**Pintos Project 4: Virtual Memory**

담당 교수 : 박성용

이름 / 학번 : 권형준, 20161566

개발 기간 : 12/07~12/23

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이번 프로젝트에서 구현할 것은 가상 메모리이다. 가상 메모리는 logical memory와 physical memory를 분리하여 한 번에 더 많은 프로그램(process)들이 메모리에 올라와 실행될 수 있게 해준다. 이는 프로그램이 실행될 때 코드의 모든 부분이 필요하지 않다는 발상에서 시작하게 된다. 가상 메모리를 구현하기 위하여 꼭 필요한 것이 바로 demand paging이다. Demand paging은 필요한 논리적 메모리 부분(page)가 물리적 메모리 부분(frame)에 없는 경우 disk에 있는 데이터를 frame으로 올리는 것을 뜻한다. 이 때 필요한 것이 이번 프로젝트에서 구현하는 page fault handler, disk swap이고 이를 통하여 구현될 수 있는 것이 stack growth이다. 처음 page table에서 필요한 page에 대응하는 frame에 메모리에 없는 경우 page fault handler가 이를 감지하고 새로운 frame을 할당하거나, 만약 메모리에 자리가 없다면(유효한 user page pool)이 없다면, 기존에 존재하면서 사용하지 않은 frame과 swap해주는 것을 한다. 이때, frame이 modified가 되었다면 다시 disk에 써주는 swap out의 과정을 거쳐야 된다.

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

각 thread마다 page table을 가지게 되어 page에 해당하는 frame의 주소, 즉 virtual memory address에 대응하는 physical memory address를 저장한다. 만약 page fault가 발생하면 이전에는 무조건 exit(-1)을 호출하여 프로그램을 종료하였지만, 이제는 page fault의 원인을 파악하여, page fault를 발생시킨 원인이 invalid memory access가 아닌 그저 memory(frame)에 올라오지 않은 것이라면 해당하는 frame을 load한다. 또는 stack의 경우 초기에는 1개의 page만 할당을 받아 0~4095byte까지 access가 가능하지만 이 영역을 초과하여 최대 8MB까지 수용할 수 있어야 된다. 즉, 처음에는 사용하지 않는 영역이기 때문에 1page만 할당하여 최대한 physical memory를 절약하는 것이다.

* 1. Disk Swap

Disk swap을 제대로 구현하게 되면, physical memory에 있는 page pool을(즉, user page pool)모두 사용하였을 때, 다른 program이 돌아갈 수 있게 현재 실행 중이지 않은 frame을 특정 알고리즘을 통하여(LRU) 구현하였다. Swap out되는 page들은 pintos에서 만들어 놓은 swap disk에 저장이 되었다가 다시 프로그램에서 필요할 때 읽어서 Page table entry에 입력이 된다.

* 1. Stack Growth

Stack으로 들어올 수 있는 데이터의 양이 늘어난다. 즉, 이전에 sc-boundary-3과 같이 유효하지 않은 주소를 호출하는 test들이 통과될 것이고, 최대 8MB까지는 변수를 수용할 수 있게 될 것이다.

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

Page fault는 프로그램이 실행 중 자신이 접근할 수 없는 영역의 주소에 접근할 때, 자신이 접근할 수 있는 주소 영역이지만 virtual memory를 사용함으로써 해당하는 주소 영역이 실제 메모리에 load되어 있지 않을 때 발생한다. 이렇게 page fault가 발생하면 control은 kernel로 넘어가고 “userprog/exception.c”의 page\_fault 함수가 호출이 된다. Page fault함수는 우선 page fault를 발생시킨 주소와 원인을 담고 있는 CR2(control register 2)를 판별한다. 만약 해당 주소가 valid하지 않으면 exit(-1)을 호출하고 valid하다면 추가적인 frame을 할당해준다.

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

Page replacement algorithm으로 LRU를 직접 구현하는 데에는 한계가 있다고 수업시간에 배웠다. 따라서 LRU를 최대한 모방하기 위한 여러가지 방법을 사용하는데, 그 중 reference bit을 사용하여 second chance algorithm을 사용하였다. Frame table에 각각 reference bit을 추가한다. 초기에는 1로 초기화하고 replacement의 대상으로 선택되었을 때 0으로 바꾸면서 한 번 더 기회를 준다. 다시 돌아왔을 때까지 사용되지 않은 frame은 reference bit이 0임으로 replacement의 대상이 된다.

