**Pintos Project 1 : User Program Basic**

**(설계 프로젝트 수행 결과)**

과목 명 : [CSE4070] 운영체제

담당 교수 : 김영재 교수님

조 / 조원 : 42조 20141501, 20141531

개발 기간 :2018.09.22 ~ 2018.10/18

**프로젝트 제목 : pintos project 1 user program basic**

**제출일 : 2O18.10.18**

**참여 조원 : 김건우 변재승**

**I. 개발 목표**

- Pintos에서의 system call을 구현하여 user program을 실행시키는 것이 최종 목표이다. 이를 위하여 system call의 argument passing을 하기 위해 명령어를 parsing해서 메모리에 적재시킨다. Thread가 일정 부분을 반복 수행하기 위하여 synchronization 기법 중 하나인 busy waiting을 사용한다.

**II. 개발 범위 및 내용**

**A. 개발 범위**

1. Argument Passing

명령어를 입력받아서 토큰화 한 다음에 stack에 PHYS\_BASE부터 쌓는다.

쌓은 방식은 아래서 부터 return address -> argc -> argv -> argv[] -> word-align -> argv[][]이다.

pintos/src/userprog/process.c의 process\_execute함수에서 page를 생성 후 파일 이름을 토큰화 하여 argv[0]에 해당하는 thread를 create함과 동시에 process\_start함수를 호출하고 그 안에서 process\_load함수를 호출한다.

process\_load함수에서 stack을 setup한 후에 esp에 저장된 PHYS\_BASE부터 stack을 쌓는다.

2. User Memory Access

user program이 invalid한 pointer를 pass하는 경우가 3가지 있지만 이번

프로젝트 에서는 2가지만 다룬다.

(1) unmapped virtual memory - pintos/src/userprog/syscall.c의 exec명령어가 system call될 때 “no-such-file”을 실행하라 할 경우 error처리를 한다.

(2) pointer to kernel address space - 이 문제를 해결하기 위해 두가지 방법을 혼합하여 사용한다. dereference전에 pointer들의 validity를 pintos/src/userprog/syscall.c에서 확인하여 address가 user program이 커널 영역 메모리 주소를 참조하는지 확인한다. 추가로 pintos/src/userprog/exception.c 에서도 kernel 영역을 참조하는지 확인한다.

3. System Call

* System call Infrastructure

system call을 처리하기 위한 기본 구조를 만든다. Intr\_frame 구조체를 활용하여 해당하는 기능을 구현해야 한다. Intr\_frame 구조체는 pintos/src/threads/interrupt.h에 선언되어 있다. 결국 이러한 기능을 구현하기 위해서는 pintos/src/userprog/syscall.c에서의 syscall\_handler 함수에서 Intr\_frame 구조체를 전달받아 구조체 안의 esp를 이용해 argument 정보들을 받고 이를 해석하여 해당하는 기능을 구현해야한다.

* System call implementation

현재 프로젝트에서 요구하고 있는 system call의 종류로는 write(), read(), exec(), exit(), halt() 함수이다. process가 정상적으로 종료되려면 구현되어야만 하는 동기화 과정을 사용한다. 이번 프로젝트에서는 busy waiting 방법으로 동기화를 구현한다. write(), read(), exec(), exit(), halt() 함수를 구현하기 위하여 pintos/src/userprog/syscall.c 안에서 함수를 구현시키고, 동기화는 pintos/src/userprog/process.c 에서의 process\_wait 함수에서 구현한다.

4. Additional Implementation

Additional Implementation은 pintos/src/example에 pibonacci함수와 sum of four

integer함수를 갖고있는 sum.c를 생성하고 Makefile을 수정하여 실행파일을 만들

어서 pintos/src/userprog/build에서 실행되게 해야 한다.

pintos/src/lib/syscall-nr.h에서 enum형의 system call number를 추가한다.

pintos/src/lib/user에 있는 syscall.c와 syscall.h에 system call하는 함수를 추가하고 syscall.c에 argument가 4개일 경우의 system call방식을 define한다.

pitos/src/userprog/ syscall.c와 syscall.h에 system call 하는 함수를 추가한다.

.

**B. 개발 내용**

이 프로젝트는 pintos에서 system call을 구현하여 user program을 실행시켜야 하기 때문에 pintos에서 수행되는 각각의 thread 즉 프로세스 내에서 실행되는 흐름들은 하나의 page를 할당 받고 각각의 thread에는 수행에 필요한 정보들이 저장되어 있다. page는 pintos의 물리적 메모리 단위 virtual memory에 존재하고 page마다 4kb로 구성되어 있다.

Argument를 받아서 stack에 쌓을때 physical memory에 속한 user memory pool에 쌓는다. stack이 kernel memory pool에 접근하게 되면 에러가 발생한다. 또한 stack에 쌓인 정보들을 system call을 이용하여 접근할 때에도 kernel memory pool에 접근하면 안된다. 우선 stack에 쌓을때 접근을 막는 방법은 esp를 user memory pool과 kernel memory pool이 분리되어 있는 기준인 PHYS\_BASE부터 아래로 쌓아가는 것이다. 즉 esp의 값과 PHYS\_BASE와 비교를 하면 된다. 이렇게 된다면 절대 kernel memory pool에 접근할 수 없다. system call로 호출할 때에는 호출 시 필요한 pointer들을 호출 전에 kernel memory pool을 가리키고 있는지 검사하는 것이다. 이 pointer들 역시 esp를 기준으로 되어 있는데, 만일 pointer들이 kernel memory pool을 가리키고 있다면 Kernel memory에 접근하는 것이므로 exit을 해주면 된다.

이 프로젝트에서의 동기화 기법은 busy waiting 기법을 사용한다. busy waiting 기법이란 부모 thread가 자식 thread를 실행시키기 전까지 멈춰있는 것이 아니라, 계속 돌면서 자식 thread 가 끝날때까지 기다리고 끝나는 즉시 부모 thread가 실행되는 방법의 동기화 기법이다. 이렇게 자식 thread가 끝났다는 것을 알기 위하여 우리는 thread 구조체에 lifeflag를 기본적으로 1로 초기화시켜놓고, 만약 thread가 끝나면 0으로 바꿔주었다. flag를 설정함으로써 부모 thread는 자식 thread의 lifeflag가 1이면 계속 멈춰서 0이 될때까지 기다리게 만들면 된다.

