Pintos 프로젝트 3. Pintos Virtual Memory

(설계 프로젝트 수행 결과)

과목명 : [CSE4070-01] 운영체제
담당교수 : 서강대학교 컴퓨터공학과 박성용

조원 : 28조 김태훈

개발기간 : 2016. 12. 05. -2016. 12. 16.

최 종 보 고 서

프로젝트 제목: Pintos 프로젝트 3. Pintos Virtual Memory
제출일: 2016. 11. 25.
참여조원: 28조 김태훈

I. 개발 목표

현재의 Pintos OS에서는 swap disk를 이용한 virtual memory를 지원하지 않는다. Pintos OS 내부 swap disk를 활용한 virtual memory 구현을 위하여 frame table과 supplementary page table을 추가하고, 이를 통하여 page(frame) swap, stack growth, memory mapped file 등을 구현하여 Pintos OS 내부에서 virtual memory의 사용이 가능하도록 한다.

II. 개발 범위 및 내용

가. 개발 범위

Virtual memory의 구현을 위해서는 demand paging이 우선적으로 구현되어야 한다. 현재의 Pintos OS의 page table로는 demand paging의 구현이 어렵다. 따라서 각 process가 사용하고 있는 physical memory의 frame에 대한 정보를 담은 frame table과 swap disk로 swap되어 있는 page에 대한 정보를 포함하고 있는 supplementary page table을 먼저 구현한다. 이후 demand paging과 memory swap을 구현하고 stack growth와 memory mapped file(mmap)을 추가한다.

나. 개발 내용

1. 각 process 별로 frame table과 supplementary page table을 추가한다. frame table과 supplementary page table에 정보를 추가하기 위해서는 각 process가 palloc_get_page(PAL_USER)를 통해서 알아서 user 영역의 page를 할당 받는 기존의 방식을 변경하여 반드시 frame.c 내부에 존재하는 frame 관련 함수를 통해서 page를 할당 받도록 한다. 이 과정에서 process.c의 load_segment 함수와 setup_stack 함수를 변경해야 한다. load_segment 함수는 demand paging을 구현할 때에도 변경해야 하므로 우선은 frame table과 supplementary table을 위한 함수들을 각각 frame.c와 page.c에 작성하고 setup_stack 함수를 변경한다.

2. Demand paging 기법은 process의 loading 과정에서 모든 segment를 load 하지 않고 supplementary page table에 관련 정보를 저장해 둔 뒤, process의 memory 영역에 실제로 access가 일어났을 때 frame을 할당 받아 segment를 load 하는 기법이다. 이를 통해서 memory를 좀 더 효율적으로 사용할 수 있으며, page fault handler에서 모든 load를 처리하기 때문에 stack growth와 mmap 구현 시에도 편리하다. 이를 위하여 process.c의 load_segment 함수에서는 suppage_insert를 통해서 supplementary page table에 최소한의 정보만 저장하고 실제 loading은 exception.c의 page fault handler 내부에서 일어나도록 한다.

단, system call 동작 과정에서 일어나는 invalid address에 의한 page fault의 경우에는 kernel이 page fault handler에 접근하게 되는데, 이 과정에서 kernel이 esp의 값을 NULL로 바꾸게 되면서 user의 esp의 값을 알 수 없게 된다. 따라서 system call의 check address 함수 내에도 관련 내용을 추가하여 system call과 관련된 demand paging을 수행하도록 한다. 이는 이후 구현할 stack_growth와 mmap에도 해당되는 내용이다.

3. 현재의 Pintos OS에서 각 프로세스의 stack은 page size인 4KB를 넘지 않는다. 따라서 process의 esp가 stack의 아래를 가리킬 경우에는 invalid address error가 발생한다. 이를 stack growth를 통하여 해결한다. esp가 가리키고 있는 위치가 stack 아래를 가리키고 있고 stack growth가 가능하면 새로운 frame을 할당받아서 stack을 늘린다. 단 system call에서 argument로 넘어온 address가 esp 아래에 위치할 경우에는 invalid address error이므로 process를 종료한다. stack이 무한히 자라게 되면 page swap이 너무 자주 일어나 OS의 성능이 저하될 수 있으므로 stack growth limit는 8MB로 제한하도록 한다.

