. */
    if (!(mme = mm_get_entry (mapid))) return;

    /* From that entry, derive the PTE list of it, and iterate
     with updating if the specific page is dirty and loaded. */
    for (entry = list_begin (&(mme->pte_list));
         entry != list_end (&(mme->pte_list));)
    {
        pte = list_entry(entry, struct pt_entry, mm_elem);

        if (pte->is_loaded &&
            pagedir_is_dirty (thread_current ()->pagedir, pte->vaddr))
        {
            lock_acquire (&access_lock);

            /* Write(store) the data in mmapped page(memory)
             onto the corresponding file in the disk. */
            size_t read_byte = pte->read_bytes;
            size_t temp = (size_t)file_write_at (pte->file,
                                              pte->vaddr, pte->read_bytes, pte->offset);

            if (read_byte != temp)
                NOT_REACHED();

            lock_release (&access_lock);

            /* Deallocate only in this case, cause if the target mmapped page
             is not dirty, there's no need to free that physical frame since
             we can reuse that frame in sometimes. (by deleting that PTE) */
            free_page (pagedir_get_page (thread_current ()->pagedir, pte->vaddr));
        }

        /* Delete the corresponding PTE from the pt. */
        pte->is_loaded = false;
        entry = list_remove (entry);
        pt_delete_entry (&(thread_current ()->pt), pte);
    }

    list_remove (&(mme->elem));
    free (mme);
}
```

'mm_get_entry'는 Static 함수로서, 현재 Running Thread의 Memory Mapping List 'mm_list'에서 mapid Parameter에 대응하는 Entry를 찾아 반환하는 함수이다. 이를 통해, 현재 munmap System Call에서 해제하고자 하는 'mm_entry'를 뽑아낼 수 있다.

뽑아낸 Entry 안에는 앞에서 확인한 바와 같이, PTE List가 존재한다. 해당 List를 이제부터 쭉 훑으면서, **Memory에 Load 되어 있으면서, 동시에 Dirty한 Page들만 실제 File에 Update**하고, Frame Deallocation할 것이다. 왜냐? 그렇지 않은 Page는 'mm_entry'의 PTE List를 해제하기만 하면 언제든지 재사용할 수 있기 때문이다. (이는 곧, Memory Mapping으로 생성한 PTE들도 결국 Demand에 따라 Load되거나, 아니거나 나눈다는 것임)

이후, 해제 작업들을 마치면 munmap이 완수된다.

이제, **System Call**을 설치만 하면 되겠다. 아래와 같이 System Call Handler에 mmap System Call과 munmap System Call을 설치한다.

```
case SYS_MMAP:          /* Map a file into memory. */
    USER_ADDR_CHECK(2, esp);
    f->eax = mmap (ARG(1, int), ARG(2, void *));
    break;

case SYS_MUNMAP:        /* Remove a memory mapping. */
    USER_ADDR_CHECK(1, esp);
    munmap (ARG(1, mapid_t));
    break;
```

```
/* Mmap routine: simply calls the function 'mm_mapping' that performs a
   memory mapping. Yes, this function is declared in 'vm/mmap.h' file. */
mapid_t
mmap (int fd, void *addr)
{
    return mm_mapping (fd, addr);
}

/* Munmap routine: simply calls the function 'mm_freeing' just like the
   mmap() above, and of course, it's declared in 'vm/mmap.h' file. */
void
munmap (mapid_t mapid)
{
    mm_freeing (mapid);
}
```

그리고, 앞서 보인 것처럼 혹여나 Process가 자신의 Memory Mapping 영역들을 해제하지 않는 경우가 있을 수 있기 때문에 'userprog/process.c'의 'process_exit'에서 현재 종료되고 있는 Process의 Memory Mapping List를 쭉 훑으며 각 Entry에 munmap을 수행하도록 하자.

이렇게 하면 Memory Mapping까지 완성되고, 이 pintOS를 'make check'로 검사하면 본 Project 4의 Test Case 뿐만 아니라 Stanford 기준 'Virtual Memory' Phase Test Case들을 모두 통과할 수 있게 된다.

더 자세한 **Code-Level Detail**은 실제 제출 **Code**와 주석들을 통해 알 수 있다. 언급했듯이, 본인은 지난 Phase들과 마찬가지로 본 Phase에서도 주석에 많은 신경을 썼기 때문에 충분한 설명들이 달려 있다. 본 보고서에서 설명을 생략한 부분들에 대해 알아볼 수 있을 것이다. (Makefile.build 파일을 수정하는 것도 잊지 말자)

C. 시험 및 평가 내용

위와 같은 절차로 개발한 pintOS는 'make check' 시 본 Project 4의 Test Case들을 포함해 Stanford 기준 'Virtual Memory' Test Case 113개를 모두 통과한다.

```
pass tests/vm/pt-grow-stack
pass tests/vm/pt-grow-push
pass tests/vm/pt-grow-bad
pass tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
pass tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
pass tests/vm/page-merge-stk
pass tests/vm/page-merge-mm
pass tests/vm/page-shuffle
```

All 113 tests passed.

이렇게 해서 'Project 4 – Virtual Memory' 개발을 마친다. 다음이자 마지막 Phase도 열심히 수행하여 이번 학기 pintOS Project를 성공적으로 마치겠다.