Project3 Design Report

20210084 김지민, 20210216 양준영

Analysis of the current implementation

1. Frame table
   1. Basics (개념, 현재 구현)
      1. Frame table: 유저 페이지 하나를 포함하는 각 프레임에 대해 하나의 엔트리를 가진다. 유저 페이지에 의해 사용되는 프레임은 palloc\_get\_page(PAL\_USER)을 호출함으로써 user pool로부터 얻어져야 한다. 프레임이 비어있다면 간단하지만, 비어있는 프레임이 없다면 특정 프레임을 골라 evict 시켜야 한다.
      2. 현재 프레임과 페이지의 크기는 4096 bytes이며, 페이지 테이블의 관리와 페이지 테이블 할당/해제는 각각 “userprog/pagedir.c”와 “threads/palloc.c”에 구현되어있다.
      3. userprog/pagedir.c
         1. pagedir\_create (void): 커널 가상 주소들에 매핑되는 새로운 페이지 디렉토리를 생성한다. (페이지 디렉토리는 페이지 테이블의 첫 주소들을 모아둔 것. 즉, 각 엔트리는 페이지 테이블의 첫 주소를 가짐)
         2. pagedir\_destroy (uint32\_t \*pd): 해당 pd를 가진 페이지 디렉토리를 없앤다. 이 디렉토리가 가리키는 모든 페이지 테이블의 엔트리에 있는 페이지들을 모두 free 시킨다.
         3. lookup\_page (uint32\_t \*pd, const void \*vaddr, bool create): pd 페이지 디렉토리에서 vaddr 가상주소를 가진 페이지 테이블 엔트리의 주소를 반환한다.
         4. pagedir\_set\_page (uint32\_t \*pd, void \*upage, void \*kpage, bool writable): pd 디렉토리에서 가상주소 upage를, 가상주소 kpage가 나타내는 물리적 프레임에 매핑한다.
         5. pagedir\_get\_page (uint32\_t \*pd, const void \*uaddr): 유저 가상주소 uaddr에 대응되는 물리 주소가 pd 안에 있는지 조사한다. 있다면 대응되는 주소를 반환.
         6. pagedir\_clear\_page (uint32\_t \*pd, void \*upage): upage에 해당하는 페이지를 not present로 마크. (나중에 여기에 접근하면 page fault 발생)
         7. pagedir\_is\_dirty (uint32\_t \*pd, const void \*vpage): 해당 pd 속 vpage 페이지에 대한 PTE가 dirty면 true 반환.
         8. pagedir\_set\_dirty (uint32\_t \*pd, const void \*vpage, bool dirty): 해당 vpage에 대한 PTE를 dirty bit를 DIRTY로 set.
         9. pagedir\_is\_accessed (uint32\_t \*pd, const void \*vpage): vpage에 대한 PTE를 최근에 접근한 적 있다면 true.
         10. pagedir\_set\_accessed (uint32\_t \*pd, const void \*vpage, bool accessed): 해당 페이지에 대한 PTE의 accessed bit을 ACCESSED로 set.
         11. pagedir\_activate (uint32\_t \*pd): 해당 pd 페이지 디렉토리를 CPU의 페이지 디렉토리 베이스 레지스터 (CR3)로 로드(디렉토리의 물리 주소를 저장).
         12. active\_pd (void): 현재 CR3에 로드된 페이지 디렉토리 반환.
         13. invalidate\_pagedir (uint32\_t \*pd): pd 디렉토리를 활성화시킴으로써 TLB를 invalidate시키는 함수다.
      4. threads/palloc.c
         1. palloc\_init (size\_t user\_page\_limit): 메모리를 초기화하고 user pool과 kernel pool로 나눠 초기화한다. 이때 user pool에 들어가는 페이지 수는 user\_page\_limit을 넘지 않도록 한다.
         2. palloc\_get\_multiple (enum palloc\_flags flags, size\_t page\_cnt): page\_cnt만큼의 연속된 free page들을 얻고 그 그룹의 시작 주소를 반환한다. 만약 flag가 PAL\_USER라면 user pool으로부터 페이지가 얻어진다. PAL\_ZERO가 세워진다면 0으로 채워진 page들이 얻어진다.