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

우선 주어진 test들로 인해 확인이 가능하다. Page grow와 같은 test들은 초기에 할당되는 stack보다 많은 stack 영역을 사용한다. 이외에도 최대 8MB까지는 stack이 늘어날 수 있게 설정하였기 때문에 지역변수를 8MB까지 선언해보아도 된다. (단, 이 경우 매우 많은 swapping이 발생할 수 있음으로 시간이 오래 걸릴 수 있다.) 또한 기존의 SC-boundary-3과 같은 test들이 통과하지 않는 다는 것은 page fault시에 무조건 죽어버리는 점을 해결했다고 해석할 수 도 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

|  |  |
| --- | --- |
|  | **일정** |
| **12/7~12/11** | **Page fault handler + page swap** |
| **12/11~12/19** | **Stack growth** |
| **12/19~12/23** | **Merge 구현 시도…** |

* 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 Fault Handler**
* 수정해야 하는 소스코드
* userprog/exception.c

프로그램에서 page fault가 발생하면 exception.c의 page\_fault함수가 호출이 된다. 이전까지는 단순히 exit(-1)을 호출하고 프로그램을 종료하였지만, 여기서 추가적으로 해당하는 page가 memory에 없는 경우까지 고려하여 해결할 수 있게 한다.

* **Page Swap**
* 수정해야 하는 소스코드
* Userprog/exception.c

Page fault가 발생하였을 때, memory에 자리가 없을 때 실행 중이지 않은 frame을 swap out하는 부분을 추가해야 된다.

* Vm/page.c

현재 pintos에서는 frame, page등 virtual memory를 사용하기 위한 구조가 형성되어 있지 않다. 이를 구현하기 위해서는 page table, frame table 등의 구조체를 만들어야 되는데, 그 중 page table entry(PTE)등을 선언하고 page에 관련된 코드가 담겨있는 소스코드이다.

* Vm/frame.c

Virtual memory를 구현하기 위하여 frame table entry(FTE)와 frame에 관련된 함수를 담고 있는 소스코드이다.

* Vm/swap.c

실질적인 swap이 일어나는 소스코드이다. 이때 사용하는 replacement algorithm은 “second chance algorithm”으로 유사 LRU방법이다.

* 수정하거나 추가해야 하는 자료구조
* Struct PTE <Vm/page.c>

각 thread마다 page table을 가지고 있다. Page table은 hash함수로 되어 있으며, 각각의 원소는 PTE구조체로 이루어져 있다. Virtual, physical address와 present bit, read, write bit, dirty bit, swap bit등이 들어있으며 swap되었을 때 저장된 위치를 찾기 위한 idx도 있다. 구현을 완료하지 못한 merge는 아마 rw\_bit또는 pinning bit등을 구현하지 못하여서 구현하지 못한 것 같다.

* Struct FTE <vm/frame.c>

실제 memory에 저장되어 있는 frame을 나타내기 위한 구조체이다. 대응된 virtual, physical address와 대응하는 thread 그리고 replacement algorithm을 위한 reference bit이 있다.

* Struct list Frame\_Table<vm/frame.c>

frame들을 저장하기 위한 list. 한 번에 여러 개의 thread가 존재하기 때문에 전 프로그램에 frame table은 한 개 존재한다.

* Struct block \*swap\_slot <vm/swap.c>

Swap out하는 내용을 담기 위한 구조체 배열.

* Bool \*swap\_check<vm/swap.c>

현재 비어 있는 swap out 공간을 확인하기 위한 flag.

* 수정하거나 추가해야 하는 함수
* Void Swap\_in (vm/swap.h)

필요한 page를 메모리 안으로 swap in하는 함수

* Int Swap\_out (vm/swap.h)

만약 replace하려는 page의 dirty bit이 set되어 있다면, 즉 수정이 된 page라면 swap out을 통해 수정된 부분을 적어주어야 된다.

* Void \*Replacement (vm/swap.h)

Second chance algorithm을 구현한 부분, frame push에서 호출을 하며, 더 이상 메모리에 할당할 physical frame이 없을 때 기존에 있는 frame하나를 swap out하여 자리를 비우고 생긴 자리의 주소를 반환한다.

* Void frame\_init (vm/frame.h)

Frame table을 생성하고 swap\_check등의 자료구조를 초기화 한다.

* Struct FTE\* frame\_push (vm/frame.h)

새로운 Frame table entry를 frame table에 넣는 함수이다. 실제 메모리를 사용하는 page가 나오면 mapping하는 과정에서 어디에 있는지 추적하기 위해 frame table을 유지할 필요가 있다.

* Struct PTE\* page\_lookup (vm/page.h)

해당 thread의 page table에서 해당하는 page를 찾는다.

* Bool page\_add (vm/page.h)

해당 thread에 page를 새로 추가한다.

