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

담당 교수 : 박성용 교수님

조 / 조원 : 20171609 김민식

개발 기간 : 10/5 ~ 11/01

1. **개발 목표**

현재 주어진 pintos에서는 많은 OS 기능이 구현되어 있지 않고, 프로그램을 실행시킬 수 없다. 따라서 이번 project에서는 user program이 실행되기 까지의 고정을 이해하고, user program을 실행시킬 수 있는 환경을 구축하는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. **Argument Passing**

User로부터 입력 받은 명령어를 parsing 하여 프로그램이름, 인자를 구분하여 stack에 push한다. 그 후 프로그램이 정상적으로 이들을 넘겨받을 수 있게 하여 프로그램이 작동되도록 한다.

1. **User Memory Access**

먼저, User space와 Kernel Space를 구분하여 서로 침범이 없게 한다. 따라서 User program이 Kernel space를 침범하는 경우 프로그램을 종료 시켰다. 또한 Memory에 접근을 못하거나, 잘못된 접근을 한 경우도 kill했다.

1. **System Calls**

우리가 프로젝트에서 실행 할 system call은 halt, exit, exec, wait, read, write, fibonacci, max\_of\_four\_int 이다. User program에서 Kernel Space에 접근할 수 있도록 기본 구조와 함께, 위의 8가지 system call을 구현하였다.

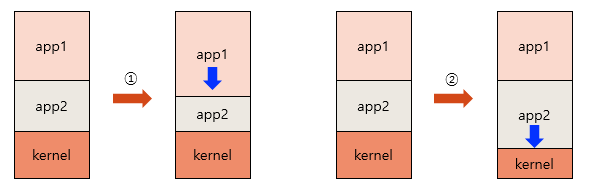
함수를 사용하여 System call number과 인자로 호출하였고, system call handler가 스택에 쌓인 system call number와 인자를 읽어 system call을 수행하도록 하였다.

* 1. **개발 내용**
* **Argument Passing**

먼저 80x86 calling convention을 참고하여 stack을 구현하였다. 사용자로부터 입력 받은 명령어를 이용하여 argument를 얻었고, esp 값을 감소해가며 stack 메모리 공간에 push하였다. 명령어는 command 부분과 argument부분으로 나누어 저장하였으며, push 한 후 word-align을, 그 다음은 0을 push하였다. 그 후 argument의 수와 return address또한 저장해주었다.

* **User Memory Access**

Pintos 상에서 invalid memory access는 아래 그림과 같이 User 프로그램이 Kernel Space를 침범하는 경우이다.



이를 막기 위해서는 두가지 방법이 있다. 먼저 User Program의 Pointer가 가리키는 주소가 PHYS\_BASE 아래에 있는지 확인하는 방법이다. 두번째로는 user provided pointer가 가리키는 주소가 user program의 영역인지 확인 하는 방법이다. 만약 확인해서 아니면 프로그램을 종료한다.

* **System Calls**

위에서 말했다시피 User Program은 Kernel Space에 접근할 수 없다. Kernel memory는 핵심 기능을 포함하고 있기 때문에 OS가 kernel memeory에 접근 하는 것을 막고 있기 때문이다. 하지만 user program이 kernel의 기능이 필요할 때, system call을 사용하면 kernel에 접근할 수 있게 된다.

이번 프로젝트에서 개발할 시스템 콜을 다음과 같다.

* + - **void syscall\_halt (void)** :

pintos를 shutdown\_power\_off( )를 통해 중단시키는 역할을 한다.

* + - **void syscall\_exit(int status)**

현재 user process를 중단할 때 사용한다. Exit status를 return 한다. Exit status는 parent 의 wait()에서 retrieve된다.

* + - **pid\_t syscall\_exec(const char \*cmd\_line)**

존재하지 않는 파일인 경우 예외처리를 해주는 함수이다.

* + - **int syscall\_wait(pid\_t pid)**

process\_wait child process가 종료할 때 까지 parent process가 terminate하지 않게 하기 위한 system call이다. 만약 child가 정상적 중단이 아닌 kill()당했다면 -1을 retur한다.

* + - **int syscall\_read(int fd, void \*buffer, unsigned size):**

file descriptor를 이용하여 input\_getc()함수를 통해 내용을 읽어 들인다.