4. file에 접근하기 위해서는 매번 file disk에 접근하여 필요한 내용을 memory에 load해야 한다. 이 과정에서 I/O overhead가 발생한다. 만약 file의 모든 내용을 main memory에 미리 올려둔다면 I/O overhead를 줄일 수 있다. 해당 file에 대한 process의 접근이 많을수록 이를 통한 효율은 더 뛰어날 것이다. 사용자는 mmap system call을 사용하여 이러한 기능을 이용할 수 있다. mapping을 수행하는 mmap system call과 unmapping을 수행하는 munmap system call을 구현한다.

5. page replacement(eviction)
현재 Pintos의 main memory는 kernel 1GB, user 3GB로 stack growth, mmap이 정상적으로 동작하기에는 부족하다. 또한 user memory size 이상으로 page를 할당할 수 없기 때문에 한번에 구동 가능한 process의 수에도 제한이 있다. 이러한 한계를 뛰어넘기 위해서 swap disk를 이용한 virtual memory를 구현하고 새로

allocate 할 frame이 없을 경우에는 기존에 사용되고 있는 frame을 evict하여 swap disk에 저장한다. 이를 위해 access bit를 활용한 modified LRU algorithm을 사용한 frame eviction target find 함수와 swap in, swap out 함수를 작성한다.

III. 추진 일정 및 개발 방법

가. 추진 일정

12.05 - 12.08 : frame table 구현
12.09 - 12.10 : supplementary page table 구현
12.11 - 12.12 : stack growth, mmap 구현
12.13 - 12.14 : page replacement (eviction, swap) 구현
12.15 - 12.16 : 프로젝트 마무리 및 보고서 작성

나. 개발 방법

Pintos manual과 강의 자료, Operating Systems 교재를 참고하여 관련 사항을 학습한다. 이번 프로젝트는 user program 프로젝트를 바탕으로 virtual memory의 관련 기능을 구현하는 것이므로 우선 기존에 작성해 둔 user program에 아무런 문제가 없어야 한다. 또한 demand paging을 구현하기 위해서는 기존의 load segment, setup stack, page fault handler, check address 함수를 수정해야 하므로 frame table과 supplementary page table을 구현하고 난 후에는 make check를 활용하여 기존의 userprog test case가 아무런 문제 없이 돌아가는지 확인해야 한다. frame table을 구현하면 모든 userprog test case를 pass 할 것이고, supplementary page table까지 구현하면 몇몇 robustness test case를 제외한 기존의 userprog test cass를 pass할 것이다. 이 모든 내용이 완료된 이후 stack growth와 mmap을 구현하도록 한다.

page replacement의 경우에는 pintos를 구동하는 linux server의 성능을 고려해야 한다. page swap 과정에서는 필연적으로 I/O overhead가 발생하게 되는데 이를 최소화하기 위해 기존의 page replacement를 변형하여 heuristic algorithm을 작성한다. 또한 malloc에 의해서 발생하는 delay를 최소화하기 위해 memory pool을 활용한다. page replacement algorithm의 종류는 다양하지만 이 중에서 가장 효율적이라고 알려져 있으며 가장 많이 사용되고 있는 accessed bit를 활용한 modified LRU algorithm을 사용하기로 한다.

