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

담당 교수 : 박성용 교수님

조 / 조원 : 20171662 이나연

개발 기간 :10/18 ~ 10/31

1. **개발 목표**

user program이 실행되기까지의 일련의 과정을 이해하여 pintos 상에서 해당 user program을 올바르게 실행시키는 것이 이번 프로젝트의 목표이다.

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

원래 구현되어 있는 형식은 들어온 filename, 즉 command와 argument가 합쳐진 문자열이 바로 실행할 파일 이름으로 설정되어 있었다. 따라서 command 부분과 argument 부분을 분리하여 command의 이름으로 된 실행할 파일이 있는지 확인하는 코드를 구현하였다.

in <userprog/process.c>

makeup\_stack(): 들어온 command와 argument들을 핀토스 매뉴얼 에서와 같은 방식으로 stack에 쌓았다.

load(): 위에서 언급한 대로 command 실행 파일을 open한다. 문제 없이 open이 완료되었다면 filename을 위와 command와 argument 부분으로 분리하여 위 makeup\_stack()함수를 호출하여 스택에 쌓았다.

1. User Memory Access

가장 중요한 점은 user memory space와 kernel memory space가 달랐다는 점이다. 따라서 user program이 kernel space를 침범하는 경우 프로그램을 종료 시켰다. check\_userr\_vaddr()를 새로 구현하였다.

in <userprog/syscall.c>

syscall\_handler(): 각 시스템 콜 함수를 호출하기 전, 먼저 command와 argument들이 저장되어 있는 stack 내의 argument 주소들이 user space 내에 있는지를 다 확인하여 준다.

in <userprog/exception.c>

page\_fault(): user변수나 접근한 주소가 kernel 주소라면 status -1을 넘겨주면서 exit()을 하게 하였다.

1. System Calls

user program에서 사용하는 system call함수들이 정상적으로 동작할 수 있게끔 하였다.

in <userprog/syscall.c>

syscall\_handler(): 위에서 메모리 영역이 정상적으로 user memory space안에 있는 것을 확인하였다면, system call type에 해당하는 system call 함수를 불러오는 함수를 호출하였다.

exit(), wait() 등 system call 이름으로 된 함수들: system call 함수를 실행할 수 있는 조건이 있다면 확인해보고 system call 함수들을 call하여 해당 함수의 역할을 다 수행할 수 있게끔 한다.

fibonacci(), max\_of\_four\_int(): 추가적인 함수가 해당 역할을 수행하게끔 구현하였다.

in <lib/user/syscall.c>

syscall4(): argument가 4개인 system call을 받기 위한 함수

fibonacci(), max\_of\_four\_int(): uerprog에서 작성한 함수가 잘 call될 수 있도록 한다.

in <lib/syscall-nr.h>

각 system call함수에 대한 syscall number를 설정하는 곳. Fibonacci()와 max\_of\_four\_int()를 위한 syscall number도 지정해 주게끔 하였다.

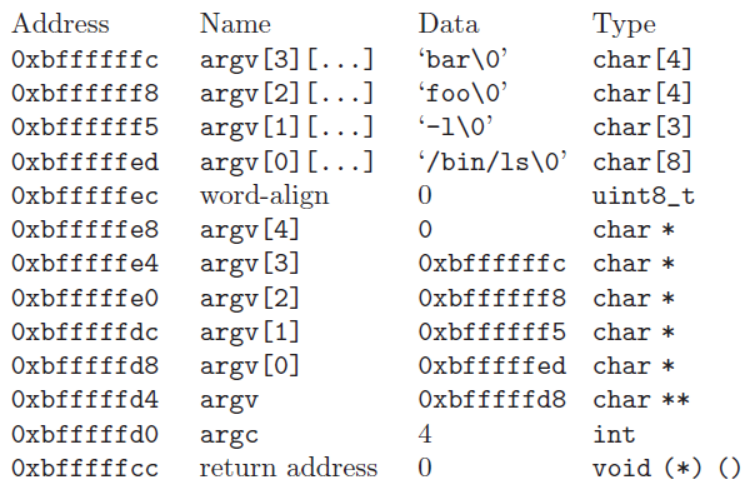
in <threads/thread.h>

struct thread: thread struct를 수정하여 필요한 추가적인 변수들 (exit\_status, semaphore 등) 역시 추가로 정보를 저장할 수 있게끔 하였다. userprog내에서 필요하기 때문에 #ifdef USERPROG 내에서 추가적인 변수를 저장하였다.

in <threads/thread.c>

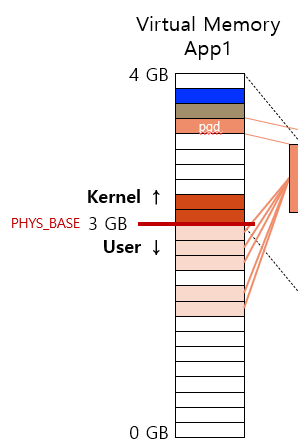
init\_thread(): 부모 쓰레드는 자식 쓰레드의 존재, status 등을 알고 있어야 하기 때문에 부모와 자식을 연결해주는 내용을 구현하였다.

* 1. **개발 내용**
* Argument Passing



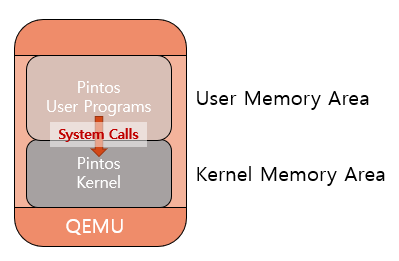
위와 같이 80x86 calling convention을 바탕으로 한 stack을 구현한다. esp의 시작점부터 위와 같은 방법으로 esp의 값을 감소해가며 적절한 메모리 공간에 적절한 data를 push 하였다. 프로그램에 입력된 명령어를 command 부분과 argument 부분들로 나누어 저장하게끔 하였다. 이는 strtok\_r()을 사용하였다. 다 push 한 후, word-align을 해주었다. 그런 다음 0을 push한 후 각 argument의 주소값을 push한 다음argument count도 구하여 push한 후 return address를 저장하였다.

