**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재

이름 / 학번 : 백상욱 / 20190388

개발 기간 : 2023/11/20 ~ 2023/12/10

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

전반적인 vm을 추가로 구현해준다. 사실 이미 pintos안에는 가상메모리가 당연히 구현되어있다. 실제로 프로젝트 1만하더라도 스택의 가상 주소에 직접 인자를 넣어줬다. 그렇다면 이번에 추가로 구현해야하는 내용은 이에 부가적인 기능을 몇개 추가해주는 것이다. 그 중에서도 특히 Lazy load를 가능하게 해준다는 것이 핵심이다. 각 페이지들을 한번에 다 올리는게 아니라 필요한 부분씩 올려준다. 이를 위해서 supplementary page table을 구현해준다. 그리고 추가로 스왑 스페이스를 사용할 수 있게 해준다. 이 또한 lazy load의 맥락에서 동일하다. 마지막으로 stack growth가 가능하게 해준다. 이는 스택을 다 채웠을 때, 스택을 넓힐 수 있는 여유가 있다면 스택을 키워주는 기능이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Page Table & Page Fault Handler  
     기본적으로 가상 메모리는 페이지를 단위로 관리되고 이를 구현하기 위해서 페이지 테이블은 필수적이다. 기존에도 페이지 테이블은 구현되어있지만 이는 정말 단순하게 매핑을 하는 기능을한다. 그리고 한번에 물리주소에서 가상주소로 한번에 올려버린다. 이를 개선하는 것이 페이지 테이블의 구현이다. 이를 위해 supplementary page table을 구현해준다. 이를 통해 어떤 페이지가 로드 되어있는지, 그리고 어떤 페이지가 어떤 파일에 연결되어있는지, 내용이 기록되었는지 아닌지등을 확인할 수 있게 되어서 메모리를 잘 동기화하고 관리할 수 있다. 그리고 page fualt handler를 구현해야하는 이유는 현재 페이지 폴트 핸들러는 페이지 폴트가 나면 비정상 접근이라고 생각하여 강제로 종료하게 되어있는 반면 실제 페이지 폴트가 발생하면 해당 부분을 메모리에 올릴 수 있는지 확인하고 올릴 수 있다면 올려줄 수 있도록 작동해야하기 때문이다.
  2. Disk Swap  
     디스크 스왑은 디스크 영역을 일부 마치 메모리 영역처럼 사용하는 것이다. 실제 메모리에 있는 것은 아니지만 메모리 영역이 확장된 것처럼 사용하는 것이다. 하지만 스왑 스페이스를 사용하는 것 자체가 굉장히 오버헤드가 큰 작업이므로 어떤 페이지를 여기에 넣을지 결정하는 것도 중요한 알고리즘이다. 이를 위해서 LRU 기반의 알고리즘을 구현함으로서 디스크스왑을 효율적으로 한다. 이렇게 된다면 물리 메모리를 보다 확장된 것처럼 활용할 수 있게 되어 프로그램을 더 유연하게 짤 수 있다.
  3. Stack Growth  
     기존에는 스택의 사이즈가 고정되어있었다. 이는 4KB이다. 그래서 이 이상의 영역에 접근하려고 하면 비정상 종료를 시켰다. 이를 바꿔서 영역을 넘어가는 지점이 되더라도 스택을 확장할 수 있는 주소인지 확인하고 확장가능하다면 확장하도록 해준다. 이때 최대 8MB까지 확장 가능하도록 해준다. 이를 개선함으로서 실제 스택을 많이 요구하는 프로그램이더라도 정상적으로 작동할 수 있는 유연함을 제공한다.
  4. **개발 내용**
* 아래 항목의 내용만 서술
  1. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술  
     페이지 폴트가 발생하는 이유는 본질적으로 내가 참고하고자 하는 메모리에 해당하는 페이지가 페이지 테이블에 존재 하지 않는 경우이다. 페이지 테이블에 존재하지 않는데 페이지를 얻어내려고 하니 폴트가 발생하는 것이다. 이를 핸들링하는 전반적인 과정은 일단 참고하고자하는 vm\_entry를 이용하여 역참조하여 disk에서 직접 읽어온다. 그리고 이를 물리 메모리의 page frame에 넣어준다. 이를 수행할 때는 리스트 자료구조가 아니라 hash table자료 구조를 사용한다. 그리고 집어넣었다면 이를 기반으로 다시 page table에 해당하는 주소를 맵핑할 수 있도록 넣어준다.
  2. Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술  
     스왑 스페이스에서 메모리로 누군가를 가져와야할 때, 만약 페이지 테이블이 꽉 차있다면 누군가를 이 테이블에서 제거(swap out)한 후에 집어넣어(swap\_in)야한다. 이때 어떤 알고리즘을 사용할 것인지에 대해서 나는 LRU 방식을 채택한다. LRU 방식은 가장 최근에 사용되지 않은 페이지를 체크하여 제거하는 알고리즘이다. 하지만 정확하게 LRU를 구현하는 것은 불가능하다. 그래서 수업시간에 배운 LRU의 근사 방법론인 second chance 알고리즘을 사용한다. 이 알고리즘은 모든 프레임을 순회하면서 access bit을 확인한다. 만약 0이라면 해당 프레임을 제거하고 그 자리에 집어넣는다. 이때 모든 페이지가 처음 들어가게 될 때, access bit은 1로 만들어준다. 그리고 접근이 되었다면 access bit을 다시 1로 만들어준다. 반면 접근이 되었지만 찾는 값이 아니라면 access bit을 0으로 만든다. 이렇게 함으로서 LRU를 근사할 수 있다. 일종의 시계처럼 돌아가면서 체크하기 때문에 알아서 순서가 가장 늦게 접근된 애를 제거할 수 있게 된다.
  3. Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술  
     스택의 어떤 영역을 접근하는지 확인할 수 있는 값은 스택포인터이다. 그래서 만약 스택포인터가 접근하려는 영역이 스택 영역을 초가환 곳이라면 스택을 확장할 수 있는지 확인해보고 그 이후에 확장이 가능하다면 확장해줘야한다. 이때 확장이 가능한지 확인하기 위해서는 스택의 최대 사이즈를 확인해야한다. 최대 사이즈가 8MB이기 때문에 그 안으로 확장하는 경우라면 확장을 해준다. 그 안임을 확인하는 방법은 PHYSBASE로부터 2\*23만큼을 빼준 값 만큼이 최대 범위이기 때문에, 이 범위보다 크거나 같으면 확장 가능한 것이고 아니라면 확장이 불가능한것이라 판단한다

