**Pintos Project 4: Virtual Memory**

담당 교수 : 김영재

이름 / 학번 : 원종윤 / 20201604

개발 기간 : 12.01 – 12.10

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

Virtual memory와 관련된 page, supplement page table, page fault handler, swap algorithm, eviction algorithm, stack growth를 구현한다.

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

기존 핀토스에선 page fault 발생시 exception.c에 존재하는 page\_fault handler를 호출하여, user/kernel memory access 여부를 확인하고 오류 메시지를 출력하고 종료한다. 이를 대체하여, fault 발생시 fault address, virtual page에서 data를 찾아내고 free할 resource를 결정한다.

* 1. Disk Swap

Physical memory에 공간이 부족할 경우, disk swap을 수행하여 page를 swap-out한다. 이 때, swap page는 다양한 page replacement algorithm 가운데 하나를 채택한다. Swap table을 구현하여 slot의 가용 여부를 관리한다.

* 1. Stack Growth

Page fault handler에서 stack 공간 부족과 적절한 공간 접근 여부를 판단하여 page를 추가로 할당 받아 stack growth를 구현한다.

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

메모리에 존재하지 않는 page에 접근하는 경우 page fault가 발생한다. 이 때, handler에서 fault\_address를 통해 발생 주소를 파악할 수 있고, skeleton code에서 제공하는 함수를 이용하여 어느 영역 (kernel/user)에 접근을 시도하였는지 확인한다. 이후, 잘못된 접근에 이어 스택 추가 할당 가능 영역 및 swap 여부를 확인한 후, page를 할당 받아 stack을 확장하거나 disk swap을 수행하거나, resource free 및 process 종료한다.

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

시간 부족으로 구현하지 못하였으나, second chance algorithm을 사용하고자 하였다. 이 알고리즘은 FIFO와 유사하지만 reference bit를 구현하여 frame에 존재하는 데이터가 접근될 경우 reference bit을 1로 set한다. 이후, 해당 데이터의 eviction turn이 돌아오면 reference bit을 0으로 set하고 다음 데이터를 evict한다.

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

우선, fault\_address가 유효한 주소 (0 ~ PHYS\_BASE)에 존재하는지 확인한다. 이후 확장 가능성을 확인하기 위해 GROW SIZE인 GSIZE (8MB)를 PHYS\_BASE에서 뺀 값 이하인지 확인하고, esp-32를 체크한다.

즉, 0 < esp-32 < fault\_addr < PHYS\_BASE-GROW SIZE < PHYS\_BASE의 상태인지 확인한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

12.01 – 12.07

기존 project에서의 error를 모두 처리한 후 구현해야 하기에, 기존 project의 error를 수정한다. (multi-oom)

12.07 – 12.10

Stack growth를 구현한다.

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야 하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수

기존 thread.c가 충돌을 일으키는 경우가 있으나, 해당 프로젝트는 thread의 기능과 무관하다고 판단하여 thread.c와 thread.h 등 기존 thread project 소스코드를 배제하고, user program project 제출물을 기반으로 구현하였다.

기존 exception.c에서 page\_fault() 함수를 수정하여 page fault handler를 구현한다. 이 때, 기존에 user, kernel을 확인한 후 exit을 호출하던 코드를 삭제한 후, page가 존재하는지 확인한 후 stack growth가 가능한지 확인하고 page를 할당 받아 stack growth를 수행하도록 구현한다.