* User Memory Access

pintos상에서 invalid memory access는 user program이 kernel memory space를 침범하는 경우 또는 vice versa의 경우가 있다. 이러한 현상을 막기 위해서는 kernel이나 user가 각각 PHYS\_BASE를 지나지 않는지 계속 확인해주어야 한다. user program내에서는 PHYS\_BASE와 0xc00000000사이의 주소를 가지는지 확인해야 하는데, 이는 is\_user\_vaddr()로 확인할 수 있다. 반대의 경우는 is\_kernel\_vaddr()을 활용할 수 있다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

system call은 바로 kernel에 접근하여 원하는 일을 수행할 수 없는 user program이 kernel에 접근할 수 있는 역할을 한다. 보통 high-level 언어들은 시스템 운영에 핵심적인 부분을 차지하는 커널에 바로 접근이 불가능하기 때문에(위험함) system call을 통해 시스템에 접근할 수 있게 된다.



* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명

- wait(): child process가 끝날 때까지 parent가 terminate하지 않게 하기 위한 system call이다. child process의 exit status도 받아오는데, 만약 exit()을 통한 정상적인 중단이 아닌 kill()당했다면, -1을 return한다. parent가 wait()하기도 전에 child가 exit()을 하는 것은 허용되지만, 이럴 때에 parent가 그래도 정상적으로 exit status를 잘 받아오게끔 해야 한다.

- halt(): pintos를 shutdown\_power\_off()를 통해 중단한다.

- exit(): 현재의 user process를 중단할 때 호출되며, kernel로 exit과 동시에 exit status를 return한다. 해당 exit status는 parent의 wait()에서 retrieve된다. 보통 0은 success, 0이 아닌 값은 nonsuccess로 간주된다.

- exec(): command로 들어온 명령어를 함께 들어온 argument들과 함께 수행한 후, new process의 pid를 return한다. 들어온 명령어가 수행 가능한 명령어인지 process\_execute에서 확인하는데, 들어온 명령어의 tokenization도 여기 내부에서 수행한다. 만약 프로그램 파일이 실행되지 못하면 -1을 return한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소

1. open()을 사용하여 system call을 call한다.

2. syscall1/syscall2/syscall3/syscall4 중 하나가 호출 되며 (argument count에 따라 달라짐) system call number와 argument가 stack에 쌓이게 된다. 그러면서 ‘int $0x30’으로 interrupt를 발생시킨다.

3. interrupt를 위한 stack 역시 쌓은 후 interrupt handler를 호출한다.

4. interrupt handler는 system call handler를 호출한다.

5. system call handler에선 system call number에 따라 address가 valid한지 확인한 후 system call 함수를 호출한다. argument는 stack에서 esp의 주솟값을 기준으로 각각 access한다. 해당 system call 함수가 return 값이 있다면 eax에 집어넣는다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

- 10/18~10/19: Pintos에 대한 전반적인 이해 및 system call의 실행 순서 코드 내에서 확인하기. argument passing 구현

- 10/20~10/25: system call handler 구현. user memory access 조건 확인해주는 부분 구현. exit, halt와 같은 기본적인 함수 및 additional 함수 구현

- 10/26~10/30: system call 정상적으로 수행되게끔 구현. exec, wait system call 구현.

- 10/31: 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.c>

makeup\_stack(): 새로 만든 함수이다. 들어온 command와 argument들을 핀토스 매뉴얼에서와 같은 방식으로 stack에 쌓기 위해 input으로 들어온 명령어를 strtok\_r()을 통하여 각 argument를 분리하여 저장한 다음, esp의 값을 변경해가며 stack의 각 부분에 알맞은 data를 push할 것이다.

80x86 calling convention을 바탕으로 한 stack을 구현한다. esp의 시작점부터 위와 같은 방법으로 esp의 값을 감소해가며 적절한 메모리 공간에 적절한 data를 push 하였다. 프로그램에 입력된 명령어를 command 부분과 argument 부분들로 나누어 저장하게끔 하였다. 이는 strtok\_r()을 사용하였다. 다 push 한 후, word-align을 해주었다. 그런 다음 0을 push한 후 각 argument의 주소값을 push한 다음argument count도 구하여 push한 후 return address를 저장하였다.

load(): makeup\_stack()함수를 이 함수 내에서 호출한다. 또한 해당 함수를 호출하기 전에도 먼저 file\_name을 parse하여 program name을 구한 후 해당 프로그램 파일을 open하게끔 수정한다.

process\_execute(): 역시 input에서 program name만 parse한 후 해당 이름으로 된 thread를 만들게끔 수정한다. 그 이전에 해당 program name으로 된 파일이 열리는지부터 확인하여 에러가 나면 -1을 리턴하게끔 한다.

2. User Memory Access

<userprog/exception.c>

page\_fault(): user 변수가 잘못되었거나 kernel virtual address를 침범한 경우 exit(-1)을 하게끔 한다.

<userprog/syscall.c>

check\_user\_vaddr(): argument의 주솟값마다 해당 주솟값이 kernel virtual address를 침범하였는지 확인한다. 침범하였을 경우 exit(-1);을 하게끔 한다.

syscall\_handler():

3. System Calls

<threads/thread.h>

struct thread: 부모 thread는 자식 thread에 대한 정보를 알고 있어야 하기 때문에 이와 관련된 멤버들을 추가적으로 구현해준다. critical section problem 해결을 위한 semaphore, child의 list, child의 exit status 등의 정보를 담을 수 있는 멤버들을 추가한다.

<threads/thread.c>

init\_thread(): user program내에서 child와 관련된 멤버들을 초기화해준다. semaphore들은 false로, child process에 대한 list를 초기화한 후 현재 프로세스의 child를 list에 추가한다.

<userprog/syscall.c>

syscall\_handler(): system call을 구현하기 switch 문을 추가하여 system call number에 해당하는 system call function을 호출하도록 한다. (system call function : halt(), exit(), exec(), wait(), write(), read(), fibonacci(), max\_of\_four\_int())

<lib/syscall-nr.h>

새로운 additional function들에 대한 system call number를 부여하도록 syscall number enum 내에 SYS\_FIB, SYS\_MAX도 추가한다.

<examples/additional.c>

들어온 argument 4개 중 첫 숫자를 n번째 피보나치를 구할 argument로 하여 Fibonacci()의 결과를 출력하고, 4개의 숫자 중 제일 큰 숫자를 출력하도록 max\_of\_four\_int()도 호출하여 결과값을 출력한다.

