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

담당 교수 : 김영재

조 / 조원 : 20161640 / 정동혁

개발 기간 : 2020/10/17~2020/10/30

1. **개발 목표**

* **이번 프로젝트의 개발 목표로는 크게 5가지가 존재한다.   
  첫 번째는 Argument Passing으로 pintos OS상에서 명령어를 입력받은뒤에 명령어를 instruction과 parameter로 구분을 지어준뒤에 stack에 저장을 해줘야 한다.   
  두 번째는 User Memory Access로 system call로부터 user memory를 보호하는 작업을 해줘야 한다.  
  세 번째는 System Call Handler로 pintos command를 입력받아 system call handler에서 command별로 처리를 해주는 전체적인 틀을 설계해야 한다.  
  네 번째는 System Call Implementation으로 이번 프로젝트는 exec, exit, write, read, halt등의 시스템콜을 구현해줘야 한다.  
  마지막은 Additional Implementation으로 4개의 숫자를 additional a b c d로 command를 입력받아 첫 번째 숫자의 피보나치 수열값, 4개중 가장큰 숫자를 pintos os에서 출력해주는 시스템콜을 구현해줘야 한다.**

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

pintos OS상에서 command를 입력하면 instruction을 통해 examples디렉토리 내 부의 프로그램을 불러오고, 이후 등장한 argument들을 Stack영역에 쌓아줘야 한 다. 이는 /userprog디렉토리 내부의 process.c에 stack\_esp라는 함수를 만들어준 뒤 argument들을 통해 Stack을 구성하는 역할을 수행하였고, load함수 내부에서 set\_up\_stack이후에 호출함으로서 Argument Passing을 구현한다.

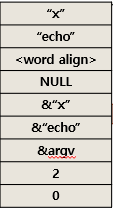
1. User Memory Access

유저메모리가 valid한지를 확인하기 위해서 threads/vaddr.h내부의 is\_user\_vaddr function을 사용한다. 사용의 편의를 위해 check\_vaddr이라는 user defined function을 userprog/syscall.c에 만들어준뒤 함수내부에서 is\_user\_vaddr를 호출하여 메모리가 유저영역을 벗어났을 경우 exit(-1)을 통해 프로세스를 종료시 켜줌으로서 invalid한 memory access를 막아준다.

1. System Calls

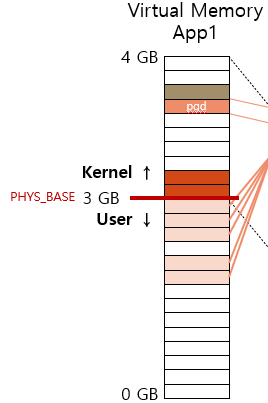
System call을 구현하기 위해서 userprog/syscall.c내부의 syscall\_handler함수를 구 현한다. 우선적으로 thread의 자식과 부모를 구분할 수 있도록 struct thread 자식 과 부모를 구분하기 위한 몇가지 멤버들을 추가해준뒤에, lib/syscall-nr.h의 enum type을 참고하여 syscall\_handler에서 case별로 나눠준다. 이후 주어진 함수들을 이용하여 halt(), exec(), exit(), read(), write()함수를 구현한다. 이를 통해 user program1의 system call들을 전부 구현할 수 있다. 마지막으로 추가적인 user progr을 구현하기 위해서 lib/user/syscall.c에서 4개의 parameter를 받는 system call을 추가적으로 구현한뒤에 userprog/syscall.c와 lib/user/syscall.c에서 fibonacci 와 max\_of\_four\_int에 맞게 함수를 구현해줌으로써 user program system call또한 구현한다.

* 1. **개발 내용**
* **Argument Passing**

주어진 강의자료를 참고하면 pintos OS상에서 ‘ehco x’라는 command가 입력되었을 때 다음과 같이 stack이 구성됨을 알 수 있다.  
 stack example of ‘echo x’  
위의 스택을 위에서부터 살펴보면 우선적으로 가장 윗부분에 argument들이 역순으로 쌓인다(argv[argc-1]~argv[0]). 다음으로는 argv를 쌓고 난뒤 word align의 관관점으로 살펴보았을 때 마지막 argv이후 공간이 남아있을 경우 4의 배수의 메모리공간을 할당해줌으로서 word alignment를 진행한다. 이후 4Byte의 공간에 NULL을 넣어주고, 다음으로 argv[argc-1]부터 argv[0]까지의 주소, 즉 argument들의 주소값들을 메모리공간에 할당해준다. 주소값의 경우 primitive한 argv와는 다르게 4의 배수로 할당이 되기 때문에 따로 word align이 필요하지 않다. 마지막으로 argv의 address를 넣어준다음 argc값과 return address(0)의 값을 memory에 할당해줌으로써 입력받은 argument에 대한 stack을 구성해주었다.

