**Pintos Project 1: User Program (1)**

담당 교수 : 박성용 교수님

조 / 조원 : 20181679 장서우

개발 기간 : 2020.10.03-2020.11.01

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

주어진 pintos에서는 system call, system call handler, argument passing 및 user stack을 비롯한 많은 OS 기능들이 구현되어 있지 않아서 program을 실행시킬 수 없다. 이번 프로젝트에서는 argument passing, user memory access, system call의 구현을 통해 user program을 제대로 실행할 수 있는 환경을 구축하는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

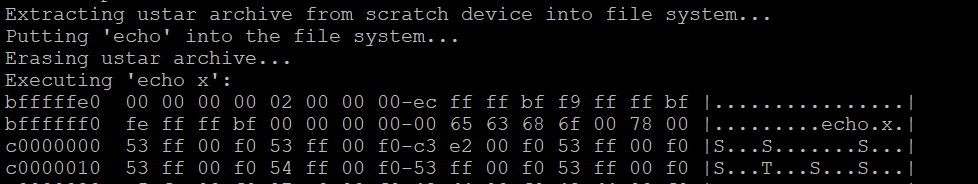
* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

사용자가 명령한 명령어를 parsing 하여 실행하려는 프로그램 이름과 인자들을 구분하여 스택에 저장하고, 실행한 프로그램이 정상적으로 인자들을 넘겨받을 수 있도록 한다.

다음은 다음 명령어를 통한 hex\_dump()를 통한 결과값이다.





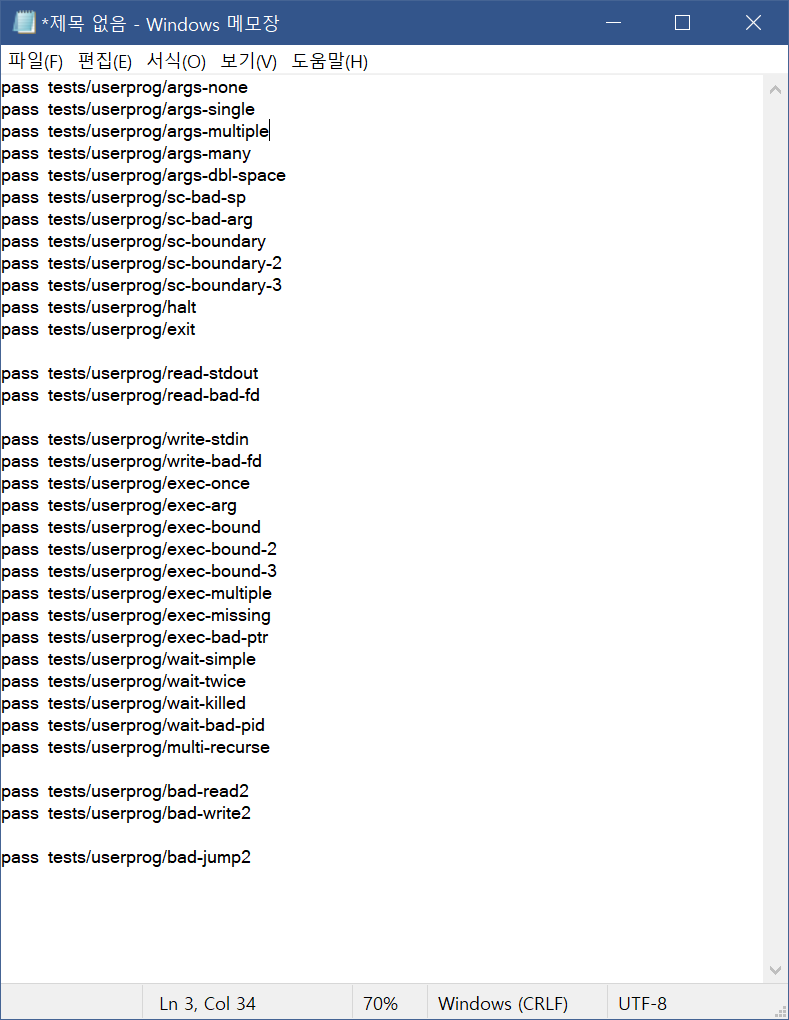
1. User Memory Access

User Program의 영역과 Kernel Program을 구분하여 서로 간의 메모리 침범이 없도록 관리하여 준다. process가 잘못된 메모리에 접근하거나 file을 load하지 못할 경우 등 여러 잘못된 접근이 일어났을 때 process를 kill을 한다. 이때 잘못된 접근을 한 thread를 종료하고 -1을 parent thread에게 반환하는 과정이다.

