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

**(설계 프로젝트 수행 결과)**

과목 명 : 운영체제

담당 교수 : 김영재

조 / 조원 : 72조 / 20141515 김연후, 20141362 천지환

개발 기간 : 10/5 ~ 11/2

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

**제출일 : 11/2일**

**참여 조원 : 20141515 김연후, 20141362 천지환**

1. **개발 목표**

* Pintos에서 명령어를 실행 시 System Call을 통해 주어진 명령어에 해당 하는 함수를 호출할 수 있게 User program 구동 상 기초적인 부분을 구현한다. 구현 시 크게 5가지 과정을 거친다. (Argument Passing, User Memory Access, System Call Handler, System Call Implementation, Additional Function Implementation)

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing
   * + - src/userprog/process.c

Pintos에서 명령어 실행 시 User program은 2개 이상의 arguments를 가질 수도 있다. 따라서 이 arguments를 parsing하는 과정이 필요하다. 그 후 ESP가 가리키는 stack에 push해준다. 이 때 parsing된 arguments를 stack에 push해주는 함수 construct\_esp(char \*file\_name, void \*\*esp)를 load()에 구현해준다.

1. User Memory Access
   * + - src/userprog/exception.c

user program의 명령어를 실행할 때 kernel의 영역을 참조하는 오류가 발생할 수 있다. 이 경우 Page\_fault()에서 user program이 kernel 영역을 참고하거나 현재 접근자가 kernel일 경우 예외처리를 해주어 exit(-1)을 호출해준다.

1. System Call Handler, Implementation
   * + - src/userprog/syscall.h , syscall.c

입력받은 명령어에 받는 system call을 실행해 주기 위해서 syscall\_handler(struct intr\_frame \*f)에서 f->esp에 저장되어 있는 syscall의 number을 이용해 system call을 분류해준다. 그 후 각 system call에 해당하는 처리를 실행해준다.

-System call 분류 및 처리

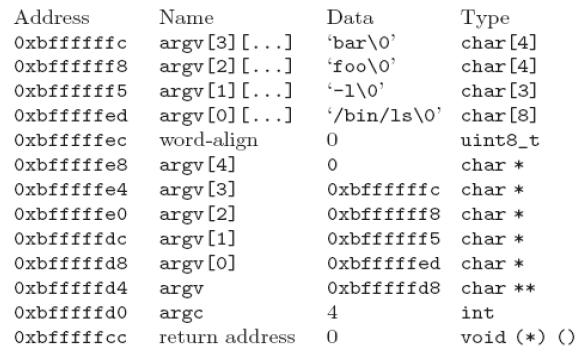
1. SYS\_HALT : Pintos를 종료시킨다.
2. SYS\_EXIT : thread의 이름과 exit\_status 출력 후 User program을 종료시킨다.
3. SYS\_EXEC : 호출된 process를 실행시킨다.
4. SYS\_WAIT : Parent process에서 child process가 종료할 때까지 대기해준다.
5. SYS\_READ : 현재 project에서는 standard input을 통해 입력받는 기능만을 수행한다.
6. SYS\_WRITE: 현재 project에서는 standard ouput을 통해 출력하는 기능만을 수행한다.
7. Additional Implementation
   * + - Project에서 요구하는 함수 2개를 추가로 구현해준다.
       1. int sum\_of\_four\_integer(int a, int b, int c, int d) : 입력받은 4개의 정수의 합을 반환해준다.
       2. int Fibonacci(int n): 입력받은 n번째 피보나치 수를 구해서 반환해준다.
   1. **개발 내용**

프로젝트를 진행하기 위해 사용할 기술에 대해 서술

1. Argument Parsing

User program의 argument가 2개 이상일 수 있기에 argument를 parsing해준 뒤 ESP에 해당하는 stack으로 가서 arguments를 push해준다. Project 명세서에 나온 예시 ‘/bin/ls –1 foo bar’를 이용해서 설명하면

해당 arguments를 ‘/bin/ls’, ‘-1’, ‘1’, ‘foo’, ‘bar’ argument를 parsing하고 stack에 다음과 같이 push해준다.



우리가 프로그래밍과정에서 사용하는 ‘echo x’를 이용해서 설명하면 해당 arguments를 ‘echo’, ‘x’로 parsing 한 후 stack에 다음과 같이 push해준다.

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Name | Data | Type |
| 0xbffffffe | argv[1][…] | ‘x\0’ | char[2] |
| 0xbffffff9 | argv[0][…] | ‘echo\0’ | char[5] |
| 0xbffffff8 | word-align | 0 | uint8\_t |
| 0xbffffff4 | argv[2] | 0 | char \* |
| 0xbffffff0 | argv[1] | 0xbffffffe | char \* |
| 0xbfffffec | argv[0] | 0xbffffff9 | char \* |
| 0xbfffffe8 | argv | 0xbfffffec | char \*\* |
| 0xbfffffe4 | argc | 2 | int |
| 0xbfffffe0 | return address | 0 | void(\*)() |

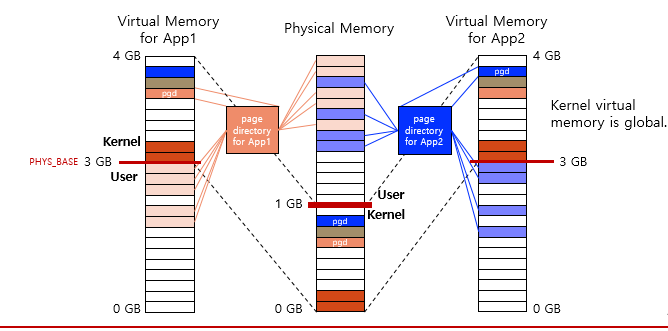
|  |
| --- |
| Stack push 순서 |
| 1. parsing 된 arguments |
| 2. word-align |
| 3. NULL |
| 4. parsing 된 arguments의 address |
| 5. argv[0]의 address |
| 6 argument의 개수(명령어 포함) |
| 7. return address |

Stack에 argument들을 push해주는 순서와 내용은 위와 같다. 이 때, 2번의 word-align은 데이터의 접근 속도를 빠르게 하기 위해 크기를 4의 배수로 맞춰주기 위해 0을 추가하는 과정이다. 4의 배수가 되기 위해 필요한 수만큼 0을 추가해준다. 3번의 NULL의 경우 C standard에 필요한 형식이다.