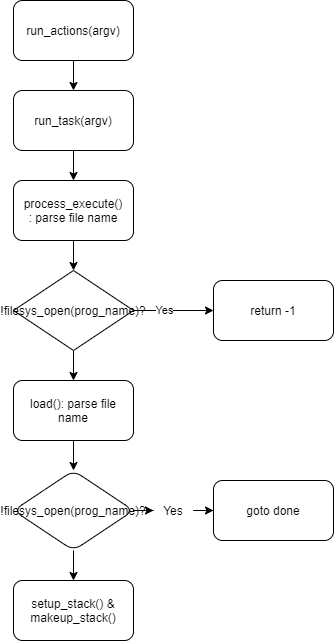
<lib/user/syscall.c>

argument가 4개인 system call을 처리하기 위하여 syscall4()라는 macro를 구현한다.

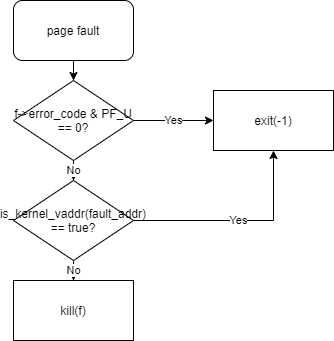
1. **연구 결과**
   1. **Flow Chart**

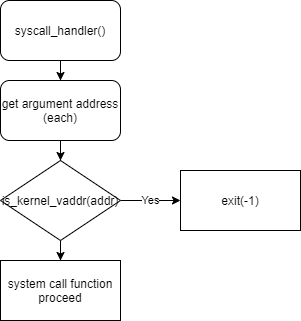
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

