**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재

이름 / 학번 : 20171659/유경연

개발 기간 : 11/12 ~ 11/24

1. **개발 목표**

이번 프로젝트에서는 가상 메모리를 구현한다. 현재 핀토스는 프로세스 실행 시 메모리에 한번에 적재하도록 구현되어 있다. 이를 page를 사용하여 가상 메모리를 사용할 수 있도록 수정한다. 이때 발생하는 page fault에 대해서는 page fault handler를 개선하여 해결하고 메모리를 swap하는 알고리즘을 적용해서 효율적인 메모리 활용을 가능하도록 한다. 또한 stack의 크기를 늘려야 하는 경우엔 stack의 크기를 늘려줄 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
   * Page Table & Page Fault Handler

현재 핀토스에서는 page fault 발생 시 프로그램이 종료된다. 가상 메모리를 사용하는 경우엔 page fault가 발생해도 조건이 만족된다면 프로그램이 계속 실행되어야 한다. 따라서 현재 page table로는 swap을 구현할 수 없으므로 page table을 개선하여 swap을 가능하도록 하고 page table이 변경되었으므로 page fault handler 또한 변경하여야 한다.

* + Disk Swap

Page fault가 발생하여 page를 새로 할당할 때 물리 메모리가 부족한 경우가 있을 수 있다. 이럴 때에는 현재 메모리에 올라와 있는 페이지를 디스크로 swap out해야한다. 이때 swap out할 페이지를 결정하는 알고리즘으로는 LRU와 LFU 등이 있다. Swap을 구현하는 경우 데이터를 필요할 때마다 lazily하게 디스크에서 swap in 할 수 있어져 효율성이 높아지게 된다.

* + Stack Growth

현재 핀토스 스택의 크기는 4KB이다. 스택을 초과하는 접근으로 발생하는 page fault에 대해 조건이 만족된다면 스택을 증가시키고 page를 새로 할당해서 page fault를 해결한다.

* 1. **개발 내용**
  + Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

어떠한 가상주소에 접근할 때 page가 page table에 없는 경우나 접근이 금지되어 있는 영역을 접근하려는 경우에 page fault가 발생하게 된다. Page fault가 발생하는 경우 supplementary table의 정보들을 사용하여 이 접근이 적절한지 판단한다. 적절한 영역에 접근하는 경우엔 swap이나 stack growth를 통해서 page fault를 handling한다. 만약 아니라면 process를 종료한다.

* + Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술

Page를 swap할 때 어떤 page를 swap out(evict)할지 결정하는 알고리즘으로는 ㅔpseudo-LRU 알고리즘인 second chance 알고리즘을 적용하였다. 이때 evict 여부는 accessed bit를 참고하여 결정한다. Accessed bit가 0인 경우엔 evict하고 accessed bit가 1인 경우엔 accessed bit를 0으로 만들고(second chance) 다음 entry로 넘어간다. 이때 dirty bit이 1인 경우엔 디스크로 swap out한다.

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

Page fault가 발생했을 때 page fault가 발생한 주소를 비교해서 스택의 확장 여부를 판단할 수 있다. 어떠한 limit를 설정하고 스택의 포인터와의 차이가 limit보다 작고 스택이 최대 크기에 도달하지 않았다면 스택을 확장한다. 이때 stack의 최대 크기는 8MB로 설정한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
  + 11/12 : vm 폴더에서 make check를 수행할 수 있도록 코드 수정
  + 11/13 ~ 11/20 : supplementary page table 구현
  + 11/21 ~ 11/22 : disk swap 구현
  + 11/23 ~ 11/24 : stack growth 구현
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
* 수정해야하는 소스코드
  + threads/init.c : disk swap에 필요한 자료구조 초기화
  + userprog/process.c : 가상 메모리가 구현되었을 때에 경우에 대해서 start\_process, process\_exit, load\_segment, setup\_stack 함수 수정.
  + userprog/exception.c : page\_fault 함수를 수정해서 가상 메모리가 구현되었을 때의 page fault를 handling하고 stack growth 처리.
  + userprog/syscall.c : string과 buffer의 주소가 적절한 주소인지 검사하는 함수 추가.
  + vm/page.c, vm/page.h : supplementary table 관련 함수 정의.
  + vm/frame.c, vm/frame.h : page 할당과 해제 관련 함수와 Pseudo-LRU 알고리즘 관련 함수 정의.
  + vm/swap.c, vm/swap.h : disk swap에 필요한 함수 정의
* 수정하거나 추가해야 하는 자료구조
  + thread/thread.h : thread 구조체에 가상 메모리 구현에 필요한 supplementary page table을 hash를 사용하여 추가. Pseudo-LRU 알고리즘에 필요한 리스트를 list를 사용하여 추가.
  + userprog/process.h : stack의 최대 크기를 8MB로 설정. stack growth의 limit 설정(stack 포인터로부터 이 limit 안에 포함되는 접근을 유효하다고 판단)
  + vm/page.h : supplementary page table의 entry를 새로 정의. page의 정보를 담고 있는 구조체 새로 정의
  + vm/swap.h : disk swap에 필요한 자료구조를 bitmap과 block으로 추가.
* 수정하거나 추가해야 하는 함수
  + userprog/process.c : start\_process, process\_exit, load\_segment, setup\_stack 함수 수정하여 가상 메모리 초기화, 추가, 삭제..
  + userprog/exception.c : page\_fault 함수를 수정하여 page fault 발생 시에 필요한 작업을 수행
  + userprog/syscall.c : check\_valid\_string, check\_valid\_buffer 함수를 추가하여 string과 buffer의 주소가 적절한지 검사.
  + vm/page.c, vm/page.h