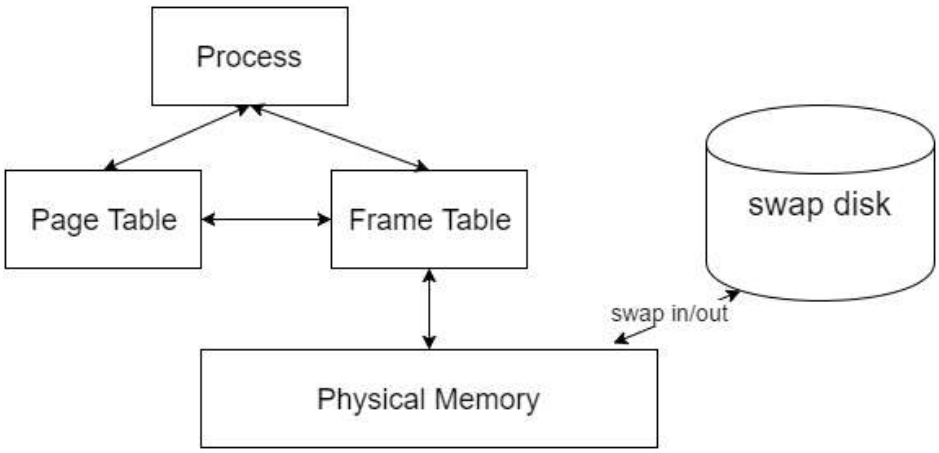
다. 연구원 역할 분담

혼자서 진행하는 프로젝트이므로 충분한 시간을 두고 철저한 계획 하에 프로젝트를 진행한다. 일정 부분 이상 진행되지 않으면 중간 결과를 확인 할 수 없으므로 구현 사항에 우선순위를 정하여 그 순서대로 프로젝트를 진행한다.

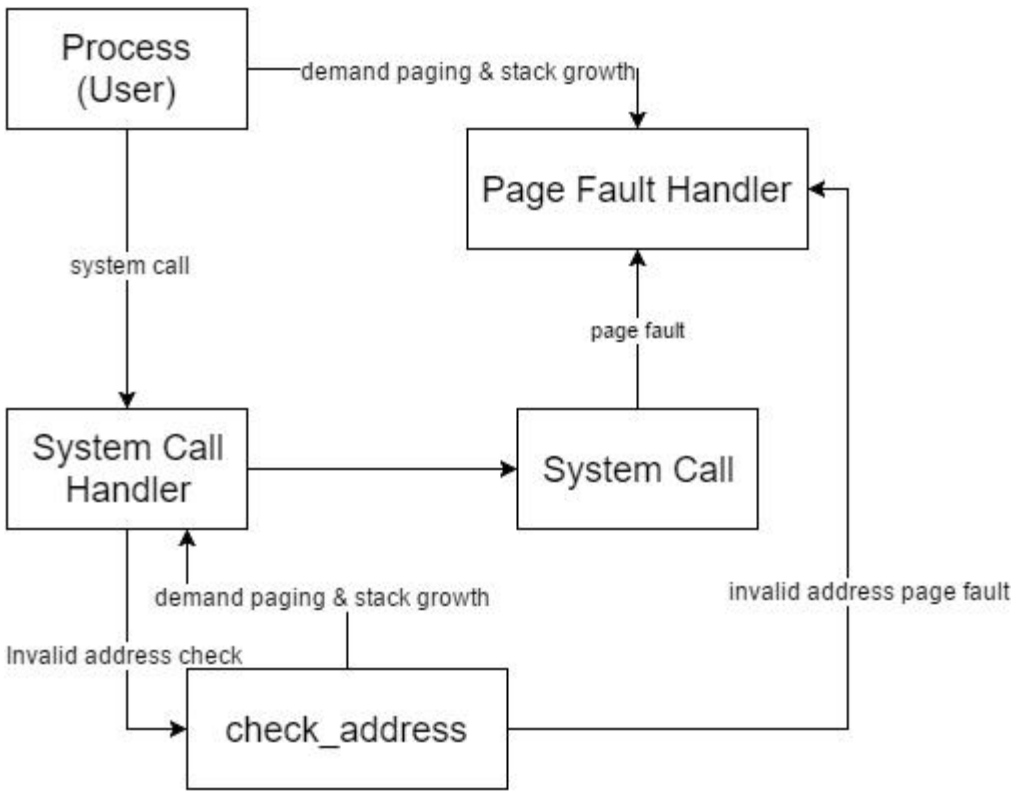
IV. 연구 결과

1. 합성 내용:

Memory management는 다음과 같은 방식으로 이루어진다.



Page fault handler와 check address에서의 demand paging은 다음과 같은 방식으로 이루어진다.