         3. palloc\_get\_page (enum palloc\_flags flags): 하나의 free page를 얻는다.
         4. palloc\_free\_multiple (void \*pages, size\_t page\_cnt): pages 주소에서부터 page\_cnt만큼의 페이지들을 free시킨다.
         5. palloc\_free\_page (void \*page): 하나만 free시킨다.
         6. init\_pool (struct pool \*p, void \*base, size\_t page\_cnt, const char \*name): base부터 시작하여 page\_cnt만큼의 페이지가 할당된, name이라는 이름의 pool을 초기화한다.
         7. page\_from\_pool (const struct pool \*pool, void \*page): 해당 페이지가 해당 pool로부터 할당되었는지 검사한다. 맞다면 true 반환.
   2. Limitations and Necessity (현재 구현의 문제점, 새 구현의 필요성)
      1. 현재 핀토스에는 프레임 테이블이 구현되어 있지 않다.
      2. free한 frame이 없을 때 victim을 골라 evict시켜야 하는데, 이때 각 프레임에 대한 정보를 관리하는 이 테이블이 있다면 훨씬 효율적으로 eviction 알고리즘을 구현할 수 있다.
   3. Blueprint (새 구현을 어떻게 만들 것인지 / 필요한 구조체 및 간단한 코드)
      1. Frame table list에 들어갈 요소 “frame”구조체를 선언한다.
         1. void \* kvaddr: 그 프레임에 매핑되는 커널 가상주소
         2. void \* upage: 해당 프레임에 매핑되는 페이지 가상주소.
         3. list\_elem frame\_list: 프레임 테이블 리스트에 들어갈 요소
         4. tid: 이 프레임을 할당한 thread를 저장
         5. not\_evict: 이게 true면 이 프레임은 evict 대상에서 제외. 등등이 필요할 것이다.
      2. vm/frame.c를 새로 생성한다.
      3. 여기에 frame\_table이라는 이름의 list와, frame 구조체를 선언한다.
      4. 주요 함수
         1. Allocate\_frame: frame을 할당하는 함수. 페이지는 User pool로부터 할당되어야 한다고 했으므로 “palloc\_get\_page(PAL\_USER)”를 호출하여 frame 구조체의 kvaddr에 반환값을 저장한다. 만약 이 함수의 반환값이 NULL이라면 남는 frame이 없다는 것이므로 “evict frame” 함수를 호출하여 evict할 frame을 반환 받고, 해당 frame의 upage를 pagedir\_clear\_page 함수에 넣어 호출함으로써 그 페이지를 not present로 마크한다. 이후 다시 “palloc\_get\_page(PAL\_USER)”을 호출하여 페이지를 할당 받는다. 이렇게 frame 구조체의 kvaddr에 페이지를 할당하고 난 뒤, frame\_table list에 이 프레임을 추가하고 해당 프레임을 반환한다. (아마 이 페이지가 축출될 때 필요하면 디스크에 write하는 함수까지 추가되어야 할 것이다.)
         2. Evict\_frame: 이 함수는 frame\_table list를 돌면서 victim을 찾아 그 프레임을 반환한다. 다양한 알고리즘 중 일단 clock 알고리즘을 기준으로 한다. 우선 최근에 리스트에서 축출되었던 요소의 바로 다음 요소부터 시작하여 해당 프레임의 페이지가 access된 적 있는지 검사한다. 이때 pagedir\_is\_accessed 함수를 이용하면 된다. 만약 true를 반환한다면 pagedir\_set\_accessed 함수를 호출하여 해당 페이지의 access bit를 0으로 설정하고 다음 요소로 넘어간다. 이런 식으로 반복하다가 만약 pagedir\_is\_accessed 함수가 0을 반환했다면 그 프레임을 반환한다.
         3. Deallocate\_frame: 인수로 프레임에 매핑되는 kvaddr를 받으면, 거기에 해당하는 프레임을 찾아 list에서 삭제한다. 만약 필요하다면 조건에 따라 palloc\_free\_page를 호출하여 페이지 해제까지 진행한다. 이후 해당 프레임을 free시킨다.