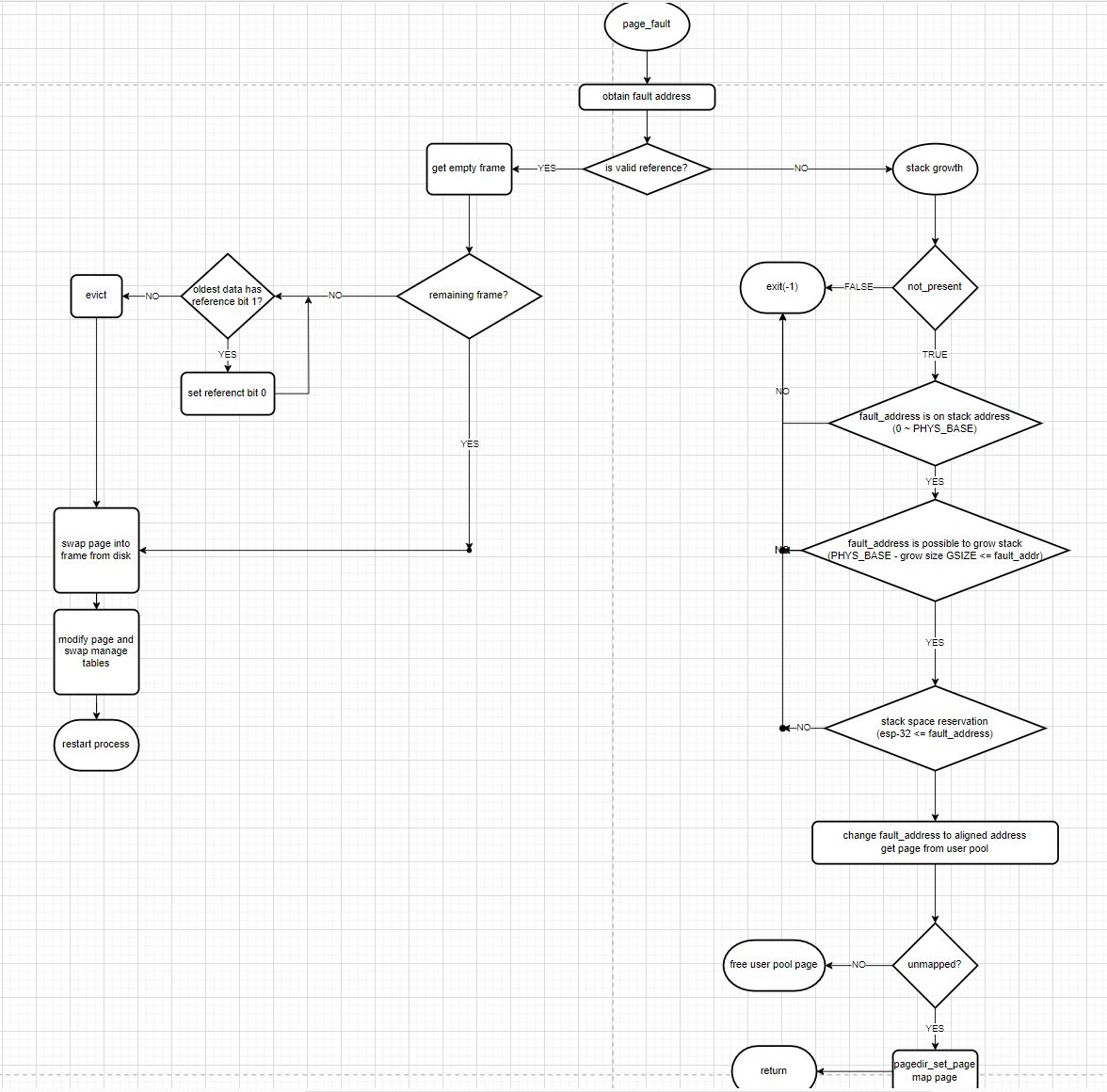
Syscall.c에서 multi-oom infinite loop를 해결하기 위해 비정상 종료 process가 eixt process에서 load\_sema를 수정하고 죽는 line을 제거한다.

Process.c에서 multi-oom을 해결하기 위해 load()에서 argv 마지막 칸에 NULL이 있는 경우 argc를 1 감소시키도록 수정한다.

