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

담당 교수 : 박성용

조 / 조원 : 20161643 정진원

개발 기간 : 10/3~11/1

1. **개발 목표**

System call과 user stack을 구현하여, user program을 수행 가능하게 한다. 구현한 system call은 halt, exit, exec, wait, read, write, fibonacci, max\_of\_four\_int이다. 또한 커널과 유저간의 data 및 control 교환을 위해 argument passing, esp를 구축하였다. 구현한 환경을 통하여 thread들을 생성하여 user program을 수행하였다.

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

Argument를 파싱하고, 스택에 삽입한다. 커맨드 라인의 첫번째 토큰을 함수의 첫 인자로 전달하여 쓰레드를 생성한다. 파싱된 명령어에 해당하는 실행 파일이 로드 된다.

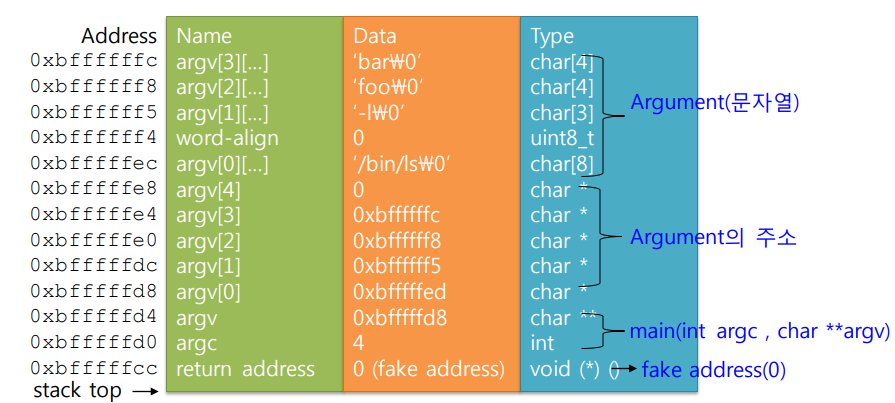
1. User Memory Access

유저 영역을 벗어난 주소를 참조할 경우 page fault를 발생시킨다. 이를 방지하기 위해 메모리 영역을 접근 할 때 메모리 주소를 확인해준다.

1. System Calls

halt, exit, exec, wait, read, write, Fibonacci, max\_of\_four\_int과 같은 system call들에 적절한 기능을 부여해준다. 항목을 구현 완료하였을 때, 프로그램이 제대로 수행된다.

* 1. **개발 내용**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명



핀토스에 명령어를 수행 시, 커맨드라인을 파싱하여 esp 스택에 프로그램의 이름과 인자를 저장한다. 인자는 오른쪽부터 왼쪽의 순서로 스택에 저장된다. 예시로 “/bin/ls -l foo bar”라는 명령어를 입력하였을 때 생성되는 스택의 구조는 아래 그림과 같다. User 영역의 PHY\_BASE(0xc0000000)부터 시작하여, 파싱된 명령어와 인자들이 저장되어 있는 것을 알 수 있다.

명령어와 인자들은 word align되어 stack에 저장되며, \*argv배열에는 명령어와 인자들이 stack에 저장된 주소 값을 저장하여 stack에 쌓아준다. \*\*argv에는 \*argv배열이 스택에 저장된 주소가 저장된다. argc에는 인자의 개수가 저장되어 stack에 저장된다. 스택의 맨 위에는 return address가 저장된다.

|  |
| --- |
| User stack |
| Bar |
| foo |
| -l |
| bin/ls |
| \*argv[4] |
| \*argv[3] |
| \*argv[2] |
| \*argv[1] |
| \*argv[0] |
| \*\*argv |
| argc |
| Return Address |

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명

핀토스에서 user program이 kernel의 메모리 영역에 접근하는 것을 invalid memory access라고 한다. 이 때, pagefault를 발생시켜준다.

* + Invalid memory access를 어떻게 막을 것인지 설명

Check 함수에서 threads/vaddr.h에 있는 is\_user\_vaddr() 함수를 이용하여 memory access가 일어 날 때 user 영역에 있는 메모리인지 확인해주었다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

시스템 콜은 운영체제가 제공하는 서비스에 대한 프로그래밍 인터페이스로, 사용자 모드 프로그램이 커널 기능을 사용하게 해주려면 필요하다.

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

Halt(): pintos를 종료시키는 시스템 콜이다. Shutdown\_power\_off() 함수를 이용한다.

