**Project 3 Design Report**

성해빈, 송창근

**1. 시스템 분석**

1. 메모리 관련 용어

**페이지(Page)** – 페이지는 **가상** **메모리**의 연속된 영역으로 크기는 4,096바이트(= 2^12), 즉 4kB이다. 페이지는 page align된 가상 주소에서 시작되며, 32비트 가상 주소는 20비트의 페이지 번호와 12비트의 페이지 오프셋 (페이지에서의 위치, 4,096)으로 나눌 수 있다. 각 프로세스는 0~0xc0000000(PHYS\_BASE)구역에 자기들만의 유저 페이지를 가지고 있고, 커널 페이지는 모든 프로세스가 공유한다. 커널은 유저 페이지와 커널 페이지에 모두 액세스 할 수 있지만 유저 프로세스는 자신이 가진 유저 페이지에만 액세스 할 수 있다.

**프레임(Frame)** – 프레임은 **물리적 메모리**의 연속 영역이다. 프레임은 페이지와 그 크기 및 크기로 나눌 수 있는 주소에서 시작된다는 규칙이 같다. 따라서 32비트 물리 주소는 20비트 프레임 번호와 12비트 프레임 오프셋으로 나눌 수 있다. 실제 주소의 메모리에 직접 액세스하기 위해 Pintos는 **커널** 가상 메모리(0xc0000000(PHYS\_BASE)~0xffffffff)를 실제 메모리에 직접 매핑한다. 커널 가상메모리 PHYS\_BASE는 실제 물리적 메모리 0에 매핑되고, PHYS\_BASE+12 은 실제 물리적 메모리 12에 매핑된다. 따라서 프레임은 **커널** 가상 메모리를 통해 1대1로 액세스 할 수 있다. 이는 pagedir\_create로 페이지 디렉토리를 만들자마자 디폴트로 주어진 매핑이다.또 당연히 **유저** 가상 메모리도 어떻게던 프레임에 매핑이 될텐데, 이는 매뉴얼 85페이지에 있는 pagedir\_set\_page로 가능하다. 하지만 이 함수를 잘 보면 **커널** 가상 메모리가 가리키고 있는 프레임을 **유저** 가상 메모리도 동시에 가리키게 하는데(alias), 이는 사실상 우리가 실제로 이용할 수 있는 램이 1GB라는 소리다.

**페이지 테이블(Page Table)** – 페이지 테이블은 CPU가 가상 주소를 물리적 주소로, 즉 페이지를 프레임으로 변환하는 데 사용하는 데이터 구조다. 페이지 테이블 형식은 80x86 아키텍처에 의해 결정되며, Pintos에는 'pagedir.c'에 페이지 테이블을 관리하는 코드가 구현되어 있다. 페이지 테이블에서는 페이지 번호만 프레임 번호로 변환하고 오프셋은 그대로 둔 채 결합하여 실제 주소를 얻어낸다.

**스왑 슬롯(Swap Slots) -** 스왑 슬롯은 스왑 파티션의 디스크 공간에 존재하는 연속적인 페이지 크기의 영역들이다. 스왑 슬롯 역시 시작 주소가 페이지 크기로 나누어 떨어져야 한다. page eviction이 이루어질 때 쫓겨난 page는 swap slot에 할당된다.

1. 관계된 함수들

**void \* palloc\_get\_page (enum palloc\_flags flags)**

**void \* palloc\_get\_multiple (enum palloc\_flags flags, size\_t page\_cnt)**

커널 풀에서 한 페이지, 또는 page\_cnt개의 연속된 페이지를 할당받아 리턴한다. 페이지를 할당할 수 없는 경우 null 포인터를 반환한다. flags에 따라 아래와 같은 결과를 얻는다. flags는 조합 가능하다.

PAL\_ASSERT - 페이지를 할당할 수 없으면 커널을 패닉시킨다. 커널을 초기화할 때만 사용해야 한다.

PAL\_ZERO - 리턴하기 전 할당된 페이지의 모든 바이트를 0으로 초기화한다.

