**Pintos Project 4: Virtual Memory**

담당 교수 :

이름 / 학번 : 이상윤 / 20171664

개발 기간 : 2022-11-20 ~ 2022-12-06

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이전 프로젝트에서는 잘못된 주소를 참조하여 page fault가 일어나면 그대로 종료되었다. 이번 프로젝트에서는 잘못된 주소 참조가 해결할 수 있는 유효한 참조인 경우에는 해결할 수 있도록 page fault handling을 한다. 이는 physical memory size에 제한되지 않도록 virtual memory을 구현하는 것과 같다. memory swapping, stack growth을 이용하여 해결할 수 있다.

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

기존 page table(pagedir) 이외에도 추가적인 정보를 저장하는 supplemental page table, frame table이 필요하다.

텍스트, 안테나이(가) 표시된 사진

자동 생성된 설명

이렇듯 pte에 dirty bit, present bit 도 존재하는데 supplemental page table, frame table이 필요한 제일 중요한 이유는, page fault가 일어난 위치에 대체 어떤 내용이 있어야 하는지 알아야 한다(이 내용은 디스크의 어느 위치에 저장되어 있다던지, 이 내용은 어느 파일을 어디어디부터 다시 읽으면 되는지 등). logical address로 찾아간 pte에선 dirty bit, present bit 이런건 알지만 정확히 어떤 내용이 있었는가의 정보는 얻을 수 없는 상태이다.

Page fault handler는 페이지의 메모리 참조가 유효한 경우에는 프레임을 할당하고, 적절한 데이터를 프레임으로 가져온다. 또한 스택의 확장이 필요한 경우에는 확장 시켜준다. 두 경우 모두 아닌 경우에는 프로세스를 죽인다. 이를 통해 가상 메모리를 사용할 수 있게 될 것이다.

* 1. Disk Swap

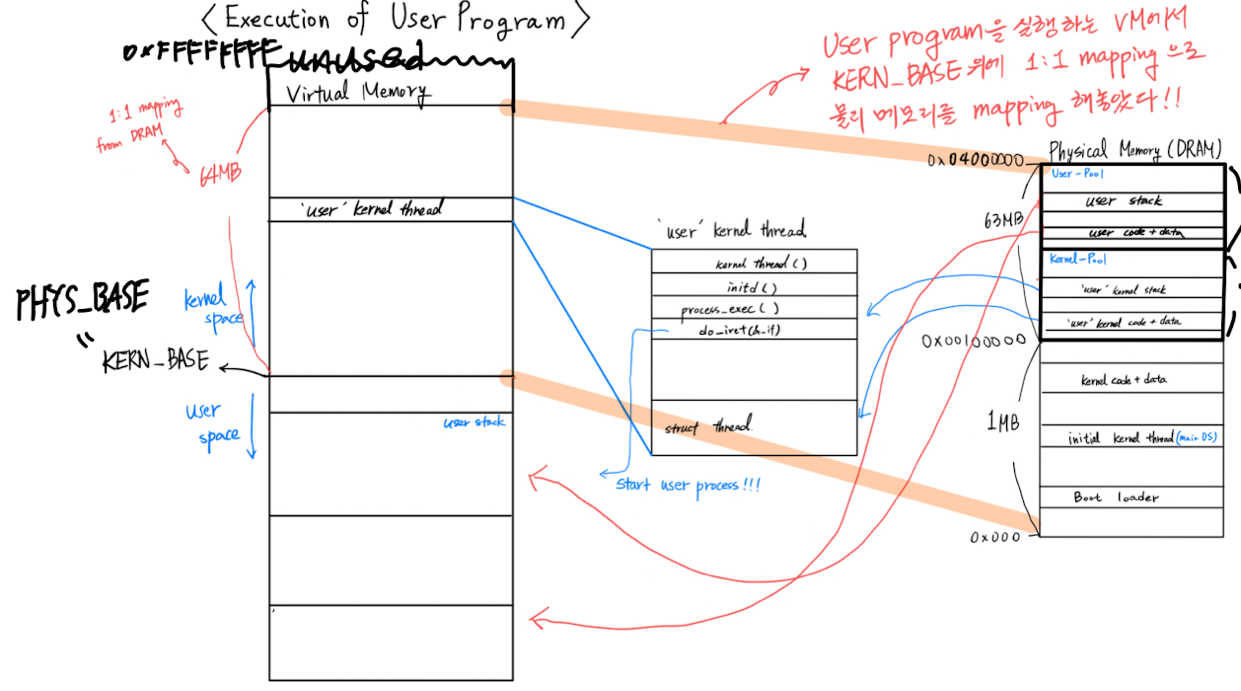
실제 physical memory 상에 공간이 부족하여 빈 페이지를 할당하지 못하는 경우에, frame replace algorithm을 이용한 eviction이 필요하다. 이 evicted frame의 내용을 저장할 공간이 필요하다. 이를 위해 swap disk가 필요하다. Swap out을 통해 swap partition에 내용을 쓰고, swap in을 통해 다시 데이터를 user pool로 들고 올 수 있다.