1. User Memory Access

(process,page의 개념, User process가 kernel memory에 접근 방지 방법)

-process: 실행중에 있는 프로그램을 의미하고, scheduling의 대상이 되는 task와 같은 의미로 쓰인다. 프로세스 내부에는 최소 하나의 thread가 존재하고 실제 scheduling은 이 thread단위로 진행된다. 디스크에 있는 프로그램을 실행하면 메모리 할당이 이루어지고 그 메모리 공간으로 코드가 올라가는데 이 순간부터 process라고 불린다.

- page: 컴퓨터가 paging이라는 메모리 관리 기법을 통해서 메인 메모리를 사용한다. 이때 virtual memory를 모두 같은 크기의 블록으로 편성하여 운용하는데 이때의 일정한 크기의 블록을 page라고 한다.

* + - * Virtual memory와 Physical memory의 구조 및 연결관계는 위의 그림과 같다. 이때 virtual memory의 3~4GB는 Kernel memory, 0~3GB는 User memory가 할당 되고, user은 kernel memory를 접근하면 안된다. 이 접근을 방지하기 위해서 src/userprog/exception.c 의 page\_fault() 함수에서 user가 kernel을 침범하거나 user program인데 현재 접근자가 kernel일 때 예외 처리를 해준다. 예외 처리 구문은 아래와 같다.

|  |
| --- |
|  |
|  |

1. System Call Handler, Implementation

입력받은 명령어에 받는 system call을 실행해 주기 위해서 syscall\_handler(struct intr\_frame \*f)에서 f->esp에 저장되어 있는 syscall의 number을 이용해 system call을 분류해준다. 그 후 각 system call에 해당하는 처리를 실행해준다.

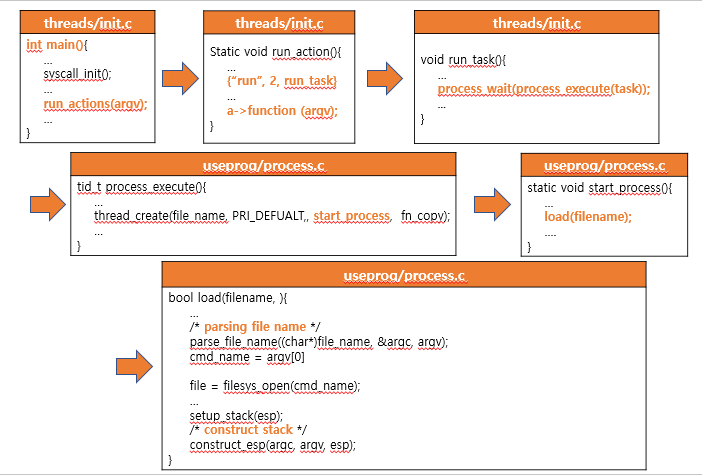
|  |
| --- |
| System number을 통한 분류 |
|  |

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 기간 | 구현 내용 |
| 2019.10.10.~20 | Pintos ppt 및 Pintos Manual 이해 |
| 2019.10.21~25 | 시험기간 |
| 2019.10.26.~29 | Pintos에 구현되어 있는 기본 코드 이해 |
| 2019.10.29 | argument passing 구현 |
| 2019.10.30 | system call 함수 구현 |
| 2019.10.31 | wait구현 및 make check를 통한 코드 수정 및 오류 체크 |
| 2019.11.01. | 추가함수(Fibonacci, sum\_of\_four\_interger) 구현 |
| 2019.11.02 | Pintos 보고서 작성 |

* 1. **개발 방법**

1. Argument Passing

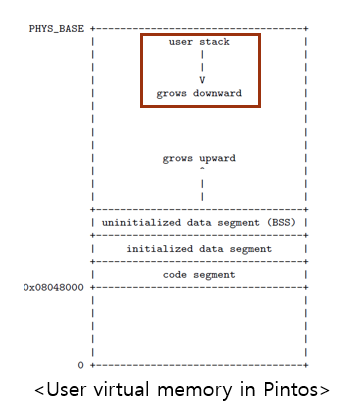
* User program에서 입력받은 명령어와 argument를 실행하는 과정은 아래 그림과 같다.
* 위의 그림과 같이 User가 명령어와 arguments를 입력하면 program은 입력받은 명령어와 argument를 passing하기 위해 run\_action()을 호출한다. 그 후 run action() 에서 run\_task가 실행되고 입력받은 명령어에 대한 process 호출을 위해서 process\_execute가 호출된다. process\_execute에서 입력받은 file\_name의 thread를 새로 만들어주기 위해 thread\_create가 호출되는데 이때 start\_process 실행된다. 그 후 호출된 start\_process에서 load()를 호출한다.

|  |
| --- |
| Stack push 순서 |
| 1. parsing 된 arguments |
| 2. word-align |
| 3. NULL |
| 4. parsing 된 arguments의 address |
| 5. argv[0]의 address |
| 6 argument의 개수(명령어 포함) |
| 7. return address |

이때 load()에서는 명령어와 arguments를 parsing해주는 함수를 만들어 수행해주고 이를 ESP가 가리키고 있는 stack에 쌓아준다.