         4. Search\_frame: 유저 프로세스 혹은 쓰레드가 의해 사용되는 프레임을 찾는다. 이는 프레임에 tid가 저장되어 있으므로, frame\_table list를 순회하면서 해당 쓰레드의 tid와 일치하는 frame을 반환하면 될 것 같다.
2. Lazy loading
   1. Basics
      1. Lazy loading: Demanding Page을 하는 것으로, 프로세스가 시작할 때 모든 페이지들이 로드되는 게 아니라 실제 필요할 때만 로드되는 방식이다.
      2. Page fault: 프로세스가 가상주소에 접근할 때, 거기에 해당하는 페이지가 실제 물리적 메모리에 없는 경우를 말한다.
      3. static void page\_fault (struct intr\_frame \*f): “threads/exception.c”에 있다. 현재는 다른 exception들과 마찬가지로 다루게끔 되어있다. not\_present (page가 존재하는지), user (유저에 의한 접근인지), fault\_addr (fault address) 변수를 이용하여 만약 “not\_present || !user || is\_kernel\_vaddr(fault\_addr)”를 만족한다면 exit(-1)를 호출하도록 되어있고, 아니라면 나중에 kill(f)하고 끝난다.
   2. Limitations and Necessity
      1. Load\_segment 함수에서 현재는 세그먼트를 바로 메모리에 로드하는 형식이다. 이를 실제로 필요할 때만 로드하여 메모리를 효율적으로 사용하도록 한다.
      2. 현재 page\_fault 함수 구현에 의하면 페이지 폴트가 발생하면 exit(-1) 혹은 kill(f)를 호출함으로써 항상 stop하도록 되어있다.
      3. 따라서 만약 할당된 페이지에 대해 페이지 폴트가 발생한다면(lazy loading에 의한 page fault라면) 이 페이지를 메모리에 로드한 후, procedure가 끝난 시점부터 다시 process operation을 재개할 수 있도록 구현한다. 즉, invalid access error가 아닌 경우에는 페이지를 로드하고 다시 재개할 수 있도록 수정해야 한다.
   3. Blueprint
      1. exception.c의 page\_fault 함수에서 wrong memory access인 경우를 먼저 처리해주어야 한다.
         1. is\_kernel\_vaddr(fault\_addr) 값을 먼저 조사하여 true라면 잘못된 메모리 접근이므로 kill 한다.
      2. 만약 not present인 경우라면 이후 나올 Load\_page 함수를 호출하여 fault\_addr에 해당하는 페이지를 frame에 올린다. 이상이 없다면 kill이 아니라 return 하여 다시 재개될 수 있도록 한다.
      3. process.c의 load\_segment를 수정함으로써 lazy load를 구현한다.
         1. While 문 내에서 페이지를 할당받고, 파일을 읽고 하는 과정 대신, Thread\_current -> supplemental page table, upage, file 등을 인수로 하여 (뒤에서 설명될) supp\_insert\_page 함수를 호출한다. 즉, 해당 페이지의 정보만 supp 페이지 테이블에 저장하고 실제로 frame을 할당하지는 않는다. 이렇게 하면 나중에 이 페이지에 접근할 때 page fault가 일어날 것이고, 핸들러에서는 supp 페이지 테이블의 정보를 기준으로 load\_page (뒤에 나옴)을 호출함으로써 필요할 때만 페이지를 로드할 수 있다.
3. Supplemental Page Table
   1. Basics
      1. Supplemental page table: page table을 보완하며, 각각의 페이지에 대한 추가 데이터를 가진다. Page fault시 커널은 이 테이블을 lookup해야 하며, 프로세스를 종료시킬 때도 이 테이블을 lookup함으로써 어떤 리소스를 free시킬지 결정한다. Hash table로 구현한다.
      2. Hash 구조체 (lib/kernel/hash.h, hash.c): key에 해시 함수를 적용하여 고유 인덱스를 생성하고 그 위치(buckets이라 불리는 list)에 value를 저장하는 구조. 주요 함수 몇 개만 알아본다.
         1. Struct hash: 해시 테이블의 요소 개수, buckets (값이 저장되는 리스트), 해시 함수 등으로 구성된다.
         2. hash\_init (struct hash \*h, hash\_hash\_func \*hash, hash\_less\_func \*less, void \*aux): 해시 테이블 h가 주어진 “hash”를 해시 함수로 갖도록하고, buckets 크기도 할당하는 등 해시테이블을 초기화한다.