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성  
  11/20 ~ 11/26 : supplementary page table/ pagefualt 구현  
  11/27 ~ 12/3 : disk swap 구현  
  12 / 4 ~ 12 /10 : stack growth 구현 & 보고서 작성
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수

Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

우선 process.c 파일에서 handle\_mm\_fualt()를 구현해야한다.해당 함수는 물리 페이지리를 할당하는 함수로 페이지 폴트가 발생했을 때 page\_fualt함수 안에서 호출된다.  
그리고 execption.c에서는 page\_fualt()함수안에서 프로세스를 kill하던 부분을 지운 후 몇가지 조건을 확인하여 handle\_mm\_fault함수를 이용하여 페이지를 로드할 수 있도록 한다. 이를 구현하기 위해서는 기본적으로 vm\_entry라는 새로운 구조체를 추가해야한다. vm\_entry는 각 vme 엔트리에 부가적인 정보를 담고 있는 것으로 멤버로는 type, vaddr, writable, is\_loaded, offset, file, read\_bytes, zero\_bytes 등 해당 vm 개체와 관련된 정보를 모두 담고 있다.이 조건들을 이용하여서 적절하게 메모리를 로드하거나 아예 프로세스를 종료시키던가 결정해야한다.  
그리고 이러한 엔트리들을 적절하게 관리하기 위해서 page.c에서 vm 자체를 초기화 하고 없애주는 vm\_init, vm\_destroy 함수, 가상 주소에 대한 vm entry를 찾아주는 find\_vme 함수, vme를 집어넣고 vme를 지우는 insert\_vme, delete\_vme 함수, 그리고 주소에 해당하는 파일을 로드하는 load\_file함수를 구현한다.

Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술

디스크 스왑을 구현하려면 기본적으로 page 자료구조를 먼저 선언해야한다. page 자료구조에는 실제 물리주소, 물리페이지와 맵핑된 vm\_entry 포인터, 해당 물리페이지를 사용중인 스레드 포인터, lru 리스트로의 연결을 위해 elem을 선언해야한다. 그리고 page table에 올라간 애들은 lru list에 모두 집어넣어야한다. 그리고 이 리스트를 쭉 순회하면서 access bit를 확인하고 조건에 맞는다면 이를 제거하고 집어넣어야한다. 이를 위해서 우선 리스트를 초기화하는 lru\_list\_init 함수를 구현하고, 리스트에 추가하고 리스트에서 제거하는 add\_to\_lru\_list, del\_from\_lru\_list 함수를 구현한다. 추가로 alloc\_page라는 함수를 구현하는데 이는 페이지를 바로 할당받고 lru list에 집어넣어주는 기능을 합쳐놓은 함수이다. free\_page는 주소를 입력으로 받고 해당 주소에 맞는 page를 찾아서 \_\_free\_page를 호출한다.. \_\_free\_page함수또한 구현하는데 이는 lru list에서 제거하고 메모리 공간 자체를 해제하는 함수이다. 그리고 try\_to\_free\_page라는 함수도 추가로 구현해야하는데, 이는 palloc\_get\_page를 통해 메모리를 할당할 수 없을 때 여유 공간을 확보하기 위해서 호출하는 함수로 성공하면 해당 페이지를 lru\_list에 넣고 아니라면 종료하게 된다.  
그리고 lru를 근사한 second chance 알고리즘을 구현하기 위해서는 get\_next\_lru\_clock 함수를 구현해야한다.이는 lru를 한번씩 이동시켜주는 함수이다. 그리고 스왑 슬롯을 사용할 수 있는지를 체크해야하는데 이를 위해서는 swap bitmap이라는 것을 이용해야한다. 이를 이용해서 스왑 공간에서 꺼내고 다시 넣기 위해서 swap\_in, swap\_out함수를 구현한다. swap\_in은 swap\_slot에 저장된 데이터를 논리주소로 복사해주는 역할을 하고 swap\_out은 다시 스왑 파티션에 기록하는 역할을 한다.

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

우선 stack을 확장할 수 있는지 조건을 체크하기 위해서는 이게 유효한지 아닌지 체크하는 함수를 먼저 호출해야한다. 이 함수 안에서 확인해야할 것은 우선 접근 주소가 스택포인터로부터 32보다 작은 값보다는 커야한다. 그리고 user\_vaddr에 해당하는지도 체크해야한다. 마지막으로 최대 크기를 넘어서지 않는지 확인해줘야한다. 이 조건에 맞아떨어진다면 스택을 확장하면 된다. 스택을 확장하는 함수로는 expand\_stack으로 구현해줘야하는데 이는 이후에 자세하게 설명한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성