2. 제작 내용:

(1) Frame Table:

Frame Table의 구현을 위해서 frame.c를 새롭게 추가 하였다. frame table에는 해당 frame에 대한 kernel virtual address와 user virtual address, writable 여부, 해당 frame이 allocate 된 thread의 정보 등이 저장된다. frame table을 구현한 이후에는 user 영역의 frame을 allocate 받을 때는 process가 palloc_get_page(PAL_USER) 대신 frame_get_page(PAL_USER)를 호출하도록 하여 frame table에 관련 정보를 추가한 뒤 palloc_get_page(PAL_USER)를 호출하도록 한다. page free 할 때에도 마찬가지로 palloc_get_page 대신 frame_get_page를 process가 호출하도록 한다. 이 내용을 process.c의 setup_stack에 반영하도록 한다. load_segment 함수는 demand paging을 구현해야 하므로 supplementary page table 구현 시에 수정한다.

(2) Supplementary Page Table and Demand Paging :

Supplementary page table은 demand paging과 swap in/out의 구현을 위해서 필요하다. 이미 구현되어 있는 page table을 변경하는 방법도 있지만 이를 위해서는 pagedir.c에 있는 함수를 상당 부분 수정해야 하므로 새로운 page table을 추가하기로 한다. supplementary page table에는 해당 page의 user virtual address를 비롯하여 swap이 일어난 경우 해당 위치를 가리키는 file pointer와 offset, 해당 page의 용도(segment, mmap) 등이 포함되어 있다. load_segment 함수에서는 suppage_insert를 호출하여 supplementary page table에 내용을 추가하고 실제 loading은 해당 page가 실제로 호출되었을 때 page fault handler에서 이루어지도록 한다.

(3) Stack Growth: Stack growth의 구현은 demand paging과 유사하게 이루어진다. 단 stack growth에서 새롭게 allocate 되는 frame은 아무 내용도 올라가있지 않은 empty frame 이어야 한다.
따라서 frame_get_page(PAL_USER|PAL_ZERO)를 호출하여 0으로 초기화된 frame을 할당받아 이를 해당 process의 stack에 추가한다. 단 user virtual address가 현재 esp의 아래에 있는 경우는 잘못된 address이므로 해당 process를 종료해야 한다. stack growth는 page 단위로 이루어져야 하므로 pg_round_down()함수를 이용하여 address를 word_align 한 뒤 사용하도록 한다. 올바른 stack address 인지 확인하는 방법은 다음과 같다.

```
address >= STACK_SIZE_LIMIT && address <= PHYS_BASE &&
f->esp <=(uint8_t*)PHYS_BASE && f->esp >= STACK_SIZE_LIMIT
```

stack growth는 page fault handler에서 이루어진다. 단, system call에서 invalid address check에서 page fault가 발생할 경우 esp가 NULL로 초기화 되므로 이 경우는 page fault로 넘어가지 않고 check_address 함수 내에서 demand paging이 일어나도록 한다.

(4) memory mapped file:

file의 내용을 전부 memory에 mapping 시키는 mmap은 file_seek와 file read를 활용한다. file의 모든 내용을 mmap system call에 의해 mapping 해두면 memory의 낭비가 심해지므로 file을 page 단위로 demand paging 하도록 한다. mmap system call을 담당하는 mmap 함수 에서는 file을 page 단위로 쪼개어 segment load와 마찬가지로 supplementary page table에 저장한다. mmap의 actual loading은 segment와 마찬가지로 page fault handler에서 이루어진다. 단, system call에서 invalid address check에서 page fault가 발생할 경우 esp가 NULL로 초기화 되므로 이 경우는 page fault로 넘어가지 않고 check_address 함수 내에서 demand paging이 일어나도록 한다.

(5) Swap In/Out and Page Replacement Algorithm:

한 번에 너무 많은 process를 execute하거나 stack growth, mmap 등이 계속 진행되다 보면 available frame이 모자라 새로운 frame을 allocate 할 수 없게 된다. 이 경우 swap table로의 frame eviction이 일어나야 한다. 이를 위해 swap.c에 관련 함수를 새로 추가 하였다. 해당 기능의 구현을 위해서 block.c의 block_write, block_read 등의 함수를 활용하였으며, memory read, write time을 줄이기 위해서 bitmap을 활용하였다.