* **User Memory Access**

(1). Invalid memory access  
pintos OS상에서 invalid한 memroy access란 user process가 kernel memory 영역을 침범하게 되는 상황을 의미한다.

(2). protect invalid memory access  
우선적으로 강의자료의 pintos OS virtual memory를 살펴보면 다음과 같다.  
 pintos OS virtual memory  
위의 그림에서 보면 알 수 있듯이 우리가 구현하는 가상메모리의 경우 user process의 memory access가 PHYS\_BASE이상으로 넘어가면 안된다는 것을 알 수 있다(이것이 곧 kernel section을 침범한 것이므로).   
다행이도 위의 기능을 구현하기 이전에 threads/vaddr.h디렉토리 내부에 is\_user\_vaddr()과 is\_kernel\_vaddr()함수가 제공되어있다는 것을 강의자료에서 확인할 수 있었다. 우선 is\_user\_vaddr이란 지금 사용하는 virtual address가 user section에 존재하면 true, 아니면 false를 반환하고, is\_kernel\_vaddr이란 지금 사용하는 virtual address가 kernel section에 존재하면 true, 아니면 false를 반환해주는 함수이다. 이 두가지 함수중 syscall\_handler내부에서 is\_user\_vaddr로 각 case실행시마다 user section인지 확인해줌으로써 invalid한 memory access를 막아주었다.

* **System Calls**

(1). system call의 필요성  
일반적으로 user program이 kernel section에서 수행되는 작업(I/O task와 같은 privileged instruction)을 필요로하는 경우 user section에서 process가 수행되어지는 것이 아닌 system call을 통해 kernel section에서 수행되어야 한다. 즉, kernel section을 보호하기 위해서는 user process가 system call이라는 방법으로 kernel로 넘어간뒤에 작업을 수행해야 한다.

(2). 이번 프로젝트에서 구현한 system call  
이번 프로젝트에서 구현해야 할 system call에는 halt, exit, exec, wait, read, write, fibonacci, max\_of\_four\_int등이 있다. 각각에 대해 살펴보면 다음과 같다.  
(2-1). halt : pintos OS를 shut down하는 system call  
(2-2). exit : 현재 user process를 종료시킨 뒤에 exit\_status를 kernel에 전달해준다. exit\_status가 0일 경우 정상적인 종료를 의미한다.  
(2-3). exec : 전달받은 parameter의 이름을 가지는 프로그램을 실행시켜준다. process를 생성해준다는 것을 의미한다.  
(2-4). wait : parent process가 child process를 fork한 이후 parent process가 child process가 exit하기 이전까지 대기하게 만들어준다.  
(2-5). read : kernel section에서 buffer를 read하는 task를 수행한다.  
(2-6). write : kernel section에서 write task를 수행한다.  
(2-7). fibonacci : 피보나치 수열의 값에서 입력받은 argument n번째 값을 계산해주는 task를 수행한다.  
(2-8). max\_of\_four\_int : 4개의 passing된 argument중 가장 큰 수를 찾아주는 task를 수행한다.

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

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| **2020/10/17** | **전체강의 시청 및 ppt, pintos manual숙지** |
| **2020/10/18** | **stack\_esp function을 통해 argument passing을 위한 stack구성** |
| **2020/10/20** | **syscall\_handler를 통해 system call을 case별로 나눠줌** |
| **2020/10/27** | **thread struct수정 및 기본적인 system call들 구현** |
| **2020/10/30** | **4argument passing을 위한 system call구현과 user function(additional)작성** |
| **2020/10/31** | **보고서 작성** |

* 1. **개발 방법**

**(1). Argument passing**

우선적으로 argument passing을 위해서는 위에서 언급하였듯이 stack에 argv, argc, &argc, NULL, word alignment, return address를 쌓아줘야 한다. 이를 위해서 /userprog/process.c 에 stack\_esp라는 user defined function을 만들어준뒤 stack을 구성하였고, start\_process에서 호출된 load함수 내부에서 stack이 만들어진뒤(setup\_stack) 호출함으로써 현재 입력받은 pintos OS command에 따라 스택을 구성해주었다.