**III. 추진 일정 및 개발 방법**

**A. 추진 일정**

-  **09.22 ~ 09.28 : Argument Passing & User Memory Access**

**09.29 ~ 10.05 : System Call Infrastructure & Additional Implementation**

**10.06 ~ 10.16 : System Call Implementation**

**10.17 ~ 10.18 : 보고서 작성**

**B. 개발 방법**

1. Argument Passing

입력받은 명령어를 process\_execute에서 토큰화 하여 argv[0]에 저장된 문자열을 thread의 이름으로 정하여 thread를 생성한다. 이 때 start\_process도 호출하여 start\_process안에서 user process를 load하고 실행한다.

process\_load를 호출할 때 명령어와 eip, esp를 parameter로 넘긴다.

setup\_stack함수를 통해서 esp에 user memory pool과 kernel memory pool의 기준인 PHYS\_BASE를 설정해준다. 그리고 strtok\_r을 이용하여 명령어를 잘라서 argv[][]에 argument를 저장하고 argument의 갯수를 argc에 저장한다. strtok가 아닌strtok\_r을 사용하는 이유는 strtok를 wrapping하는 함수이므로 static char로 선언된 변수를 함수의 인자로 받아서 처리하기 때문에 thread에서 안전하다. argument들이 저장되어 있는 argv의 address들은 argv\_address배열에 저장한다. 그 후에 각각의 배열에 저장되어 있는 것들을 stack에 쌓는다.

제일 먼저 argv[][]에 저장된 argument를 쌓고 argument들이 정확히 4의 배수가 아닐경우 word align을 사용하여 pc를 4의 배수로 맞춘다. 4의 배수로 맞추는 이유는 명령어의 byte의 총합이 4의 배수일수록 데이터의 접근속도가 빨라지기 때문이다. word align의 데이터는 0을 넣는다. 그 후에 NULL을 넣고 argv\_address에 저장된 argv argument들이 저장되어 있는 주소를 stack에 쌓는다. 주소를 모두 쌓은 후에는 argv를 가리키고 있는 주소를 쌓고 argument의 갯수인 argc를 쌓아준다. 마지막으로는 return address를 넣어서 stack을 완성한다.

2. User Memory Access

(1) unmapped virtual memory

“no-such-file”을 execute할 경우 unmapped virtual memory로 포인터가 향하게

된다. 이러한 상황을 방지하기 위하여 exec system call에서 process\_execute를

호출하여 실행할 때 인자로 넘어온 file\_name이 no-such-file일 경우 return -1을

해준다.

(2) pointer to kernel address space

dereference전에 pointer들의 validity를 pintos/src/userprog/syscall.c에서

확인하여 address가 user program이 커널 영역 메모리 주소를 참조하는지

확인한다. 추가로 pintos/src/userprog/exception.c 에서도 kernel 영역을

참조하는지 확인한다.만약 address가 user program이 커널 영역 메모리 주소를

참조하게 된다면 exit(-1)을 호출해야 한다. 이를 구현하기 위하여

pintos/src/userprog/syscall.c안에서 check\_vaddr() 함수를 만들어서 address를

참조할 때마다 check\_vaddr()함수를 활용하여 정상적으로 메모리 주소를

참조하는지 확인한다.. 추가로 pintos/src/userprog/exception.c 에서도 user라는

변수를 활용하여 kernel 영역을 참조한다면 exit(-1)을 해준다.

3. System Call

* System call Infrastructure

pintos/src/userprog/syscall.c에서의 syscall\_handler 함수에서 Intr\_frame 구조체를 전달받아 구조체 안의 esp를 이용해 argument 정보들을 받고 이를 해석하여 해당하는 기능을 구현해야한다. esp는 elem형식으로 되어있고, elem은 pintos/src/lib/syscall-nr.h 에서 제공해준다. flame의 esp가 SYS\_HALT, SYS\_EXIT, SYS\_EXEC, SYS\_WAIT, SYS\_READ, SYS\_WRITE, SYS\_PIB, SYS\_SOFI에 따라서 해당 기능을 하는 함수를 호출하여 system call을 한다.

* System call implementation

pintos/src/userprog/syscall.c안에서 flame의 esp가 SYS\_HALT, SYS\_EXIT, SYS\_EXEC, SYS\_WAIT, SYS\_READ, SYS\_WRITE, SYS\_PIB, SYS\_SOFI에 따라서 해당 기능을 하는 함수를 호출한다.

* system call이 halt인 경우 아예 power\_off 해야하므로 shutdown\_power\_off() 함수를 호출한다.
* system call이 exit인 경우 현재 thread를 exit 하는 것이다. 따라서 thread의 이름과 exit하는 상태인 status를 출력해준다. 또한 동기화를 위하여 부모 thread가 자식 thread의 상태를 알고 있어야 한다. 따라서 exit을 하는 thread의 부모 thread에 exit status를 부모 thread 구조체 안 child\_status라는 변수를 지정하여 저장해놓는다. 그리고 thread\_exit() 함수를 사용함으로써 해당 thread를 exit한다. thread\_exit() 함수에서 process\_exit() 함수도 호출을 하여 exit시킨다.
* system call이 exec인 경우 process\_execute 함수를 호출하고 thread 번호인 tid를 return 함으로써 입력받은 argument 정보를 실제로 calling하고 실행시킨다.
* system call이 wait인 경우 process\_wait() 함수를 호출하고 부모 thread가 자식 thread의 실행이 끝날때까지 기다린다. 이때 동기화를 사용하는데, 우리가 사용하는 동기화 기법은 busy waiting이므로 process\_wait() 함수 안에서 busy waiting을 이용하여 동기화를 진행한다. process\_wait() 함수의 parameter는 자식 thread number인 child\_tid이다. 그런데 여기서 tid를 번호로 가지는 thread를 찾기 위하여 우리는 모든 thread list를 저장하고 있는 linked\_list인 every\_list를 변수로 설정하였다. 이 thread list인 every\_list는 pintos/src/threads/thread.h 에 설정해두었다. every\_list를 처음부터 끝까지 돌면서 child\_tid인 thread를 확인하여 자식 thread를 확인하고, 부모 thread는 자식 thread가 죽을때까지 계속 무한 loop를 돌며 execute하지 않고 기다린다. 그리고 자식 thread가 죽으면 loop를 벗어나며 해당 thread가 실행이 되는 구조이다. 이를 위해서 우리는 thread 구조체마다 lifeflag 라는 flag 변수를 만들어놓고, 1로 초기화해두었다. 만약 thread가 끝나면 lifeflag 변수를 0으로 바꾸어주었고 이 flag를 활용하여 부모 thread는 자식 thread의 lifeflag가 0이 될때까지 무한 loop가 실행되는 것이다.
* system call이 read인 경우 fd가 0일때 pintos/src/device/input.c에 있는 input\_getc함수를 이용하여 keyboard로 부터 읽어서 buffer에 저장한다. buffer에 저장한 size를 return한다. fd가 0이 아닌 경우는 이번 프로젝트에서는 구현하지 않는다.
* system call이 write인 경우 fd가 1일때 pintos/src/lib/kernel/console.c에 있는 putbuf함수를 이용하여 console에 write한다. console에 write한 size를 return한다. fd가 1이 아닌 경우는 이번 프로젝트에서는 구현하지 않는다.
* system call이 pib인 경우 parameter로 넘어온 n을 갖고 피보나치 수열의 n번째 값을 return해주는 pib()함수를 호출한다.
* system call이 sofi인 경우 parameter로 넘어온 네가지 argument의 합을 return해주는 sofi()함수를 호출한다.