* Bool handle\_mm\_fault (userprog/exception.c)

Page fault handler에서 valid한 reference인 경우 원하는 page가 단지 in memory가 아니어서 page fault가 발생한 것이기 때문에 이를 해결하기 위해 호출하는 함수이다. 메모리 내에 자리가 있다면 새로운 frame을 할당하고 끝나지만 없다면 replacement algorithm을 호출하게 된다.

* **Stack Growth**
* 수정해야 하는 소소코드
* Userprog/exception.c

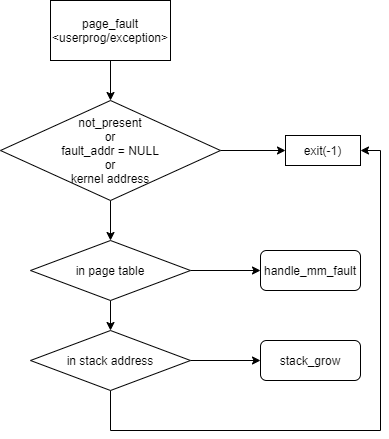
Stack은 처음부터 physical memory로 올라오는 것이 아니라, 한 덩어리씩 올라온 다음 더 많은 영역을 요구하게 되면 추가적으로 demand paging을 통하여 memory로 올라온다. 이 또한 page fault handler에서 처리해주어야 된다.

* 수정하거나 추가해야 하는 함수
* Bool stack\_grow (userprog/exception.c)

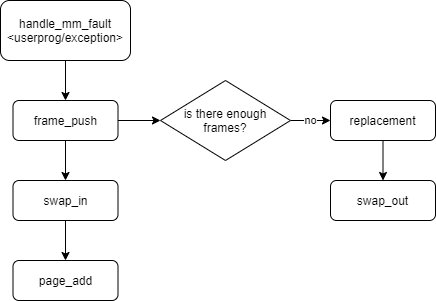
Page fault가 stack grow로 인하여 발생하였다고 판단이 되면 호출하는 함수이다. PHYS\_BASE부터 Fault가 난 주소까지 page를 할당해준다. 이때 palloc\_get\_page를 통하여 얻을 수 있는 frame이 더 이상 없다면, replacement algorithm을 통하여 새로운 주소를 받아온다.

1. **연구 결과**
   1. **Flow Chart**

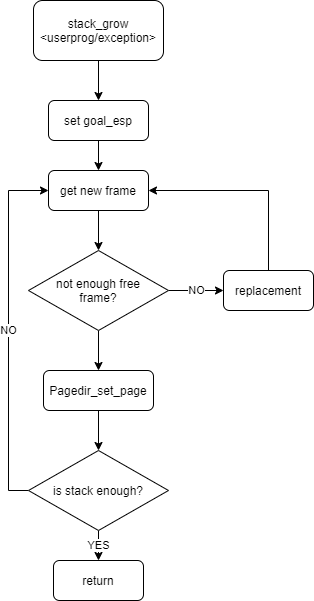
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성
* Page fault handler



* Page swap



* Stack growth



* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)

Page fault handler는 stack growth또는 page swap 둘 중 하나를 호출해주는 역할을 한다(그 외에도 완전히 invalid한 reference 때 exit을 해주기도 한다) 그렇기 때문에 코드는 stack growth, swap page 두개의 관점에서만 설명을 하며, 각각의 처음에 page fault handler가 어떻게 사용되는지 표시하였다.

* Page Swap

우선 exception.c에서 page fault handler가 호출이 된다. Page fault handler가 호출되었다고 해서 무조건 page swap이 필요한 것이 아니다. 우선 page fault를 발생시킨 fault address가 유효한 주소인지 확인할 필요가 있고(kernel address가 아님, 이상한 주소가 아님), 유효한 주소일 경우 중에서도 free frame이 없는 경우에만 page swap이 필요하다. 각각의 thread는 생성되면서 page table도 같이 생성된다. 따라서 page fault를 발생시킨 주소가 page table에는 있지만 fault address인 것으로 미루어 보아, 해당 주소의 page가 physical memory에 load되지 않았다고 유추할 수 있다. 이러한 경우 handle\_mm\_fault함수를 호출하게 된다. Handle\_mm\_fault 함수는 physical memory에 해당 주소의 page를 올리는 역할을 한다. 해당 주소의 page start address를 구한 뒤, thread의 page table을 검색하여 load해야되는 page를 찾는다. 다음 frame table에 해당하는 page를 frame\_push함수를 사용하여 추가한다. 이는 바로 뒤에서 자세하게 설명하겠다. Frame\_push함수가 수행하는 영역은, 새로운 frame table entry를 생성하고 새로운 physical address를 할당한 뒤 추가할 뿐이다. 실제 physical address에는 내용이 load된 것이 아니다. (즉, 접근만 가능하도록 frame table에 entry가 추가된 것) 따라서 실질적인 내용을 load해주는 함수가 바로 swap\_in이다.

