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

담당 교수 : 김영채

조 / 조원 : 김현우

개발 기간 : 20210915 ~ 20211002

1. **개발 목표**

처음에 핀토스가 제대로 구현이 되어 있지 않아서, process 가 실행이 되지 않는다. 스택에 메모리를 쌓기 위해 argument passing을 구현하고, 각 상황에 맞는 system call을 실행하기 위해 system call handler가 stack 의 esp 포인터에 따라 움직이고, 원래 의도한 대로 프로그램이 굴러가도록 알맞은 system call을 구현해 준다. 여기에서 프로그램 실행중에 올바른 메모리를 가르킬 수 있도록 user memory access 작업도 같이 해준다. 프로세스가 실행 중일 때는 semaphore를 사용해서 자녀와 부모 프로세스의 작업 속도도 맞춰 주도록한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

* Stack에 의도한 argument가 넘어가서 system call을 부를 때 사용 가능

1. User Memory Access

* 올바른 메모리만 접근가능 하게 됨. 만약 올바른 메모리가 아니면 exit을 실행해서 스레드가 종료된다.

1. System Calls

* System call handler에서 알맞은 시스템 콜이 실행됨 / 부모 자식 스레드의 실행 순서가 정해짐
  1. **개발 내용**
* Argument Passing

pintos의 user program의 입력으로 다수의 argument가 들어올 수 있어 이를 80x86 Calling Convention에 맞추어 메모리에 배치해야 한다. Program에 입력된 명령어를 strtok\_r을 사용하여 띄어쓰기를 기준으로 token해준다. 그리고 token한 명령어의 주소를 포인터 배열에 저장. 그 후 esp를 PHY\_BASE부터 해서 명령어와 명령어의 주소값을 저장한다. 단어 명령을 저장한 후에는 word-align을 통해 메 모리 주소가 4의 배수 단위로 끊기게 설정한다.

* User Memory Access

핀토스에서 invalid memory access란 널 포인터에 접근하는 것 , 할당 되지 않은 가상메모리 공간에 접근, (PHY\_BASE)위인 커널 메모리에 접근하는 것을 의미한다. 널 포인터의 접근은 NULL이라면 exit을 하게 하면 되고, 할당 되지 않은 공간 접근은 pagedir\_get\_page() 으로, 커널 주소에 접근하는 것은 is\_kernel\_vaddr()로 방지한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

유저 프로그램이 만약에 메모리에 직접 쓰기를 하거나, 장치가 돌아가는 것을 제어 할 수 있게 된다면 안정성이나, 보안 측면에서 문제가 생기므로 kernel만이 수행할 수 있는 명령의 범위를 정하고, system call을 통해서 user process는 유효한 명령만 system call을 통해 실행하면 kernel은 이를 받아서 간접적으로 명령을 실행해준다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

Halt()

-shutdown\_power\_off()를 통해서 핀토스를 종료한다.

exit()

-현재 유저 프로세스를 종료하고, 종료 상태를 커널에 반환한다.

exec()

-자식 프로세스 생성하고 userprog/process.c에 있는 process\_execute의 반환값을 리턴한다.

Wait()

-자식 프로세스가 자신의 일을 마치기 전까지 기다린다. 자식의 pid가 유효한지 확인(process\_wait 호출해서 확인하고 기다린다.) 자식 스레드가 종료되면 종료 원인인 숫자를 넘겨받는다.

Write(), Read()

-해당 파일 지시자에 쓰거나 읽는다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명
* 시스템콜을 호출하면 /lib/user/syscall.c에 있는 해당 시스템콜로 이동해서

user stack에 인자를 쌓는다. 그 다음에 int 30 을 호출해서 interrupt vector 테이블로 이동한 이후에는 userprog/syscall.c로 이동해서 숙제로 구현한 시스템콜을 시행한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 기간 | 내용 |
| 1단계 (9/15 ~ 9/22) | Argument passing 및 stack 만들기 |
| 2단계(9/22 ~ 9/25) | Memory access 및 system call handler 구현 |
| 2단계(9/22 ~ 9/25) | Thread 구조 수정 및 semaphore로 동기화 |

* 1. **개발 방법**
* **Argument Passing** (userprog/process.c)

- Argument Passing을 위하여 userprog/process.c의 load()에서 명령어를 parsing하고 stack을 쌓을 것이다.