4. Additional Implementation

(1) pintos/src/liib/syscall-nr.h & pintos/src/userprog/syscall.c & pintos/src/userprog/syscall.h

pintos/src/liib/syscall-nr.h 에 존재하는 enum type의 system call number에 SYS\_PIB ( pibonacci )과 SYS\_SOFI ( sum\_of\_four\_integers)를 추가하고 , pintos/src/userprog/syscall.c에 system call number SYS\_PIB와 SYS\_SOFI를 받는

if문을 추가한 후 int pib와 int sofi코드를 구현한다.

(2) pintos/src/lib/user/syscall.h

피보나치 배열의 n번째 숫자를 구하는 함수 int pib(int n)와 정수 4개를 더해주는 함수 int

sofi(int a,int b,int c,int d)를 선언해준다.

(3) pintos/src/lib/user/syscall.c

기존 코드에는 argument를 3개까지 받을 수 있었는데 int sofi함수에서는 argument가

4개이므로 #define syscall4(NUMBER,ARG0,ARG1,ARG2,ARG3)을 선언해주고 코드를

작성한다. 그 다음에 syscall.h에 선언한 int pib함수와 int sofi함수에 pib와 sofi에 대한 system call을 구현한다.

(4) pintos/src/examples

pib랑 sofi함수가 제대로 구현되고 실행되는지 확인하기 위해 sum.c를 구현하고 makefile이

sum의 실행파일을 생성하게끔 고친 후 make를 통해서 sum의 실행파일을 생성한다.

최종적으로 pintos/src/userprog/build에서 sum명령어를 이용하여 프로그램을 실행하여 pib함수와 sofi함수가 system call number를 받아서 제대로 돌아가는지 확인한다.

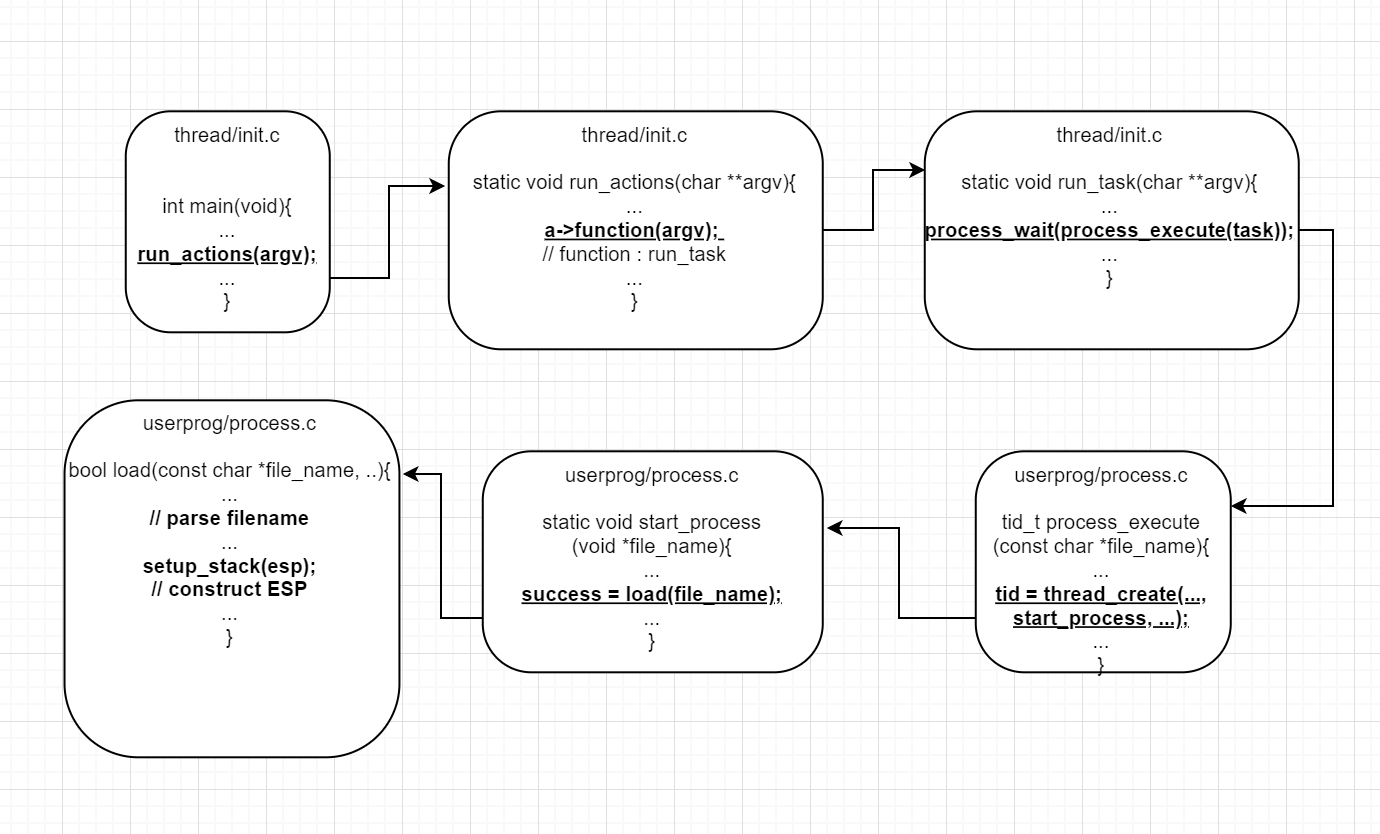
-

**C. 연구원 역할 분담**

-  **project 1의 흐름을 같이 공부하여 개발 방식을 다 구상하여 놓은 후에 비교적 쉬운 난이도의 Argument passing과 Additional Implementation을 변재승이 구현하였고, User Memory Access와 System Call Infrastructure를 김건우가 구현하였다. 그리고 복잡하고 어려운 난이도의 System Call Implementation을 같이 구현하였다.**

**IV. 연구 결과**

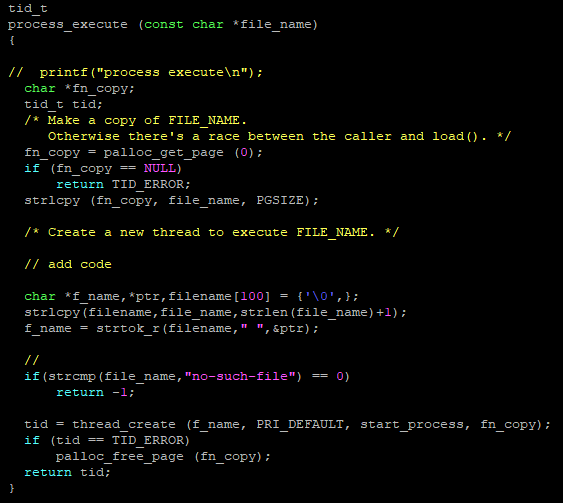
**A. 합성 내용**

-  ****

**B. 제작 내용**

1. Argument Passing

**→ tid\_t process\_execute(const char \*file\_name) [pintos/src/userprog/process.c]**

****

\* 함수 설명 : filename으로 부터 load된 user program을 실행하는 새로운 thread를 시작한다. 새로운 thread는 process\_execute()가 return되기 전에 schedule된다. 함수가 끝난 후에는 new process의 thread id가 return되고, thread가 생성되지 않았을 경우 TID\_ERROR가 return된다.