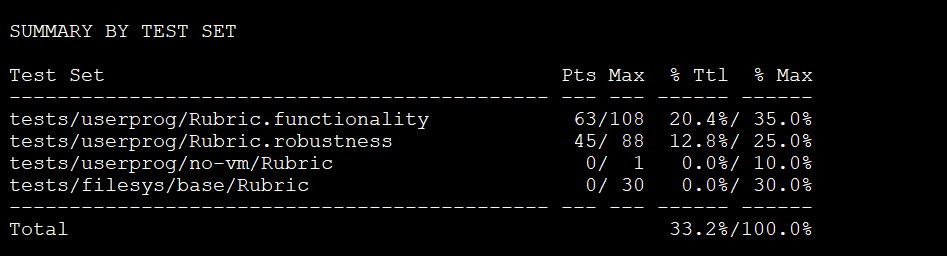
1. System Calls

user program에서 kernel mode에 접근 가능하도록 system call을 처리하기위한 기본 구조와 process 관련된 system call 함수 6개 halt(), exit(), exec(), wait(), read(), write()을 구현하였다. additional system call로 fibonacci(), max\_of\_four\_int()를 구현하였다. process termination message 출력 또한 구현하였다.

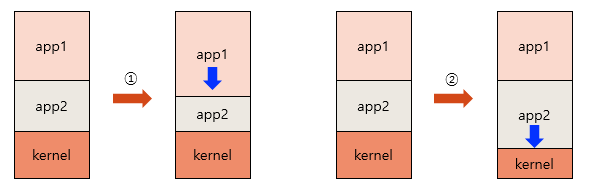
* make check를 통해 최종 결과값을 확인할 수 있다.



* make grade



* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명
    - construct esp를 하기 위해서 가장 위에 각각의 argument들의 value 값과 마지막에 ‘\0’ 문자를 추가로 넣어준다. 단 push해 주는 순서는 반대로 push 해야 한다. push 하다보면 push 한 횟수가 4의 배수가 아닐 수 있는데 pointer의 값을 4의 배수로 맞추기 위해 나머지 byte는 0으로 채운다. 또한 각각의 argument의 주소값을 push하고 argc와 return address를 push해 주어야 한다.
* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
    - pintos는 메모리를 user memory와 kernel memory 두 영역으로 나눈다. 이러한 메모리 영역을 직접적으로 사용하게 되면 메모리를 관리하기 어려워진다. 다음과 같이 각 process가 서로의 영역을 침범하게 되거나 process가 kernel code 영역을 손상시킬 수 있기 때문이다.



* + - 이를 해결하기 위해 운영체제는 가상 메모리 시스템을 채택한다. 가상메모리이기 때문에, 각 process에는 자체 메모리 영역이 있으며 process가 전체 메모리를 차지하는 것처럼 이를 사용할 수 있다. pintos 또한 가상 메모리로 메모리 영역을 관리한다. 가상메모리도 user virtual memory와 kernel virtual memory 두 영역으로 나뉜다.
    - 각 process는 각자의 virtual memory를 가지고 있으며, pintos는 kernel에 phys\_base(3GB)~4GB 까지의 1GB 영역을 할당한다.
    - pintos에서 memory 단위는 page로 size는 4KB이다. user program은 physical memory에 page directory 와 page table을 통해 가상 주소를 변환하여 접근할 수 있다.
    - 이때 process가 잘못된 메모리에 접근하거나 file을 load하지 못할 경우 등 여러 잘못된 접근이 일어날 수 있다.
  + Invalid memory access를 어떻게 막을 것인지 설명
    - user-provided pointer의 유효성을 확인(PHY\_BASE 아래에 있는지 확인)하고 dereference한다. 만약 pointer가 유효하지 않으면 이는 page fault를 일으킨다. 이는 userprog/exception.c의 page\_fault()를 수정하여 해결할 수 있다.
    - user program에서 PHY\_BASE와 0xc0000000사이의 주소를 갖는지 system call handler에서 지속적으로 확인한다.
* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명
    - Pintos는 각 process와 kernel code를 보호하기 위해 메모리를 user virtual memory와 kernel virtual memory로 나눈다. OS는 user program이 핵심 기능을 포함하고 있는 kernel memory에 access 하는 것을 막고 있다.
    - user program이 kernel의 기능을 사용하기 위해서는 system calls이 필요하다.
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
    - void syscall\_halt (void): shutdown\_power\_off함수를 호출하여 pintos를 종료한다.
    - void syscall\_exit(int status): process 종료 시 출력하는 message를 출력하고, thread\_exit 함수를 호출한다.
    - pid\_t syscall\_exec(const char \*cmd\_line): 존재하지 않는 파일인 경우 예외처리를 한 후 process\_execute 함수를 호출한다.
    - int syscall\_wait(pid\_t pid): process\_wait 함수를 호출한다. 이는 child process가 완료될 때까지 대기하는 일을 한다.
    - int syscall\_read(int fd, void \*buffer, unsigned size): file descriptor이 0인 경우(STDIN), Input\_getc() 함수를 이용하여 한 글자씩 읽는다.
    - int syscall\_write(int fd, void \*buffer, unsigned size): file descriptor이 1인 경우(STDOUT), putbuf() 함수를 이용하여 출력한다.
    - int fibonacci(int n): n번째 피보나치 수를 구한다.
    - int max\_of\_four\_int(int a, int b, int c, int d): a, b, c, d 중 최대값을 구한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명
    - main()에서 시작되어 run\_actions(argv)를 호출한다.
    - run\_actions(argv)안의 a->function(argv)가 run\_task를 호출한다.
    - run\_task가 process\_wait에 들어갈 process\_execute를 호출한다.
    - process\_execute안의 thread\_create가 user program 이름을 등록한다. 그리고 start\_process를 통해 user program을 실행시킨다.
    - process scheduling이 호출되면 child process(user program)은 \_start()에 의하여 실행된다. 이는 user program이 종료되면 exit이 호출되고 이는 process\_wait로 돌아간다.
    - process\_execute에서 실행된 부모 process는 process\_wait에서 child process가 끝날 때까지 기다린다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