**(2). User memory access**

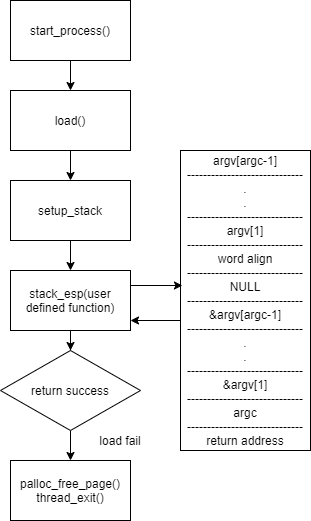
이 또한 위에서 언급하였듯이 user가 kernel section에 접근하였을 때 그 process를 종료시켜야 하므로 /threads/vaddr.h내부의 is\_user\_vaddr을 이용하였다. /userprog/syscall.c에서 user defined function인 check\_vaddr(void \*vaddr)을 만들어준뒤, 유효하지 않은 virtual address일 경우 exit(-1)을 해줌으로써 error message를 전달하고 process가 종료하게끔 만들어주었다. 이후 syscall\_handler에서 system call case마다 사용할 argument들의 virtual memory address를 검사해줌으로써 invalid한 memory acess를 방지해주었다.

**(3). system call**

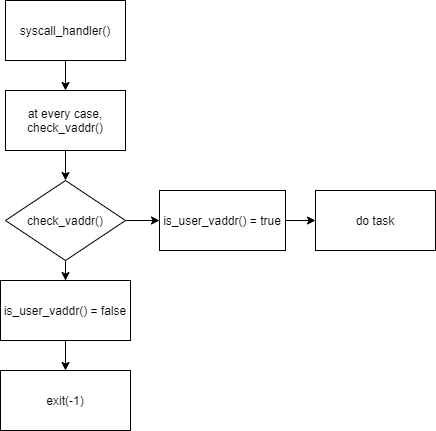
(3-1). halt : ppt에 명시된 대로 system call halt가 발생한 경우 shutdown\_power\_off() function을 통해 pintos OS를 종료시켜주는 function void halt(void)를 생성해준뒤에 syscall\_handler에서 f->esp == SYS\_HALT인 경우 호출해준다.  
  
(3-2). exit : current thread에 exit\_status를 passing해준뒤에 종료된 thread의 이름을 thread\_name function을 통해 출력해주고 thread\_exit function을 통해 종료시켜주는 function void exit(int st)를 생성해준뒤에 syscall\_handler에서 f->esp == SYS\_EXIT인 경우 호출해준다.  
  
(3-3). exec : /userprog/process.c 내부에 구현되어 있는 process\_excute에 전달받은 command를 passing해줌으로써 pintos OS를 통해 전달받은 command의 instruction과 같은 이름의 program을 examples directory에서 찾아 실행시켜주는 function tid\_t exec(const char \*comm)를 생성해준뒤에 syscall\_handler에서 f->esp == SYS\_EXEC인 경우 호출해준다.  
  
(3-4). read : 전달받은 size만큼을 전달받은 buffer에서 읽어들인 뒤에 읽어온만큼의 size를 return해주는 function int read(int fd, void\* buf, unsigned size)를 생성해준뒤에 syscall\_handler에서 f->esp == SYS\_READ인 경우 호출해준다.  
  
(3-5). write : putbuf function과 전달받은 size를 이용하여 write를 하고 write한 size를 return해주는 function int write(int fd, void \*buf, unsigned size)를 생성해준뒤에 syscall\_handler에서 f->esp == SYS\_WRITE인 경우 호출해준다.

(3-6). fibonacci : dynamic programming기법을 사용하여 n번째 피보나치 수열의 값을 찾아주는 function int fibonacci(int n)을 만들어준 뒤에 syscall\_handler에서 lib/syscall-nr.h에 새롭게 정의한 SYS\_FIBONACCI와 f->esp가 같은 value를 가질 경우 호출해준다.  
  
(3-7). max\_of\_four\_int : 입력받은 4개의 argument들중 가장 큰 값을 찾아주는 function int max\_of\_four\_int(int a, int b, int c, int d)를 만들어준 뒤에 syscall\_handler에서 lib/syscall-nr.h에 새롭게 정의한 SYS\_FOURMAX와 f->esp가 같은 value를 가질 경우 호출해준다.