         3. hash\_find (struct hash \*h, struct hash\_elem \*e): 해시 테이블 h에서 e와 같은 element를 반환한다.
   2. Limitations and Necessity
      1. 페이지 테이블은 가상주소를 물리 주소로 변환해주는 역할을 한다. 그러나 페이지 폴트가 발생했을 때 커널은 그 페이지를 찾아 데이터가 어디 있는지 알아내야 하고 프로세스가 종료될 때 어떤 리소스를 해제시켜야 하는지 결정해야 하는데, 페이지 테이블만으로는 이 기능들을 구현하기 어렵다. 따라서 페이지에 대한 추가 정보를 담은 이 테이블이 필요하다.
      2. 각 페이지에 대해 데이터가 어디 존재하는지, 이 페이지가 대응되는 커널 가상주소, 이 페이지가 현재 dirty인지 등등을 저장한다.
   3. Blueprint
      1. 우선 supplemental\_page\_table 구조체를 만든다.
         1. struct hash supp: 해시 테이블을 이용하므로 해시 구조체를 멤버로 갖는다.
      2. 그리고 thread.h에 이 구조체를 멤버로 추가한다.
      3. 그리고 supplemental\_page\_table의 요소가 될 엔트리 구조체인 supp\_entry 구조체도 만든다.
         1. Void \*upage: 이 페이지의 가상주소
         2. Void \* kvaddr: 페이지가 할당된 프레임의 가상주소. 즉, 만약 upage가 프레임에 할당되어 있지 않다면, 이 값은 NULL!
         3. hash\_elem ha: supplemental\_page\_table이 해시 테이블로 구현된다. hash.c의 많은 함수들이 hash\_elem을 사용하므로 함수 사용을 위해 이 멤버를 추가한다.
         4. Int status: 페이지의 데이터가 현재 어디 있는지 상태를 나타낸다. 예를 들어, 이 값이 1이면 현재 메모리 프레임에, 2면 swap slot에 있다는 의미가 될 것이다.
         5. Bool dirty: 수정되었는지 여부.
         6. swap\_index: 뒤에 나올 swap table에서의 인덱스.
         7. Struct file\* file: 만약 페이지가 파일로부터 로드되었다면 해당 파일을 저장.
         8. Off\_t offset: 파일의 오프셋. 등등이 필요할 것이다.
      4. 또한 hash\_init 함수에서 쓰일 hash 함수와 less 함수를 구현해야 한다.
         1. Supp\_hash: 우리는 주어진 페이지 주소를 통해 해시 테이블에서 엔트리를 찾아야 하므로 hash 함수가 페이지 주소 즉, upage를 기준으로 해시할 수 있도록 구현해야 한다. 암호화하는 hash\_int 함수에 upage 값을 넣어주고 호출한다.
         2. Less: 이 함수 역시, 두 element의 upage값을 비교하여 그 결과를 반환한다.
      5. vm/page.c를 새로 생성한다.
      6. 우선 새로운 페이지를 이 테이블에 추가하는 함수가 필요할 것이다.
         1. supp\_insert\_page: 우선 엔트리인 supp\_entry를 malloc으로 엔트리 크기만큼 할당받은 뒤, 해당 페이지의 주소 upage를 이 엔트리의 멤버인 upage에 저장한다. 이후 supp 테이블의 해시 구조체에 이 엔트리를 삽입한다(hash\_insert 호출). 이때, 이 페이지의 데이터가 현재 어디있냐에 따라 함수의 매개변수가 달라질 것이다. 만약 파일에 존재한다면 해당 file과 오프셋 등을 추가로 받아야 할 것이다. 따라서 이 함수를 위치 각각에 대해 따로 선언해주어야 할 것이다.
      7. 페이지 폴트 시, 커널은 이 테이블을 참조한다고 했으므로 관련 기능을 구현한다.
         1. lookup: 페이지 폴트가 발생한 페이지 주소를 받으면, supp 테이블을 lookup함으로써 해당 페이지 주소를 가진 엔트리를 반환한다. 해시 테이블이므로 hash\_find 함수를 이용함으로써 쉽게 찾을 수 있을 것이다.