- userprog/process.c의 process\_execute()에서 파일 이름을 추출하고 파일이 있는 지 확인하고 없으면 -1을 리턴한다. 만약에 존재하는 파일이라면 파일 이름으로 스레드를 만든다.

* **User Memory Access** (userprog/exception.c,userprog/syscall.c)

- userprog/exception.c의 page\_fault()에서 user memory가 아니라면 exit(-1)를 호출하여 유저 메모리가 커널 메모리에 접근 하는 것을 방지한다.

- user memory access를 확인하기 위하여, userprog/syscall.c에서 is\_user\_vaddr함수를 통해 user memory인지 체크한다.

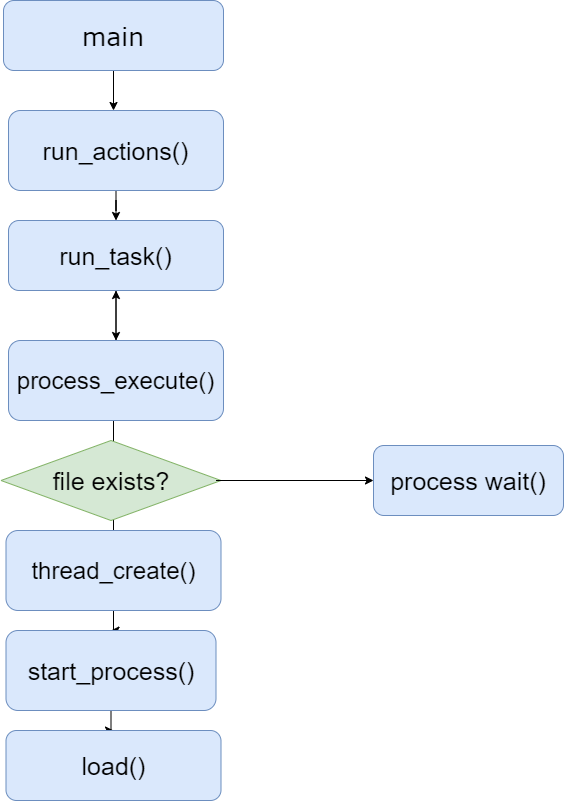
- **System calls**

- exit()과 wait()을 구현하기 위해 threads/thread.h에서 thread 구조체 내에 변수를 추가함

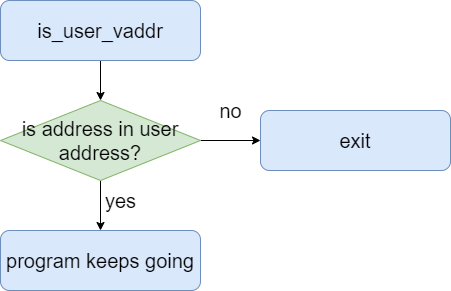
- system call을 구현하기 위해 userprog/syscall.c에서 syscall\_handler()에 switch 문을 추가하여 각자 case에 맞는 함수를 매칭한다.