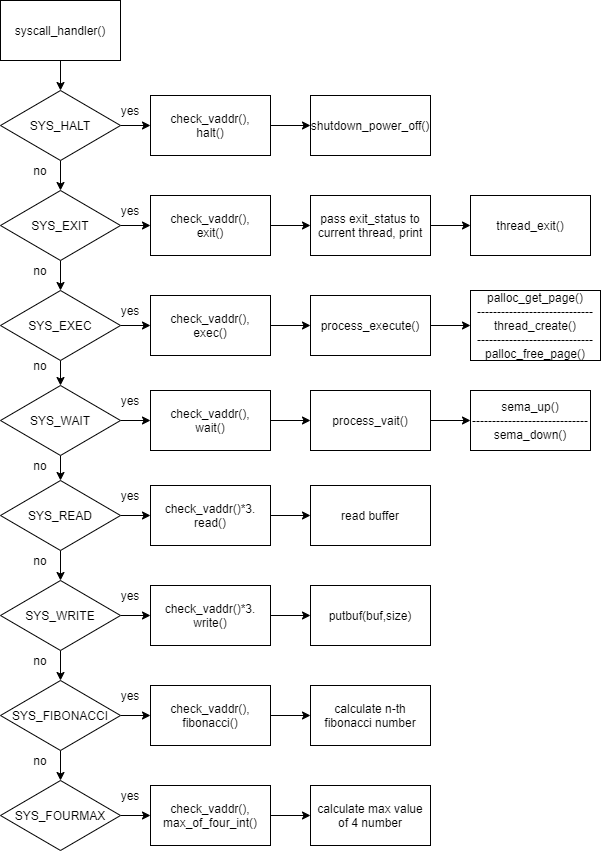
1. **연구 결과**
   1. **Flow Chart**
2. **Argument Passing**



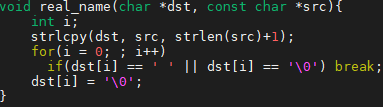
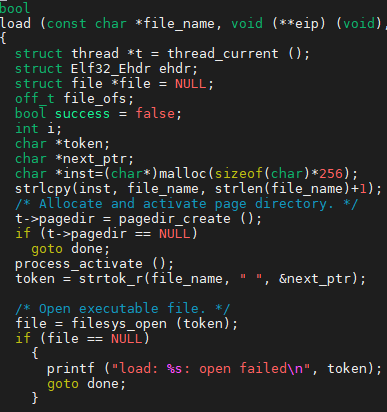
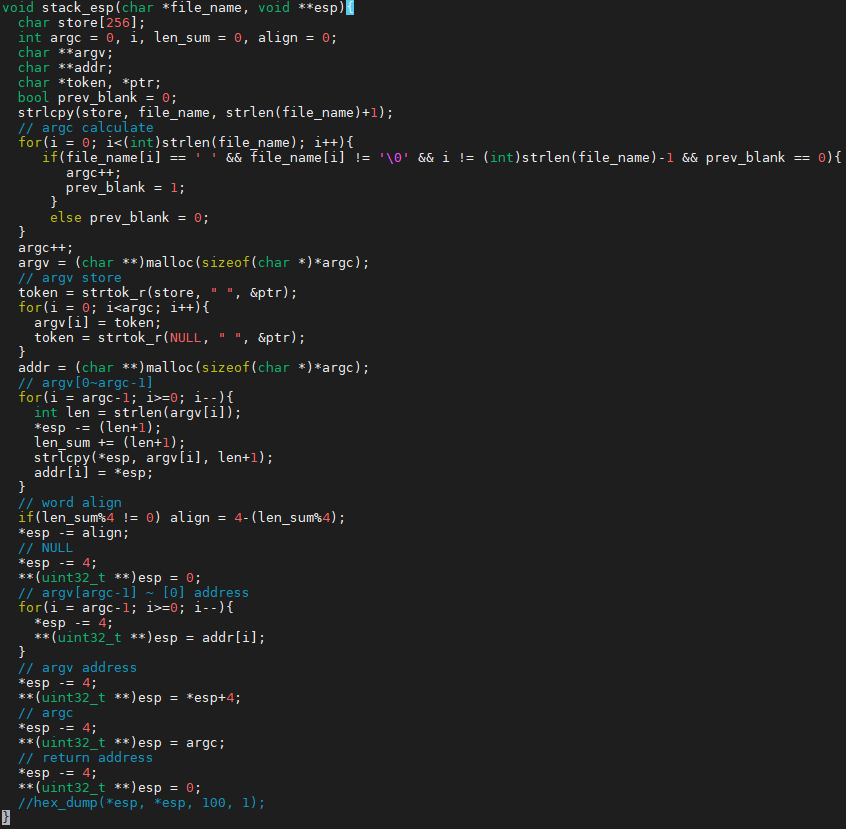
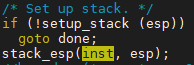
1. **User Memory Access**