도표, 텍스트, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명

page fualt & page\_fault\_handler

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

disk swap

도표, 스크린샷, 텍스트, 평면도이(가) 표시된 사진

자동 생성된 설명

stack growth

* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
* 위의 내용들을 모두 구현하기 위해서 우선 vm\_entry 자료구조를 구현한다. vm\_entry는 기본적으로 type, vaddr, writable, is\_loaded, \*file, offset, read\_bytes,zero\_bytes, swap\_slot, hash \_elem을 필드로 가지게된다. 이중 특히 hash\_elem을 가지는 이유는 이들을 관리할 때 리스트가 아닌 해시 테이블을 이용하기 때문이다. 그래서 vm\_entry를 vm에 넣고 제거하는 것을 수행하는 insert\_vme, delete\_vme를 구현하기 위해서는 hash\_insert, hash\_delete함수를 사용해야한다. 이 함수는 간단하게 해쉬테이블에 넣어주는것, 그리고 제거해주는 것을 의미한다. 이외에도 vm\_init을 초기화하는 vm\_init 함수, vm을 없애는 vm\_destroy 등을 추가로 구현해준다. 이 안에서는 마찬가지로 hash\_init,hash\_destroy 함수를 호출해주는 역할을 한다. 그리고 가상 메모리가 들어왔을 때 이 메모리에 해당하는 vme를 찾는 과정이 필요하기 때문에 이를 구현하기 위해서 find\_vme라는 함수를 구현했다. 이는 입력으로 들어온 주소를 pg\_round\_down으로 페이지 번호를 얻어내고 이걸 hash\_find 함수의 인자로 주어서 hash elem을 얻어낸다. 만약에 존재하지 않는다면 null을 반환한다. load\_file 함수를 구현하는데, 구현하는 방식이 여러가지가 있겠지만 나는 주어진 오프셋으로부터 파일을 읽어오는 File\_read\_at함수를 사용했다. 그리고 이 반환 값으로 나온 읽은 만큼의 bytes를 vme의 ready\_bytes을 비교한다. 만약 동일하다면 입력으로 받은 kaddr로부터 read\_bytes만큼을 0으로 초기화시켜주기 위해서 memset을 호출한다. 만약 동일하지 않다면 file\_Read\_at의 반환값이 null이라면 false를 반환한다.  
  그런데 위의 내용을 구현했을 때 vm\_init을 호출할 위치가 어디일지 고민했고 이는 thread\_init부분에서 호출했지만 이는 올바른 경로가 아니었고 start\_process에서 호출해줬다.. 하지만 이를 구현해서 끝이 나는 것이 아니라 이들을 작동하게 하기 위해서는 process.c에 존재하는 load\_segment, setup\_stack 함수를 수정해줘야한다. 우선 load\_segment에는 프로세스의 가상 메모리와 관련된 자료구조를 초기화해주는 기능을 추가해준다. 그래서 파일을 처음에 읽은 이후에 vm\_entry를 malloc을 통해 하나 새롭게 할당해준다. 그리고 read\_bytes와 zero\_bytes를 계산해준다. 이때 ready\_bytes가 PGSIZE보다 크다면 이때는 그냥 PGSIZE로 준다. 그리고 PGSIZE로부터 이 값을 뺀 것을 zero\_bytes로 지정해준다. 그리고 이걸 vm\_entry의 field값으로 준다. 그리고 이때 중요한 것이 vm\_type을 VM\_BIN으로 주는 것이다. VM\_BIN은 바이너리 파일로부터 데이터를 가져왔다는 것을 의미한다. 이를 포함해서 다른 정보들을 vme에 초기화해준 뒤에 insert\_vme로 현재 스레드의 vm에 vme를 넣어준다. 이렇게 함으로서 vme를 넣어준다. 그리고 다음에는 setup\_stack을 수정해줘야한다. setup\_stack에서는 4KB만큼의 vm\_entry를 만들어야한다. stack도 결국 페이지 단위로 관리해야하기 때문에 vm\_entry를 넣어줘야한다. 그리고 이를 초기화해준 뒤에 다시 vm 테이블에 집어넣어준다. load\_segment와 비슷한다. 이를 위해서 일단 virtual\_address 를 계산해주고 alloc\_page함수를 이용해서 페이지를 할당한다. 그리고 만약 페이지가 할당이 되었다면 install\_page를 호출한다. 