1. parse\_file\_name() : parsing할 문자열과 parsing한 문자열을 저장할 argv, parsing한 문자열의 수를 저장할 argv를 input 값으로 받는다. user가 입력한 argument들을 parsing하는 과정을 수행하는 함수로 Parsing한 명령어 및 argument들은 argv(이중 포인터)에 문자열 형태로 순차적으로 저장되며 parsing된 argument의 수는 argc에 저장된다.(명령어 포함) 그리고 argv를 return 해준다.
2. construct\_esp(): argument를 쌓을 stack을 가리키는 esp와 parse\_file\_name을 통해 구한 argv와 argc를 input값으로 받는다.

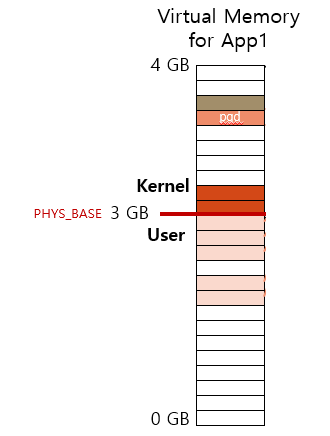
esp가 가리키는 stack에 오른쪽 표의 순서로 value들을 push해주는 함수이며 value들을 push 할때에는 아래 그림과 같이 위에서 아래로 쌓인다. 따라서 value가 쌓일 때 마다 esp에 +가 아닌 -값을 취해준다.

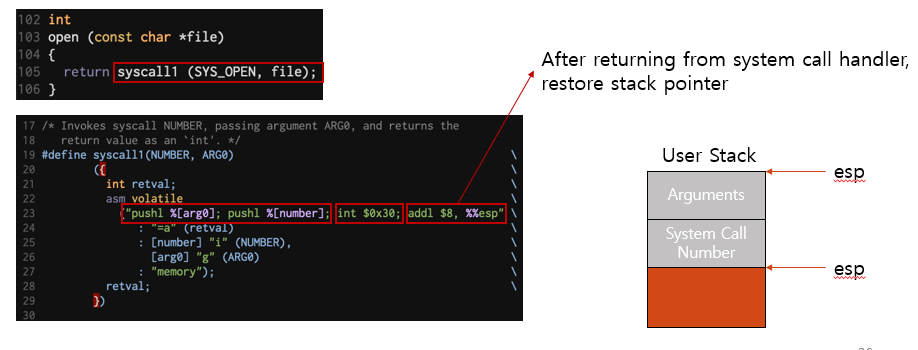


위의 표의 내용을 그림과 같이 downward 형태로 stack에 push 해줄 때 esp값 변화는 아래와 같이 해준다.

1. Parsing된 argument들을 차례로 넣어주는데 이때 stack의 memory공간 확보를 위해 \*esp에서 ‘parsing된 argument의 길이 + 1(NULL 포함)’의 값만큼 빼준다. 그 후 argument를 push한다.
2. 데이터의 접근 속도를 빠르게 하기 위해 word-align을 추가해 크기를 4의 배수로 맞춰준다. 이를 위해 필요한 word-align의 수만큼 \*esp의 값에서 빼준다. 그후 0을 필요한 수 만큼 push한다.
3. C standard에 맞추어 argument의 값과 주소들을 구분하기 위해 \*esp에서 4를 빼준후 NULL 값을 PUSH한다.
4. 각 argument의 값들이 저장되어 있는 주소를 stack에 넣어야한다. 이때 주소값들은 모두 4 byte이므로 \*esp에 4byte를 빼고 argument의 주소값 memcpy()를 통해 stack에 push하는 과정을 argument의 수인 argc만큼 반복한다.
5. argv의 첫번째 address를 stack에 넣어주는 과정이다. 4번과 마찬가지로 주소값은 4 byte이기에 \*esp에서 4를 빼준후 argv의 첫 주소를 push해준다. 이때 argv의 첫 주소는 \*esp+4의 값이므로 이 값을 push해주면 된다.
6. argc의 값도 stack에 넣어주기 위해서 4 byte가 필요하기에 \*esp에서 4를 빼준후 argc의 값을 넣어준다.
7. Return address의 값 또한 stack에 넣어주기 위해 4 byte가 필요하기에 \*esp에서 4를 빼준 후 0을 넣어준다.
8. User Memory Access

-일단 기본적으로 program 실행 도중 page fault가 발생할 경우 src/userprog/exception.c에 있는 page\_fault() 함수가 발생하여 오류 체크를 해준다.

여기에 추가하여 virtual memory상에서의 kernel의 memory는 오른쪽의 그림과 같이 3GB~4GB이고 user의 memory는 0GB~3GB인데 user가 kernel address로 접근하려 하거나 현재 program은 user program인데 접근자가 kernel로 뜰 경우 exit(-1)을 해준다.

1. System Call Handler, Implementation

위의 그림에서 알 수 있듯이 userprog/syscall.c의 syscall\_handler에서 input으로 받는 intr\_frame \*f의 f->esp에서 system call의 number를 알 수 있다. 이를 이용해 아래 그림과 같이 switch로 case를 나눠 각 case에 맞는 function을 호출해준다. 이 때 함수의 return 값이 존재할 경우 Pintos Manual에 적혀 있는 대로 f->eax에 저장해준다.

|  |
| --- |
| System number을 통한 분류 |
|  |

1. SYS#1. SYS\_HALT

argument값이 system call number만 있는 syscall0에 해당한다. shutdown\_power\_off() 함수를 통해 Pintos를 종료시켜준다.