10. 03 ~ 10. 09: 매뉴얼 분석 및 pintos 코드 이해

10. 10 ~ 10. 16: argument passing 구현

10. 17 ~ 10. 23: system call 구현 (additional 포함)

10. 24 ~ 10. 30: accessing user memory 구현

10. 31 ~ 11.01: 마무리 작업, 보고서 작성

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. Argument Passing

* userprog/process.c의 load() 윗부분에서 명령어를 parsing해주는 코드를 추가한다.
* userprog/process.c의 load()에서 setup\_stack()이 끝난 후, 커널 내 스택에 argument를 쌓는 코드를 추가한다.
* userprog/process.c의 process\_execute()에서 파일 이름을 받고, file open했을 때 NULL이면 return 처리해준다.

1. User Memory Access

* userprog/exception.c의 page\_fault()에서 user memory가 아니면 exit(-1)를 호출하여 kernel memory에 접근하는 것을 방지한다.
* userprog/syscall.c에서 is\_user\_vaddr함수를 통해 user memory인지 지속적으로 체크해준다.

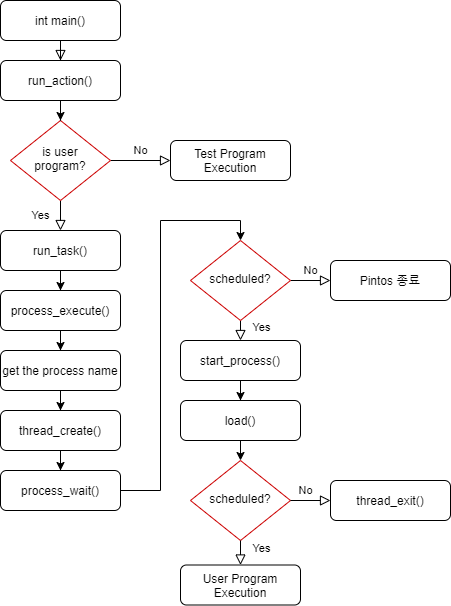
1. System Calls

* userprog/syscall.c에서 syscall\_handler()에 switch 문을 추가하여 system call number에 따라 해당하는 system call function을 호출하도록 한다. 이는 lib/syscall-nr.h에서 system call number를 참고하여 코드를 짠다.
* userprog/syscall.c에 system call function을 구현한다: syscall\_halt(), syscall\_exit(), syscall\_exec(), syscall\_wait(), syscall\_read(), syscall\_write()
* userprog/syscall.c에 additional system call function을 구현한다: fibonacci(), max\_of\_four\_int()
* 이에 맞게 userprog/syscall.h를 수정한다.
* 또한 additional system call 구현을 위해 lib/syscall-nr.h에 system call number를 추가하고, lib/user/syscall.h에 system call API의 prototype을 적는다. 또 lib/user/syscall.c에 syscall4()를 새로 정의하고 system call API도 정의한다.
* exit과 wait 기능을 수행하기 위해 threads/thread.h에서 thread struct 내에 변수를 추가하고 threads/thread.c의 thread\_create()에서 추가한 변수를 초기화한다.

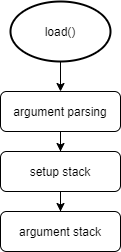
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