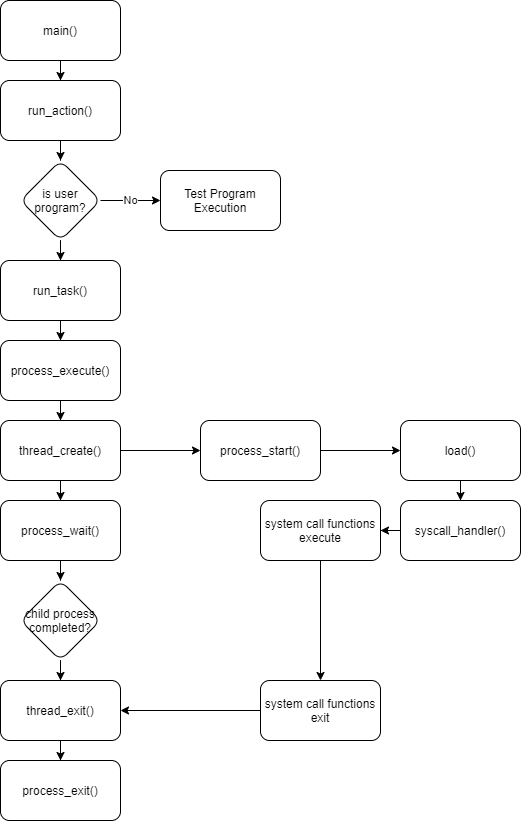


1. User Memory Access

<userprog/exception.c>

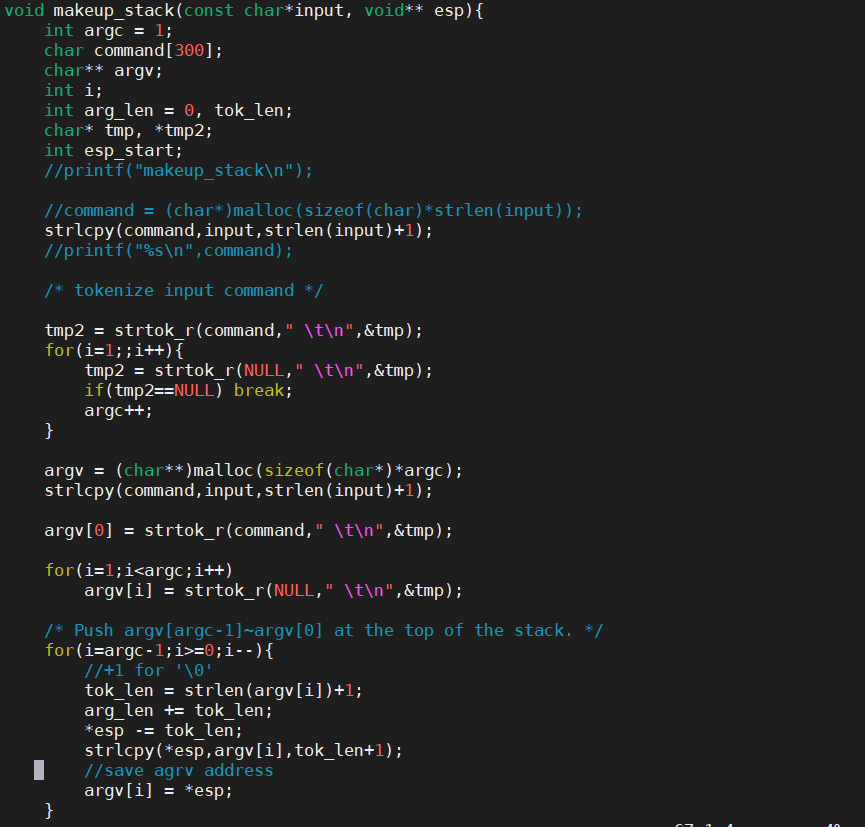
<userprog/syscall.c>

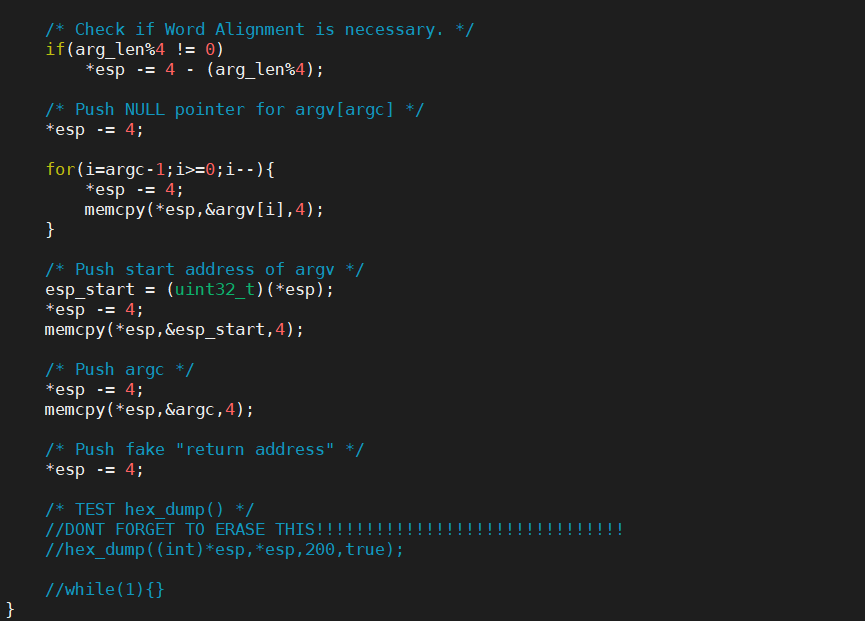
1. System Calls



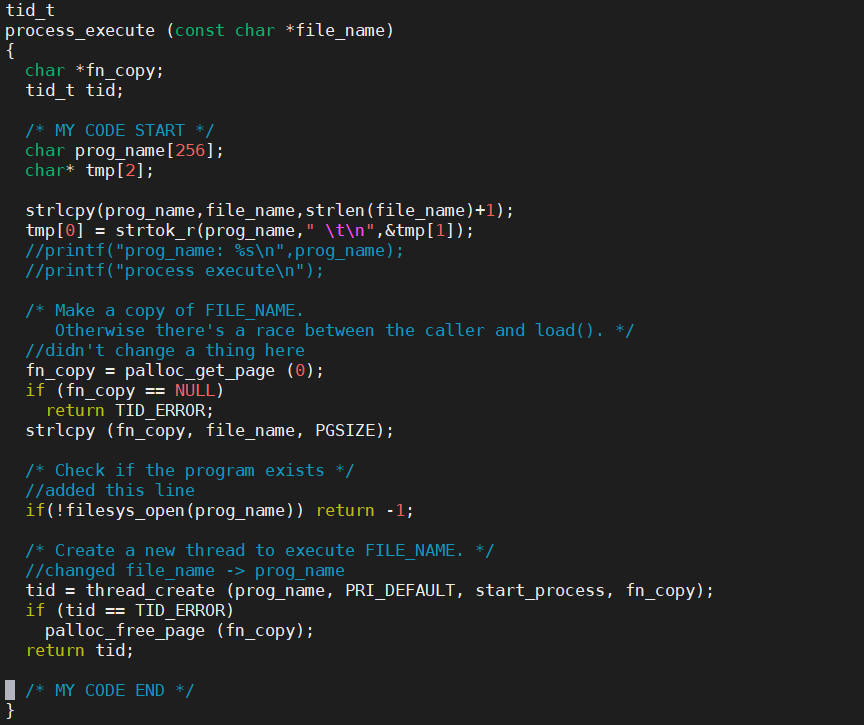
* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

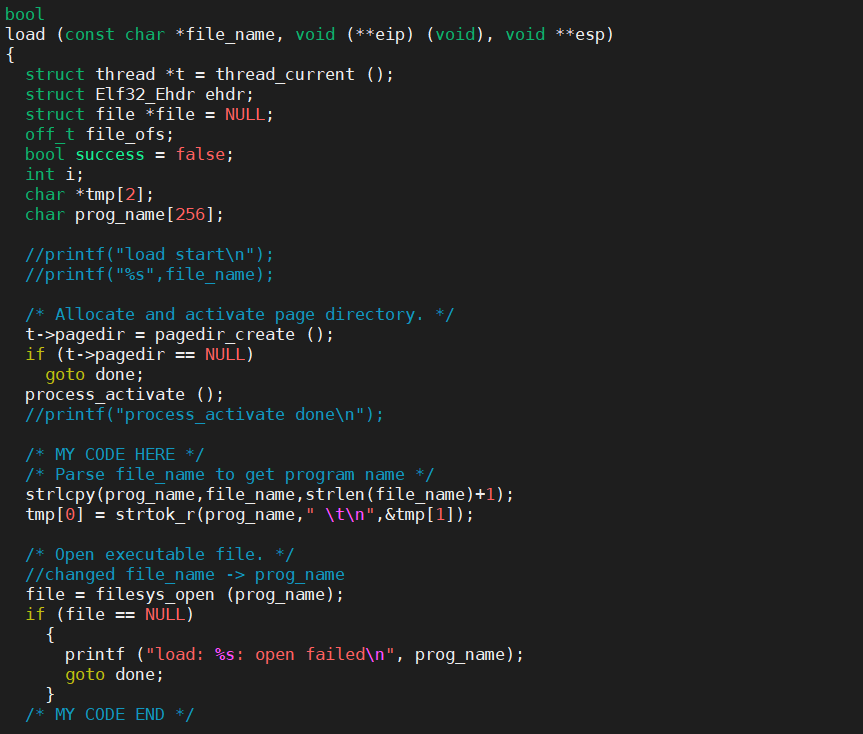




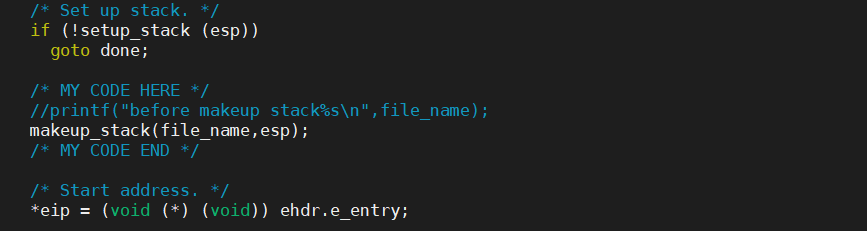
위에서 언급한대로 stack에 data를 집어넣는 과정이다. command로 주어진 input을 parsing하여 program name과 argument 부분을 strtok\_r를 사용하여 나누어 각각 argv[i]에 저장하였다. 우선 argument count를 먼저 확인하기 위해 token의 수를 먼저 센 다음 tokenization을 실행하였다. 그런 다음 esp의 값을 감소시키며 알맞은 data를 알맞은 memory space에 push하였다. 중간에 word alignment도 해주었다.



process\_execute에서도 알맞은 program name의 이름으로 된 file을 열 수 있게끔 program name parsing을 해주었다. 마찬가지로 strtok\_r을 사용하였다. filesys\_open(prog\_name)이 false면, 즉 해당 파일이 정상적으로 열리지 않았다면 -1을 반환하게끔 했다. thread\_create도 prog\_name을 사용하였다.