* + - **int syscall\_write(int fd, void \*buffer, unsigned size):**

file descriptor이 1인 경우(STDOUT), putbuf() 함수를 이용하여 출력한다.

* + - **int fibonacci(int n)**

n번째 피보나치 number을 구해준다.

* + - **int max\_of\_four\_int**

네가지 숫자 중 가장 큰 값을 출력해주는 system call이다.

User Program에서 필요한 argument를 넘겨 함수를 호출한다. 이 함수는 system call number와 argument를 인자로 system call을 호출하게 된다.

이따 인자들은 stack에 쌓이게 되고, interrupt handler를 호출한다. Interrupt handler는 system handler를 호출한다. System call handler는 system call number에 따라 주소를 확인 한 후 system call 함수를 호출한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/5 ~ 10/9 : 매뉴얼 분석 및 코드 이해

10/10 ~ 10/18 : argument passing 및User memory access 구현

10/19 ~ 10/21 : System call handler 구현

10/22 ~ 10/29 : system call 함수 및 additional 함수 구현

10/29 ~ 11/1 : 보고서 작성

* 1. **개발 방법**

1. **Argument Passing**

80x86 calling convention을 바탕으로 한 stack을 구현한다. Strtok\_r()을 사용하여 명령어를 command와 argument 부분들로 나누어 저장하였다. 다 push하면 word align을 해주었고, 0을 push하고 각 argument의 주소값과 argument count를 순서대로 push하였다. 그 후 return address를 저장하였다.

Load(): load()함수 위에 명령어를 parsing해주는 코드를 추가한다. 또한 setup\_stack()이 끝난 후 kernel 내 스택에 argument를 쌓는 코드를 추가하였다.

Process\_execute() : 파일 이름을 받고, 그 파일을 열었을 때 NULL이면 return 처리 해주었다.

1. **User Memory Access**

Exception.c의 소스 코드 중 page\_fault()에서 만약 user memory인지 아닌지 확인하여 user memory가 아닐 경우 exit(-1)을 호출하여 kernel memory에 접근하는 것을 방지하고자 하였다.

Syscall.c의 user\_vaddr() 에서는 argument의 주소값마다 해당 주소값이 kernel virtual address를 접근하였는지 확인하였다. 만약 접근하였을 경우에는 exit(-1)을 호출하였다.

1. **System Calls**

Thread.h.에서 앞에서 설명한 것들을 구현하기 위해 thread구조체를 추가하였다. Child process의 list, exit status. Semaphore 등의 정보를 저장하였다.

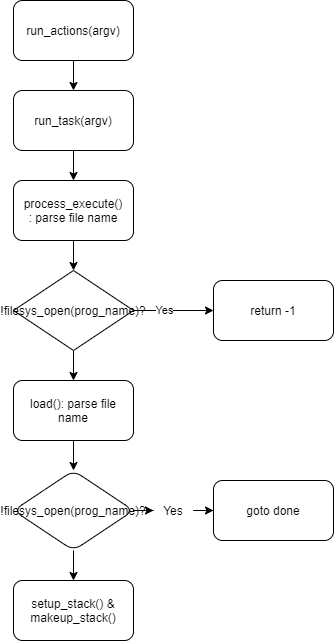
Thread.c의 init\_thread()에서는 process를 초기화할 때 앞의 struct thread와 함께 component들도 초기화 할 수 있도록 하였다. semaphore들은 false로 바꿔주었다.

Syscall.c의 syscall\_handler()는 switch문을 사용하여 system call number에 해당하는 함수들만 호출하도록 만들었다.

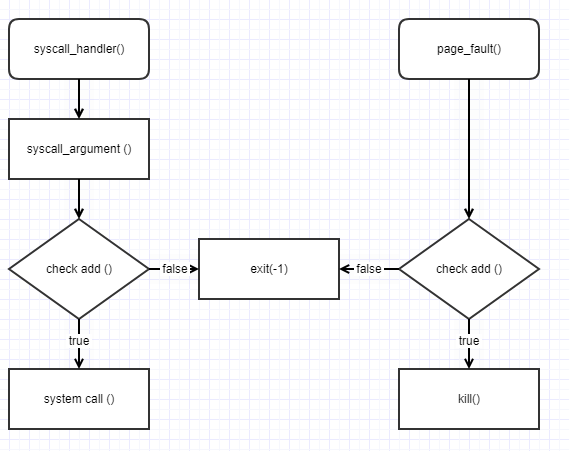
Lib/syscall-nr.h에서는 피보나치수열 함수 등 additional 함수들에 대한 system call number를 부여하도록 수정하였다.