* 1. Stack Growth

기존에 process.c : setup\_stack() 을 보면 프로세스마다 PHYS\_BASE – PGSIZE에 한 페이지 크기의 스택을 마련하는 것을 볼 수 있다. 이렇게 정해진 한 페이지 크기의 스택은 더이상의 확장이 불가능하였다. Page fault가 일어날 때의 user end stack pointer와 fault address를 비교해서 최대 스택 범위 내에서 스택의 확장이 가능하다고 판단되면 확장한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술
  1. Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

우선 최초로 부팅 시 threads/loader.S 가 real mode에서 실행된다. 그 후 threads/start.S를 실행한다. 이때 readl mode에서 protected mode로 변환이 일어난다. 이후에 주소는 가상주소로만(pagedir을 통해) 사용이 가능하게 된다.

실제 physical memory는 kernel space와 일대일 대응된다. user process는 user space에서 돌아가고, 즉 이는 physical memory의 user pool에서 돌아가는 것이다. (모든 user virtual page는 kernel virtual page와 aliased임을 볼 수 있다)

CPU가 pagedir을 통해 physical memory에 접근하려하는데, 프레임이 실제로 physical memory에 존재하지 않는다면(=present bit가 꺼져있음) page fault interrupt가 발생하고 exception.c : page fault()가 실행된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이때 원래 기존에 %esp는 user pool 영역을 가르키다가, kernel pool의 kernel stack을 가르키게 된다. 그리고 intr\_frame 구조체 형태로 기존 user pool 영역에서의 registers 정보를 %esp부터 저장하게된다.

page fault handling 과정은 다음과 같다. 우선 이 page fault가 kernel에 의해 접근 되었는지, user에 의해 접근 되었는지 확인하고 kernel에 의해 접근되었다면 process를 종료한다. 이 이유는 코드 설명에서 추후 서술한다.

그 후 vaild reference인지 확인한다. 만약 vaild reference라면 적절한 데이터를 frame table 과 supplemental page table을 이용하여 메모리에 위치시키고 process를 restart한다.

valid reference를 확인했는데 아니라면, fault\_addr과 f->esp를 비교하여 stack growable region인지 확인한다. 만약 stack growable region이라면 스택을 확장하고 process를 restart한다.

모두 아니라면 그냥 죽인다.

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

선형 합동법을 이용한 난수생성을 통해, random replacement algorithm을 사용하였다. frame table과 supplemental page table 모두 hash 자료구조를 이용해 구현하였는데, 생성한 난수값을 기반으로 frame table hash를 순회하면서 evict할 프레임을 골랐다.

이때 evict할 프레임이 진짜로 evict되어도 되는지 여부를 따져보아야 한다. user pointer로 사용되는 프레임인 경우에는 evict되면 안된다. 만약 evict되어서는 안되는 프레임이라면 같은 방법으로 다시 뽑는다.

모든 user virtual page는 kernel virtual page와 aliased임을 보았는데, 이 때문에 access bit, dirty bit가 서로 동기화되지 않는 문제가 있다. Dirty bit는 해결하였지만, access bit는 완벽히 해결하지 못하였다. 만약 완벽히 해결한다는 가정하에 second chance, clock 알고리즘을 사용하는 것이 성능에 더 좋을 것이다.

Second chance 알고리즘은 frame을 circular list로 따로 관리한다. 처음 로딩된다면 1로 되어있다. eviction할때 1인경우는 0으로, 0인 경우는 eviction 한다. 그리고 접근된 프레임은 1로 다시 바꾼다.