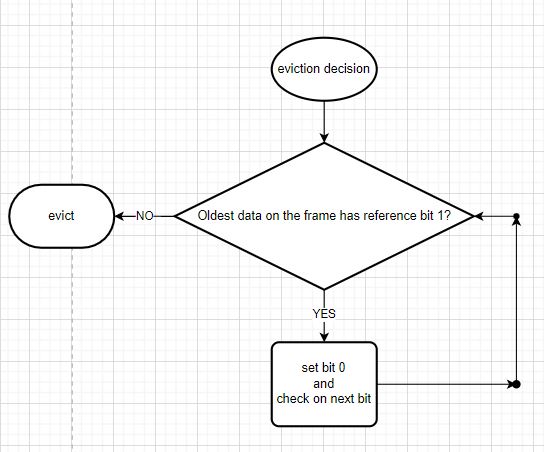
1. **연구 결과**
   1. **Flow Chart**

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

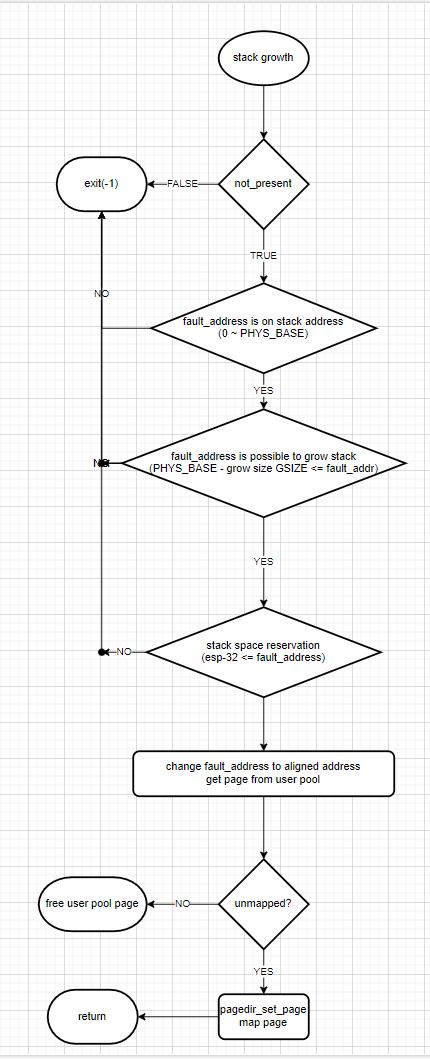
Page fault handler



Second chance algorithm



Stack growth 판단



* 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

Thread project는 구현에 영향을 끼치지 않는다고 생각하여, 배제하고 기존 user program project 제출물을 기반으로 구현하였다.

Exception.c/page\_fault

*if* (!not\_present) exit(-1);

*else* *if* (fault\_addr > 0 && fault\_addr < PHYS\_BASE && PHYS\_BASE - GSIZE <= fault\_addr && *f*->esp - 32 <= fault\_addr) {

   void \*page\_addr = pg\_round\_down(fault\_addr);

   void \*kpage = palloc\_get\_page(PAL\_USER);

*if* (pagedir\_get\_page(thread\_current()->pagedir, page\_addr) == NULL)  pagedir\_set\_page(thread\_current()->pagedir, page\_addr, kpage, true);

*else*  palloc\_free\_page(kpage);

*return*;

   }

*else*  exit(-1);

not present 여부를 확인하고, stack growth 가능 여부를 확인하기 위해 여러 조건을 확인한다. 우선, fault\_addr이 0과 PHYS\_BASE 사이에 존재하는지 확인한다. 이후, stack growth size (GSIZE)를 PYHS\_BASE에서 뺀 값이 fault\_addr보다 작거나 같 같은 확인한다. 또한, 확장을 위해 f->esp-32가 fault\_addr보다 작거나 같은지 확인한다.

모든 조건을 만족한다면, stack growth가 가능하므로 현재 thread의 page directory 정보와 fault address를 pagedir\_get\_page에 넘겨 해당하는 kernel virtual address를 반환한다. 이 때, pg\_round\_down을 사용하여 page alignment에 적합한 주소 값을 전달한다. 이후 unmapped 상태라면 pagedir\_set\_page를 호출할 때, palloc\_get\_page에 PAL\_USER를 전달하여 user pool에 해당하는 page를 사용하도록 하고, fault\_address와 user pool virtual address를 mapping하고 종료한다.

하지만 mapped 상태라면 user pool에서 가져온 kapge를 free시키고 종료한다.

**이슈**

#1\_1. User program project: multi-oom 수정

기존 user program project의 문제를 모두 해결하고 virtual memory project를 시작하는 것이 권장되기에, multi-oom의 오류를 해결하고자 gdb debugging을 통해 원인을 파악하였다. 그 결과, 기존 test에선 mutli-oom n의 시작과 동시에 page fault가 발생하여 process가 이상 종료되는 것을 확인하였고, 이를 자세히 확인하기 위해 다시 stack을 해체하여 input file name의 blank parsing 과정에 오류가 존재하여 process가 이상 종료되는 것을 깨달았다. 따라서 이를 수정하고자, 기존 process.c의 argv parsing을 마친 후 마지막 argv에 NULL이 존재할 경우, argc를 1 감소시키도록 하여 process의 정상 진행을 보장하였다.

Process.c/load

*if*(argv[argc-1] == NULL) argc--;

#1\_2 multi-oom 89

1\_1의 수정 후 mutli-oom이 time over로 종료되는 경우는 제거되었으나, 80~90번 사이, 주로 multi-oom 89에서 FAIL이 발생하였다. 역시 gdb debugging을 통해 확인하니, 기존 time over(infinte loop) 현상을 처리하는 과정에서, 이상 종료되는 thread가 부모 thread의 load semaphore를 변경하지 않고 종료되는 것을 방지하기 위해 system call exit에서 변경 여부를 확인하고, 변경한 후 종료하는 line이 문제를 일으킨 것을 알게 되었다.

Process.c /start\_process

  sema\_up(&(thread\_current()->parent->sema\_load));

syscall.c/exit

*if*(thread\_current()->load\_flag==0){

    sema\_up(&(thread\_current()->parent->sema\_load));

    thread\_current()->load\_flag=1;

  }

따라서, 1\_1의 문제를 해결한 후 해당 line도 제거되는 것이 옳기에 이를 제거하여 multi-oom을 pass하였다.

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