\* 사용 변수 :

char \*f\_name : filename의 자른 부분을 저장하는 변수

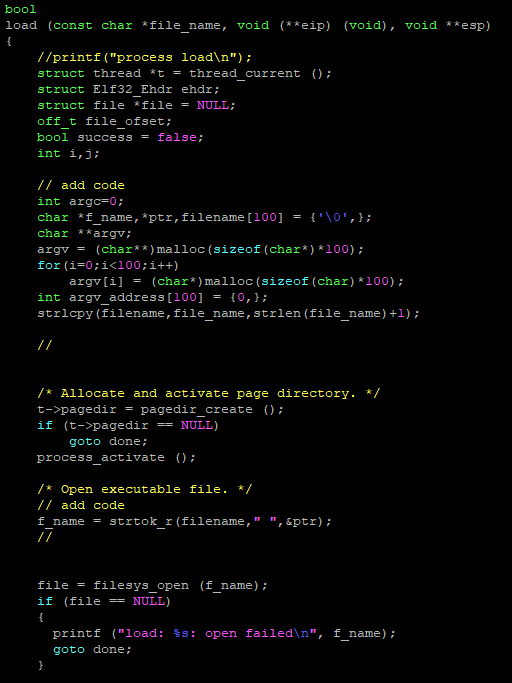
char \*ptr : strtok\_r을 사용할때 다음 처리를 위한 위치를 저장하는 pointer, 이 변수는 직접 handling하지 않는다.

char filename[100] : parameter로 넘어온 file\_name의 맨뒤에 NULL을 추가해서 저장하는 변수

\* 내 용 : file\_name뒤에 NULL을 붙여서 filename에 저장하고 filename을 strtok\_r을 이용하여 잘라낸 후 맨 처음 잘라낸 문자열 f\_name이 no-such-file이면 return -1을 하고 no-such-file이 아닐 경우 thread를 create하면서 process를 시작한다

**→ static bool load(const char \*cmdline, void (\*\*eip) (void),void \*\*esp) [pintos/src/userprog/process.c]**

(1)

****

\* 함수 설명 : 현재의 thread에 file\_name으로부터 ELF executable을 load하고 executable의 시작 포인트를 \*eip에 저장한다. 그리고 처음 stack pointer를 \*esp에 저장한다. 성공적으로 load 되었을 경우 true를 아닐경우 false를 return한다.

\* 사용 변수 :

int argc : argv의 argument저장하는 변수

char \*f\_name : filename의 자른 부분을 저장하는 변수

char \*ptr : strtok\_r을 사용할때 다음 처리를 위한 위치를 저장하는 pointer, 이 변수는 직접 handling하지 않는다.

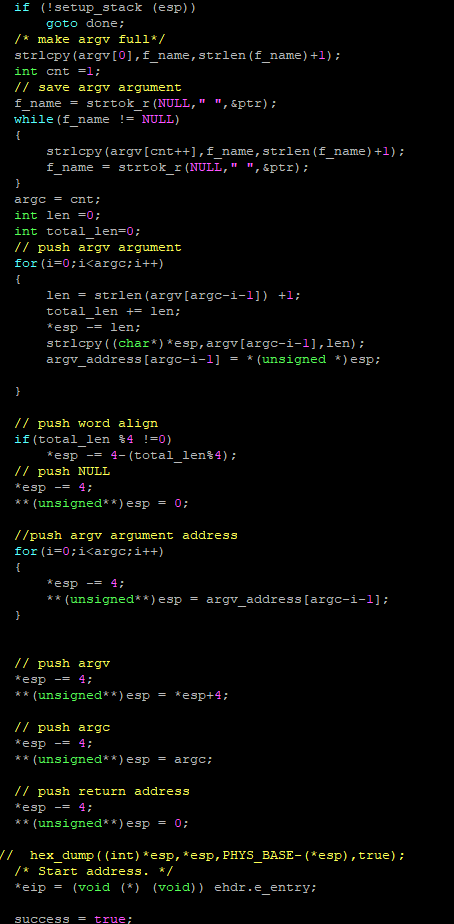
char filename[100] : parameter로 넘어온 file\_name 맨뒤에 NULL을 추가해준 변수

char \*\*argv : paramteter로 넘어온 명령어를 strtok\_r해서 나눈 argument들을 저장하는 변수

int argv\_address[100] : argv의 argument들이 저장된 배열의 주소를 저장하는 변수

\* 내 용 : parameter로 넘겨받은 file\_name을 strtok\_r을 이용하여 자른 뒤 맨 처음에 존재하는 문자열을 f\_name에 저장하고 f\_name의 excutable file을 open한다. open에 실패할 경우 open failed을 출력한다.

(2)



\* 사용 변수

int cnt : argv배열에 filename을 토큰화하여 저장할때 저장개수를 count하는 변수

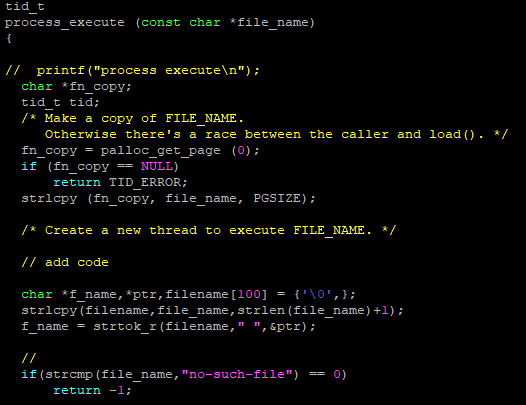
int len : argv 배열을 돌면서 argument들의 길이를 구해서 저장하는 변수

int total\_len : argument들의 총 길이를 저장하는 변수

\* 내 용 : 위에서 자른 f\_name을 argv[0]에 저장하고 계속 잘라나가면서 argv[]에 저장한다. argv[]에 저장된 내용들을 하나하나 읽으면서 stack에 쌓아준다 argv[]를 다 쌓고 나면 데이터가 4의 배수가 맞는지 아닌지를 확인하고 아닐경우 word align을 해준다. 그 다음 NULL을 쌓고 argv의 argument들이 저장되어 있는 address를 쌓는다. 다 쌓고 난 후에 argv의 address를 쌓고 argument의 개수를 의미하는 argc를 쌓은다음 마지막으로 return address를 쌓아준다.