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

스택의 크기 제한은 8Mb로 제한한다.

일반적으로 fault\_addr과 user end stack pointer가 일치하고, 스택 크기 제한을 만족하면 스택을 확장하면 될 것이다.

또한, 핀토스 매뉴얼 4.3.3에 따르면, 80x86 PUSH 명령어는 stack pointer를 조정하기 전에 접근 허가부터 한다. 그래서 fault\_addr는 user end stack pointer보다 4bytes 밑에 있을 것이다. 비슷하게 PUSHA 명령어는 32bytes 밑에 있을 것이다.

즉, fault\_addr == f->esp, fault\_addr == f->esp – 4, fault\_addr == f->esp – 32를 만족하고, 스택 크기 제한을 만족할때 스택을 확장하면 될 것이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2022-11-20 ~ 2022-11-25 : supplemental page table, frame table 자료구조 구현, 함수 구현, priotiry scheduling 오류 수정.

2022-11-25 ~ 2022-11-27 : syscall에서 user provided pointer 유효한지 확인

2022-11-28 ~ 2022-11-28 : swap 함수 구현

2022-11-29 ~ 2022-11-29 : swap\_in 적용, stack grow 구현

2022-11-30 ~ 2022-11-30 : swap\_out 적용, random replacement algorithm 적용

2022-12-01 ~ 2022-12-04 : 프로세스 종료 시 메모리 할당 해제, file system semaphore 오류 수정, memory sharing 자료구조 추가

2022-12-04 ~ 2022-12-06 : mmap, munmap system call 추가(page-merge-mm test)

2022-12-06 ~ 2022-12-07 : 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

frame table은 user page를 저장하고 있는 frame에 대한 정보를 저장한다. 즉, 모든 프레임들을 저장하지 않고, per system이다. frame table entry를 빠르게 탐색할 수 있도록 해시테이블을 사용한다(key, value) = (frame, frame\_table\_entry). sharing을 고려하여 중복을 허용하는 해쉬 테이블을 frame-table-hash.c에 구현한다. 그리고 frame\_table\_entry 구조체와, entry에 대해 조작하는 함수들을 frame.c에 구현한다.

supplemental page table은 추가적인 정보로 page table을 보완한다. 즉, page table을 나타내고, per process이다. supplemental page table entry를 빠르게 탐색할 수 있도록 해시테이블을 사용한다(key, value) = (user\_page, supplemental\_page\_table\_entry). supplemental page table entry 구조체와 spte에 대해 조작하는 함수들을 page.c에 구현한다.

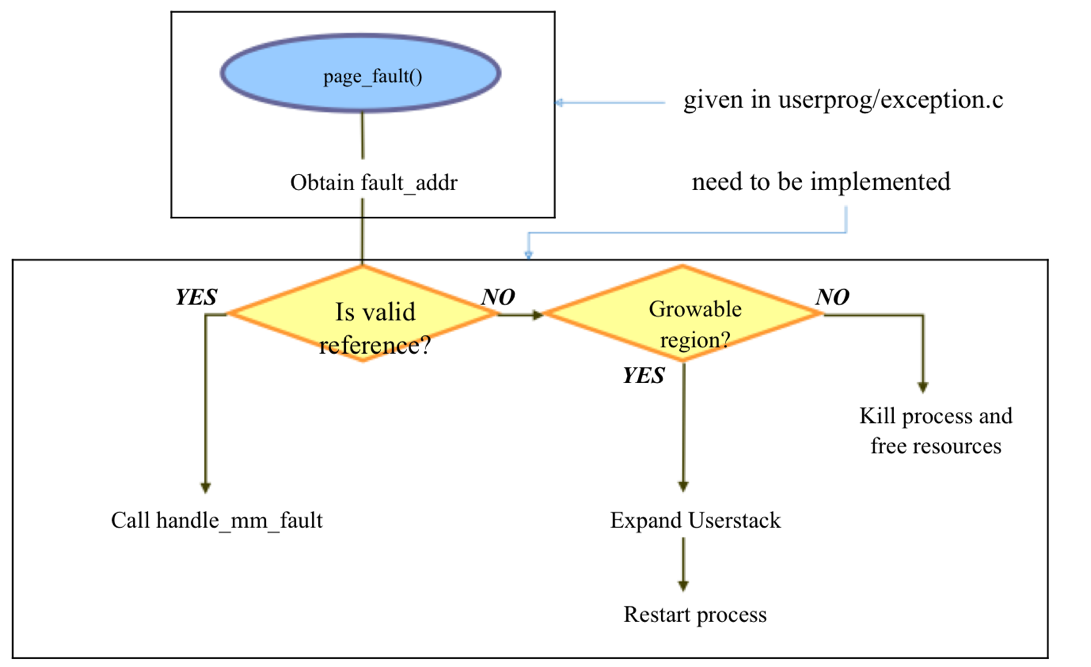
swap table은 단순한 배열이다. 한 sector의 크기는 512bytes, 페이지의 크기는 4096bytes 이므로, 8개의 섹터가 한 프레임을 나타내게 된다. 각 인덱스는 8개-aligned sector 묶음을 가르킨다. 그 값은 현재 이 프레임을 참조하고 있는 프로세스의 수를 나타낸다. swap table과 swap disk, swap disk 조작 함수가 swap.c에 구현된다.