PAL\_USER - 사용자 풀에서 페이지를 가져온다.

**void palloc\_free\_page (void \* page)**

**void palloc\_free\_multiple (void \* pages, size\_t page\_cnt)**

page 또는 pages에서 시작하여 한 페이지 또는 page\_cnt개의 연속된 페이지를 각각 해제한다. 해당되는 페이지는 palloc\_get\_page() 또는 palloc\_get\_multiple()을 사용하여 할당되었던 페이지여야 한다.

**unsigned pg\_ofs (const void \*va)**

**unsigned pg\_no (const void \*va)**

va 주소에서 각각 페이지 오프셋과 페이지 번호를 추출해 리턴한다.

**void \* pg\_round\_down (const void \*va)**

**void \* pg\_round\_up (const void \*va)**

각각 va를 가상 페이지의 시작점(0)/끝점으로 설정한 va를 반환한다.

**bool is\_user\_vaddr (const void \*va)**

**bool is\_kernel\_vaddr (const void \*va)**

각각 va가 유저/커널 가상 주소에 들어가 있는지 체크한다

**uint32\_t \* pagedir\_create (void)**

새 페이지 테이블을 만들고 반환한다. 새로운 페이지 테이블은 Pintos의 일반적인 커널 가상 페이지 매핑을 포함하지만 사용자 가상 매핑은 포함하지 않는다. 테이블을 만들 메모리를 확보 할 수없는 경우 널 포인터를 리턴한다.

**void pagedir\_destroy (uint32\_t \* pd)**

페이지 테이블 자체와 매핑하는 프레임을 포함하여 pd가 보유한 모든 리소스를 해제한다.

**void pagedir\_activate (uint32\_t \* pd)**

pd를 활성화한다. CPU가 메모리를 변환하는 데는 활성화된 페이지 테이블을 사용한다.

**bool pagedir\_set\_page (uint32\_t \* pd, void \* upage, void \* kpage, bool writable)**

사용자 페이지 upage와 커널 가상 주소 kpage로 식별되는 프레임의 매핑을 pd에 추가한다. writable이 true이면 페이지는 읽기/쓰기로 매핑되고, 그렇지 않으면 읽기 전용으로 매핑된다. upage는 pd에 아직 매핑되어 있지 않아야 한다. 커널 페이지 kpage는 palloc\_get\_page (PAL\_USER)로 사용자 풀에서 얻은 커널 가상 주소 여야 한다. 성공하면 true를 반환하고, 페이지 테이블에 필요한 추가 메모리를 확보 할 수없는 경우 실패하여 false를 반환한다.

**void \* pagedir\_get\_page (uint32\_t \* pd, const void \* uaddr)**

pd의 uaddr에 매핑 된 프레임을 찾는다. uaddr이 매핑되어 있으면 해당 프레임의 커널 가상 주소를 반환하고 그렇지 않으면 null 포인터를 반환한다.

**void pagedir\_clear\_page (uint32\_t \* pd, void \* page)**

pd에 "not present"라고 표시한다. 이후 페이지에 액세스하면 페이지 폴트가 발생한다. 페이지에 대한 페이지 테이블의 다른 비트는 보존되고 액세스 및 더티 비트는 체크할 수 있도록 허용된다. 페이지가 매핑되지 않은 경우 아무 일도 하지 않는다.

또 hash table 관련과 bitmap 관련 함수도 사용할 계획이다.

**2. 문제 해결 (구현)**

1. **Frame Table** (5 points)

프레임을 효과적으로 관리하기 위한 프레임 테이블을 구현한다. 프레임 테이블은 user page에 매핑되어있는 frame 에 대해서 추가 정보를 담고 있는 테이블이다. 프레임 테이블을 만드는 가장 큰 이유는 메모리가 부족해서 쓸 수 있는 프레임이 없을 때, eviction을 판단하기 위해서이다. eviction 당한 page는 swap partition의 swap slot으로 배당된다. 만약 eviction이 필요한 상황인데 swap slot이 남아있지 않다면, 커널 패닉을 띄운다.