2. User Memory Access

**→ tid\_t process\_execute(const char \*file\_name) [pintos/src/userprog/process.c]**

****

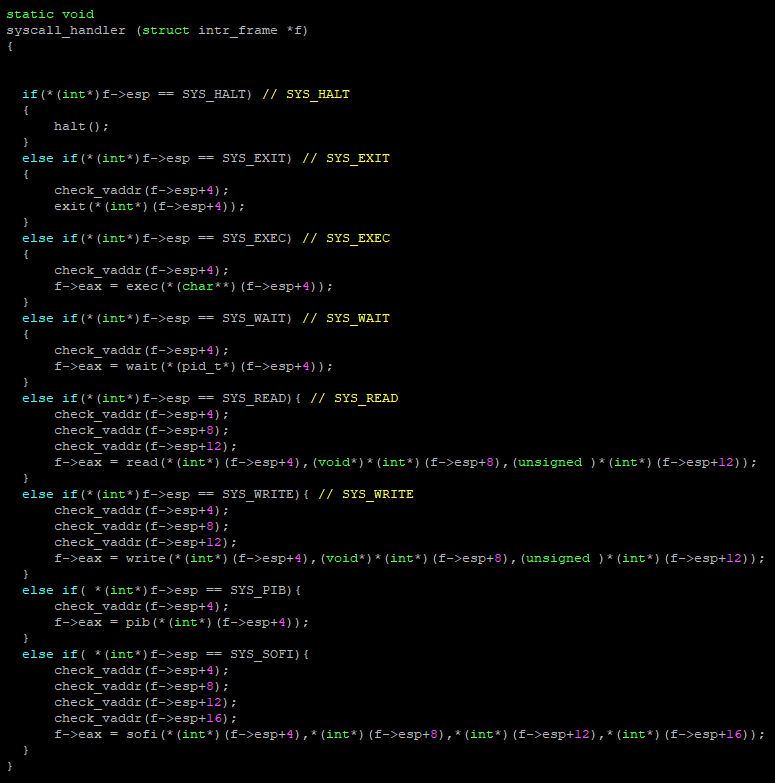
\* 함수 설명 : filename으로 부터 load된 user program을 실행하는 새로운 thread를 시작한다. 새로운 thread는 process\_execute()가 return되기 전에 schedule된다. 함수가 끝난 후에는 new process의 thread id가 return되고, thread가 생성되지 않았을 경우 TID\_ERROR가 return된다.

\* 사용 변수 :

const char \*file\_name : parameter로 명령어를 받는다

\* 내 용 : file\_name에 no-such-file이 저장되어 있을 경우 return -1을 해준다.

**→ static void syscall\_handler(struct intr\_frame \*f) [pintos/src/userprog/syscall.c]**

****

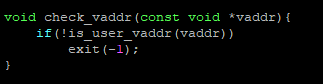
\* 함수 설명 : system call number에 해당하는 system call을 진행한다

\* 사용 변수 :

struct intr\_frame \*f : system call을 진행할 때 접근할 stack pointer를 갖고 있는 변수

\* 내 용 : system call을 진행할때 f에 저장된 stack pointer를 이용하여 memory에 접근하는데 memory에 접근하는 pointer들이 valid address인지 check\_vaddr함수를 이용하여 확인한다.

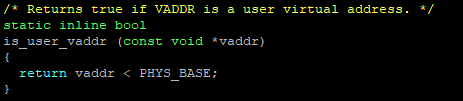
**→ void check\_vaddr (const void \*vaddr) [pintos/src/userprog/syscall.c]**

****

\* 함수 설명 : parameter로 넘어온 \*vaddr을 검사한다.

\* 사용 변수 : 없음

\* 내 용 : \*vaddr이 user memory를 가리키고 있지 않을 경우 exit(-1)을 한다.

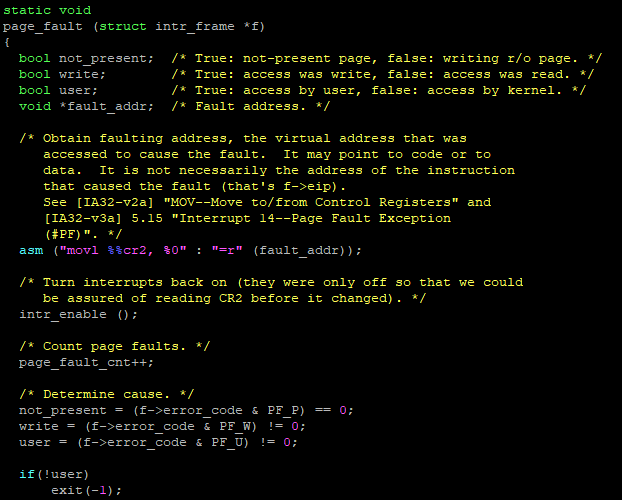
**→ static inline bool is\_user\_vaddr(const void \*vaddr) [pintos/src/thread/vaddr.h]**

\* 함수 설명 : parameter로 넘어온 \*vaddr을 검사한다.

\* 사용 변수 : 없음

\* 내 용: vaddr이 user memory와 kernel memory의 경계인 PHYS\_BASE 보다 아래에 있으면 user memory를 가리키고 있으므로 true를 return한다.

**→ static void page\_fault(struct intr\_frame \*f) [pintos/src/userprog/exception.c]**



\* 함수 설명 : page가 제대로된 주소를 찾아가는지 확인하는 함수이다. pintos os에서 기본적으로 제공하는 함수이다.

\* 사용 변수 :

bool user : True라면 user program memory에 접근한 것이고, False라면 kernel program memory에 접근한다는 것을 저장하는 flag이다.

\* 내 용 : 우리는 이 함수에 if(!user) exit(-1); 문구를 추가하였다. 이는 user memory에 접근한 것이 아니고 kernel memory에 접근했을 경우를 뜻하기 때문에 잘못된 메모리 접근으로 exit(-1)를 시켰다.

3. System Call

**→ void halt(void) [pintos/src/userprog/syscall.c]**

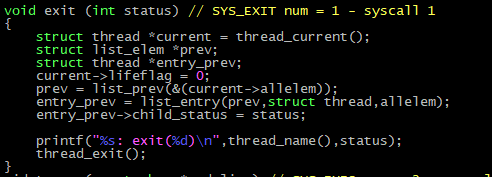


\* 함수 설명 : system call이 halt 인 경우 실행되는 함수이다. power off를 위하여 pintos/src/devices/shutdown.c에 있는 shutdown\_power\_off() 함수를 호출한다. shutdown\_power\_off() 함수는 우리가 실행하고 있는 machine을 poweroff 해주는 함수인데, 이는 pintos에서 기본적으로 제공하는 함수이다.