frame eviction에는 accessed bit를 활용한 LRU(Least Recently Used) Algorithm을 적용하였다. accessed bit의 값은 pagedir_is_accessed()를 통하여 접근 할 수 있다. 현재 적용된 demand paging은 pure demand paging이므로 frame queue에 들어있는 frame이 아직 한 번도 load되지 않은 page일 수도 있다. 따라서 이 경우에는 accessed bit를 확인하여 true일 경우 eviction을 하지 않고 queue의 다음 node로 넘어간다(second chance). false일 경우에는 eviction을 진행하여 해당 frame을 swap disk로 옮긴다(swap out). 해당 frame의 내용이 process에 의해 다시 호출될 경우에는 swap disk에서 찾아 새로운 frame을 할당받아 해당 process의 page table에 추가하고 swap disk에 저장되어 있던 내용을 frame에 load 한다.

frame에 특정 내용을 저장하는 도중 해당 frame의 eviction이 일어나는 일을 막기 위해 frame table에 pinned 변수를 추가하여 frame의 내용이 변경되는 도중에는 해당 frame이 eviction target에서 제외되도록 하였다.

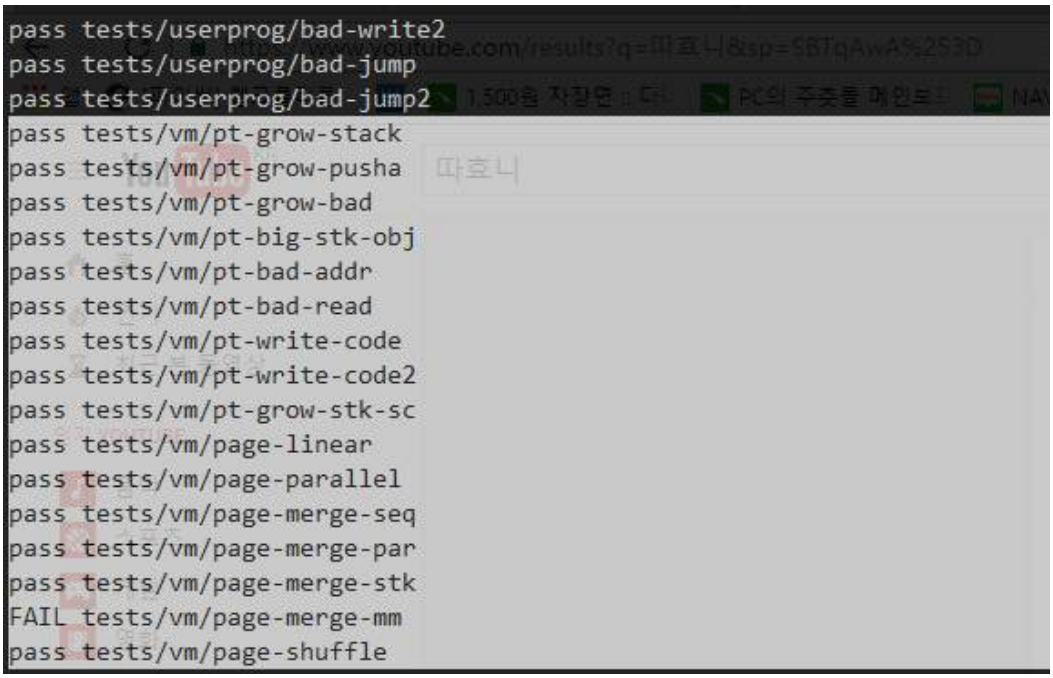
(6) Trial and Error:

프로젝트를 진행하면서 Project 2. User Program에 발견하지 못한 invalid address 관련 error를 발견하고 이를 수정하였다. 또한 알려진 LRU Algorithm을 Pintos OS에 그대로 적용하기에는 qemu emulator가 구동되는 cspro 서버의 성능이 너무 느려서 malloc을 사용하는 경우를 최대한 줄이고 대신 외부에 따로 선언한 table entry pool을 이용하였다. 또한 각 process가 가지고 있는 supplementary page table과 frame table은 hash table로 구현하여 속도를 높였다. 이외에도 기존의 user program의 source code를 상당 부분 수정하였다.