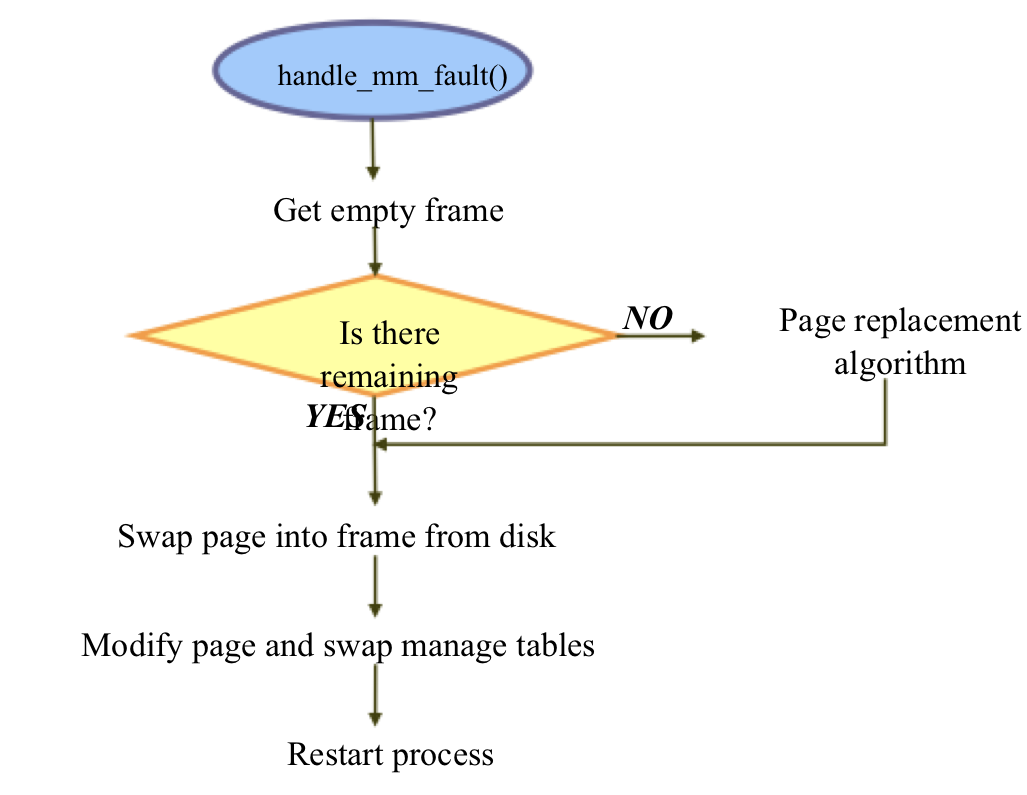
syscall.c에서 mmap, munmap, user pointer validate function들을 구현한다.