그리고 만약 page가 설치가 된다면 malloc으로 vme를 선언한다. 그리고 load\_segment에서 한 것 처럼 vme 멤버들을 초기화해준다. 이때 타입을 VM\_ANON으로 선언해주는 것이 load\_segment와 다르다. VM\_ANON 스왑 가능한 영역임을 표시해주는 부분이다. 나머지 부분은 동일하고 이를 초기화 한 후에는 insert\_vme로 Vm을 넣어준다. 그리고 이후에 메뉴얼에서는 syscall에서 사용하는 buffer의 메모리를 유요한 가상 주소인지 체크해줄 필요가 있다고 하는데 나는 해당 부분을 구현하지 않고도 다른 테스트 케이스에서 통과해서 별다른 처리를 해주지 않았다. 이후에는 페이지 폴트 함수를 수정하고 handle\_mm\_fault를 구현한다. 지금 페이지 폴트에서는 바로 프로세스를 죽이지만 프로세스를 죽이는 부분을 제거한다. 그리고 fualt\_addr를 기반으로 vme를 우선 찾는다. 만약 vme가 없다면 stack 확장을 고려해야하는데 이는 이후에 다룬다. 만약 vme가 존재한다면 handle\_mm\_fault를 호출해준다. handle\_mm\_fault안에서는 우선 페이지를 먼저 할당해준다. 할당이 안된다면 false를 반환한다. 그리고 vme의 type을 확인해준다. 만약 type이 VM\_BIN이라면 load\_file을 호출해준다. Load\_file 함수는 디스크의 페이지를 물리 메모리로 가져오는 함수이다. 나는 file\_read\_at함수를 사용하여 구현했다. 이를 통해 file을 로드하는데 만약 실패했다면 기존의 page 자체를 Free해줘야한다. 그리고 만약 타입이 VM\_SWAP이라면 swap space를 구현해야하는데, 이는 이후에 다시 설명한다. 그것도 아니라면 잘못된 경우이므로 false를 반환한다. 그리고 페이지 설치까지 무사히 되었다면 vme의 Is\_loaded를 true로 만들어 준다. 하지만 여기까지 구현했을 때 계속 kernel panic과 함께 프로세스가 종료했다. 이를 찾아본 결과 thread\_create에서 thread\_yield를 마지막에 호출해주는 부분에서 #ifdef USERPROG를 호출하지 않았기 때문에 이런 문제가 발생하는 것이었다..그래서 앞뒤로 조건을 걸어줌으로서 이를 통과한다.  
  이제는 스와핑을 구현해야한다. 일단 page 자료 구조를 추가해준다. page 자료구조는 물리페이지를 관리하기 위한 자료구조로 필드로는 물리주소인 kaddr, 연결된 vm\_entry, 해당 물리 페이지를 사용 중인 thread, 그리고 eviction policy를 구현하기 위한 lru list에 연결하기 위해서 lru를 선언한다. 여기서 리스트 자체, 그리고 리스트에 사용할 lock, 그리고 victim을 가리킬 clock을 모두 초기화한다. 그리고 우선 이 lru 리스트를 관리하기 위해서 lru\_list\_init 함수를 선언한다. 이는 init.c 함수에서 main 함수 안에서 수행해준다. 그리고 lru\_list에 넣고 빼주는 add\_page\_to\_lru\_list, del\_page\_from\_lru\_list를 호출한다. 이때 중요한 것은 넣고 뺄 때 락을 잘 걸어줘야지 테스트 케이스에서 처리가 가능할 수 있어야한다. 그리고 여기서 alloc\_page를 구현한다. 일단 기본적으로 palloc\_get\_page함수를 사용하여 page를 새로 할당받는다. 이는 기본적인 핀토스 구현 함수로 flag를 받아서 그 flag에 맞게 page를 할당해준다. 그리고 새로운 페이지를 malloc을 통해서 할당받는다. 이는 곧 palloc\_Get\_page로 받은 page 자료구조의 kaddr 필드에 할당될 값이다. 그리고 이후에 이 페이지를 lru\_list에 넣어준다. 그리고 free\_page, \_\_free\_page를 구현해준다.free\_page안에서는 lru\_list를 순회하며 입력으로 받은 kaddr와 동일한 lru\_page를 찾는다. 그리고 찾았다면 멈추고 그것에 대해 \_\_Free\_page를 호출해준다. 이 \_\_Free\_page는 page를 lru\_list에서 제거한 후에 palloc\_free\_page로 완전하게 페이지를 제거한 후에 마지막으로 free를 호출해 메모리를 완전히 해제한다. 다음으로 get\_next\_lru\_clock을 호출하는데 이는 clock 알고리즘에 따라 lru 리스트를 순회하기 우해서 사용하는 함수로 lru list의 다음 노드의 위치를 반환해준다. 