Exit(): 현재 수행하고 있는 쓰레드를 종료시키는 system call이다. 종료시에 프로세스의 이름과 종료 상태를 출력해준다. 종료상태는 thread\_current()->exit\_status에 저장되어 있으며 정상적으로 종료 될 때는 exit\_status가 0이다.

Exec(): process\_execute 함수를 이용해 자식 프로세스를 생성하고, 프로그램을 실행시키는 시스템 콜이다. 프로세스 생성에 성공한 경우 생성된 프로세스의 pid 값을 반환한다. 현재 스택 f의 eax 자리에 새로 생성한 process의 pid를 저장하여 생성된 process로 context switching 시켜준다. 세마포어를 이용하여 부모 프로세스는 자식 프로세스가 메모리에 적재될 때까지 대기한다.

Wait(): 자식 프로세스가 모두 종료 될 때까지 대기시켜주는 역할을 하는 시스템콜이다. 현재 쓰레드에서 child\_list 내의 자식 thread를 찾아 child\_tid가 thr->tid와 같은 경우 해당 쓰레드를 child\_list에서 제거한다. 이때 semaphore 2개를 이용하여 child process가 제거되는 동안 child process의 메모리를 유지시킨다.

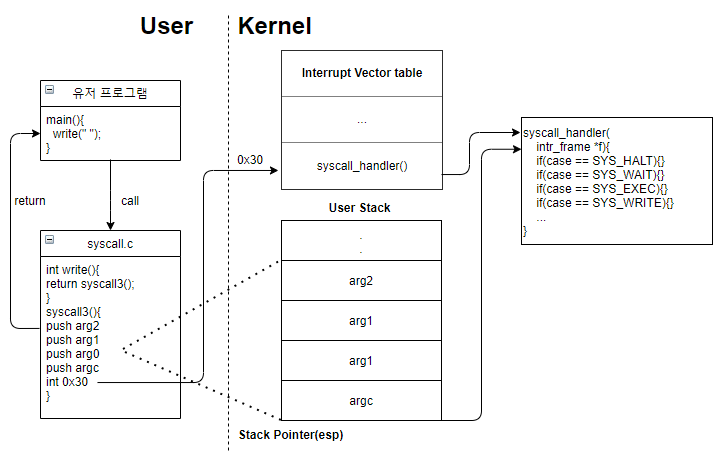
Read(): stdin/stdout을 통한 입출력만 구현하면 되기 때문에 fd == 0인 경우 input\_getc() 함수를 통해 입력을 받는다.

Write(): stdin/stdout을 통한 입출력만 구현하면 되기 때문에 fd == 1일때만 putbuf() 함수를 통해 출력을 해준다.

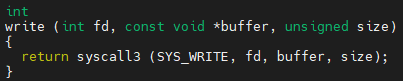
Fibonacci(): 인자로 전달받은 숫자 n에 대해 n번째 피보나치 수를 구해주는 프로그램을 실행시켜주는 system call.

Max\_of\_four\_int(): 인자로 전달받은 숫자 네 개중 가장 큰 숫자를 구해주는 프로그램을 실행시켜주는 system call.

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



유저레벨에서 시스템 콜 API를 호출 한 이후 다시 유저 레벨로 돌아 올 때까지의 요소를 그림으로 나타내면 위와 같다. 예를 들어 유저 프로그램에서 write라는 시스템콜을 하면 pintos/src/lib/user/syscall.c를 통해 stack에 argument들이 push 되고 system call 이 invoke되어 kernel 영역으로 전달된다.



Write의 경우 세개의 인자가 있어 sycall3가 사용되고, 전달된 system call은 syscall\_handler에 의해 번호에 맞는 기능을 수행한다. 시스템 콜 함수의 리턴 값은 인터럽트 프레임의 eax에 저장되어 다시 유저 레벨로 돌아오게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/3 ~ 10/5 프로그램 요구사항 파악

10/10 ~ 10/14 argument passing, user memory access에 관한 부분을 구현하였다.

10/28 ~ 11/1 system call handler, system call implementation, additional implementation 구현을 하였다.

* 1. **개발 방법**
     1. Argument Passing

Src/userprog/process.c의 process\_execute()함수와 start\_process에 command line을 파싱하는 코드를 추가하였다. 또한 user stack을 구성하기 위해 build\_esp함수를 추가해주었다. Build\_esp함수는 esp에 전달받은 인자들을 저장해주는 역할을 한다.