1. SYS#2. SYS\_EXIT

argument값이 system call number를 포함해 2개가 존재하는 syscall1에 해당한다. 이 때 추가로 받는 argument는 exit status 값이다. 이를 현재의 thread의 exit\_status인 thread\_current()->exit\_status에 넣어준 후 exit 양식에 맞게 printf를 실행하고 thread\_exit()을 호출한다. (이 때 exit status의 값이 음수이면 thread\_current()->exit\_status의 값을 -1로 저장한다.)

1. SYS#3. SYS\_EXEC

argument값이 system call number를 포함해 2개가 존재하는 syscall1에 해당한다. 이 때 추가로 받는 argument는 실행시킬 process의 이름이다.이 이름에 해당하는 process를 process\_execute()를 통해 실행시켜준다.

1. SYS#4. SYS\_WAIT

argument값이 system call number를 포함해 2개가 존재하는 syscall1에 해당한다. 이 때 추가로 받는 argument는 기다릴 child의 pid이다.

pid값을 갖는 child에 대해 process\_wait() 함수를 호출한다. 이를 통해 child함수가 종료할 때까지 대기하고 child process가 종료되면 parent process를 진행해준다.

(process\_wait은 parent process가 child process 완료 전에 종료되는 것을 막기 위한 함수이다. Input 값으로 child thread의 id를 받는다. 이 id와 list의 함수를 이용해 입력받은 tid에 해당하는 child thread를 찾는다.

그 후 semaphore을 이용해 child가 끝날 때 까지 기다려준다. (sema\_down()) child thread가 종료 되면 process\_exit()에서 sema\_up()) child thread가 종료 된 뒤 이 항목을 list\_remove를 통해 child list에서 지워줘야하는데 이때에 child thread가 종료 되었을 때 list\_remove를 할 메모리가 남아있는지 확실히 알수 없기에 memory lock을 위해 semaphore을 한번 더 이용해준다.(process\_exit() 마지막 부분에 sema\_down()함수를 추가하고 process\_wait의 list\_remove가 완료 된 뒤에 sema\_up())

child thread가 종료되었을 경우 child thread의 exit\_status값을 반환해주고 입력받은 child thread의 id에 해당하는 child를 찾기 못하였을 경우 (-1)을 반환해준다.).

1. SYS#8. SYS\_READ

argument값이 system call number를 포함해 3개가 존재하는 syscall2에 해당한다. 이 때 추가로 받는 argument는 read를 위해 file open을 실행할 fd, read 한 data를 저장할 buffer, read할 size이다.

먼저 buffer의 위치가 user memory가 맞는지 확인하고 kernel memory일 경우 exit(-1)해준다. 그 후 fd가 0의 값이라면(fd=0은 standard input을 가리키고 현재 project에서는 standart input만 구현) size만큼 input\_getc()를 통해서 읽어온 data를 저장한다. 주어진 size만큼 buffer에 저장하는데 성공 하였을 경우 size를 return하고 나머지 경우 -1을 return한다.

1. SYS#9. SYS\_WRITE

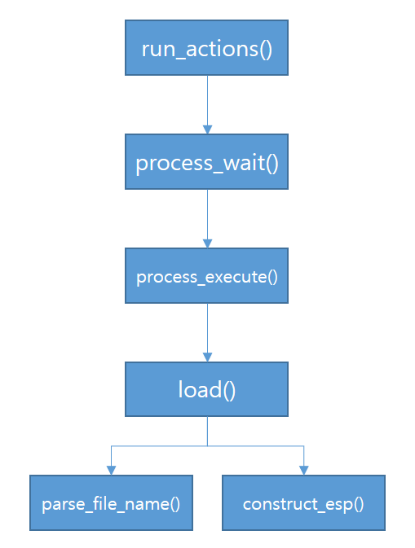
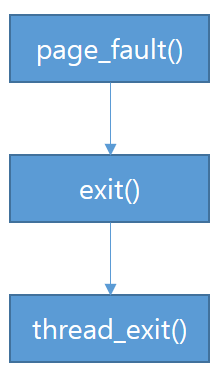
argument값이 system call number를 포함해 3개가 존재하는 syscall2에 해당한다. 이 때 추가로 받는 argument는 wirte을 위해 file open을 실행할 fd, write할 data가 있는 buffer, write할 size이다.

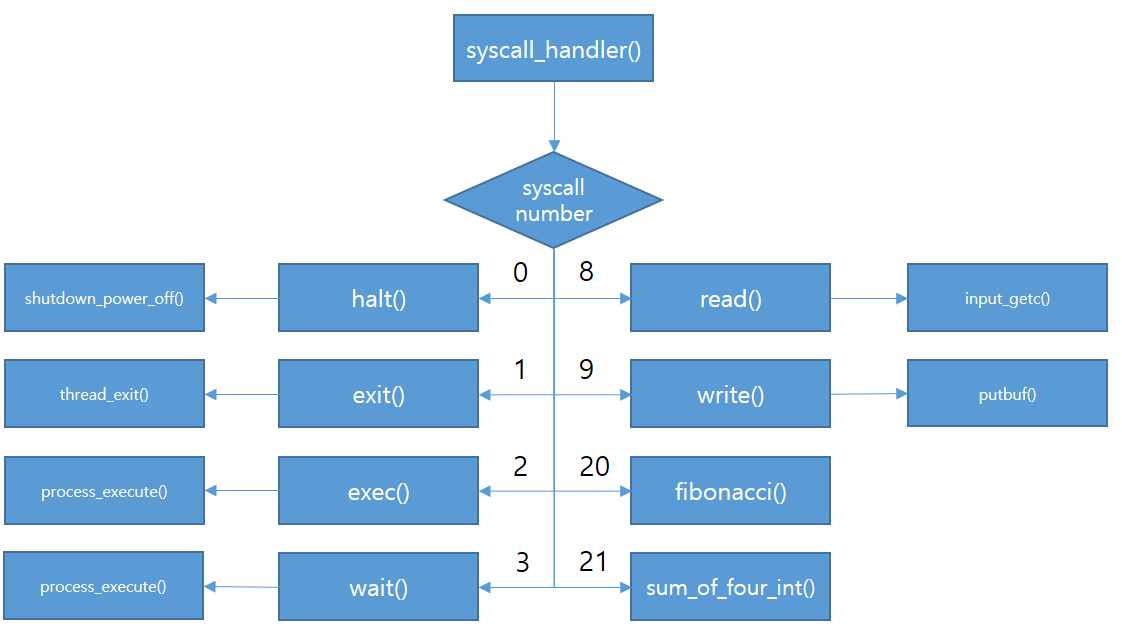
먼저 buffer의 위치가 user memory가 맞는지 확인하고 kernel memory일 경우 exit(-1)해준다. 그 후 fd가 1의 값이라면 (fd=1은 standard output을 가리키고 현재 project에서는 standart output만 구현) putbuf를 통해 size만큼 buffer의 값을 standard output으로 write해주고 size를 반환한다. 그 외의 경우 -1을 반환한다.(ppt 상 input\_putc를 이용하라 되어있지만 이 경우 priority queue에 해당하는 정보를 알아야되는데 이가 확실치 않아 Pintos Manual에 쓰여져 있는 데로 putbuf()를 이용한다.)