텍스트이(가) 표시된 사진

자동 생성된 설명

* + vm/frame.c, vm/frame.h

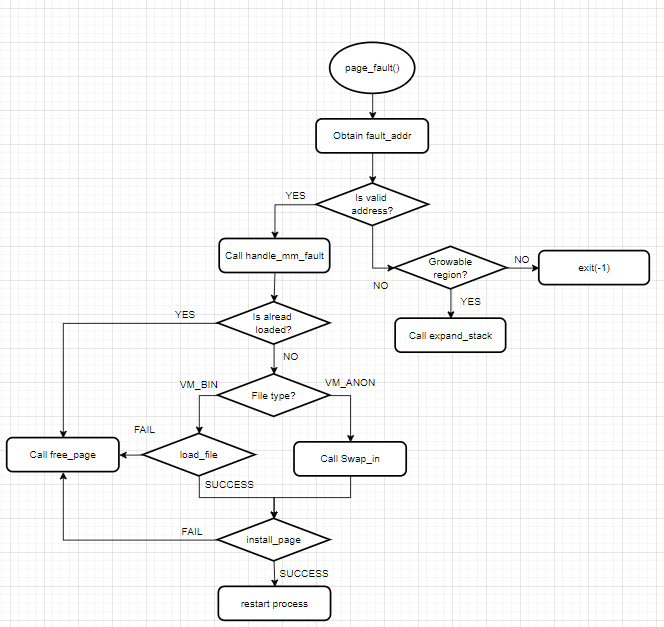
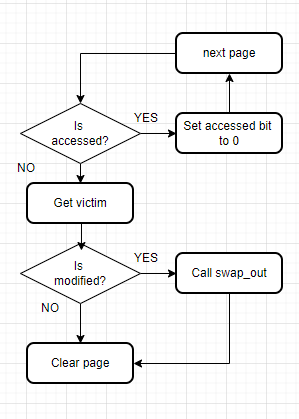
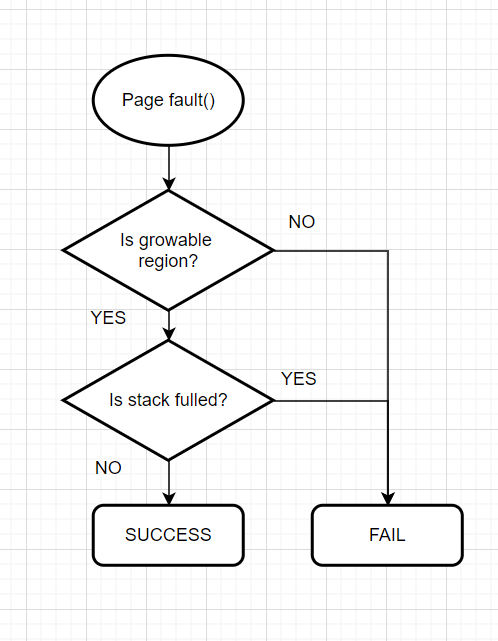
텍스트이(가) 표시된 사진

자동 생성된 설명

* + vm/swap.c, vm/swap.h

텍스트이(가) 표시된 사진

자동 생성된 설명

1. **연구 결과**
   1. **Flow Chart**
   * Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술
   * Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술
   * Stack growth 구현 시 stack 확장 여부를 판단할 수 있는 방법에 대해 서술
   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
  + Page fault가 발생하는 이유와 이를 handling하는 전반적인 과정을 서술

Page fault가 발생하면 userprog/exception.c의 page\_fault 함수가 호출되고 page fault가 발생한 주소를 얻는다. 우선 이 주소가 user 영역인지 검사하고 user 영역인 경우에만 진행한다. user 영역이 아닌 경우엔 exit(-1)을 호출해 thread를 종료하게 된다.

이후엔 page fault가 발생한 주소가 supplementary page table에 있는지 새로 구현한 vm/page.c 의 find\_vme 함수를 사용하여 검사한다. 이 함수는 현재 thread의 주소 공간에 가상 주소가 있는 지 확인하는 함수이다. 페이지 번호를 알아내기 위해서 pg\_round\_down 함수를 사용하고 page table을 hash로 구성하였기 때문에 hash\_find 함수를 사용한다.