이제, frame\_push와 swap\_in 함수를 더 자세하게 살펴보자. Frame push에서 해주는 것은 앞에서 설명했듯이 frame table에 새로운 entry를 추가하는 것이다. Frame table은 physical memory를 표현한 것으로 새로운 entry를 추가한다는 것은 physical memory의 frame으로의 접근을 가능하게 함을 뜻한다. Palloc\_get\_page를 사용하여 user poll영역의 빈 page(frame)을 얻어오려고 시도한다. 이때 실패한 경우 이는 user가 사용할 수 있는 physical memory가 꽉 차 있다는 뜻이다. 따라서 page replacement algorithm을 사용하여 victim frame을 정해야 되는데 이때 호출되는 함수가 replacement이다. Replacement 함수를 second chance로 구현하기 위하여 각 frame table entry내에는 reference bit이 있다. 초기 frame\_push에서 새로운 frame이추가되는 경우 이 bit을 1로 초기화 하였다. 또한 second chance algorithm의 경우 매번 victim frame을 정할 때 이전까지 돌았던 곳에서 다시 출발한다. 이를 저장하기 위하여 e라는 전역변수를 지정하였다. 처음에 e가 NULL인 경우, 즉 한 번도 replacement algorithm이 실행된 적이 없는 경우 frame table의 첫 원소로 지정하였다. 각 frame table entry를 돌면서 1을 0으로 만들고 처음으로 만나는 reference bit이 0인 frame을 victim으로 선정하여 swap out을 한다. 이후 victim frame을 swap\_out한 뒤, swap\_out한 disk의 위치를 page table entry안의 idx라는 변수에 저장하였다. 그 뒤에 victim frame을 frame table에서 제거한 뒤 새로 사용 가능한 주소를 반환하였다. 이를 frame\_push에서 새로운 frame에 할당하여 사용하게 된다.

Swap\_in과 swap\_out의 경우 disk 상의 공간으로 swap out, in하는 함수이다. Devices/block.h에 구현되어 있는 block구조체를 사용하여 page를 저장하고 불러올 수 있다. Swap in, out을 해주는 과정에서 대응하는 page table entry에 idx변수를 조절하여 만약 in memory page라면 idx를 -1로 설정하고 만약 in memory가 아닌 swap out된 page라면 swap out된 block의 number를 저장한다. 이 number는 swap.c에서 swap\_check라는 배열의 index로 들어간다.

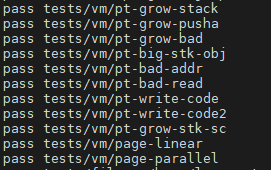
* Stack Growth

Stack growth도 page swap와 마찬가지로 page fault handler에서 출발한다. Thread는 stack을 처음에 1page만큼 할당하기 때문에 더 필요하게 되면, 즉 stack이 자라게 되면 추가적으로 stack의 영역에 demand paging을 통하여 physical memory를 mapping해주어야 된다. 우선 PHYS\_BASE와 PGSIZE의 차이를 통하여 처음의 esp를 구한다. 다음 목표 esp, 즉 얼마나 많은 page가 필요한지 구하기 위하여 fault address의 page start address를 구한다. 이제 처음 esp값에서 PGSIZE만큼 빼면서 fault address의 page start address와 같거나 더 작아질 때까지 반복한다. 이는 현재 fault를 발생시킨 주소와 stack의 시작 사이의 모든 page를 demand paging하는 것과 같다. 이 과정에서 만약 frame이 부족한 경우 앞에서 구현한 replacement algorithm을 사용하여 새로운 frame을 가져온다. Stack의 경우 기존의 page table에 entry가 없기 때문에, pagedir\_set\_page로 pintos상의 pagedir에 추가해주고 page\_add를 사용하여 구현한 page table에 추가한다.

* + 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
* Merge를 구현하지 못하였다.

Merge의 코드는 각각 1MB의 데이터를 16등분하여 sort한 뒤 merge하는 테스트 케이스인 것으로 파악하였다. System call시에 victim frame으로 선정되면 안되는 frame들이 생기게 되는데 이를 처리하지 못하여 테스트 케이스를 통과하지 못한 것 같다.

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



(merge는 구현하지 못하였고, 구현하지 않은 상태에서 make check을 하게 되면 시간이 너무 오래 걸리게 되어 src/tests/vm/Make.tests를 수정하여 애초에 check을 하지 않았다.)