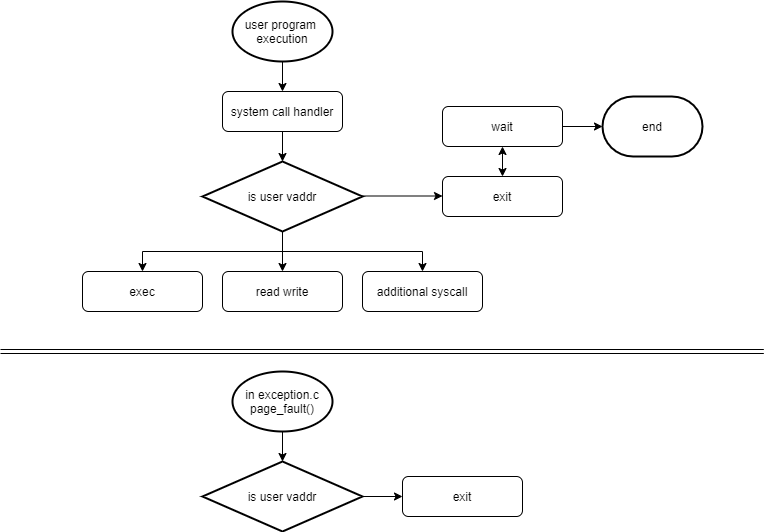
1. 전체적인 flow chart



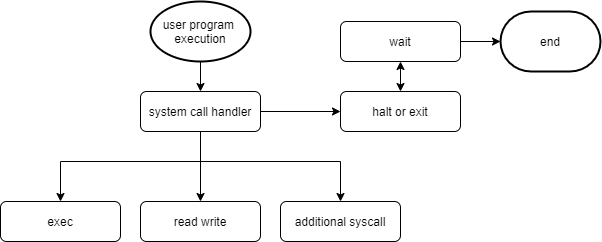
1. Argument Passing



1. User Memory Access



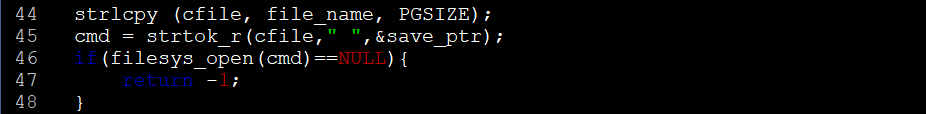
1. System Calls

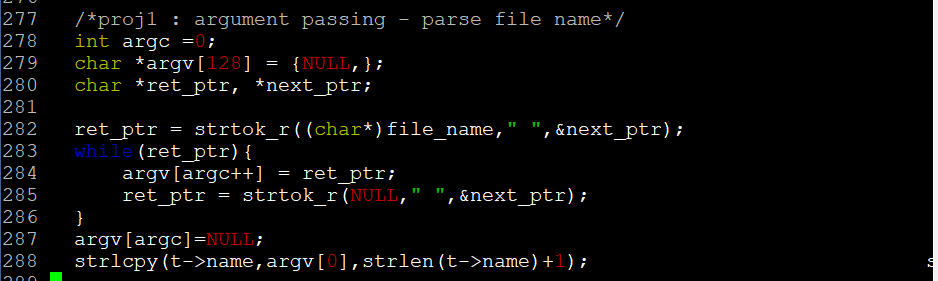
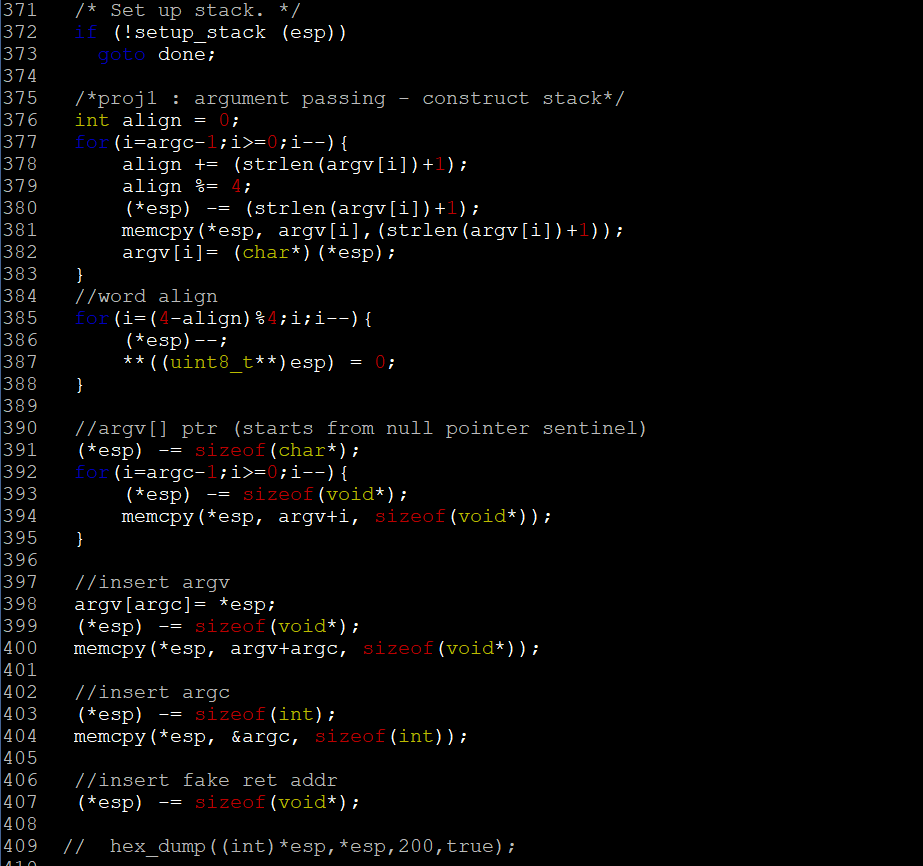


* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

* userprog/process.c의 process\_execute()에서 파일 이름을 받고, file open했을 때 NULL이면 return 처리해준다.