Supplementary page table에 page fault가 발생한 주소가 있다면 handle\_mm\_fault 함수를 호출한다. 이 함수는 page fault가 발생하는 경우에 물리 페이지를 할당하는 함수이다. 이때 바이너리 파일로부터 데이터를 load하는 경우엔 이번 프로젝트에서 새로 구현한 vm/page.c의 load\_file 함수를 사용한다. 이 함수는 disk에 있는 file을 물리 페이지로 load하기 위해 filesys/file.c에 구현되어 있는 file\_read\_at 함수를 사용해서 read\_bytes만큼을 읽고 나머지 공간엔 0을 할당한다.

Swap 영역으로부터 데이터를 load할 수도 있다. 이런 경우엔 이번 프로젝트에서 구현한 vm/swap.c의 swap\_in 함수를 사용한다. 이 함수는 이후 swap에 대한 부분에서 자세히 설명한다.

이후 물리 메모리에 load가 완료되면 가상 주소와 물리 주소를 페이지 테이블에 mapping하기 위해서 userprog/process.c의 install\_page 함수를 사용하고 load flag를true로 설정한다. 여기까지 하면 handle\_mm\_fault가 종료된다.

Supplementary page table에 없는 경우에는 stack을 확장할 수 있는지 확인하고 stack을 확장한다. 만약 supplementary page table에도 없고 stack도 확장할 수 없다면 exit(-1)을 호출하여 thread를 종료한다.

* + Disk swap 발생 시 사용한 page replacement algorithm에 대해 서술

이번 프로젝트에서 page fault가 발생하였는데 물리 메모리가 부족한 경우에 page를 교체하기 위해서 second chance 알고리즘을 사용하였다. 이 알고리즘은 pseudo-LRU 알고리즘으로 accessed bit를 기준으로 하여 evict할 페이지를 결정한다. Accessed bit가 1인 경우에 0으로 만들고 evict하지 않는다(second chance 부여). 만약 accessed bit가 0인 경우엔 그 page를 victim으로 선정하고 evict한다. evict할 후보를 결정하기 위해서 frame.h에 lru\_list와 lru\_clock 변수를 선언하였다.

이번 프로젝트에서는 메모리를 할당 받기 위해서 vm/frame.c에 alloc\_page라는 함수를 새로 만들었다. 이 함수에서는 palloc\_get\_page함수를 호출하는데 만약 이 함수를 호출하고도 page를 할당 받지 못 했다면 try\_to\_free\_pages 함수를 호출한다. 이 함수가 second chance 알고리즘을 적용한 코드다. get\_next\_lru\_clock이라는 함수를 호출하여 evict될 후보를 하나 받아온다. 이 후보의 accessed bit이 1인지 확인하기 위해서는 핀토스에서 제공된 pagedir\_is\_accessed 함수를 활용하였다. 만약 true가 return됐다면 pagedir\_set\_accessed 함수를 호출해서 0으로 만들고 다시 get\_next\_lru\_clock 함수를 호출하여 다음 후보를 받아온다.

만약 1이 아니었다면 pagedir\_is\_dirty 함수를 호출해서 이 page가 수정되었는지 확인한다. 만약 수정이 되었다면 이번 프로젝트에서 구현한 vm/swap.c의 swap\_out 함수를 사용해서 swap 영역에 다시 쓴다. 수정되지 않은 경우엔 pagedir\_clear\_page와 \_\_free\_page 함수를 호출하여 page를 free한다. \_\_free\_page 함수는 이번 프로젝트에서 page를 free하기 위해서 새로 정의한 함수로 물리 메모리를 해제하고, lru\_list에서 page를 삭제한다.

Disk swap에서 사용하는 swap\_in과 swap\_out 함수의 경우 핀토스에서 제공된 bitmap과 block을 사용하여 구현하였다. Block을 단위로 읽거나 쓰기 위해서 block\_read, block\_write 함수를 사용했다.

이때 발생할 수 있는 concurrency 문제를 해결하기 위해서 lru\_list\_lock과 swap\_lock 변수를 만들어서 critical section을 정의하였다.

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

Stack의 확장 여부를 판단하기 위해서 page fault가 발생한 주소가 현재 스택의 포인터(esp)와 차이가 얼마나 나는 지를 확인한다. 이때의 차이가 미리 설정해준 STACK\_GROW\_LIMIT보다 작다면 stack을 증가시키기 위해서 userprog/process.c의 expand\_stack 함수를 호출한다. 이 limit는 process.h에 정의되어 있다. 임의로 32로 설정하였는데 테스트 케이스를 통과하는 데에 무리가 없었다.

Expand\_stack 함수가 호출되면 현재 stack이 full인지를 검사한다. 이때 stack의 최대 크기는 8MB로 설정하였다. 이 또한 process.h에 MAX\_STACK\_SIZE로 정의되어 있다. 만약 stack이 full이라면 stack의 크기를 더 이상 증가시킬 수 없으므로 stack을 확장할 수 없다.

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

텍스트이(가) 표시된 사진

자동 생성된 설명