* 1. **Flow Chart**

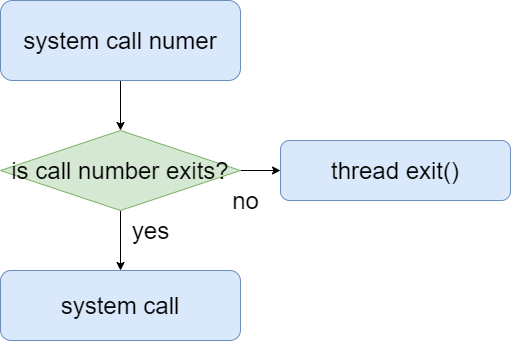
1. Argument Passing



1. User Memory Access



1. System Calls



* 1. **제작 내용**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

process\_execute()에서 명령어를 파일이름을 넘겨주고, 그 이름을 가진 thread를 생성한다. 그리고 load()에서는 명령어의 주소와, 해당 명령어들의 주소를 저장하는 배열을 만들들었다. 그리고  esp 포인터를 반복문을 돌며 마지막 명령어부터 첫번째 명령어 만큼 (해당 명령어의 길이 +1(null 문자)) 만큼 각각 주소를 내리고, 그 주소에 명령어를 저장하는 것을 반복하였다. 또한 이 반복문이 실행될 때 명령어가 저장된 위치를 저장해준다.반복문이 끝나면 현재 위치에서 가장 가까운 esp 이하의 4의 배수로 esp 를 이동 시켜준다음, null pointer를 하나 삽입한 이후에, 다시 반복문을 돌려서 앞의 반복문에서 저장했던 명령어의 주소와 string을 모두 stack에 쌓은 후 주소, argument 개수, return address를 주소값을 순서대로 저장하였다. 이렇게 해서 완성된 구조의 예시는 다음과 같다.

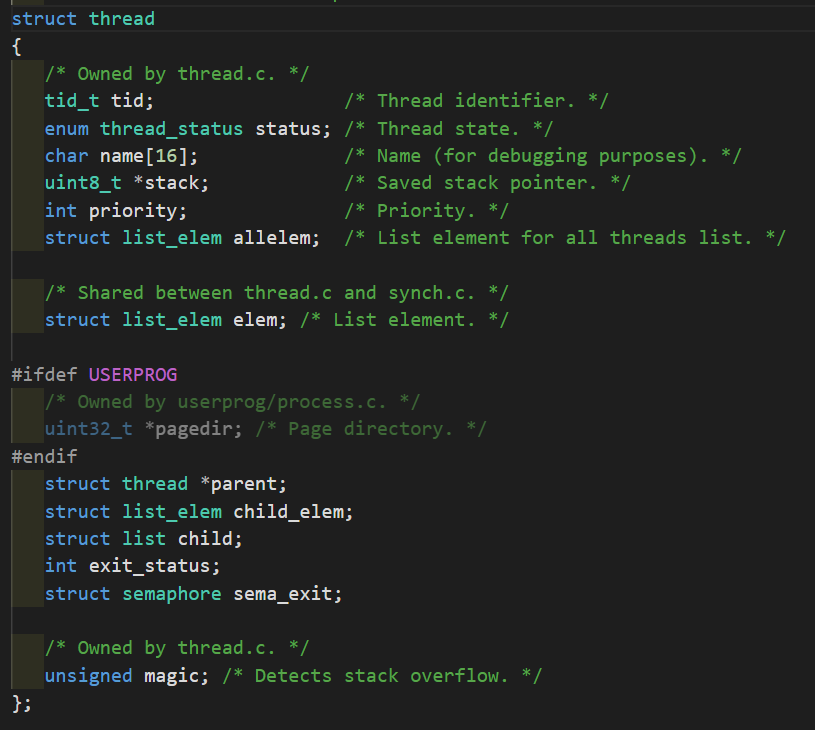


1. User Memory Access

userprog/exception.c의 page\_fault()에서 user memory가 아니라면 exit(-1)를 호출하여 유저 메모리가 커널 메모리에 접근 하는 것을 방지한다.

1. System Calls
2. Thread

-이번 프로젝트에서 핵심이 바로 쓰레드를 통해 처음 작업에서 다른 작업으로 넘어가는 것을 구현하는 것이다. 다른 작업이 진행되는 동안 부모 쓰레드가 종료 되면 안되므로 semaphore를 통해 관리한다. 필요한 자료들을 추가해서 쓰레드를 다음과 같이 설정하였다.



1. Halt()

shutdown\_power\_off()를 실행하여 핀토스를 종료하는 함수이다.