      8. 페이지를 메모리 프레임에 할당하는 함수도 필요할 것이다.
         1. Load\_page: 우선 위에서 구현한 lookup 함수를 호출하여 해당 페이지가 supp 테이블에 있는지 조사한다. 즉, 유효한 페이지 주소를 가리키고 있는지 검사한다. 이후 frame.c에서 구현한 allocate\_frame함수를 호출하여 프레임을 할당받은 후, 이 프레임에 해당 페이지의 데이터를 올린다. 이때, swap disc에 있다면 이후 나올 swap in 함수를 호출하고, 파일에 있다면 load\_from\_file 함수를 호출한다. 마지막으로 pagedir\_set\_page 함수를 호출하여 해당 주소와 프레임을 연결한다.
         2. Load\_from\_file: 해당 supp\_entry의 file값 등을 이용하여 file\_read 함수를 호출한다.
      9. 페이지가 쫓겨날 때, 만약 수정되었다면 파일에 이를 반영해야 한다.
         1. 해당 supp의 엔트리의 dirty가 true라면 이를 파일에 써야 하므로 file\_write\_at 함수를 호출함으로써 해당 페이지의 값을 파일에 쓴다. 이후 프레임을 free시키고, pagedir\_clear\_page 함수를 호출하여 해당 페이지를 not present로 마크한다.
      10. 페이지가 쫓겨날 때, 해당 엔트리는 frame 테이블, swap 테이블에서도 삭제되어야 한다.
          1. 페이지의 status에 따라, 만약 frame에 있다면 deallocate\_frame을 호출하여 해당 페이지의 프레임을 프레임 테이블에서 삭제한다.
          2. 만약 swap 테이블에 있다면 swap\_index에 해당하는 엔트리를 그 테이블에서 삭제한다.
      11. 2번 lazy loading에서 봤듯이, Exception.c의 page\_fault 함수에서 page fault가 일어났을 때 not\_present인 경우 load\_page를 호출함으로써 페이지를 프레임에 올린 후 return한다.
4. Stack Growth
   1. Basics: 지금까지는 1 page 크기로 size가 고정되어 있었다. 이제는 가변적으로 만든다.
   2. Limitations and Necessity: 1 page밖에 없으니 공간이 부족할 수 있다. 이 경우 페이지를 load하지 못할 수 있다. 따라서 스택 크기를 유동적으로 키워 유용하게 사용하도록 한다. Page fault가 발생했을 때 스택 크기에 관한 거라면 스택 크기를 키우도록 한다.
   3. Blueprint
      1. 우선 thread 구조체에 esp 멤버를 추가한다.
         1. 이 값은 user에서 커널모드로 초기에 변할 때 저장해야 한다. 따라서 syscall.c의 syscall\_handler 함수에서 현재 쓰레드의 esp 값을 f->esp 값으로 저장한다.
      2. Thread 구조체에 bottom을 추가하여 스택의 bottom을 가리키도록 한다.
         1. Setup\_stack 함수에서 현재 쓰레드의 bottom을 PHYS\_BASE – PGSIZE로 저장한다.
      3. 이후 page\_fault 핸들러에서 만약 user 값이 true라면 f->esp 값을 그대로 쓰고, 만약 아니라면 현재 쓰레드 구조체에 있는 esp 값을 사용한다. 이 값을 새로운 변수 curr\_esp에 저장한다.
      4. fault\_addr 값이 만약 PHYS\_BASE - MAX\_STACK\_SIZE 보다 크거나 같고, PHYS\_BASE보다는 작거나 같고, curr\_esp – 4 (푸시를 할 때 한 번에 32비트를 감소시키므로) 보다 크거나 같으면, stack\_growth 함수를 호출한다.
      5. Stack\_growth 함수: ‘thread\_current() -> bottom - PGSIZE’값을 인수로 받는다. 1개의 페이지를 만든 후 이 주소와 매핑한다. 이후 thread\_current-> bottom을 PGSIZE 만큼 감소시킨다.
5. File Memory Mapping
6. Swap Table
7. On Process Termination