만약 이게 마지막 노드라면 Null을 반환한다. 만약 현재 clock이 list\_end와 같거나 null이라면 이는 곧 시작 위치를 반환해야하는 경우이다. 그리고 lru\_clock의 다음이 list\_end와 같지 않은 경우라면, next를 반환하지만 만약 end와 같다면 시작 위치를 반환해준다. 그리고 try\_to\_free\_pages를 구현하는데, 이는 paclloc\_get\_page를 통해 메모리를 호출할 수 없을 때 여유 공간 확보를 위해 사용하는 함수다. 일단 반복문을 계속 돌아준다. 그리고 계속 get\_next\_lru\_clock으로 lru elem을 뽑아준다. 이때 뽑아지지않는다면 return을 해준다. 그리고 잘 얻어졌다면 list\_entry로 page를 가져온다. 그리고 이 page에서 thread를 얻어서 해당 threaddml page\_dir가 접근되었는지 확인한다. 접근되었다면 set\_accessed로 접근 되었음을 확인한다. 그리고 page가 dirty인지도 확인하여 만약 dirty라면 type을 swap으로 만들어주고 swap\_slot을 swap\_out한다. 그리고 이러한 스왑을 구현해주기 위해서는 swap\_in, swap\_out, swap\_init을 구현해야한다. 기본적으로 swap은 swap\_bitmap과 swap\_partition을 이용해야하고 bit 연산으로 구현한다. 이를 위해서 swap\_init은 우선 swap\_block을 block\_get\_role이라는 함수를 통해서 받아온다. 이는 devices/block.h에 구현된 함수로 swap block을 가져와주는 기능응ㄹ한다. 그리고 이 block을 이용해서 bitmap\_create 함수로 swap\_map을 만들어준다. 그리고 이들을 모두 0으로 초기화해준다. 그리고 swap\_in 함수에서는 일단 used\_index에 해당하는 bitmap이 0인지 확인한다. 그리고 0이 아니라면 page에 해당하는 부분을 쭉 훑으면서 block을 읽어서 물리 메모리로 다시 넘긴다. 여기서 block\_read함수를 사용한다. 이는 블럭에서 해당하는 값을 가져오는 것이다. 그리고 읽어낸 부분은 0에서 1로 값을 바꿔준다. 그리고 swap\_out함수이다. 이부분은 일단 swap\_Free한 index를 찾아내고 그부분에 값을 채워준 뒤 다시 0으로 만들어줘야한다. 이를 위해서 free\_index를 찾는에 이때 bitmap\_scan\_and\_flip함수를 사용한다. 근데 이 index가 BITMAP\_ERROR와 인지 확인한다 맞다면 에러를 반환한다. 그게 아니라면 반환하면서 블럭을 순회함녀서 이번에는 block\_write함수로 값을 써준다. 이렇게 하면 스왑 스페이스를 구현할 수 있다.  
  마지막으로 stack growth를 구현하기 위해서는 우선 입력으로 받은 주소 자체가 스택을 확장할 수 있는지 아닌지를 체크해야한다. 이를 위해서 일단 user\_vaddr에 해당하는지 체크한다. 그리고 stack이 자라날 수 있는 최대치인 8MB보다 큰지를 확인한다. 이보다 크게 확장하려고 한다면 종료해야하는데 이를 확인하기 위해서 PHYSBASE - 2\*23를 보다 작은 지점을 참조하려고 하면 비정상 종료한다. 위의 경우에서 확장가능한 범위라면 확장해주는데 이를 위해서 일단 vme를 malloc으로 새로 할당받는다. 그리고 vme에 addr에 해당하는 page number를 채워주고, Type은 vmswap으로 잡아준다. 그리고 현재 스레드의 vm에 vme를넣어준다. 마지막으로 page를 설치해주면 끝이다.  
  이제 swap과 stack growth를 모두 구현했다. 전반적인 플로우를 다시 살펴보기 위해서는 handle\_mm\_fault를 봐야한다 .앞서 handle\_mm\_fault에서 vme\_type이 VM\_BIN인 경우에 대해서만 알아봤는데 만약 VM\_ANON이라면 Swap을 할 수 있는 영역이므로 swap\_in을 호출한다. 그 경우가 아니라면 스택을 확장할 수 있는지를 마지막으로 확인해야하고 위에서 만들어놓은 확장 가능여부를 확인하고 가능하다면 stack을 확장해준다.
  1. **시험 및 평가 내용**
* (채점 대상 테스트 케이스에 해당하는) make check 수행 결과를 캡처하여 첨부  
  텍스트, 폰트, 흑백, 스크린샷이(가) 표시된 사진

  자동 생성된 설명