3. 시험 및 평가 내용:

Project 결과에 대한 평가는 make check의 pass/fail을 기준으로 진행하였다.
평가는 cspro1.sogang.ac.kr에서 진행하였으며 simulator로 qemu를 사용하였다.
그 결과는 다음과 같다.

총 16개의 test 중 page-merge-mm을 제외한 15개의 test를 pass 하였다.



(1) 다음 8개의 test를 통해 생산성을 test하였다.

pass tests/vm/pt-grow-stack
pass tests/vm/pt-grow-pusha
pass tests/vm/pt-big-stk-obj
pass tests/vm/pt-grow-stk-sc
pass tests/vm/page-merge-seq

pass tests/vm/page-merge-par
pass tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm

page-merge-mm을 제외한 모든 test case를 문제없이 수행하여 의도한 결과를 출력하여 생산성을 만족하였다.

(2) 다음 8개의 test를 통해 내구성을 test 하였다.
pass tests/vm/pt-grow-bad
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-shuffle

각종 예외 상황을 적절하게 대처하여 Pintos OS의 비정상적인 종료를 방지하였으며 이로서 내구성을 만족하였다.

(3) 보건 및 안정성 test
OS가 마주하는 예외 사항은 다양하다 . 이에 대해 적절하게 대처하지 못하면 사용자의 불편을 초래하게 된다. Virtual memory management 과정에서 synchronization이 제대로 이루어지지 않으면 page fault가 발생하면서 Pintos OS가 비정상적으로 종료될 수도 있다. 안정성을 확인하기 위해 make check를 계속 돌려보고 이를 통해 안정성을 확인하였다. 또한 평가 항목에서는 제외된 mmap 관련 test와 기존의 userprog test case를 대부분 통과하여 Pintos OS의 user 영역이 문제 없이 안정적으로 구동됨을 확인하였다. 해당 결과는 아래에 첨부하였다.

```
pass tests/vm/pt-grow-stack
pass tests/vm/pt-grow-pusha
pass tests/vm/pt-grow-bad
pass tests/vm/pt-big-stk-obj
pass tests/vm/pt-bad-addr
pass tests/vm/pt-bad-read
pass tests/vm/pt-write-code
pass tests/vm/pt-write-code2
pass tests/vm/pt-grow-stk-sc
pass tests/vm/page-linear
pass tests/vm/page-parallel
pass tests/vm/page-merge-seq
pass tests/vm/page-merge-par
pass tests/vm/page-merge-stk
FAIL tests/vm/page-merge-mm
pass tests/vm/page-shuffle
pass tests/vm/mmap-read
pass tests/vm/mmap-close
FAIL tests/vm/mmap-unmap
pass tests/vm/mmap-overlap
pass tests/vm/mmap-twice
pass tests/vm/mmap-write
FAIL tests/vm/mmap-exit
pass tests/vm/mmap-shuffle
pass tests/vm/mmap-bad-fd
FAIL tests/vm/mmap-clean
FAIL tests/vm/mmap-inherit
pass tests/vm/mmap-misalign
FAIL tests/vm/mmap-null
pass tests/vm/mmap-over-code
pass tests/vm/mmap-over-data
pass tests/vm/mmap-over-stk
pass tests/vm/mmap-remove
pass tests/vm/mmap-zero
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
FAIL tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
7 of 109 tests failed.
make: *** [check] 오류 1
cse20121577@cspro1:~/pintos/src/vm/build$
```

V. 기타

1. **연구 조원 기여도**: 김태훈 (100%)

2. 해당 프로젝트를 진행한 cspro1.sogang.ac.kr은 서버의 낙후화로 다양한 문제점들이 발생하였다. bochs simulator를 사용할 수 없어 qemu simulator만을 사용하여야 했으며, git이 설치되어 있지 않아 source code version control에 어려움이 있었다. 하지만 느린 서버 환경이 다양한 synchronization 문제를 발생시켜 이를 해결하며 Pintos OS 자체의 성능을 향상하는데 도움이 되었다. 또한 page swap in/out의 overhead를 최대한 줄이기 위해 bitmap과 hash 등의 자료구조를 적절히 활용하였다.