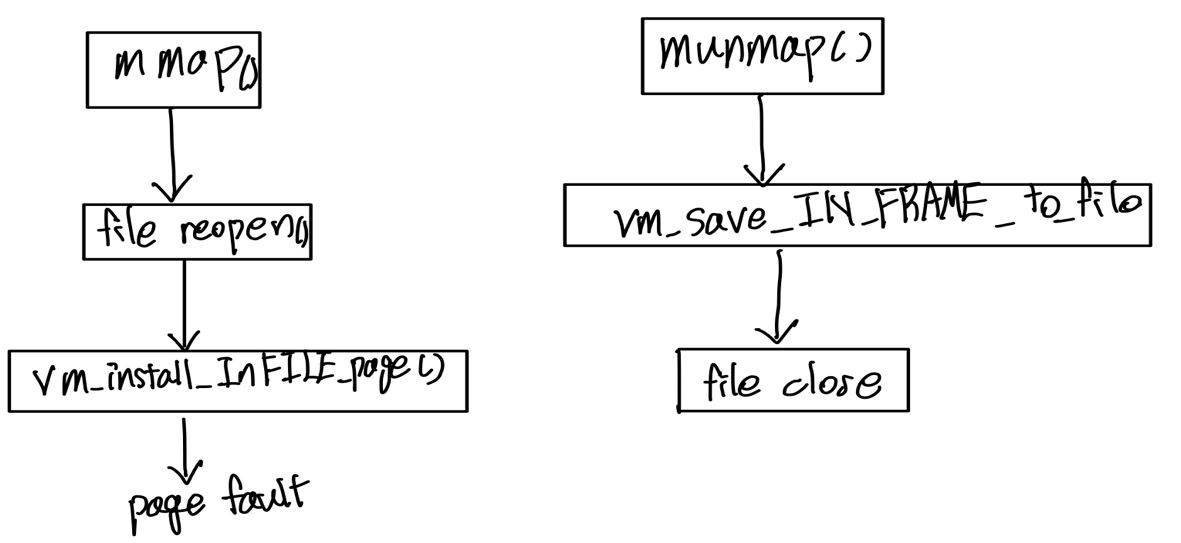
exception.c에서 stack grow를 구현하고 기존 page fault()를 수정한다.

1. **연구 결과**
   1. **Flow Chart**

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







* 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

우선 stack growth를 구현하기 위해서는 f->esp의 값을 정확히 알아야 한다. 위에서 잠시 서술하였듯이, %esp는 user pool을 가르키다가, interrupt가 발생하면 %esp는 kernel pool의 kernel stack을 가르킨다. 그리고 intr\_frame 형태로 기존 user pool에서의 register 정보들을 %esp부터 저장한다. 즉, CPU는 user->kernel모드로 전환될 때만 기존 user pool에서의 register 정보들을 (intr\_frame 형태로) kernel stack에 저장한다. 따라서 kernel에서 page fault가 발생했다면 intr\_frame에 esp 정보는 유효한 값이 아니다.

system call 에서 kernel은 user program이 넘겨준 포인터로 메모리에 엑세스 해야한다. 그러나 이때 kernel은 매우 신중해야한다. user 가 null pointer을 넘길수도 있고, unmapped된 virtual memory에 접근하는 pointer를 넘길 수도 있고, kernel space에 접근하는 pointer를 넘길 수도 있다. kernel이 user pointer로 메모리에 엑세스 하는데, 유효하지 않다면 page fault가 발생하고, 이는 곧 kernel에서 page fault가 발생한 것이 된다. 따라서 page fault handler에서 f->esp는 유효한 값을 가지지 못하기 때문에, user pointer의 유효성을 검증한 후 user pointer로 메모리에 엑세스하는 방법을 택한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

user pointer부터 bytes만큼 유효한지 확인하는 함수이다. vm\_spt\_lookup()을 통해 매핑 되어있는지를 확인한다.

또한, user\_pointer\_inclusive부터 bytes만큼 physical memory에 고정시킬 필요가 있다. BLOCK\_FILESYS, BLOCK\_SWAP 와 같은 block device를 컨트롤하는 block device driver가 있는데, BLOCK\_FILESYS에서 block\_read(), block\_write()를 호출하는 도중에 page\_fault()가 발생하여 BLOCK\_SWAP에서 block\_read(), block\_write()를 호출하는 상황이 발생하면 안된다. 서로 lock resource를 공유하기 때문이다. 따라서 user pointer에 대해, eviction이 절대 되면 안된다고 명시하고, swap partition에 있는 데이터는 미리 메모리로 swap in 할 필요가 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

user pointer부터 bytes만큼 미리 메모리로 swap in 하는 함수이다. vm\_spt\_lookup()을 통해 현재 thread의 supplemental page table으로부터 정보를 찾고, 미리 swap in을 한 다음, vm\_frame\_set\_for\_user\_pointer()을 통해 frame table에다 이 프레임은 eviction이 발생하면 안된다고 명시한다.