1. Additional Implementation
2. fibonacci(): Fibonacci 점화식이 fn = fn-1+ fn-2 (n은 0이상의 정수) 이므로 for 반복문을 통해 n번째의 피보나치 수를 구한 후 그 값을 반환하도록 구현한다.
3. sum\_of\_four\_integer(): 입력받은 4개의 값의 합을 구해 반환하도록 구현한다. (현재 syscall의 argument는 4개까지인데 sum\_of\_four\_integer은 명령어와 숫자4개로 argument가 5개이므로 해당하는 syscall을 새로 define해준다.)
   1. **연구원 역할 분담**

**각자 코드를 구현한 다음 더 나은 코드를 제출하고 보고서는 함께 작성하였다.**

1. **연구 결과**
   1. **합성 내용**

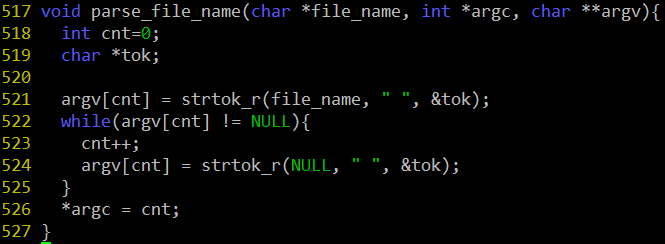
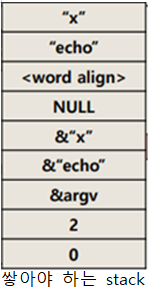
* Argument Passing
* User Memory Access

* System Call ( + Additional system calls)
  1. **합성 내용**

**Argument Passing**

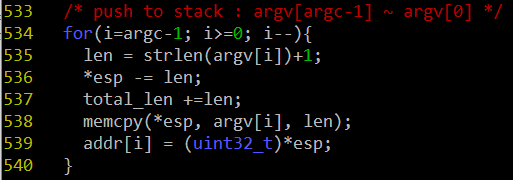
file\_name이 process\_execute() 함수, thread\_create() 함수 및 start\_process() 함수를 거쳐 load() 함수로 들어오면 file\_name을 cmd\_name과 argv 문자열들로 띄어쓰기 기준으로 parsing을 해주고 이들을 이용하여 user stack을 만들어 주는 construct\_ESP() 함수에 사용하였다.

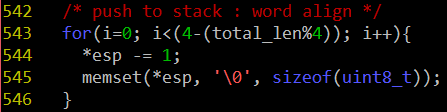
file\_name을 parsing 한 방법은 parse\_file\_name(char \*file\_name, int \*argc, char \*\*argv) 함수를 아래와 같이 구현하였다.

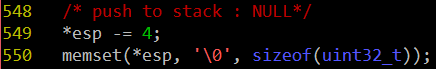


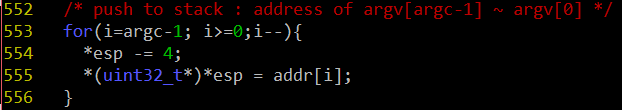
이 함수를 이용하여 구한 file\_name에서 나온 argument 개수 (argc) 와 각 argument들의 정보를 construct\_ESP(int argc, char \*\*argv, void \*\*esp) 함수에서 이용하여, esp부터 stack에 쌓아야 할 정보들을 우측과 같은 순서로 차례차례 쌓았다.

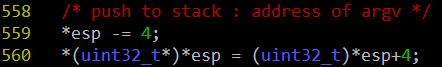
아래와 같은 방식으로 argument들이 stack에 push 되도록 구현하였다.

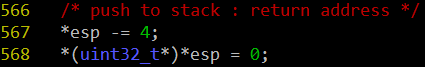
Push arguments at the top of the stack

Word Align

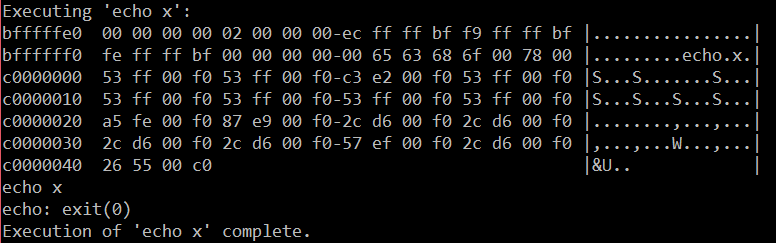
NULL

Push argument’s addresses to the stack

Push argv’s address to the stack

Push return address to stack

pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run 'echo x' 를 실행했을 때



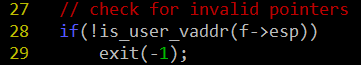
**2. User Memory Access**

User program이 invald pointer를 보낼 수도 있는 경우가 있다.