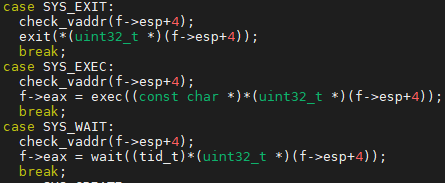
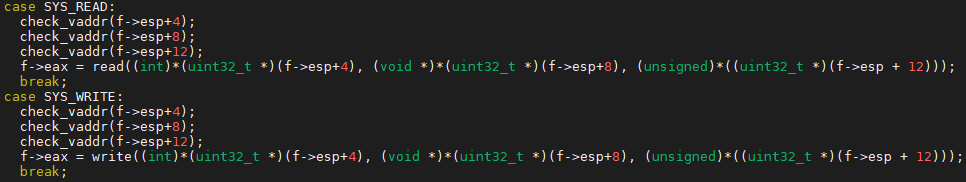


1. **System Calls**

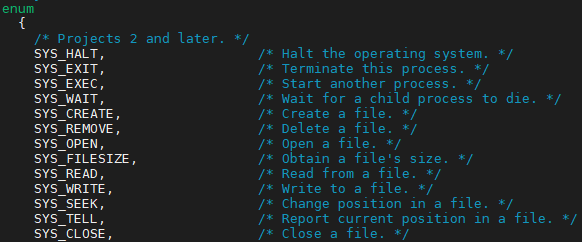
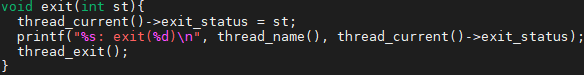
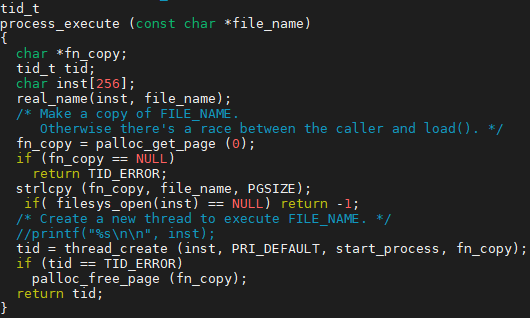
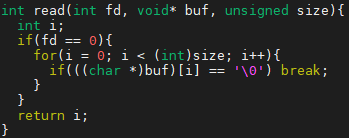
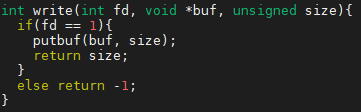


* 1. **제작 내용**

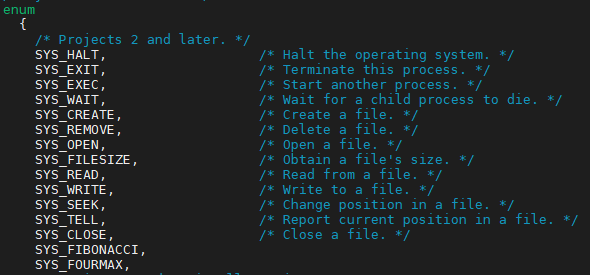
1. **Argument Passing**우선적으로 process가 시작할 때 /userprog/process.c 내부의 start\_process를 호출하게 된다. 이후 그 내부에서 load function을 호출하는데, load function에서 전달받은 file\_name을 출력해보면 pintos OS상에서 echo가 아닌 echo x가 출력이 된다. 그러나 우리가 load function에서 필요한 정보는 instruction과 argument들이 분리된 형태가 요구되므로 다음과 같은 void real\_name function을 통해 file\_name과 argument들을 load함수내부에서 분리해주었다.  
    get instruction  
   이후 load함수에서 입력받았던 file\_name을 새로운 buffer변수인 inst에 복사하여 저장을 해준뒤, file\_name을 instruction만 남긴채로 이용해주었다.  
    just get instruction from file\_name  
    call with inst(full coammand)  
   또한 이전에 언급하였던 argument passing을 위한 stack을 build하기 위해 stack\_esp function을 작성하였다.   
   본인이 구현한 stack\_esp의 경우 우선적으로 full command name을 가지고 있는 inst를 parameter로 받는다. 그리고 안에서 공백을 기준으로 argc값을 계산해주고 그 argc값과 “ “을 이용한 strtok\_r function을 이용하여 argv배열을 생성해준다.  
   이후 강의자료에서 명시된대로 스택의 esp를 이용하여 위에서부터 argv[argc-1]부터 argv[1]까지의 값을 쌓아주고, 그 과정에서 각 argv배열의 값들을 addr배열에 넣어준다. 이후 stack memory를 4의 배수형태로 만들어줘 word alignment를 진행하였고, NULL value또한 stack에 넣어주었다. 이후 이전에 저장하였던 argv배열의 address value들도 넣어주고 마지막으로 argc value와 return address를 넣어줌으로써 우리가 원하는 stack을 설계하였다. stack\_esp는 다음과 같다.  
     