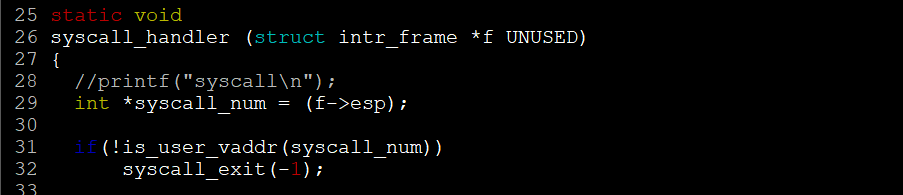
* userprog/process.c의 load()에서 parsing을 위해 다음과 같은 코드를 추가하였다. thread의 name에 명령어가 잘 넘어가도록 마지막 줄을 추가하였다.  
  
* userprog/process.c의 load()에서 setup stack이 끝난 후 stack에 위에 설명한 순서대로 stack을 쌓는 과정이다. 마지막의 hex\_dump를 통해서 중간과정을 확인할 수 있다. 각 주석에 해당하는 부분에 맞는 코드가 적혀 있다.   
  

1. User Memory Access

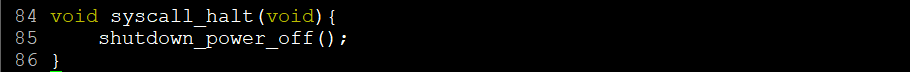
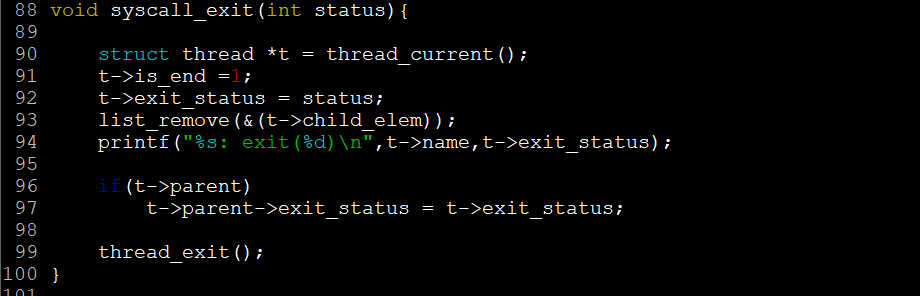
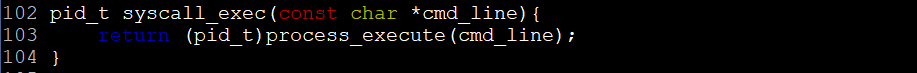
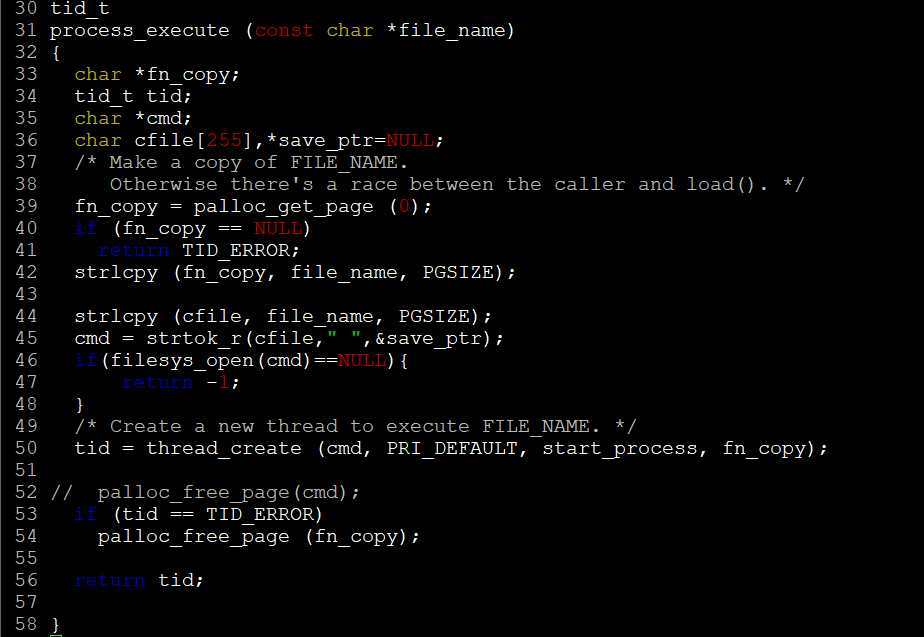
* userprog/exception.c에서 page\_fault() 안에 다음과 같은 코드를 추가하여 user memory access를 확인해주었다. 이를 통해 exec\_bad\_ptr, sc\_bad\_sp, wait-killed를 해결할 수 있었다.