* NULL pointer : open(NULL); in tests/userprog
* Unmapped virtual memory
* Pointer to kernel address space

위와 같은 Invalid Pointer들은 반드시 rejected 되어 kernel이나 다른 실행중인 process에 영향을 주지 못하도록 해야한다.

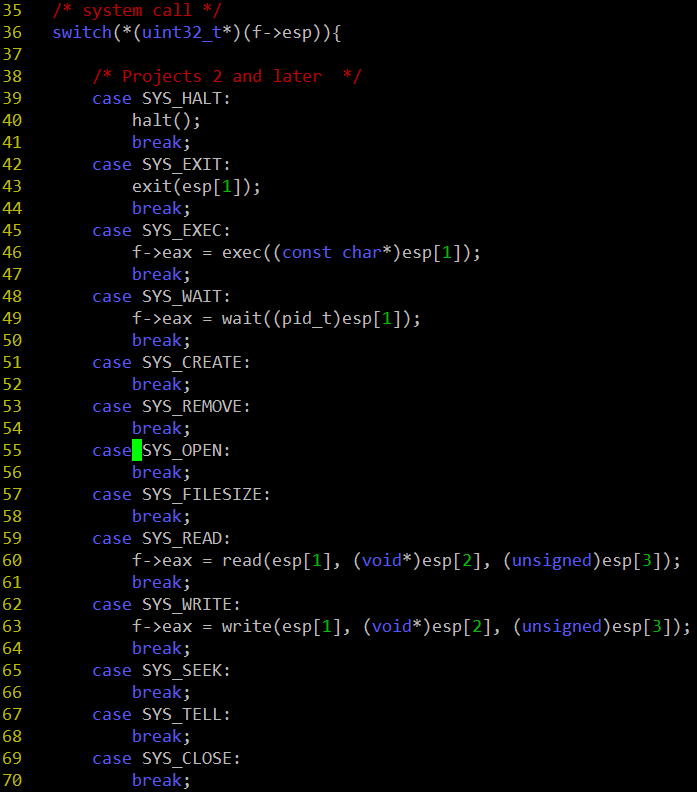
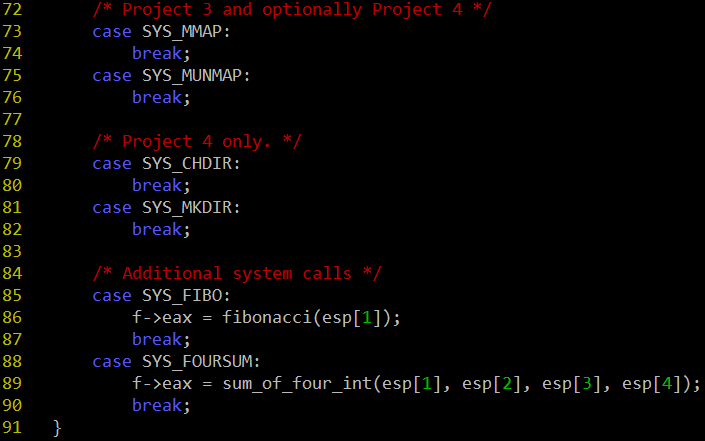
이를 구현하는 방법은 두 가지가 있는데 아래의 방법을 이용한다.

1) Verify the validity of a user-provided pointer, then dereference it.

2) Check only that a users pointer points below PHYS\_BASE, then dereference it. If the pointer is invalid, it will cause a “page fault” that you can handle by modifying the code page\_fault() in ‘userprog/exception.c’

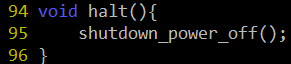


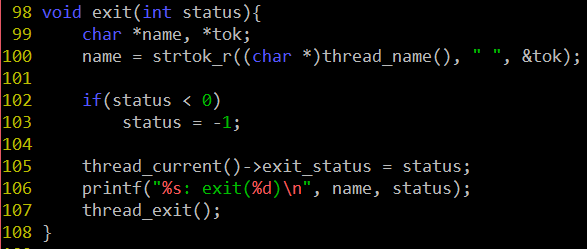
**3. System Call**

우선 system\_handler 에서 interrupt frame에 의해 얻은 esp를 이용하여 system call number를 받고 해당하는 number를 갖는 system call 함수를 호출하도록 case문으로 구현하였다.

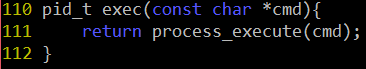
이번 프로젝트에서 system call을 사용하기 위해 구현한 system call 함수들은 다음과 같다.

* halt()

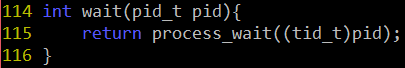
“devices/shutdown.h” 에서의 shutdown\_power\_off() 함수를 이용하여 현재 실행중인 pintos 프로그램을 종료한다. 정보 손실의 우려가 있어서 자주 사용되지는 않는다.