   이로써 pintos OS상에서 command를 입력받은뒤 command에 해당하는 stack에 대한 구성을 하였다.  
   이후 다음과 같이 setup\_stack을 진행한후 stack\_esp(inst, esp)를 호출함으로써 정상정으로 stack을 build할 수 있었다.   
    make stack
2. **User Memory Access**

위에서 언급하였듯이 pintos OS상에서의 user section의 memory는 0x00000000부터 0xC0000000까지라는 사실을 이용하여 kernel memory접근을 방지하기 위해서 /threads/vaddr.h에 존재하는 is\_user\_vaddr()함수를 사용하였다.   
사용의 편의를 위해 다음과 같이 user section memory가 아니라면 exit(-1)을 통해 종료하게 하였다.  
  
이후 syscall\_handler function에서 case별로 명령어를 수행하기 이전에 앞서 구현한 stack을 확인하여 들어온 argument가 valid한 경우는 그냥 실행해주지만 invalid한 경우에는 check\_vaddr의 exit(-1)을 통해 종료시켜주었다.  
  
  
이렇게 system call을 수행할 때마다 address를 검사해줌으로써 user memory access issue를 handle할 수 있었다.

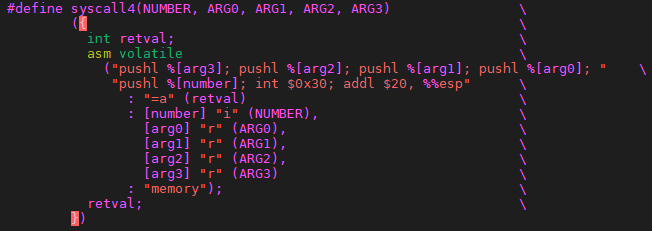
1. **System Calls**

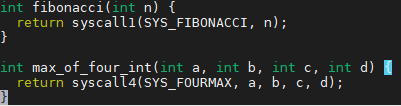
system call을 구현하기 위해서 우선적으로 다음과 같이 lib/syscallp-nr.h에 정의된 enum type을 참고하였다.  
  
이후 userprog/syscall.c에서 다음과 같이 각 enum case(f->esp)별로 syscall\_handler function에서 switch case문을 통해 나눠주었다.  
  
**3-1). halt**  
halt에 전달되는 argument는 존재하지않으므로 vaddr이 valid한지는 검사하지 않고 바로 halt function을 호출해주었다. halt function의 경우 다음과 같이 shutdown\_power\_off() function을 호출해줌으로서 pintos OS를 종료시켜주는 역할을 수행하였다.  
  
  
**3-2). exit**  
exit에 전달되는 argument가 하나 존재하므로 f->esp+4의 memory가 valid한지 검사하였고 이후 exit funtion을 호출해주었다. exit function에서는 current thread에게 exit\_status를 전달해줘야므로 thread 구조체에 exit\_status를 추가하여 현재의 exit정보를 넘겨주었다. 또한 thread가 종료될 시 종료된 thread의 이름을 넘겨줘야 했는데, 이는 강의자료를 참고하여 thread\_name을 통해 종료된 thread의 이름을 출력해주었다. 이후 제공된 함수인 thread\_exit() function을 통해 정상적으로 thread를 종료시켜 주었다.  
  