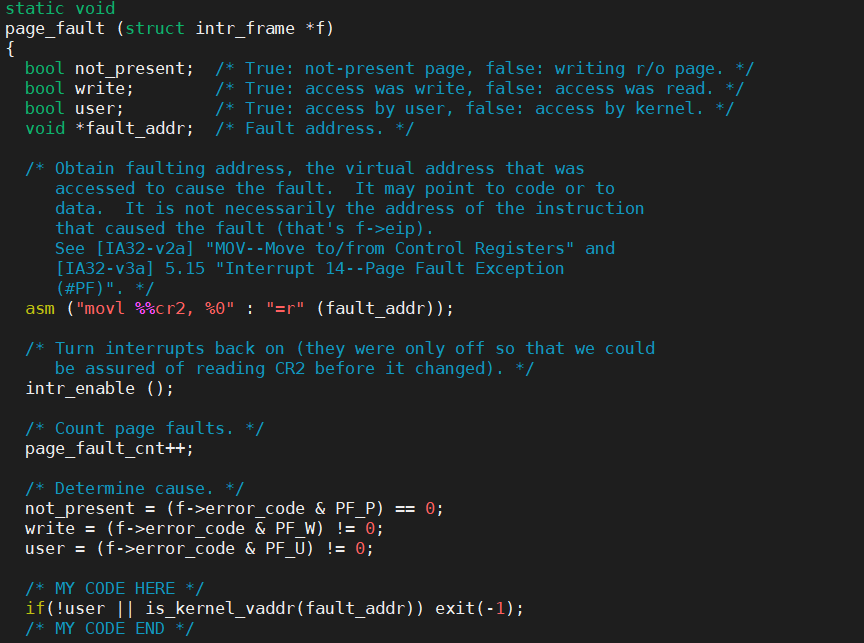


load()에서도 parsing을 진행해주었다. 해당 프로그램명으로 된 파일이 open되게끔 하였다.

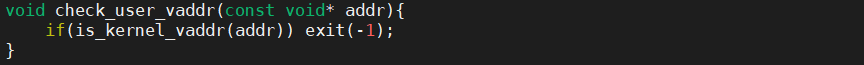


stack을 먼저 만들어 준 후 성공했다면 makeup\_stack()을 통하여 stack에 argument 관련 data를 모두 집어넣었다.

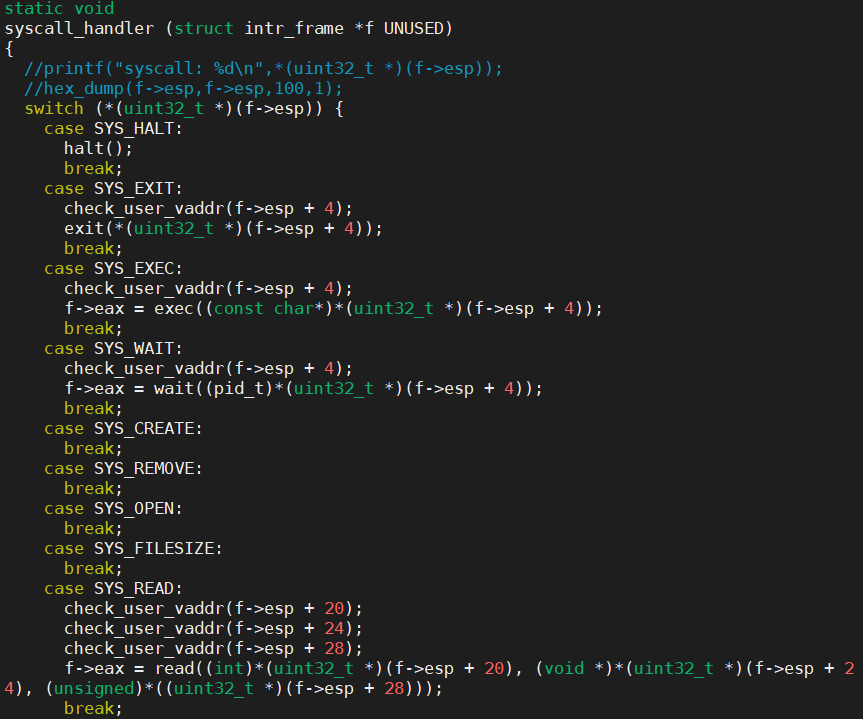
1. User Memory Access

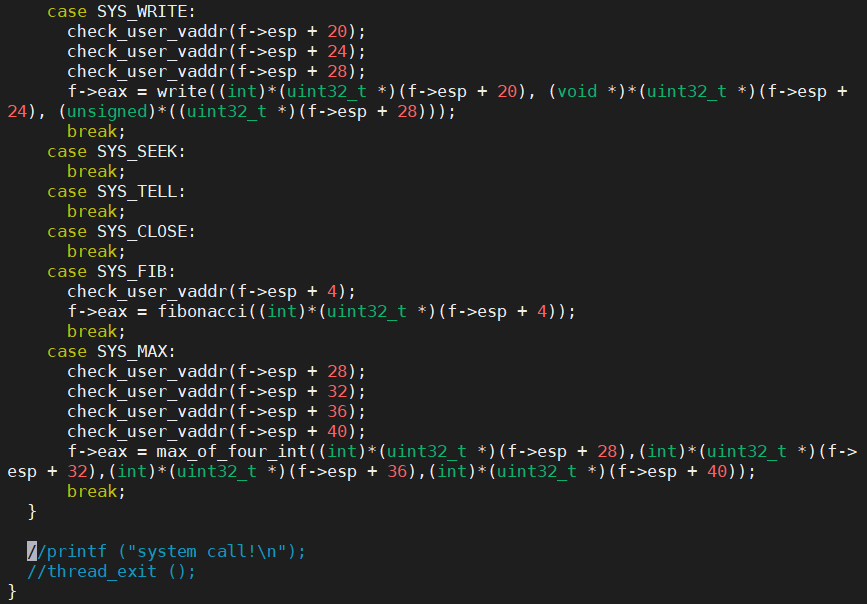


page fault가 invalid address access에 의해서 생겼을 경우에 대한 코드를 구현하였다. user변수는 access by kernel인 경우에 false를 담고 있으므로, 해당 경우에 exit(-1)을 해주어야 했다. 또한 fault\_addr 주소값을 봤을 때 kernel에 해당하는 memory space에 있는 것을 확인할 경우에도 exit(-1)을 해주어야 했다.