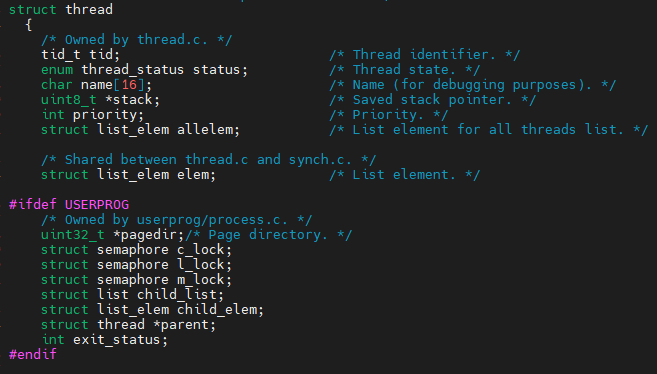
* + 1. User Memory Access

Src/usserprog/syscall.c에서 메모리 영역에 접근 할 때, user memory address의 범위 안에 있는 접근을 하는지 확인해주는 check함수를 구현하였다.

Src/userprog/exception.c에서 user를 이용해서 커널 영역을 참조 할 때 exit을 해주는 코드를 추가한다.

* + 1. System call handler & system call implementation

구현해야하는 system call halt, exit, exec, wait, read, write의 system call number에 맞는 기능을 수행 할 수 있도록 src/userprog/syscall.c에 if문과 함수들을 구현해주었다. 또한 parent process와 child process간의 프로세스 계층 구조 구현을 위해 src/threads/thread.h의 thread 구조체에 필요한 필드들을 추가하였고, semaphore를 사용하였다.



* + 1. Additional implementation

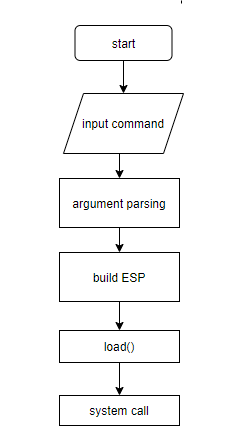
Syscall.nr-h 파일에 fibonacci와 max\_of\_four\_int에 해당하는 system call 번호를 부여했다. 또한 src/lib/user/syscall.c와 src/lib/user/syscall.h에 fibonacci와 max\_of\_four\_int에 필효한 syscall4를 구현하고 함수들을 추가해주었다. Src/userprog/syscall.c 와 src/userprog/syscall.h에도 fibonacci함수와

max\_of\_four\_int에 대한 구현을 하였고, system call에 알맞게 수행될 수 있도록 syscall\_handler에서 필요한 부분들을 코드로 작성하였다.

1. **연구 결과**
   1. **Flow Chart**

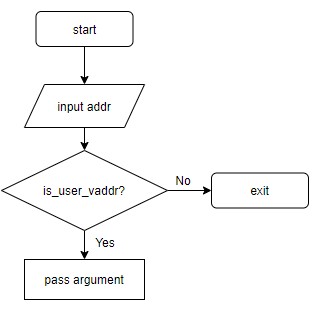
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