\* 사용 변수 : 없음

\* 내 용 : power off를 위하여 pintos/src/devices/shutdown.c에 있는 shutdown\_power\_off() 함수를 호출한다.

**→ void exit(int status) [pintos/src/userprog/syscall.c]**



\* 함수 설명 : thread가 exit이 필요한 경우 호출되는 함수이다.

\* 사용 변수

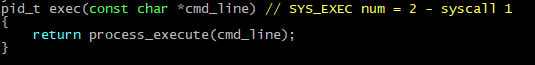
struct thread \*current : 현재의 실행되고 있는 thread(구조체)를 담고있다.

struct list\_elem \*prev : 현재 실행되고 있는 thread의 부모 thread linked\_list를 저장한다.

struct thread \*entry\_prev : 현재 실행되고있는 thread의 부모 thread(구조체)를 저장한다.

\* 내 용 : 현재 execute 되고있는 thread를 exit시키기 전에 thread 구조체 안의 몇가지 변수를 변경시켜준다. 현재 thread의 lifeflag 변수를 0으로 바꾸어주고, 부모 thread의 구조체에 exit status를 저장시킨 후에 thread\_name과 exit status를 출력하고 thread\_exit() 함수를 호출한다.

**→ pid\_t exec(const char \*cmd\_line) [pintos/src/userprog/syscall.c]**



\* 함수 설명 : 호출한 system call을 execute해주는 함수이다. 즉 system call이 exec인 경우 호출되는 함수이다.

\* 사용 변수 : 없음

\* 내 용 : process\_execute() 함수를 호출하는데, 이 함수는 argument passing에서 미리 설명했다.

**→ int wait(pid\_t pid) [pintos/src/userprog/syscall.c]**



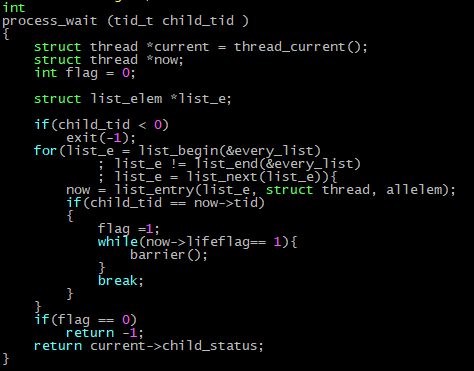
\* 함수 설명 : thread 마다의 고유번호인 thread id를 parameter로 받아 해당하는 thread를 wait시켜주는 함수이다. system call이 wait인 경우 호출된다.

\* 사용 변수 : 없음

\* 내 용 : process\_wait()함수를 호출한다. 이 함수는 바로 아래에서 다시 설명하겠다.

**→→ int process\_wait(tid\_t child\_tid)**

**[pintos/src/userprog/process.c]**



\* 함수 설명 : 실질적으로 system call이 wait일 때 실행되는 함수이다. thread wait가 일어나기 위해서는 반드시 필요한 함수이고, thread 상호간에 실질적인 동기화가 일어나는 함수이다.

\* 사용 변수 :

struct thread \*current : 현재의 실행되고 있는 thread(구조체)를 담고있다.

struct thread \*now : parameter로 전달된 child\_tid를 thread id로 가지는 thread(구조체)를 담고있다.

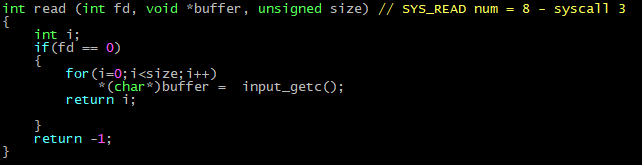
int flag : parameter로 전달된 child\_tid를 thread id로 가지는 thread가 존재하면 1, 존재하지 않으면 0으로 설정된다. 그래서 함수 마지막에 flag == 0 이면 -1을 return해주는 조건문이 존재한다.

struct list\_elem : thread의 linked\_list로써 thread를 검색할 때 사용한다.

\* 내 용 : 자식 thread의 thread id를 parameter로 제공받아 now라는 변수에 저장한다. 그리고 자식 thread가 종료되었는지 알 수 있는 flag인 lifeflag를 계속적으로 확인하면서 자식 thread가 끝나기 전까지 계속적으로 확인한다. 그리고 자식 thread가 종료되면 lifeflag가 0으로 설정되므로 loop문을 벗어나게 되고, 그때 부모 thread의 wait가 끝나게 된다. 이때 barrier()를 사용하는데, 이는 명세서에서 busy wait 부분을 참고하였다. barrier()는 해당 process의 실행을 기다리는 기능을 가진다. 따라서 loop 문 안에서는 계속 barrier()를 사용하면서 , loop문이 끝나면, 즉 자식 thread의 execute가 끝나면(종료되면) 부모 thread의 wait이 끝나는 것이다.

**→ int read(int fd, void \*buffer, unsigned size)**

**[pintos/src/userprog/syscall.c]**



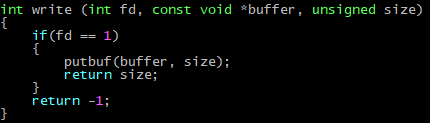
\* 함수 설명 : 입력값을 buffer에 저장하는 함수

\* 사용 변수 : 없음

\* 내 용 : fd가 0일경우 input\_getc()함수를 이용하여 keyboard로 부터 글자를 입력받아서 buffer에 저장하고 입력받은 글자의 총 길이인 i를 return한다.

**→ int write(int fd, const void \*buffer, unsigned size)**

**[pintos/src/userprog/syscall.c]**

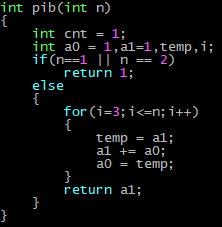


\* 함수 설명 : console창에 buffer에 저장된 값을 출력하는 함수

\* 사용 변수 : 없음

\* 내 용 : fd가 1일 경우 putbuf를 이용하여 buffer가 가리키고 있는 곳에서 부터 size만큼의 내용을 출력하고 size를 return한다.

**→ int pib(int n) [pintos/src/userprog/syscall.c]**



\* 함수 설명 : 피보나치 수열을 수행하는 함수이다. system call에서 pib를 실행 시키는 경우 호출된다.

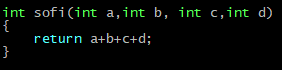
\* 사용 변수

int cnt, a0, a1, temp, i : 단순히 피보나치 수열을 구하기 위하여 설정한 변수이다.

\* 내용 : 재귀함수로 구현하였으며 parameter로 받은 n번째 수열을 return한다.