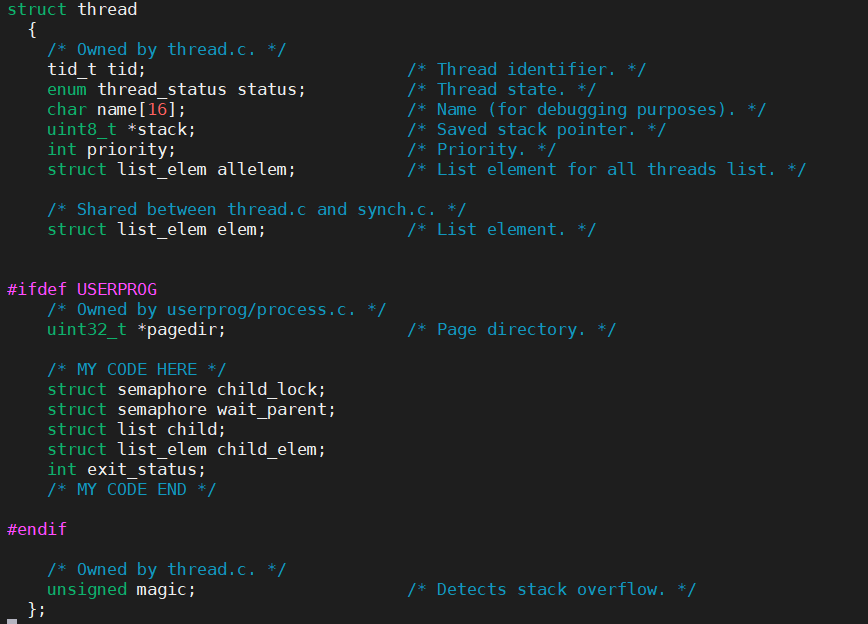
syscalll.c에서 check\_user\_vaddr()함수를 구현하여 만약 argument가 들어있는 주소값이 kernel 공간에 들어있다면 바로 exit(-1)을 해주는 기능을 넣었다.



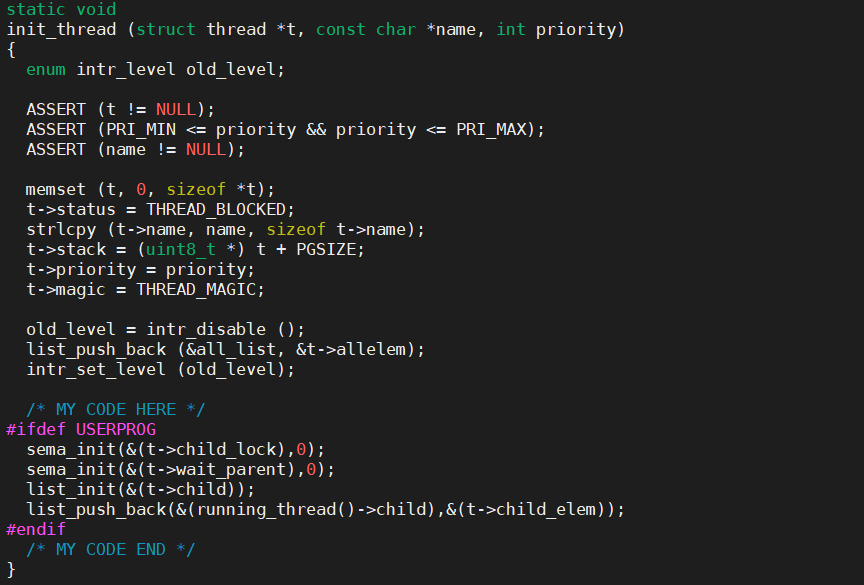


syscall\_handler()에서 모든 system call을 호출하기 전에, 해당 system call이 사용할 argument의 모든 주소값들을 다 check하여 kernel address 공간에 있다면 exit(-1)을 하게 하기 위해 각 주소값을 저장하고 있는 곳마다 check\_user\_vaddr()를 실행해주었다.