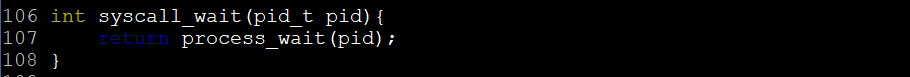


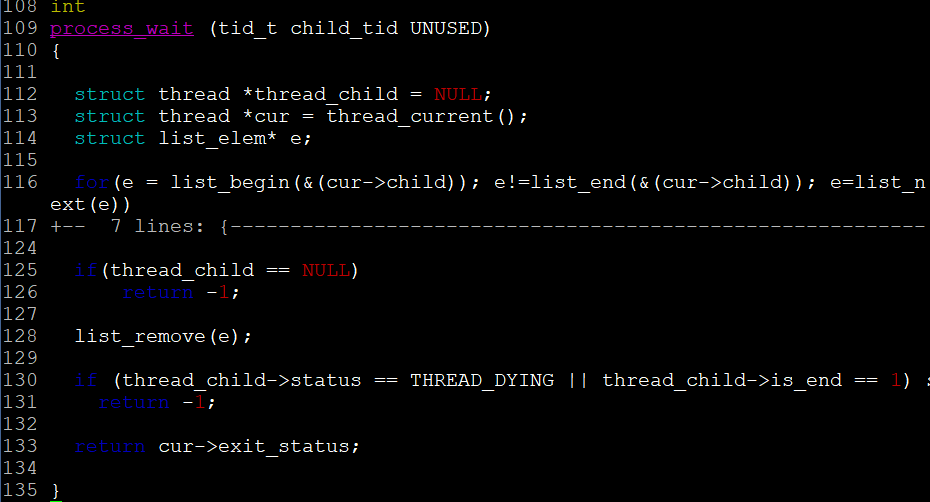
* userprog/syscall.c에서 다음과 같은 방법을 반복적으로 사용하여 user memory access를 확인해주었다. 아래 syscall\_handler 설명에서 더 자세한 코드를 확인할 수 있다.

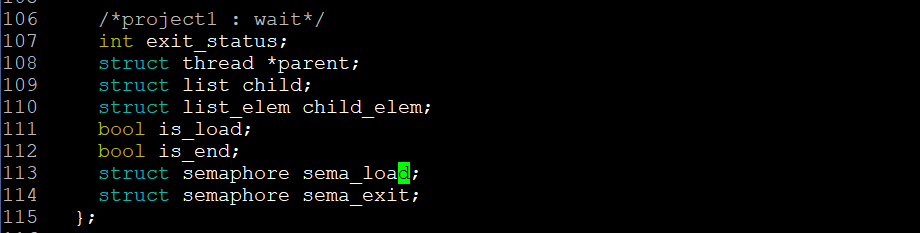


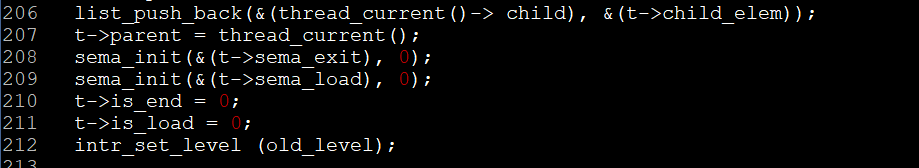
1. System Calls

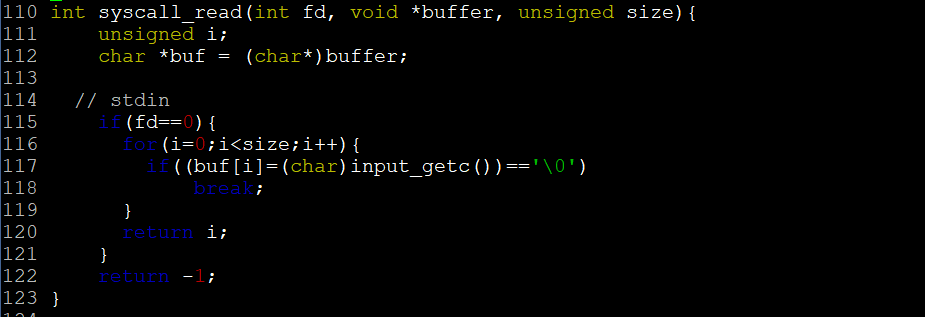
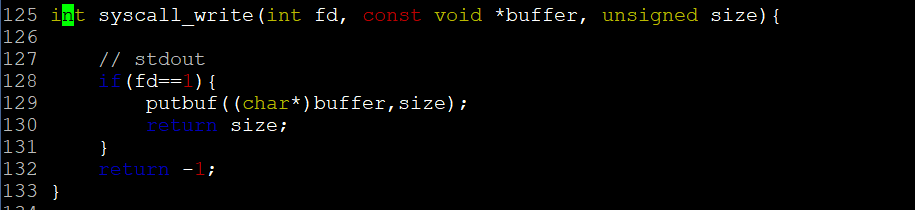
* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
* system handler를 userprog/syscall.c에서는 switch문을 사용하여 다음과 같이 구현한다. 해당하는 case number는 lib/syscall-nr.h에서 확인할 수 있다.  
  