1. exit()

현재 프로세스를 종료하고 부모 프로세스(커널도 될 수 있음)으로 넘겨주는 시스템 콜이다. 여기서는 부모에게 자신의 종료 상태만 넘겨주고 thread\_exit()을 통해서 프로세스를 종료한다. thread\_exit()에서는 없어지는 프로세스가 자신의 자식 목록을 삭제하고 sema\_up을 통해 exit 세마포어를 해제해서 부모 process가 다시 작동 할 수 있도록 한다. 이후에는 schedule 함수 내부에서 남겨진 자식 스레드 처리가 시작된다.

1. wait()

process\_wait 을 통해서 대기하며 자식 프로세스가 실행될 때 까지 기다린다.

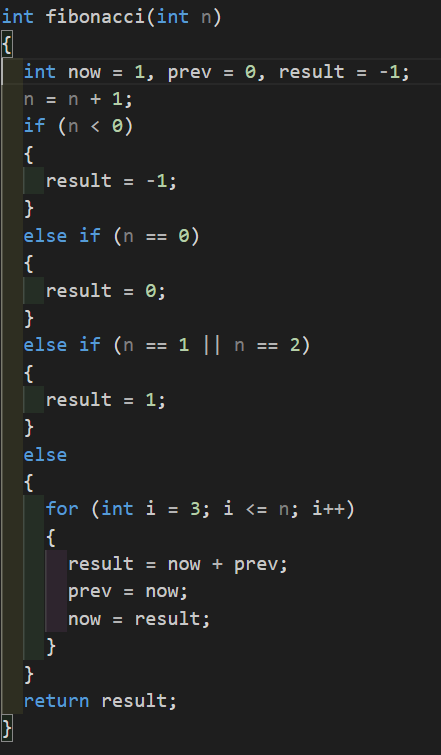
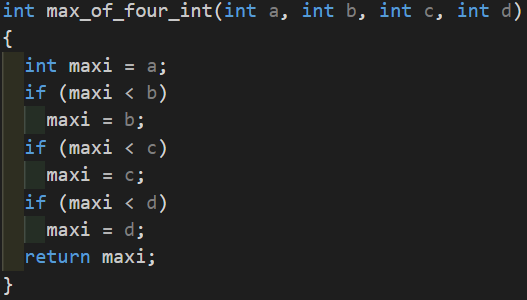
1. exec()