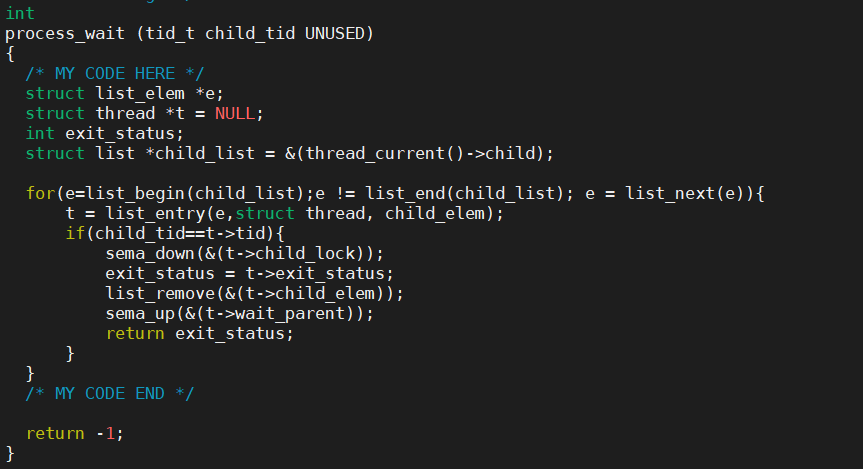
1. System Calls



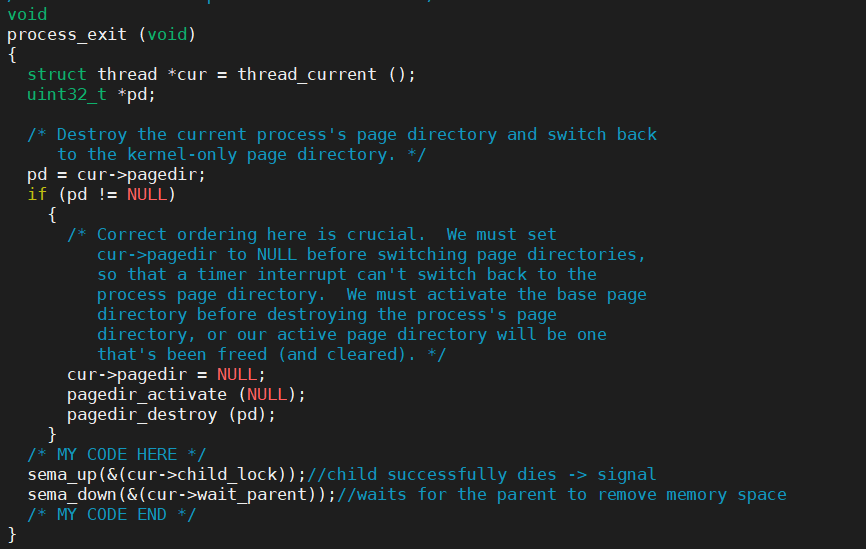
parent thread는 child process의 존재 및 접근이 가능해야 한다. 원래 thread 구조체에는 child의 정보를 저장할 수 있는 멤버가 구현이 되어있지 않았으므로 추가적으로 구현해주었다. user program내에서 사용되는 멤버들이기 때문에 #ifdef USERPROG #endif사이에 구현해주었다. child가 오기 전에 parent는 wait을 해야 하므로 기다리게끔 하는 semaphore child\_lock을 생성하였고, parent가 list\_remove()를 수행하기 전에 child가 죽게 되면 제대로 메모리가 할당 해제가 되지 않을 수 있기 때문에 parent가 list\_remove()를 다 수행할 때까지 child가 완전히 terminate되지 않게끔 wait\_parent semaphore를 생성하였다. 그런 다음 child의 list를 저장할 수 있도록 child라는 list 멤버를, list\_elem 변수 (list 구조체 사용을 위한) child\_elem을 생성하였다.



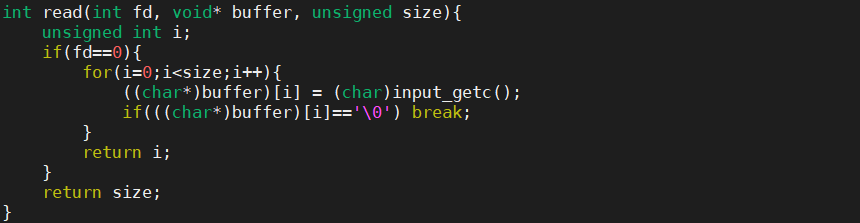
init\_thread()에서도 역시 user program 내에서만 사용하면 되니 #ifdef 내에 구현하였다. 모든 semaphore를 false로 초기화하였고, child list를 생성한 뒤 child노드를 push하였다.



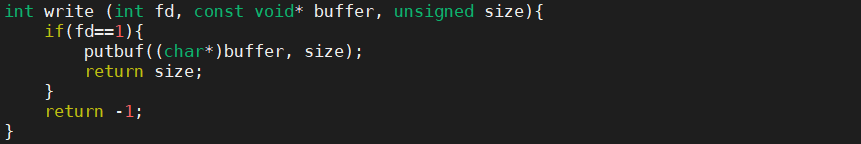
process wait()에선 지금 기다리고 있는 child를 현재 thread의 child list에서 찾아 wait()기능을 수행한다. parent는 child가 끝나기를 기다렸다가 child의 exit status를 retrieve한 다음, child가 사용하고 있던 resource를 반납해야 한다. 이게 정상적으로 작동하려면 우선 child가 실제로 수행을 끝내기 전에는 기다리기를 해야 한다. 따라서 child가 수행을 끝냈다는 child\_lock semaphore가 true로 되기 전까지 sema\_down(&(t->child\_lock))로 기다려야 한다. signal이 오면 해당 semaphore 조건에서 벗어나 exit\_status를 저장할 수 있게 된다. 그런 다음 list\_remove()를 수행하여 child의 resource를 정상적으로 반납한다. child가 수행을 끝냈을 때 list\_remove()전에 사라지는 것을 방지하기 위해 wait()을 걸어놨던 wait\_parent semaphore를 list\_remove() 수행 후 다시 sema\_up해준다. 그런 다음 exit\_status를 반환한다. child tid가 존재하는 list에 없다면 -1을 반환한다.



해당 쓰레드가 수행을 끝낼 때, parent에 정상적으로 exit status를 돌려주기 위해선 semaphore로 critical section을 잘 조절해주어야 한다. 우선 child\_lock을 sema\_up해주며 parent에 수행이 끝났다는 것을 알린다. 그런 다음 wait\_parent semaphore가 up되길 기다리며 sema\_down에 있는다. 만약 parent가 list\_remove()를 정상적으로 종료했다면 wait\_parent에 대해서 sema\_up을 할 것이기 때문에 그제서야 해당 쓰레드는 수행을 정상적으로 마칠 수 있을 것이다.



read는 우선 stdout이 돌아가게끔 작동하게 간단히 구현하였다. input\_getc()를 size만큼 계속 받으며 만약 size에 도달하기 전에 NULL을 받았다면 그대로 저장을 중지한 후, 해당 buffer의 size만큼을 반환한다.



write도 stdin이 돌아가게끔 구현하였다. putbuf()를 사용하여 size만큼 buffer에서 fd에 저장하게 하였다.



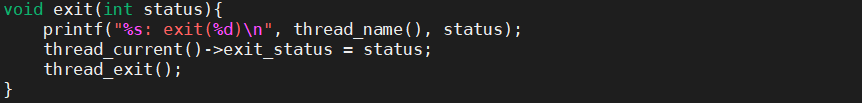
wait() system call은 위에서 설명한 process\_wait()을 호출하여 parent가 child를 정상적으로 기다리고 status도 받아올 수 있게끔 한다.



역시 위에서 언급한 process\_execute()를 호출하여 입력받은 command를 argumnet를 사용하여 수행할 수 있게끔 한다.