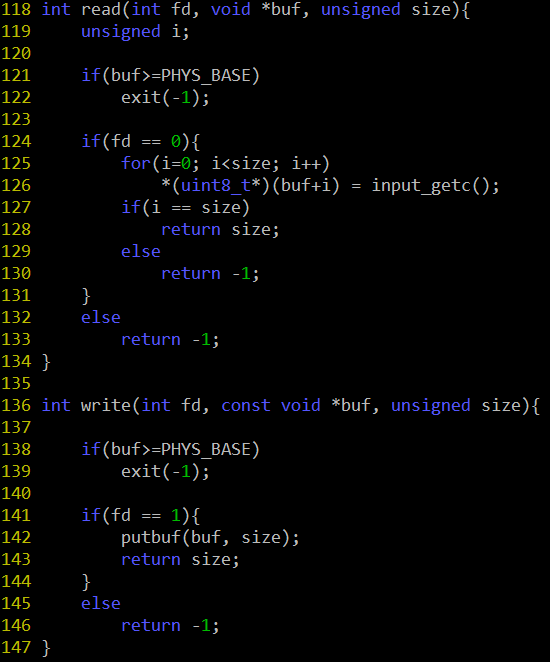
* exit()

kernel에 status를 반환하며 현재 실행중인 thread에 exit\_status를 업데이트 해주고 thread를 종료한다. 종료하기 직전에 printf를 이용하여 Requirements의 Process Termination Messages를 구현한다.

* exec()

parameter로 받아온 명령어를 보고 해당 명령어를 이름으로 갖는 파일을 실행하고, 새로운 thread를 만든다. 이 때, process\_execute 함수로 새로운 thread를 만들면서 얻은 새로운 process의 tid를 반환한다.

* wait()

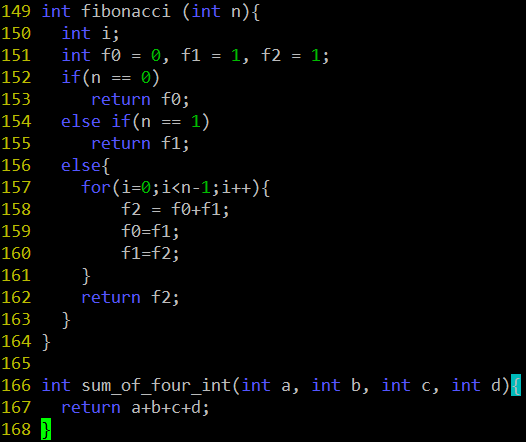
process\_wait() 함수를 이용하여 입력 받은 tid 값을 갖는 thread를 wait 한다.

* read()

standard read를 구현한다. file descripter의 경우 STDIN = 0 이고, buffer와 size를 이용한다. input\_getc() 함수를 이용하여 한 byte씩 문자를 입력 받아 buffer에 저장한다.

* write()

standard write로 구현된다. file descripter의 경우 STOUT = 1 이고, buffer와 size를 이용한다. putbuf(buffer,size) 함수를 이용하여 한 byte씩 문자를 입력 받아 buffer에 저장한다.

**4. Additional System Call**

pintos/src/userprog/ 의 syscall.c과 syscall.h

fibonacci()

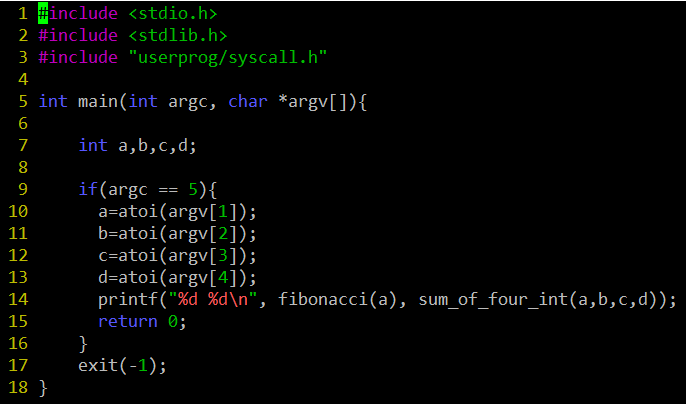
n번째의 피보나치 수열 값을 구하도록 구현하였다. 피보나치 점화식이 fn = fn-1+ fn-2 (단 , f0=0, f1=1, n=2, 3, 4, ….) 임을 이용하였다.

sum\_of\_four\_int()

4개의 정수의 합을 구하도록 구현하였다. <syscall.c>

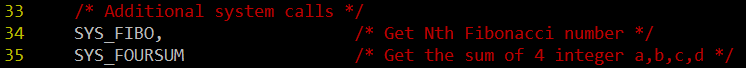
<syscall.h>

sum.c

위의 두 새로운 system call을 사용하는 user level program을 pintos/src/examples에 만들었다.

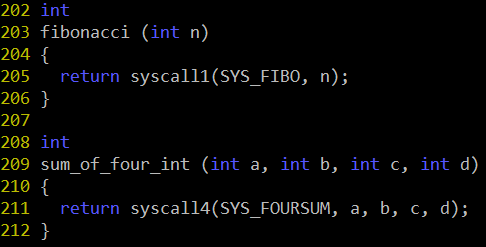
입력 받은 argument 개수가 딱 5개일 때만 (예시. ‘sum 10 20 40 56’ 으로 5개 입력) pintos/src/lib/user/ 의 syscall.c, syscall.h에 있는 fibonacci(), sum\_of\_four\_int() 함수를 이용하여 결과값들을 출력한다.

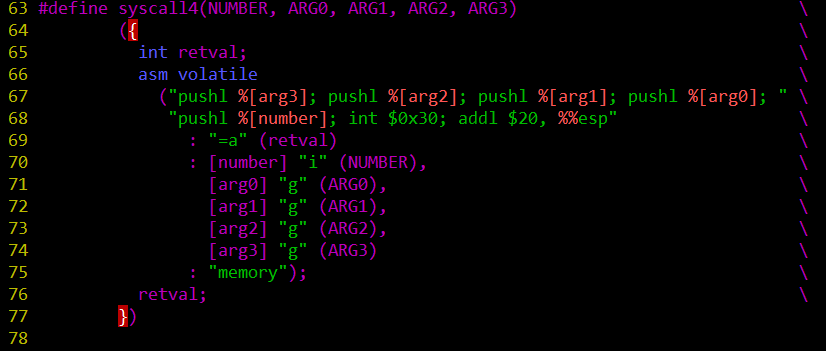
pintos/src/lib/ 의 syscall-nr.h

pintos/src/userprog/ 의 syscall.c 에서 사용하는 case문에서 새로 추가된 system call을 사용하기 위해 SYS\_FIBO와 SYS\_FOURSUM을 추가해준다.