  
**3-3). exec**  
exec에 전달되는 argument가 하나 존재하므로 f->esp+4의 memory가 valid한지 검사하였고 이후 exec funtion을 호출해주었다. exec function의 경우 userprog/process.c에 존재하는 process\_execute(comm)을 호출해주는 역할을 수행한다.  
하지만 process\_execute또한 앞서 말했던 것처럼 ‘echo’가 아닌 ‘echo x’를 parameter로 받으므로 다음과 같이 parsing을 해준뒤에 thread\_create에 넣어줌으로써 정상적으로 instruction을 받아 thread를 만들어주도록 하였다. 또한 make check의 test case중 exec-missing을 살펴보니 filename이 examples디렉토리에 존재하지 않을 경우 -1을 return해줘야 한다고 언급되어있기 때문에 filesys\_open function을 통해 실행파일을 찾아본뒤 없으면 -1을 return 하도록 설계해주었다.  
  
  
**3-4). read**  
read의 경우 전달되는 argument가 총 3개 존재하므로 f->esp+(4, 8, 12)의 memory가 valid한지 검사하였고 이후 read function을 호출해주었다. read function의 경우 전달받은 1st argument값이 0인경우 3rd argument인 size만큼 2nd argument인 buffer를 읽어주는 역할을 수행하였다. 이후 읽은 size를 return해주었다.  
  
  
**3-5). write**  
write의 경우 전달되는 argument가 총 3개 존재하므로 f->esp+(4, 8, 12)의 memory가 valid한지 검사하였고 이후 write function을 호출해주었다. write function의 경우 전달받은 1st argument값이 1인 경우 kernel에 정의되어 있는 putbuf function을 수행하게 하였고 size를 return하게 하였다. 만약 1st argument가 1이 아닌경우에는 정상적으로 수행되지않았으므로 -1을 retrun하게 하였다.  


1. **Additional System calls**

가장먼저 새로운 system call을 구현하기 위해서는 enum에 두 가지 case를 추가해줘야하므로 lib/syscall-nr.h에 다음과 같은 SYS\_FIBONACCI, SYS\_FOURMAX를 만들어주었다.  


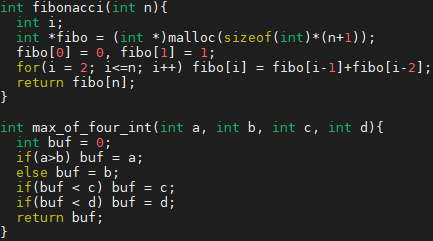
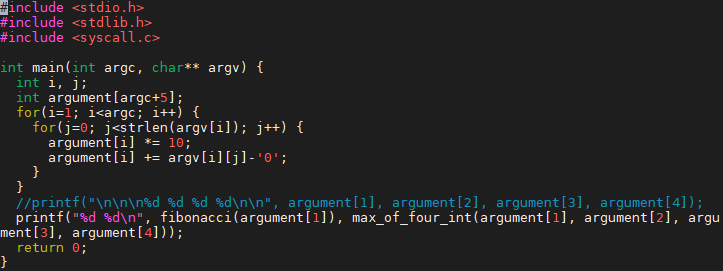
이후 /lib/user 디렉토리에 존재하는 syscall.h에 다음과 같은 두개의 함수원형을 정의해주었다.  
 /lib/user/syscall.h

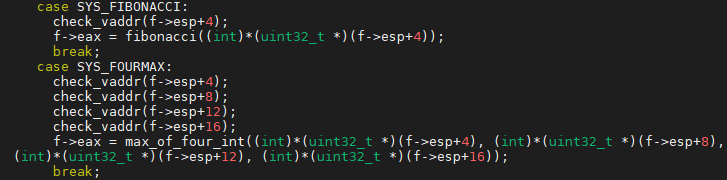
또한 구현한 pintos os에서는 3개까지의 argument passing만 구현되어 있으므로 추가적으로 4개의 argument를 가지는 syscall4를 새롭게 define해주었다. syscall4는 다음과 같다.  


이후 /lib/user/syscall.c 코드의 마지막부분에 이전에 헤더파일에서 선언한 두개의 함수의 경우 다음과 같이 각각 syscall1, syscall4를 호출함으로써 정상적으로 parameter를 넘겨주었다.  
 /lib/user/syscall.c

이제 system call의 argument passing에 대한 처리는 끝났으니 handler를 통해 두 가지 case에 대한 처리를 구현하면 된다.