**→ int sofi(int a, int b, int c, int d)**

**[pintos/src/userprog/syscall.c]**



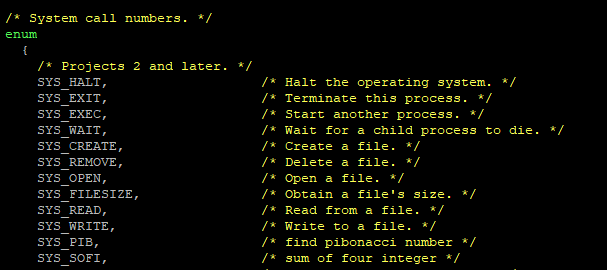
\* 함수 설명 : parameter로 받아온 a, b, c, d 정수형 변수를 모두 더해 return 해주는 함수이다. system call 중 sofi를 호출할 때 사용된다.

\* 사용 변수 : 없음

\* 내 용 : 없음

4. Additional Implementation

**→ enum [pintos/src/lib/syscall-nr.h]**

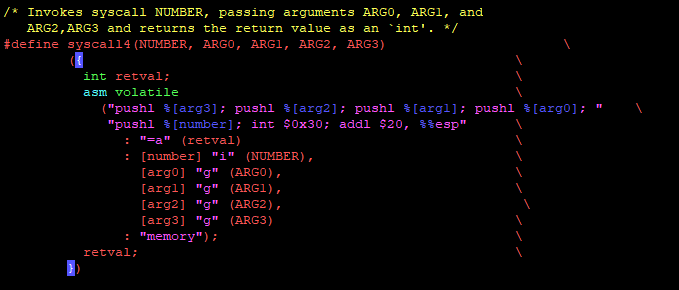
****

**\*** 변수 설명 : system call number를 저장하고 있는 enum타입의 변수

\* 사용 변수 : 없음

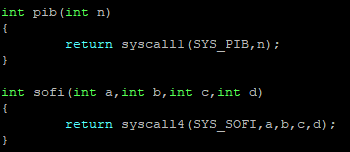
\* 내 용 : 기존에 존재하지 않는 SYS\_PIB랑 SYS\_SOFI를 호출하는 system call number를 추가하였다

**→ #define sysc all4(NUMBER, ARG0, ARG1, ARG2, ARG3) [pintos/src/lib/user/syscall.c]**

****

\* 내 용 : 기존에는 argument가 4개인 system call을 처리할 수 없었는데 새롭게 만들어서 네 가지 변수를 더하는 system call을 처리할 수 있게 한다.

**→ int pib(int n) / int sofi(int a,int b,int c,int d) [pintos/src/lib/user/syscall.c]**

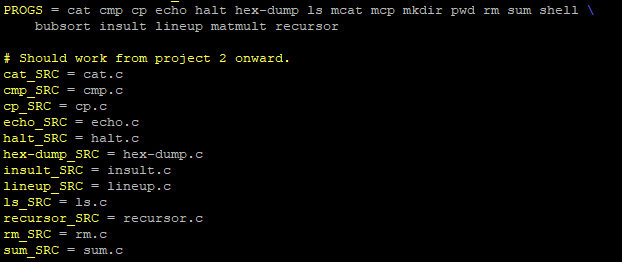
****

\* 함수 설명 : system call이 실행될 경우 memory에 올린다.

\* 사용 변수 : 없음

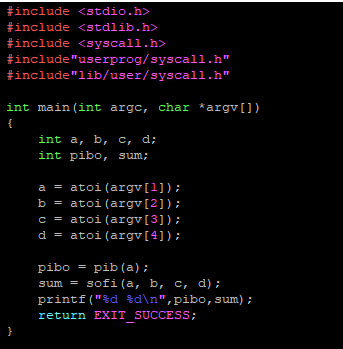
\* 내 용 : system call은 argument의 개수에 따라 다르게 다르게 호출하는데 pib같은 경우는 argument가 1개이므로 syscall1을 호출하고 sofi는 argument가 4개이므로 syscall4를 호출한다.

**→ Makefile [pintos/src/example/]**

****

\* 내 용 : 기존의 Makefile은 sum.c에 해당하는 실행파일과 sum.o sum.d를 생성하지 못하였지만 Makefile의 PROGS에 sum을 추가하고 sum\_SRC = sum.c를 추가하여 sum실행파일과 sum.o sum.d가 오류없이 정상적으로 생성되게 하였다.

**→ sum.c [pintos/src/example/]**

****

\* 함수 설명 : sum명령어를 실행하였을 때의 실행파일을 생성한다.

\* 사용 변수 :

int a,b,c,d : argv에 저장된 argument를 저장하는 변수이다.

int pibo : pib함수의 return 값을 저장한다.

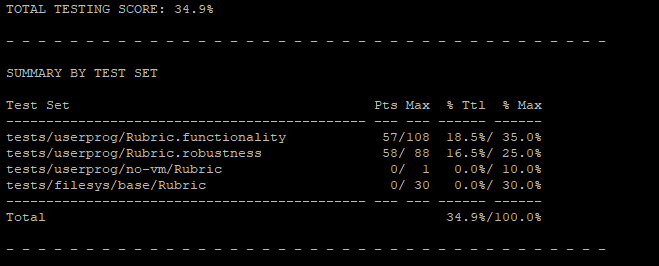
int sum : sofi함수의 return 값을 저장한다.

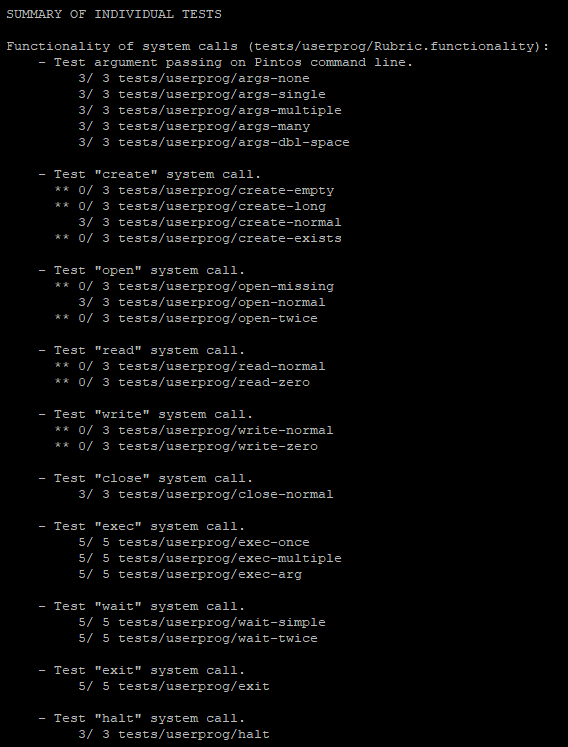
\* 내 용 : sum명령어를 실행하였을 경우 memory에 올라가져 있는 argument들과 argc를 parameter로 받아서 atoi함수를 이용하여 문자열에서 int형으로 변환하고 pib함수와 sofi함수에 넣어서 적절한 return값을 받은후 출력한다. 정상적으로 끝나면 EXIT\_SUCCESS를 return한다.