**현재 구현** – 페이지와 프레임의 크기는 4096 바이트다. 페이지 테이블의 관리를 위한 함수는userprog/pagedir.c 에, 페이지 테이블의 할당/해제는 threads/palloc.c에 구현되어 있다.

**필요한 구현** –각 프레임의 정보(프레임 번호, 쓰레드 ID, 할당 가능 여부)를 저장하는 프레임 테이블의 구현. 다음 기능들을 포함해야 한다.

* 프레임의 할당/해제.
* 만약 비어 있는 프레임이 없다면, 적절한 page을 쫓아내는 eviction 수행

eviction policy : load 이후 modify된 page(dirty bit으로 판별) 중에서 LRU

* 유저 프로세스가 사용하는 프레임 검색

유저 페이지가 사용하는 프레임은 ‘유저 풀’에서 할당되어야 한다.

frame table은 list형태로 구현할 것이며, frame table entry는 tid, user virtual address, 할당 가능 여부, 프레임 번호 등의 정보를 가지고 있어야한다.

1. **Supplemental Page Table** (5 points)

더 많은 기능을 가진 S-페이지 테이블을 구현한다.

**필요한 구현** –S-페이지 테이블과 그 관리 기능을 구현해야 한다. S-페이지 테이블은 페이지 번호, 프레임 할당 가능 여부, 프레임 번호, 스왑 가능 여부를 포함한다. 다음 기능을 포함해야 한다.

* Lazy Loading, 스왑 테이블, 파일 메모리 매핑의 정상 작동.
* Lazy Loading으로 각 페이지에 프레임을 할당한다.
* 수정된 데이터를 파일이나 스왑 디스크의 프레임에 저장한다.
* 페이지가 삭제될 때, 연관된 엔트리들도 프레임 테이블이나 스왑 테이블 등에서 찾아 같이 삭제해야 한다.

threads/exception.c에 있는 페이지 폴트 핸들러는 새로 만들어진 S-페이지 테이블에 적용되어야 한다.

supplementary page table은 hash table로 구현할 것이며, 각 thread마다 하나씩 있을 것이다. table의 element인 supplementary page entry에는 page에 해당하는 user vaddr, read byte와 zero byte 개수, load됐는지의 여부, swap slot index 등의 정보가 들어가야 한다.

1. **Stack Growth** (5 points)

스택의 크기를 늘릴 수 있도록 한다.

**현재 구현** – 스택의 크기가 1페이지로 고정되어 있다.

**필요한 구현** –페이지 폴트 핸들러가 스택이 증가해야 하는지 여부를 체크한다. 스택이 더 필요한 경우, 필요한 페이지 수를 계산해서 인터럽트 프레임에서 스택 포인터를 가져온다. 그 후 페이지를 로드하고 프레임 테이블과 S-페이지 테이블의 정보를 넣는다. 스택이나 힙 충돌이 일어날 수 있으므로 사전에 스택의 최대 크기는 정해둔다. 스택이 더 필요한 경우를 휴리스틱하게 판단하라고 돼있는데, 보통 user program의 stack access는 자잘한 양만큼 차례대로 access하므로 주변 스택의 참조 비율을 확인해 볼 예정이다.

1. **File Memory Mapping** (5 points)

메모리에 파일이 매핑될 수 있도록 구현한다. 이는 진짜 메모리에 파일을 박아넣는 방법으로, read 와 write로 파일에 접근하는 방법 이외의 방법이다.