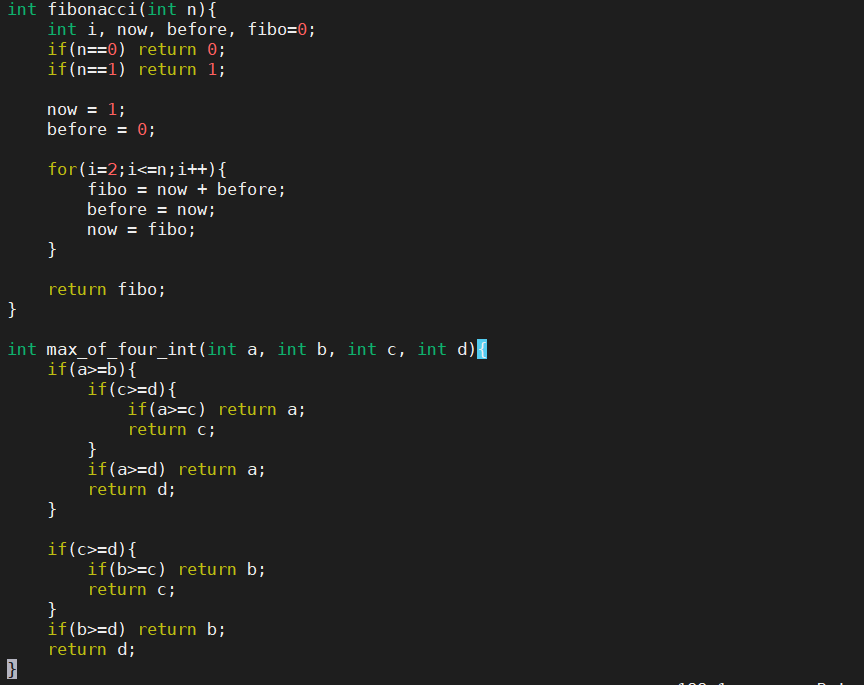
shutdown\_power\_off()를 호출함으로써 핀토스를 종료시킨다.



exit() system call은 우선 어느 thread에서 어떤 status를 반환하며 종료되었는지를 print하게끔 하였고, 현재 thread의 exit\_status에 전달받은 status를 저장하였다. 그런 다음 thread\_exit()을 호출하여 해당 thread가 정상적으로 exit하게끔 하였다. thread\_exit() 다음에 process\_exit()이 수행되며 위에서 언급하였던 child의 termination이 정상적으로 진행될 것이다.

2. user memory access에서 보여주었던 syscall\_handler 코드를 보면, 파라미터로 전달받은 interrupt frame에서의 esp를 가지고 system call의 argument에 접근한다. 해당 system call의 argument 개수에 따라 argument가 저장되어 있는 곳이 esp로부터 달라지기 때문에 argument 개수에 따라 esp에서 얼만큼 떨어져있는 argument에 접근해야하는지를 다 따로 넣어주었다. argument 자체끼리는 4만큼의 주소값만큼 떨어져있기 때문에 각 argument에는 4 단위로 접근하였다. 어떤 system call인지는 system call number를 통하여 switch()문을 사용하였다.

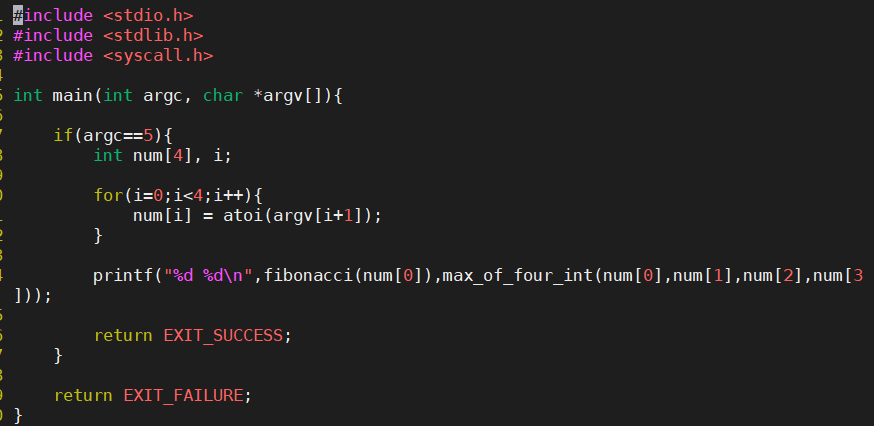
1. Additional System calls



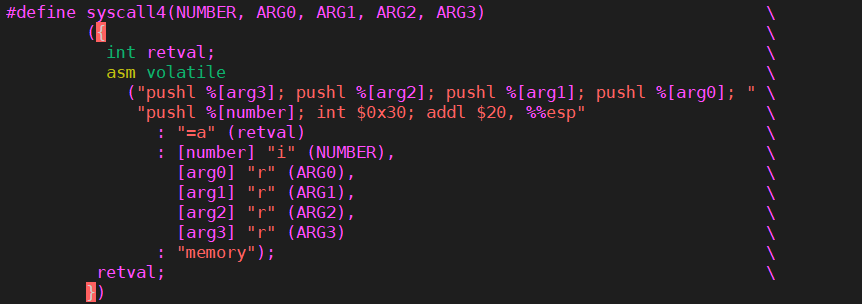
userprog/syscall.c에서 Fibonacci()함수와 max\_of\_four\_int()를 구현하였다. fibonacci는 주어진 숫자를 인덱스로 갖는 피보나치 수열 값을 반환하고, max\_of\_four\_int는 4개의 숫자 중 제일 큰 값을 반환한다.



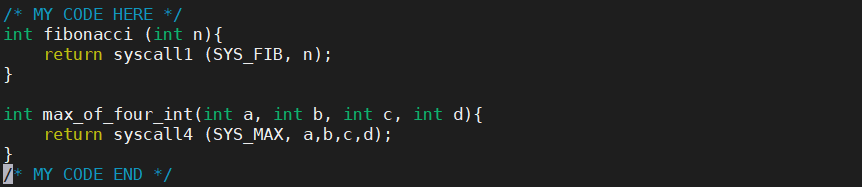
lib/syscall-nr.h에 SYS\_FIB, SYS\_MAX의 system call number를 추가로 구현하였다. 이는 위에서 syscall\_handler()에서 어떤 systemcall을 호출해야 하는지를 구분하기 위한 인자로 쓰인다.



examples/additional.c에서 두 함수를 수행한다. 피보나치 수열은 첫 argument번째 피보나치 수열 숫자를, max\_of\_four\_int는 네 argument 중 제일 큰 숫자를 print하게끔 하였다. 다 성공적으로 수행하였다면 EXIT\_SUCCESS를, 그렇지 않다면 EXIT\_FAILURE를 반환하게끔 하였다.



lib/user/syscall.c에선 우선 argument가 4개인 함수도 받을 수 있도록 syscall4()라는 subroutine을 구현하였다. argument들이 stack에 push될 수 있게 된다.



또한 위와 같이 syscall1, syscall4 subroutine을 통하여 Fibonacci, max\_of\_four\_int() system call 함수를 수행할 수 있도록 system call number와 인자를 전달한다.

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