* syscall\_halt(): 시스템 자체의 종료를 위해 shutdown\_power\_off()를 호출한다.  
  
* syscall\_exit(): 해당 process name과 함께 termination message를 출력하고 exit status를 전달하고 thread\_exit()를 실행한다.  
  
* syscall\_exec()/process\_execute(): 주어진 cmd\_line을 실행하고 새로운 process의 pid를 반환한다. syscall\_exec()는 process\_execute()를 호출하여 구현하였다.   
   
* syscall\_wait()/process\_wait(): syscall\_wait()에서는 pid를 받아 process\_wait()로 넘겨주는데 이는 종료된 child thread의 식별자이다. process\_wait는 입력으로 이를 받아 해당하는 pid를 가진 child thread를 찾아 child thread의 is\_end가 1이 될 때까지 parent thread를 semaphore를 사용하여 sleep state로 유지시킨다. child process가 종료된 것이 확인되면 child process의 status를 return한다. 사용되는 semaphore를 위해 thread.h에서 thread struct에 변수를 추가하였다. 또한, thread.c의 thread\_create()에서 추가한 변수를 초기화한다. busy waiting을 사용하려하였지만 어려움을 겪어 간단한 semaphore 방법을 구현하였다.



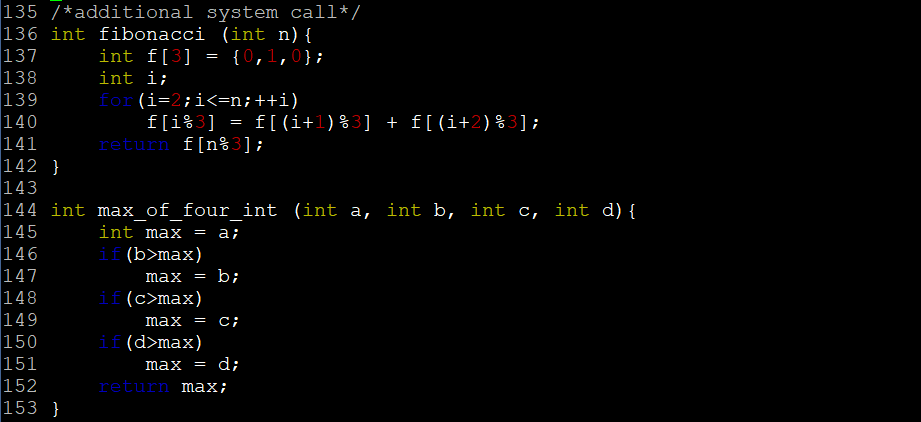
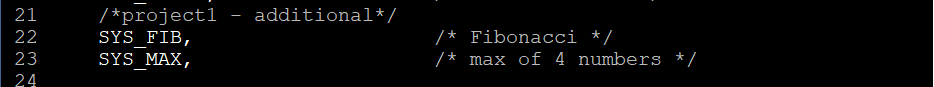
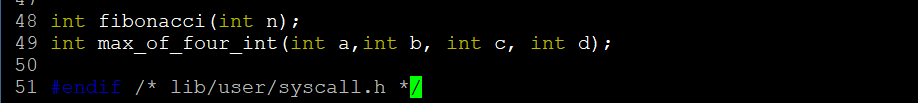
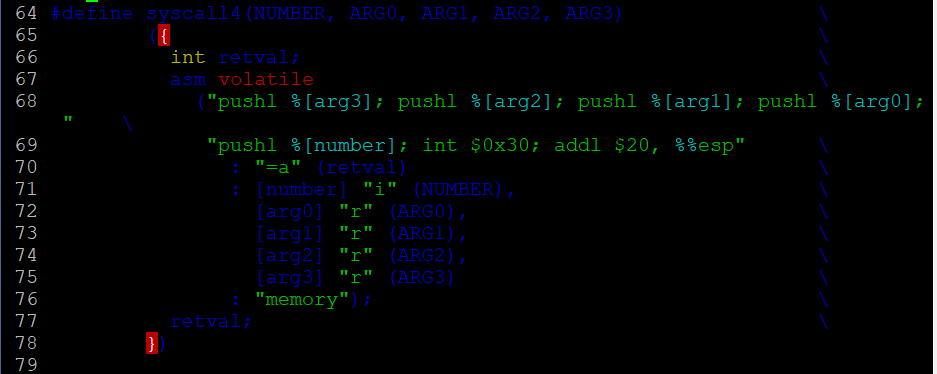
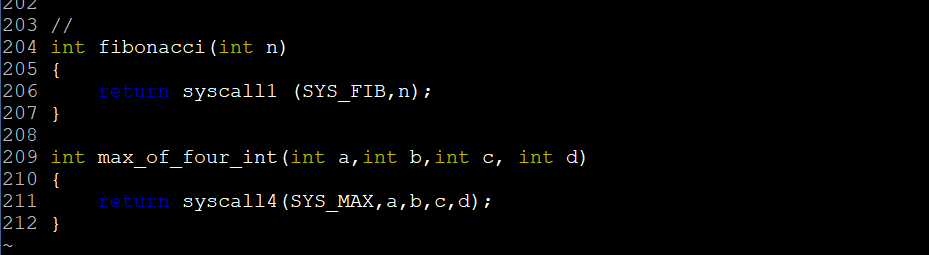
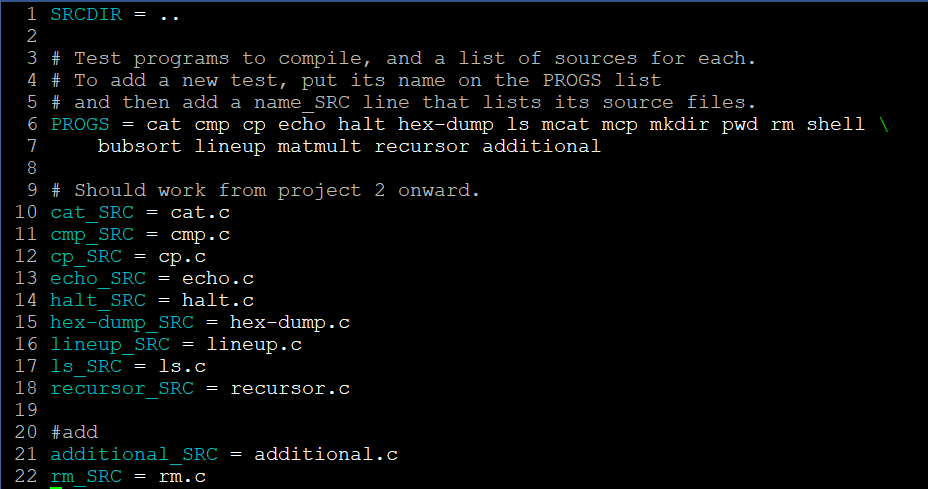