**필요한 구현** –파일과 페이지의 연결을 관리하는 파일 매핑 테이블을 만든다. mapid\_t mmap(int fd, void \*addr) 과 void munmap(mapid\_t mapping) 시스템 호출을 구현한다. 각각 mapping과 unmapping을 수행한다.

mmap도 lazy loading을 적용해 eviction을 당하면 그때가서야 제대로 load한다.

mmap은 연속적인 페이지를 할당해 file을 메모리에 보관하므로, 마지막 페이지에는 남는 공간이 생기기 마련이다. eviction 당했을 때는 빈 공간을 0으로 채우고, 페이지가 디스크에 swap되었을때는 빈 공간 0을 다 버린다.

file 이 zero byte일때, addr가 page-align이 안 돼있을때, addr부터 할당될 page가 다른 page와 충돌을 일으킬 때, addr가 0일때, fd 가 0또는 1일때 mmap함수는 fail한다.

memory mapped file은 hash table 로 구현될 것이며, entry는 file 포인터, mapid\_t, start address, page size등의 요소가 있어야한다.

1. **Lazy Loading** (5 points)

메모리에 페이지를 로드하기 위해 Lazy Loading을 구현한다.

**현재 구현** – 프로세스 시작에 필요한 실행할 코드가 메모리에 바로 로드된다. 페이지 폴트가 발생하면 잘못된 액세스 오류로 간주하고 프로그램 실행을 항상 중지한다.

**필요한 구현** –프로세스가 시작에서 메모리 할당을 위한 로딩 과정 중에 먼저 스택 설정 부분만이 로드된다. 다른 부분은 메모리에 로드되지 않고 그냥 페이지만 할당된다. 이제 페이지 폴트가 일어나면, 메모리에 로드시키는 것이다. 전부 다 load시키면 page fault handler가 다시 user program의 instruction을 수행한다.

페이지 폴트 핸들러는 I/O(Lazy Loading)가 필요한 상황과 I/O(메모리 접근 오류)가 필요없는 상황이 동시에 발생하면 전자를 기다리지 말고 후자를 먼저 처리해야 한다.

페이지 폴트 핸들러는 threads/exception.c에 있는 page\_fault() 함수를 수정하면 된다.

lazy loading을 구현하려면 userprog/process.c의 load\_segment() 안의 루프를 수정해 page\_read\_bytes 와 page\_zero\_bytes를 조율해야 한다.

1. **Swap Table** (5 points)

eviction 당한 페이지는 swap table에 의해 관리된다.

**필요한 구현**

* 스왑 테이블 – 스왑 디스크를 만들고 스왑 디스크를 관리하기 위한 스왑 테이블을
* 스왑 아웃 – 선택된 페이지를 스왑 디스크에 복사해 넣어 여유 공간을 확보한다. (선택하는 기준은 dirty bit + LRU)
* 스왑 인 – 페이지 폴트가 발생했을 때, 스왑 디스크에 있는 페이지를 새 프레임에 할당한다.

읽기/쓰기 함수는 devices/block.h에 구현되어 있는 섹터 기반으로 구현해야 한다. lib/kernel/bitmap.c에 구현되어 있는 비트맵 구조를 사용하는 것이 좋다. 스왑 디스크를 생성해야 하는데, "pintos-mkdisk swap.disk n"이 "vm / build"에서 실행되면 "swap.disk"라는 이름의 파티션이 n MB 생성된다.

1. **On Process Termination** (5 points)

프로세스가 종료될 때, 사용하던 리소스를 전부 할당 해제한다.

**필요한 구현** –프로세스가 종료될 때, S-페이지 테이블, 프레임 테이블, 스왑 테이블의 관련 내용을 삭제하고 파일 역시 전부 닫는다. (Dirty page needs to be written back)

**3. Conclusion**

이번 프로젝트는 굉장히 많은 부분에 손을 대야 하는, 핀토스에서 가장 어려운 프로젝트로 유명하다. 그래도 오피셜 매뉴얼이나 조교님의 ppt에서 어디서부터 시작해야하는지 비교적 명확하게 명시되어 있어, 그대로 따라가면 큰 문제 없을 것이라 믿는다. 아직 구체화하지 않은 부분이 많은데, 너무 양이 많은 관계로 차근차근 하나씩 해나갈 계획이다. 이제 virtual memory를 완벽하게 구현하면 disk를 메모리 백업 스토리지로 사용해, 사용 가능한 메모리보다 많은 프로세스를 구동하는 데에 성공할 것이다.