수정한 system call의 하나의 예시를 들면 아래와 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

exec system call이다. f->esp +4에 저장된 문자열 포인터를 참조하여, 해당 문자열을 참조해야하는 상황이다. 우선 f->esp +4, 즉 문자열 포인터가 저장된 공간이 유효한지부터 확인한다. 유효하다면 문자열 포인터가 저장된 시작주소를 얻는다. 그리고 문자열이 모두 유효한 주소공간에 있는지 또한 확인해야한다. 유효하다면 eviction을 금지시키고, physical memory 상으로 데리고 온다. 그리고나서 exec system call을 실행하고, 실행이 끝나면 금지시킨 eviction을 원래대로 되돌린다.

이렇게 user pointer validation을 해결한다면, kernel이 호출한 page fault()는 커널 내의 버그인 경우만 해당될 것이다. 따라서 page fault handling에서 kernel에 의한 접근이 일어난 경우는 프로세스를 죽이고 디버깅을 위한 출력만 뽑고 프로세스를 종료한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이 코드는 page fault() 내부이다. 이제 참조가 유효한지를 확인한다. 유효한 참조라면, page replace algorithm 적용 후, 메모리로 데이터를 load한다. 유효한 참조이기 위한 조건은, supplemental page table 상에 정보가 존재해야하고(!= NULL), kernel 영역에 대한 접근이 아니어야 하고(!is\_kernel\_vaddr), read only 영역에 write하는 접근이 아니어야한다(not\_present || !write).

텍스트이(가) 표시된 사진

자동 생성된 설명

이 코드는 page.c에 있다. swap partition에 현재 데이터가 저장되어 있는 것이 확인되면, 이 함수가 호출된다. vm\_swap\_in()을 통해 새로 할당받은 kernel\_virtual\_page에다가 데이터를 쓰고, reinstall\_page()로 pagedir상에서 user page와 새로 할당받은 kernel\_virtual\_page를 연관시키는 것을 볼 수 있다. 여기 vm\_frame\_allocate()에서 eviction 등이 일어나므로 좀더 살펴본다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

함수의 기능은 주석과 같다. palloc\_get\_page() 내장 라이브러리이고, vm\_ft\_hash\_insert()는 중복이 허용되게 수정한 hash\_insert() 내장 라이브러리와 비슷하다. 그리고 frame table에는 user pool의 데이터만 넣는 것을 볼 수 있고, frame\_table은 per system이므로 lock을 잡아주는 것을 볼 수있다. eviction algorithm을 수행하는 vm\_evict\_a\_frame\_to\_swap\_device() 부분을 좀더 살펴본다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

random replacement algorithm을 적용한 부분을 볼 수 있다. 여기서 선형 합동법을 이용한 난수생성을 통해, evict할 frame을 골랐다. 그리고 evict될 수 없는 경우는 다시 난수생성을 통해 고르는 부분을 볼 수 있다. dirty bit를 동기화 하는 부분이나, 나머지 내용은 주석에서 설명하는 바와 같다.

page\_fault()에서 vm\_load\_IN\_FILE\_to\_user\_pool() 을 적용하는 부분은 mmap과 관련이 있다. 작동 방식은 위에서 설명한 방식과 거의 일치한다.

마지막으로, stack growth는 user context에서만 page fault가 일어나므로 f->esp는 유효하므로 쉽게 구현 가능하다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

마찬가지로 vm\_frame\_allocate()를 통해 user pool에 빈 프레임을 마련하고 install\_page()를 호출하는 것을 볼 수 있다.

개발 중 발생한 문제 중 하나로, 프로세스가 종료할 때 page fault handling을 위해 구현한 자료구조와 실제 pagedir 상의 동기화가 필요하다.

텍스트이(가) 표시된 사진

자동 생성된 설명

process\_exit()가 호출될 때 이 함수가 호출되는데, 이 함수를 수정하여 frame table을 동기화 시키도록 한다. 또한 supplemental page table에 대해선 내장 라이브러리인 hash\_destroy() 함수를 호출하도록 한다.

* 1. **시험 및 평가 내용**
* (채점 대상 테스트 케이스에 해당하는) make check 수행 결과를 캡처하여 첨부

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명