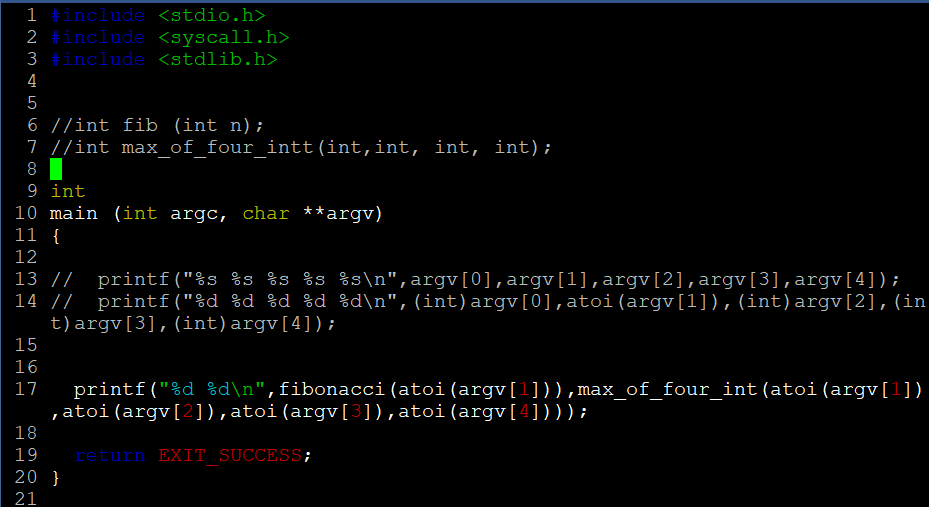


* syscall\_read(): stdin을 input\_getc()를 사용하여 구현한다.  
  
* syscall\_write(): stdout을 putbuf()을 사용하여 구현한다.
* 

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
* userprog/syscall.c에서 함수를 구현한다.  
  
* lib/syscall-nr.h에 system call number를 추가하였다.   
  
* lib/user/syscall.h에 system call API의 prototype을 추가하였다.  
  
* lib/user/syscall.c에 syscall4()를 새로 정의하고 system call API도 정의한다.   
    
  
* src/examples에서 makefile을 수정하고, additional.c를 생성하였다. 다음을 통해 아래와 같이 pintos에서 ‘additional’이 실행이 가능해졌다.  
    
  

**pintos/src/userprog$** pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run 'additional 10 20 62 40'



* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