Additional.c에서는 숫자 4개를 입력 받고, 첫번째 숫자를 n으로 하여 피보나치 수열의 결과를 출력하였다. 또한 max\_of\_four\_int()에서 입력받은 4개의 숫자 중 가장 큰 숫자를 출력하도록 하였다.

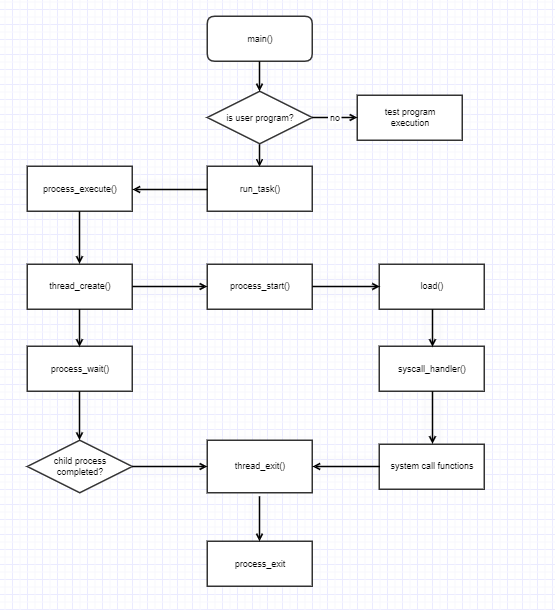
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



1. System Calls

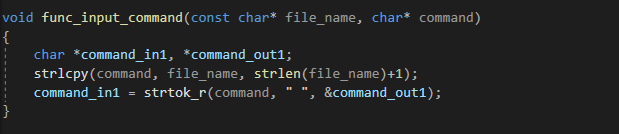


* 1. **제작 내용**

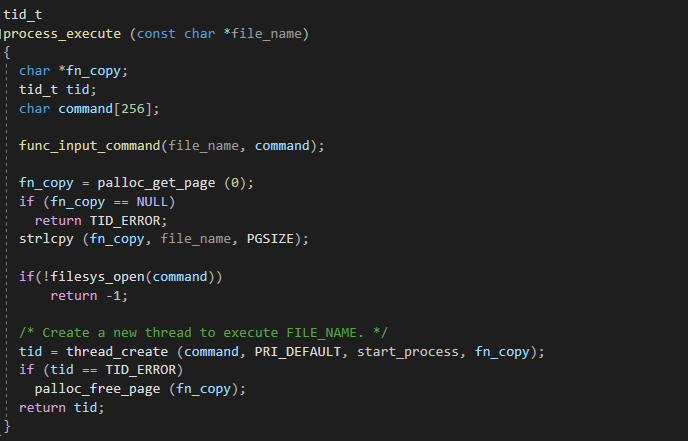
1. **Argument Passing**

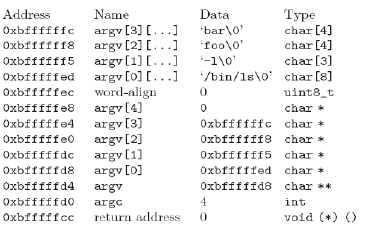
<userprog/process.c>

Func\_input\_command라는 함수를 만들었다. Strtok\_r()을 이용하여 공란을 기준으로 파일 이름을 나누어 저장하도록 하였다.



Process\_execute()에서 명령어를 파싱하여 파일이름을 넘겨주었다. 그리고 그 이름을 가진 thread를 생성하였다. 또한 process\_load()에서는 명령어의 주소와, string을 저장하는 구조체를 만들어서 명령어를 파싱하여 구조체의 string에 저장하였다.