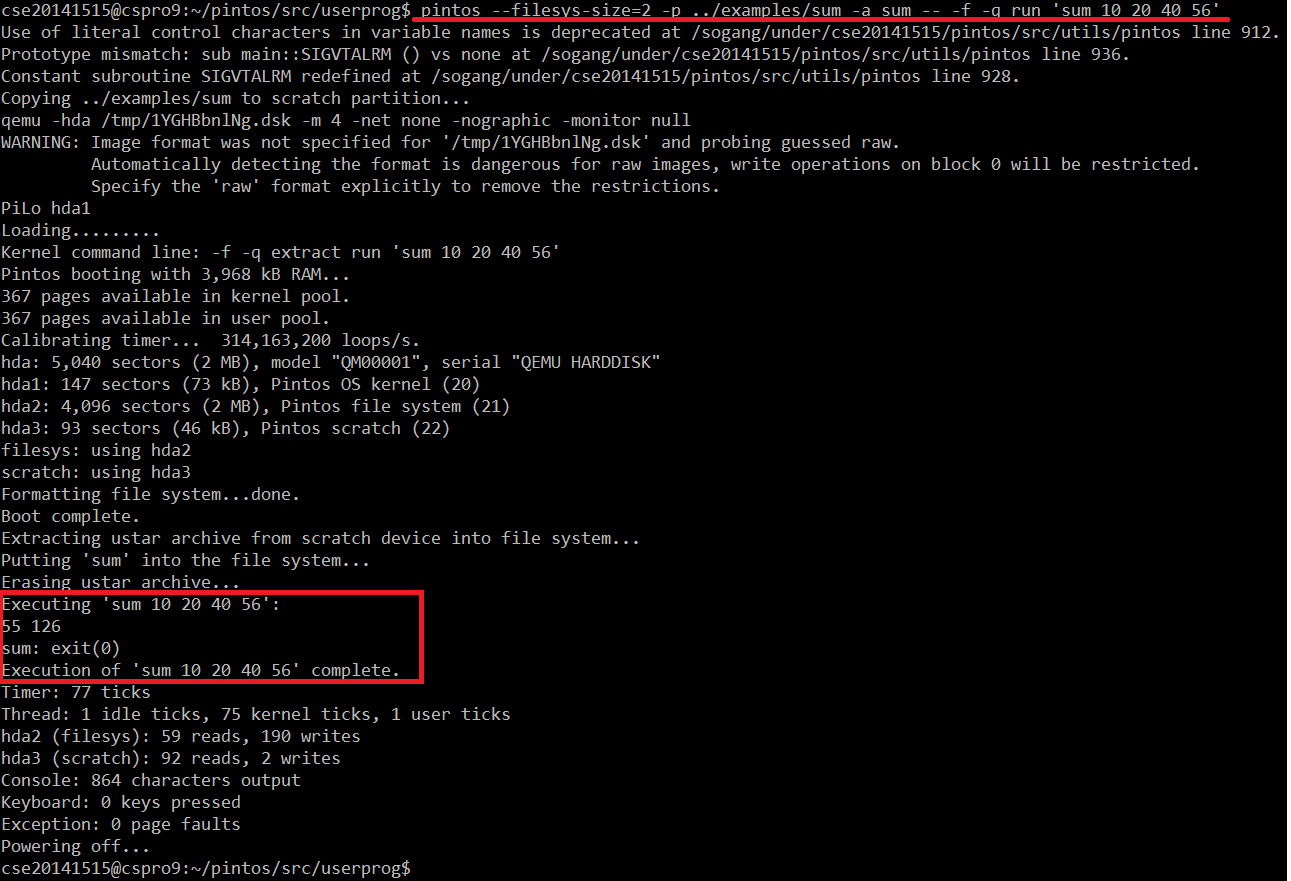
pintos/src/lib/user/ 의 syscall.c와 syscall.h

syscall.h에 다음과 같이 함수 fibonacci(), sum\_of\_four\_int() 들을 추가해준다.

syscall.c에 다음과 같이 함수 fibonacci(), sum\_of\_four\_int() 들을 추가해준다.

이 때, argument가 5개인 경우의 입력을 받아오기 위해 다음과 같은 #define을 만들어준다.

* 1. **시험 및 평가 내용**
* fibonacci(), sum\_of\_four\_int()의 수행결과

****pintos/src/userprog/ 에서 pintos –filesys-size=2 –p ../examples/sum –a sum -- -f –q run ‘sum 10 20 40 56 입력

1. **기타**
   1. **연구 조원 기여도**

* 20141515 김연후 : 50% 20141362 천지환 : 50%
  1. **소감**
* 김연후

전반적인 운영체제의 동작 과정을 단순히 이론으로만 배우는 것이 아닌 전부는 아니지만 직접 구현해서 체감해보는 것으로써 더더욱 지식이 나 자신에게 각인됨을 느꼈다.

* 천지환

익숙하지 않은 LINUX를 통해 구현하는 것에 어려움이 많았지만 이번 프로젝트로 운영체제를 구현하는 것에 어느정도 이해하고 적응한 것 같고, 구동 과정을 좀 더 자세히 이해할 수 있었다