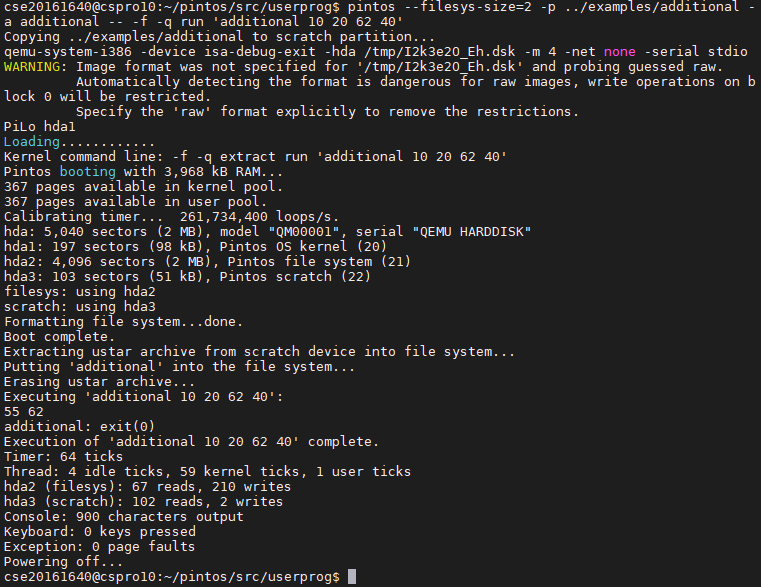
다음으로 userprog디렉토리 내부의 syscall.h에 다음과 같이 두개의 함수원형을 정의해주었다.  
 /userprog/syscall.h  
이후 syscall.c에서 각 함수의 의도, 기능에 맞게 다음과 같이 함수를 작성해주었다.

 /userprog/syscall.c  
 그 다음 syscall\_handler에 의해서 case를 나눠주기 이전에 examples 디렉토리내부에서  
 다음과 같이 additional.c를 만들어준뒤에 Makefile에 additional case를 추가해줌으로써 additional의 실행파일을 만들어주었다.  
  /examples/additional.c

마지막으로 다음과 같이 case에 맞게 syscall\_handler함수에서 파라미터를 함수에 넘겨준 뒤 eax에 저장함으로써 정상적으로 additional system call을 구현하였다.  
   
 /userprog/syscall.c

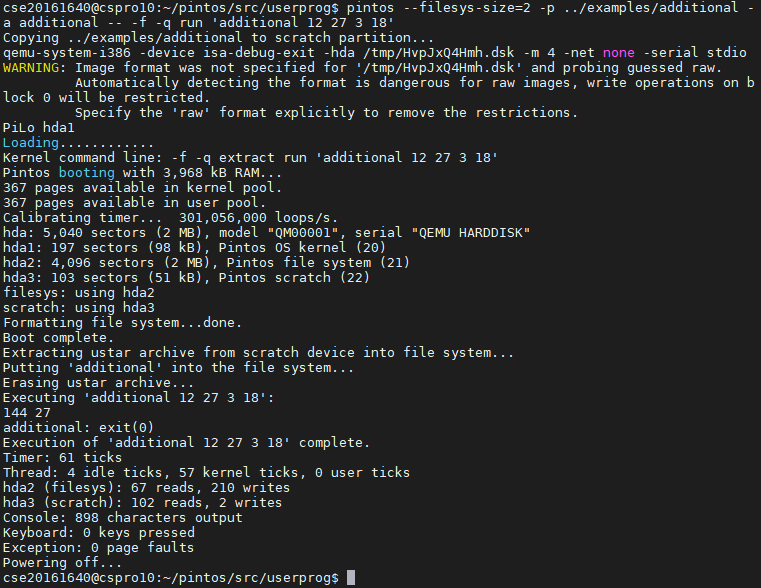
* 1. **시험 및 평가 내용**

**1st test case(fibonacci(10), max\_of\_four\_int(10,20,62,40))**



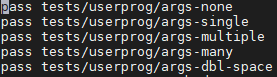
피보나치 수열의 10번째 value는 55이고 10, 20, 62, 40중 가장 큰 값을 가지는 수는 62이다. pintos os에서 additional 10 20 62 40을 수행한 결과가 이와 일치하므로 정상적으로 구현하였다고 볼 수 있다.

2nd test case(fibonacci(12), max\_of\_four\_int(12,27,3,18))



피보나치 수열의 12번째 value는 144이고 12, 27, 3, 18중 가장 큰 값을 가지는 수는 27이다. pintos os에서 additional 12 27 3 18을 수행한 결과가 이와 일치하므로 정상적으로 구현하였다고 볼 수 있다.

**최종결과  
(1). Funtionality**



**(2). Robustness**

**ppt에 명시된 21개의 test case를 모두 통과하였다.**