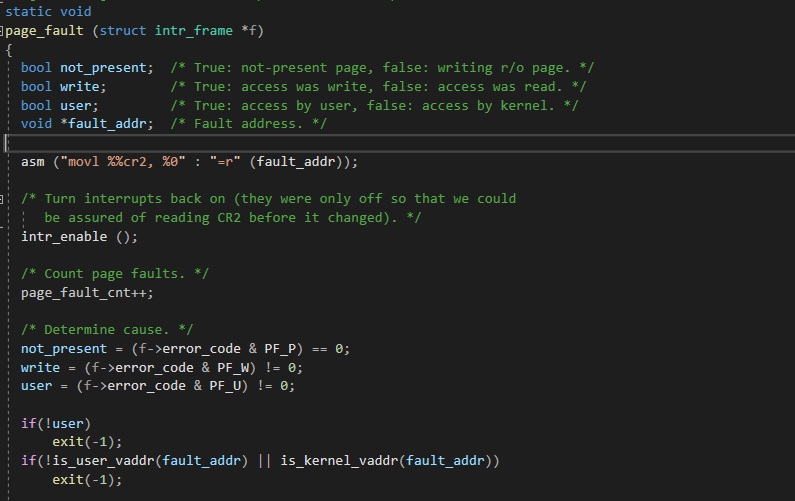
esp 포인터를 PHY\_BASE부터 시작해서 반복문을 통하여 마지막 명령어 인덱스부터 명령어의 길이 + null포인트 사이즈 만큼 내린 후, 그 주소에 명령어를 저장하는 것을 반복하였다. 또한 이 반복문이 실행될 때 명령어를 저장하는 esp포인터의 주소를 구조체의 주소에 저장하였다. 그 후 명령어 전체 길이를 저장한 변수를 통하여 word\_alignment를 해주고, 다시 반복문을 이용하여 구조체에 저장한 명령어의 주소값을 차례대로 stack저장한다. 명령어의 주소와 string을 모두 stack에 쌓은 후 주소, argument 개수, return address를 주소값을 내려가며 차례로 저장하였다. 최종적으로 구현한 구조는 다음과 같다.  


1. **User Memory Access**

<exception.c>

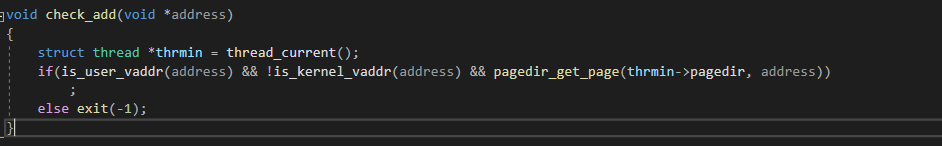
Page\_fault()안에 아래와 같은 코드를 추가하여 구현하였다. user변수가 access by kernel인 경우에 false이므로 그 경우에 exit(-1)을 해주었다.

그리고 fault\_addr 주소값을 받았을 때, kernel에 해당하는 memory space에 있는 것을 확인한 경우에도 exit(-1)를 해주도록 프로그래밍 하였다.



<syscall.c>

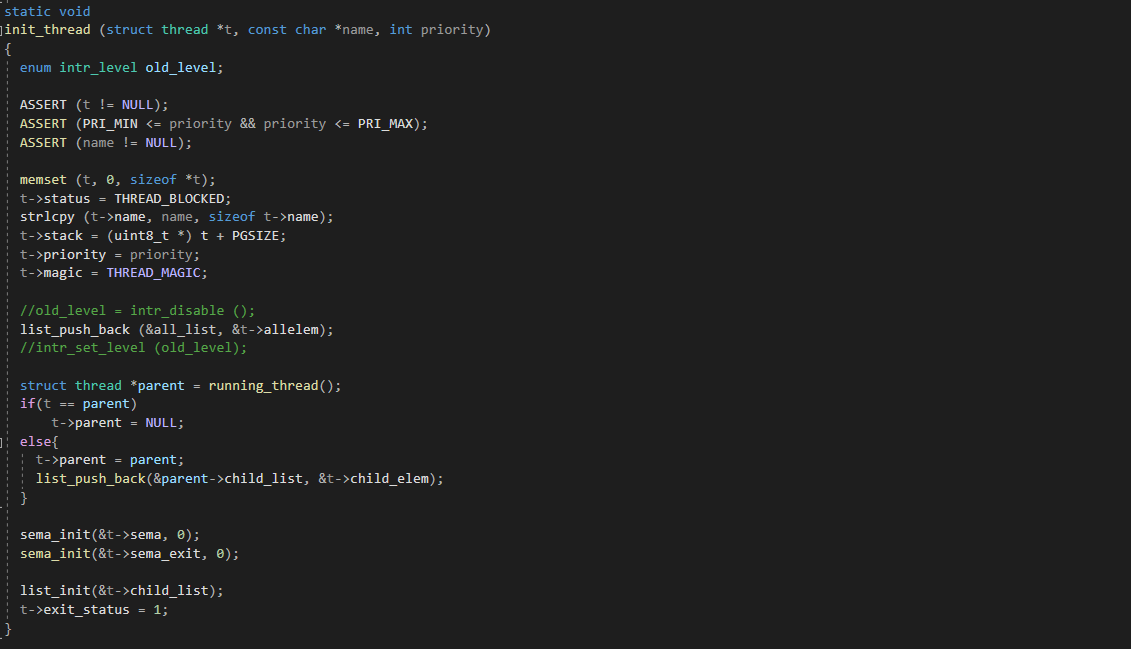
주소가 user program의 영역인지 아닌지 확인하기 위해 is\_user\_vaddr() 함수와 is\_kernel\_vaddr() 함수를 이용하여 확인gkduTrh, 아니면 exit(-1)하였다.