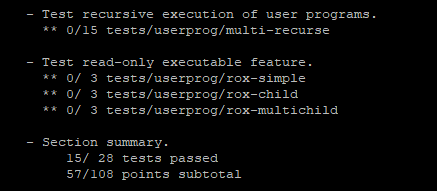
-

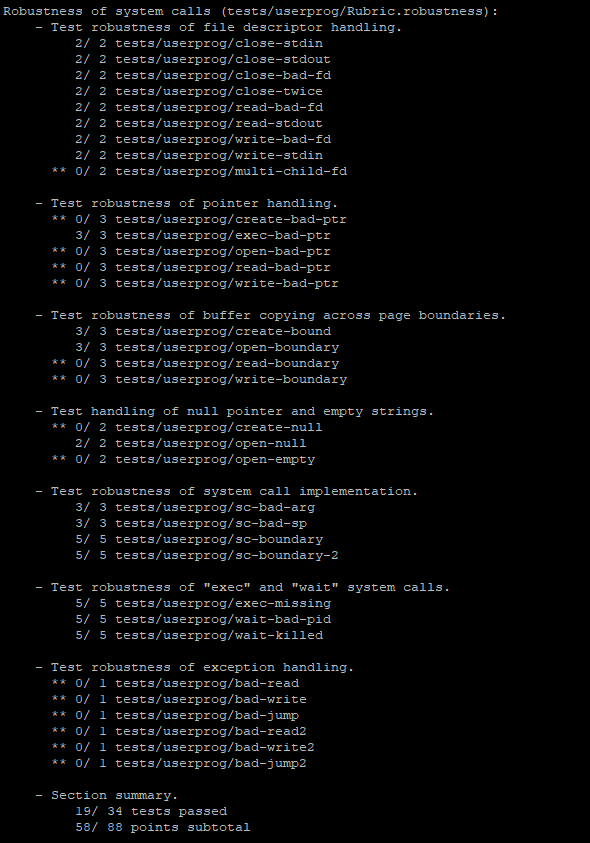
**C. 시험 및 평가 내용**

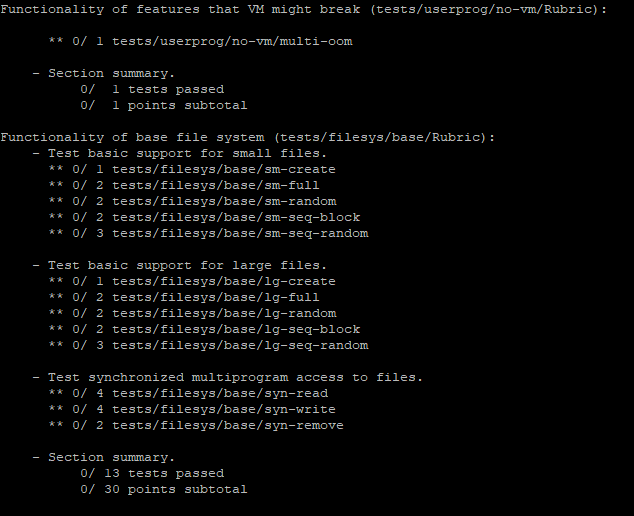
**make grade**

****

****

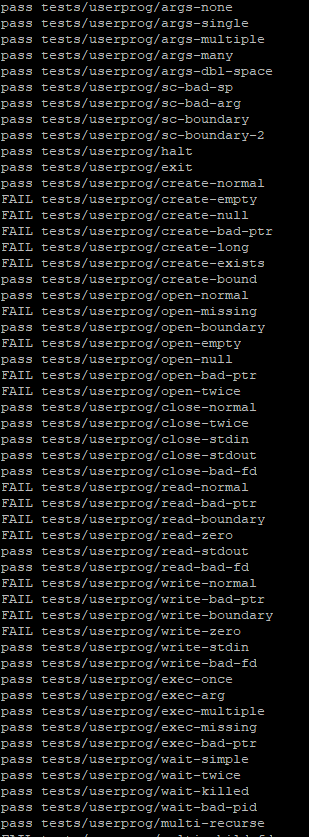
****

****

****

-  ****

**[make check]**



**V. 기타**

**A. 연구 조원 기여도**

-  **김건우 50% / 변재승 50%**

**B. 소감**

**김건우**

**pintos의 기본 구조를 이해하는 것이 굉장히 어려웠다. 처음 프로젝트를 보고 명세서를 확인했을때에는 어디부터 접근해야 하는지, 무슨 코드부터 확인해야하는지 너무 막막했다. 일단 명세서와 매뉴얼을 천천히 읽어보았다. 명세서와 매뉴얼을 읽어보니 그때서야 제대로 무엇을 하는 프로젝트인지 이해가 되었다. 무엇을 하는 프로젝트인지 이해를 하고 나서부터는 난해하고 새로운 개념을 계속 반복해서 공부하고 직접 print해보며 실행해보았다.**

**팀원과 함께 진행하는 것에는 어려움이 없었다. 맡은 부분에 대해서 서로 설명해가며 틀린 부분이 있는지, 잘못 이해한 부분이 있는지 확인하면서 서로에게 좋은 시너지 효과를 낼 수 있었다. 서로의 소통을 통해서 thread 대해서 이해하게 되었다. 개인 프로젝트보다 팀 프로젝트로 진행하다 보니 개인 프로젝트보다 더 즐겁게 진행할 수 있었고 훨신 수월했다.**

**이번 프로젝트를 진행하며 os 내부에서 일어나는 일을 실제로 확인해보고 loading과 몇가지 system call을 구현하면서 확실하게 thread에 대해서 깊은 지식을 가지게 되었고 운영체제에 대한 기반을 다졌다는 느낌이 굳게 들었다. 다음 프로젝트는 system call을 몇개 더 추가하고 진행하는 것으로 알고 있는데, 이번 프로젝트가 아주 큰 도움이 된 것 같다.**

**변재승**

**맨 처음 프로젝트에 접했을 때, 하나도 이해하지 못했지만 수업시간에 교수님이 주신 힌트들과 pintos 메뉴얼 그리고 학생들이 올린 질문을 조교님들이 답변해주신 것을 보고나서야 조금이나마 가닥을 잡을 수 있었다. 팀원과 함께 전체적인 틀에 대한 구상을 마치고 나서 부터 코딩을 시작하였는데, 구상에서와는 다르게 직접 코딩을 해보니 여러가지 장벽에 부딪혔다. 이러한 문제들을 해결하기 위해 인터넷에서 검색도 하고 팀원과 토론하여 결과를 도출하다 보니 어느샌가 프로젝트를 완성할 수 있었다. 아직 pintos의 완벽한 진행방식에 대해서는 잘 모르는 것 같아서 매우 궁금하고 앞으로 남은 프로젝트들로 인해서 이러한 궁금증을 해결할 수 있을 것 같다.**