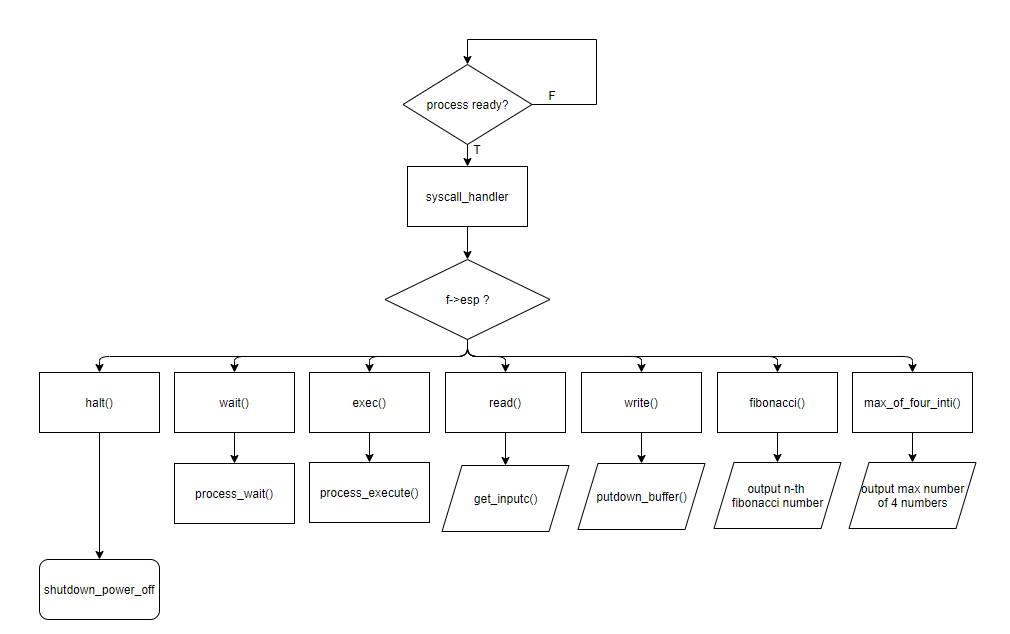


Change order

1. User Memory Access

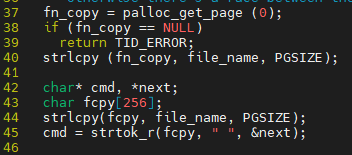


1. System Calls

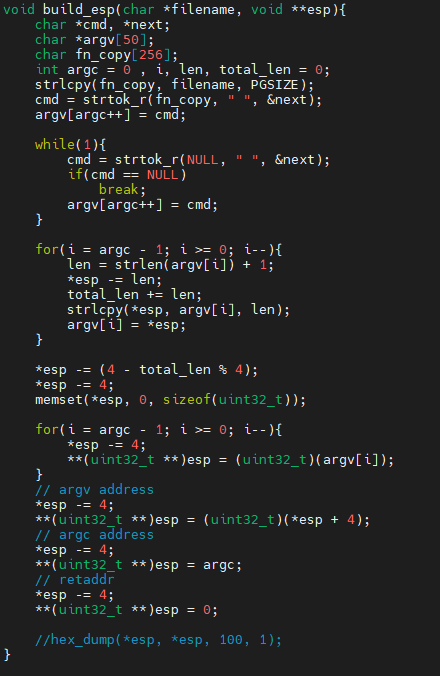


* 1. **제작 내용**

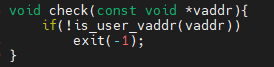
1. Argument Passing



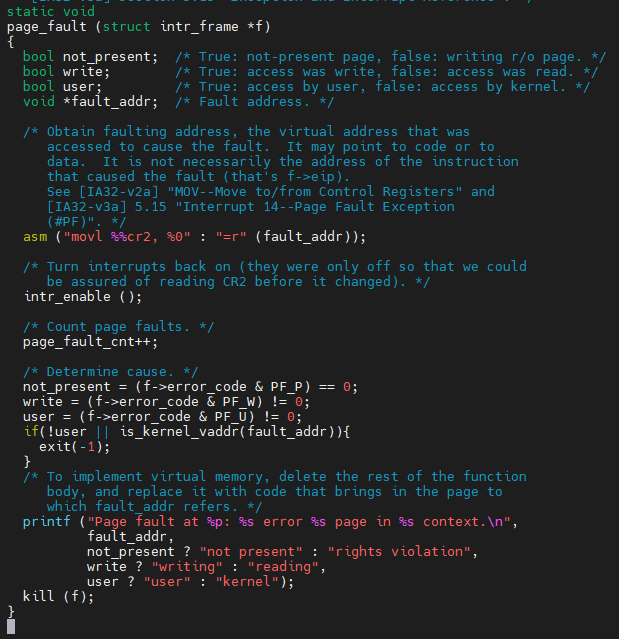
Argument passing을 위해서는 먼저 argument를 parsing 하는 과정이 필요한데, 이는 strtok\_r 함수를 이용하여 전체 명령어를 토큰화해주어 수행하였다. parsing하는 과정은 process\_execute 함수와 start\_process함수에 추가해주었다. 전체 명령어가 아닌 파싱된 명령어를 load해주어 제대로 load가 되게 한다. 이후 user stack에 토큰화된 argument를 저장하기 위해 아래와 같은 build\_esp 함수를 구현했다. PHY\_BASE로부터 downward로 argument들을 쌓아주었다. 이때, 스택에 저장된 값을 확인하기 위해 hexdump함수를 이용했다.