1. **System Calls**

먼저 thread 구조체를 구현하여 child process의 정보를 저장할 수 있게 구현하였다. Child가 오기 전에 parent는 wait()를 호출해야 하므로 semaphore child\_lock을 만들었다. 또한 만약 parent가 list\_remove를 실행하기 전에 child process가 종료하게 되면 안되기 때문에 parent가 다 수행할 때 까지 child가 terminate되지 않게 semaphore wait\_parent를 만들었다.

Init\_thread()에서는 모든 semaphore를 초기화 하도록 하였고 child를 push하게 하였다.



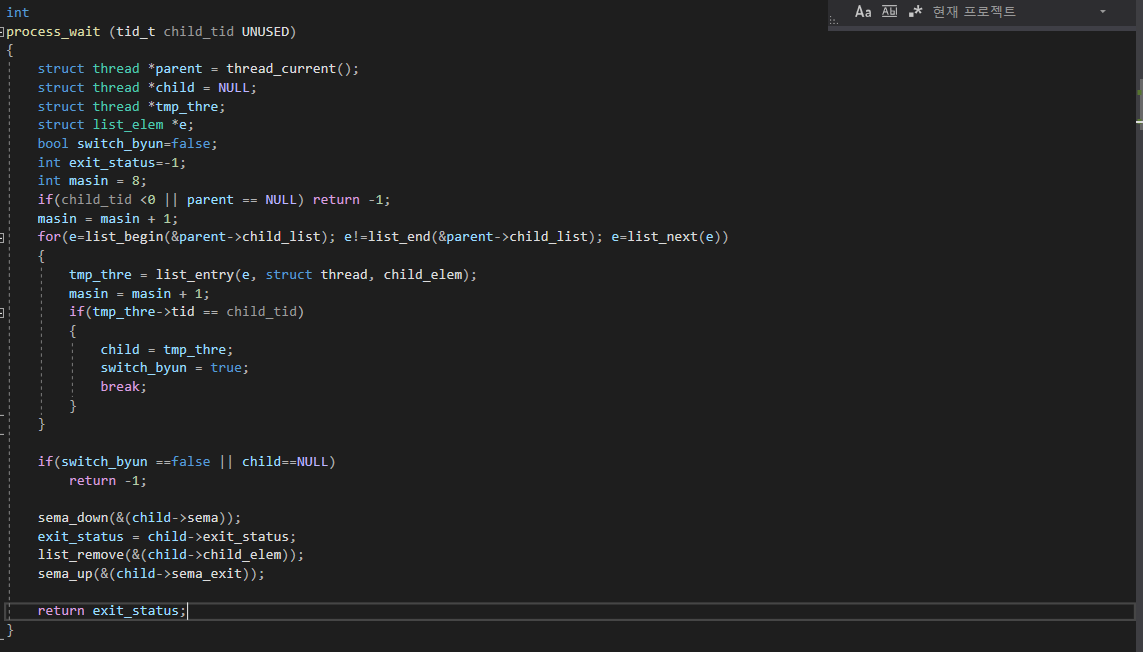
<syscall.c>

Switch문을 사용하여 system handler를 구현하였다. 해당하는 case number는 lib/syscall-nr.h에서 확인하였다.