process\_execute를 통해 받아온 함수를 실행하여 자식 함수의 pid를 리턴한다

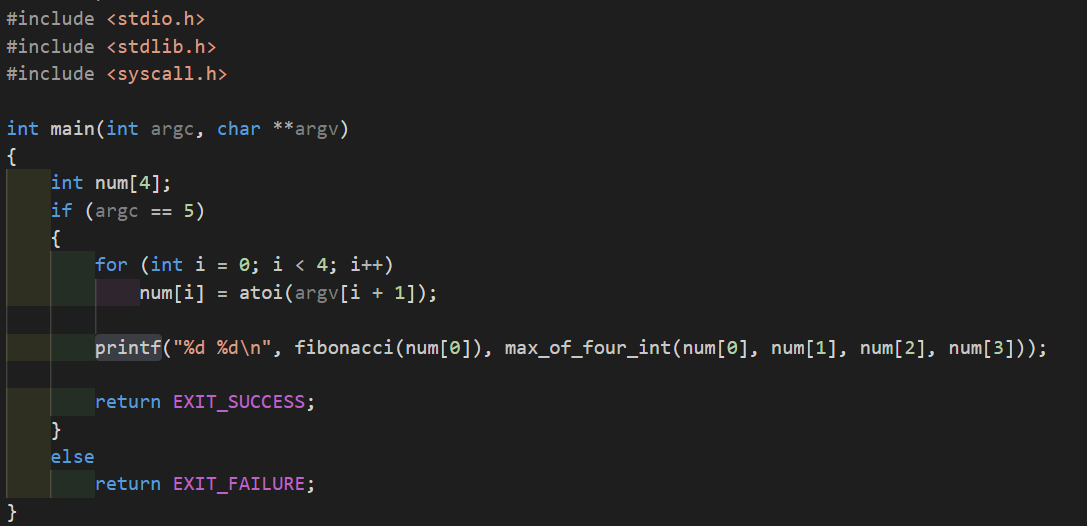
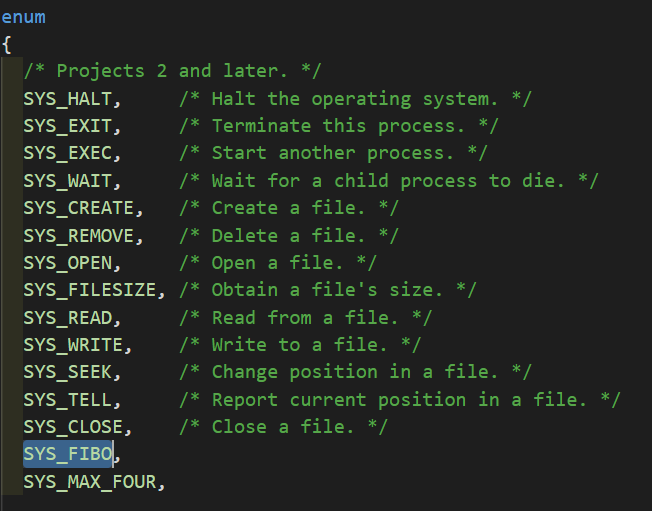
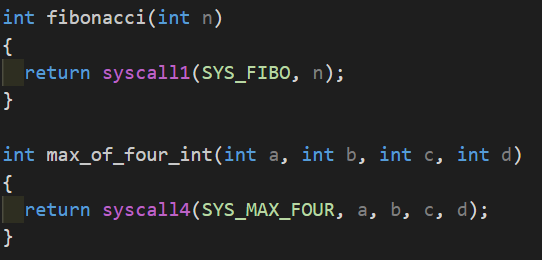
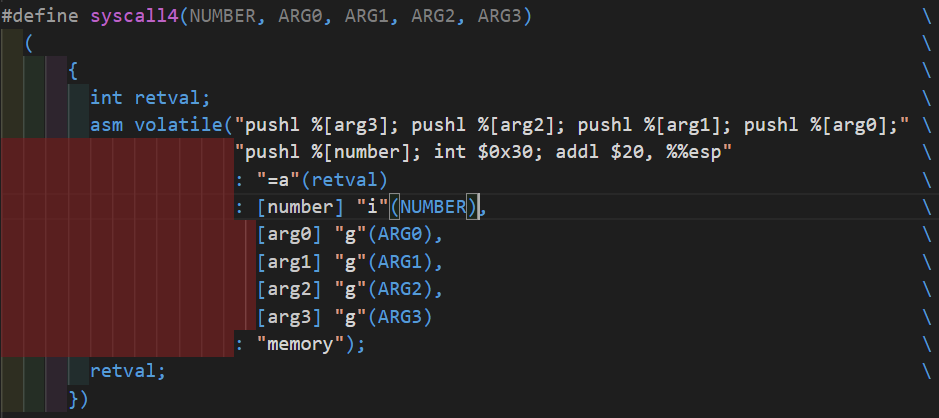
1. read(), write()

-해당 파일 지시자에 쓰거나 읽는다.

4. Additional System calls

1. Userprog/syscall.c  
    

syscall.c에서 additional system call의 동작을 정의한다. Fibonacci()는 n이 들어왔을 때 피보나치 수열 n번째 값을 반환하고 max\_of\_four\_int()는 입력으로 들어온 네 개의 숫자중 최대 값을 반환한다. Syscall.h에는 함수의 프로토타입을 작성하였다.

1. Examples/additional.c  
     
   additional 명령어의 결과를 출력하는 additional.c 함수를 examples 폴더에 생성한 후 make 해주었다.
2. Lib/syscall-nr.h  
     
   새로운 system call인 fibonacci와 max\_of\_four\_int에 상응하는 system call number를 등록해준다 (Fibonacci : 13, max\_of\_four\_int : 14)
3. Lib/user/syscall.c  
    

argument의 개수가 4개인 경우가 정의되지 않았으므로, 정의해주고 syscall.c과 syscall.h에 Fibonacci()와 max\_of\_four\_int()를 저장한다.

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