1. User Memory Access

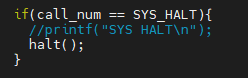


다음과 같은 check 함수를 구현하여 접근하려는 메모리 영역이 user 영역인지 확인해주었다. 이 함수는 syscall\_handler 함수에서 메모리에 접근 할 때마다 사용되었다. 이때 사용된 is\_user\_vaddr함수는 threads/vaddr.h에 구현되어 있는 함수로 esp의 값과 PHYS\_BASE의 값을 비교해준다. 또한 userprog/exceptions.c의 page\_fault함수에 유저가 커널 영역을 참조하면 exit(-1)을 해주는 코드를 추가해주었다.



1. System Calls

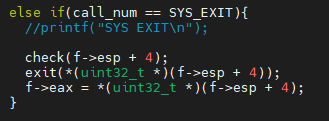
* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
* Halt(): pintos를 종료시키는 시스템 콜이다. f->esp에 저장된 번호가 sys\_halt에 해당하는 번호면 halt()함수를 불러주어 pintos를 종료시킨다.

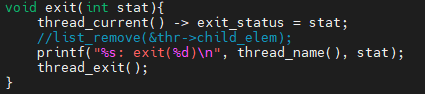




Pintos의 shutdown\_power\_off() 함수를 이용해주었다.

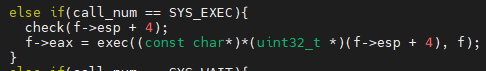
* Exit(): 현재 수행하고 있는 쓰레드를 종료시키는 system call이다.

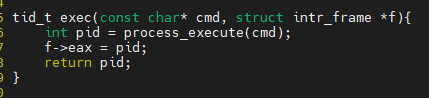




Syscall\_handler()에서 call\_num == SYS\_EXIT인 경우 exit함수를 호출해주어 프로세스 이름과 상태를 출력해주고 종료시킨다.

* Exec(): process\_execute 함수를 이용해 자식 프로세스를 생성하고, 프로그램을 실행시키는 시스템 콜이다.



Syscall\_handler()에서 call\_num == SYS\_EXEC인 경우 exec함수를 호출해주어 

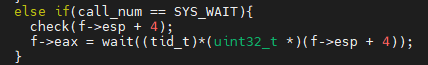
Child process의 pid를 반환해주고, context switching 시켜준다.



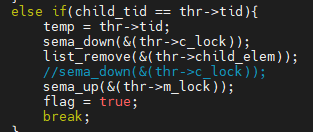
Process\_execute() 함수에서 semaphore l\_lock(load\_lock)을 down 시켜주고 start\_process에서 process가 load에 제대로 load되면 semaphore l\_lock을 up 시켜주어 load가 제대로 될 때까지 부모 프로세스가 기다리게 해준다.

* Wait(): 자식 프로세스가 모두 종료 될 때까지 대기시켜주는 역할을 하는 시스템콜이다.

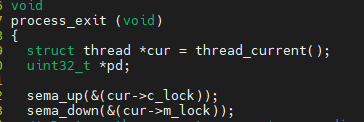




Sys\_handler에서 call\_num이 SYS\_WAIT인 경우 수행되며 process\_wait함수를 수행해준다. 현재 쓰레드에서 child\_list 내의 자식 thread를 찾아 제거한다.



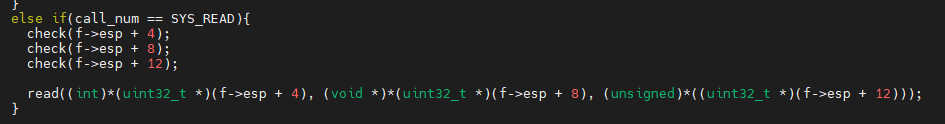
이때 semaphore 2개를 이용하여 child process가 제거되는 동안 child process의 메모리를 유지시킨다.



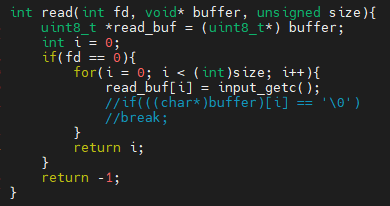
Child process가 삭제되는 경우 sema\_down(&(thr->c\_lock))을 통해 부모 프로세스가 기다리고 있다가 child가 종료되는 경우 process\_exit이 되므로 sema\_up(&(thr->c\_lock))을 통해 부모 프로세스가 수행 될 수 있도록 한다. m\_lock의 경우 memory lock의 약자로 자식 프로세스의 메모리를 부모 프로세스가 list\_remove를 수행할 때까지 남겨두는 역할을 하는 semaphore이다.