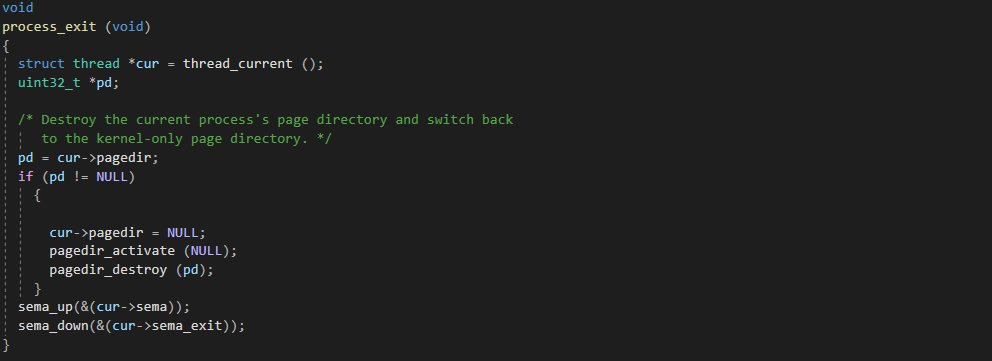
<process.c>

Process wait()에서는 parent가 입력으로 받은 pid를 가지는 child를 찾아 child의 is\_end가 1이 될때까지 semaphor를 이용하여 sleep을 유지한다. 만약에 chid가 종료된 것이 확인되면 child의 status를 return한다.



Process\_exit에서 종료를 할 때는 parent에 정상적으로 exit status를 주어야 한다. 따라서 semaphore로 critical section을 구해야 한다. 변수 sema를 sema\_up하여 parent process에게 종료되었음을 알리고, sema\_exit가 up되길 기다리면서 sema\_down한다.

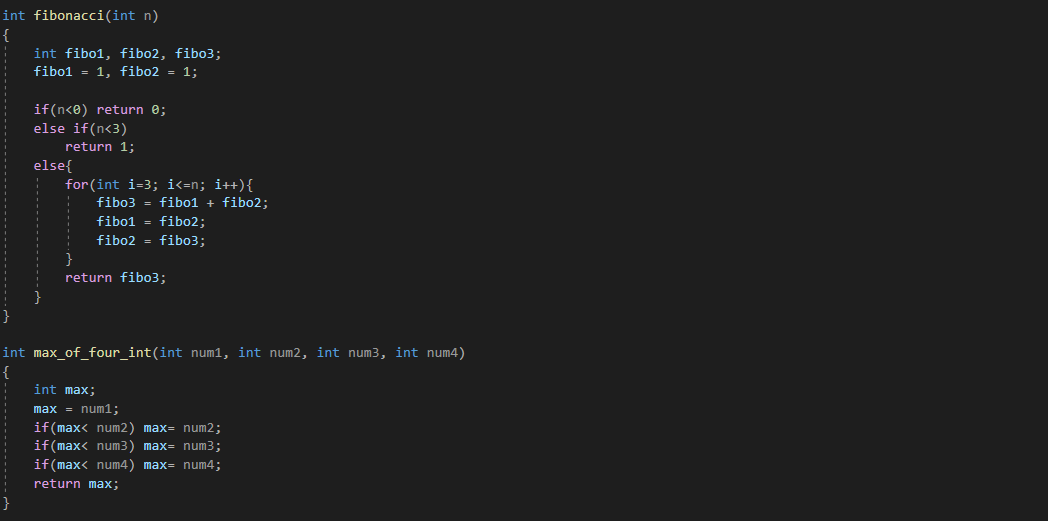
만약 parent가 정상적으로 종료했다면 해당 thread는 정상적으로 수행을 마칠 수 있을 것이다.



1. **Additional System calls**

추가적으로 syscall.c.에 Fibonacci()함수와 max\_of\_four\_int()함수를 구현하였다.

Fibonacci()는 4개의 숫자를 입력받아 첫번째 숫자의 피보나치 수열 값을 리턴하는 함수이며 max\_of\_four\_int()함수는 입력받은 4가지 숫자 중 가장 큰 숫자를 리턴하여 보여주는 함수이다. 이를 위해 Lib/syscall-nr.h.에 system call number를 추가해주었다.



* 1. **시험 및 평가 내용**

아래의 사진으로 볼 수 있듯이 10 20 62 40을 입력하였을 때 처음에는 10의 피보나치 수열값인 55가, 두번째로는 네가지 숫자중 가장 큰 62가 출력된다.

* 