* Read(): stdin/stdout을 통한 입출력만 구현하면 되기 때문에 fd == 0인 경우

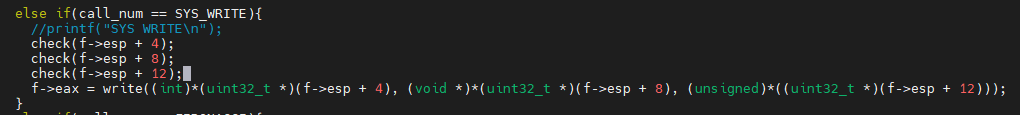
제공된 함수인 input\_getc() 함수를 통해 입력을 받는다.



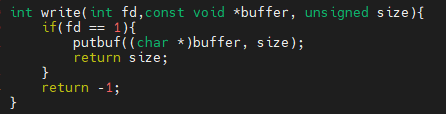
SYS\_READ에 해당하는 system call이 들어온 경우 전달받은 인자들이 user영역 안에 있는지 확인해주고, read함수로 인자들을 전달해준다.



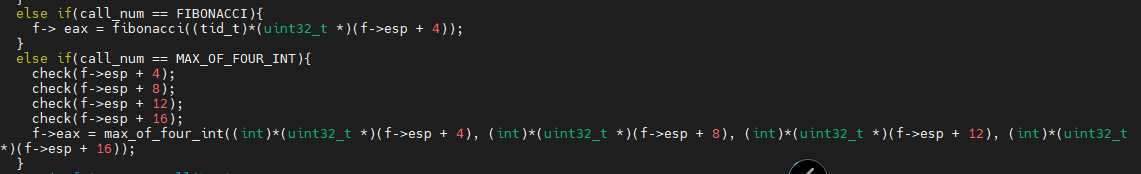
* Write(): stdin/stdout을 통한 입출력만 구현하면 되기 때문에 fd == 1일때만 putbuf() 함수를 통해 출력을 해준다.

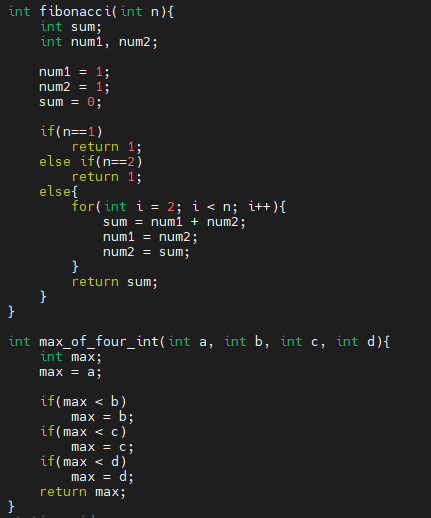


SYS\_WRITE에 해당하는 system call 이 들어오면 인자들의 범위를 체크 해준 뒤, write함수에 전달해준다.

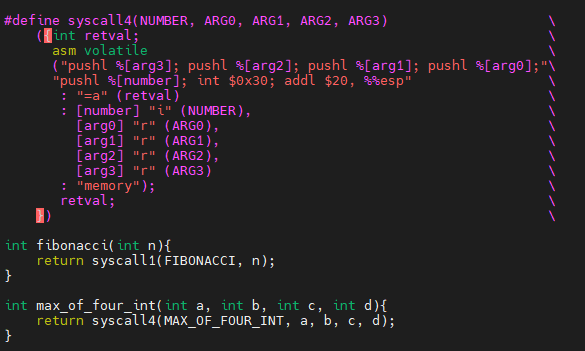


1. Additional System calls

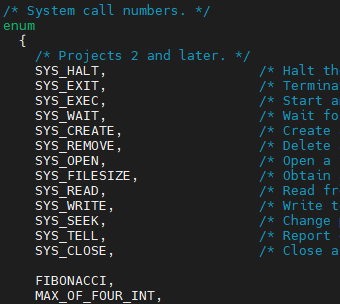




* Syscall\_handler에 Fibonacci 기능과 max\_of\_four\_int기능을 하는 함수들을 추가하였고, 해당하는 system call이 들어온 경우 인자들의 범위를 확인하고 함수들로 전달해주는 것을 구현하였다.



또한 src/lib/user/syscall.c에서 max\_of\_four\_int의 경우 인자가 추가적으로 하나가 더 필요하기 때문에 syscall4라는 매크로를 정의하였다. 어떤 syscall을 이용하는지에 대한 함수들도 추가해주었다.



또한 Lib/syscall-nr.h에 fibonacci와 max\_of\_four\_int에 해당하는 system call 번호를 부여하였다.

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

