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

담당 교수 : 박성용

조 / 조원 : 20161566\_권형준

개발 기간 : 10.3~11.1

**목차**

1. **개발 목표**
2. **개발 범위 및 내용**
   1. **개발 범위**
   2. **개발 내용**
3. **추진 일정 및 개발 방법**
   1. **추진 일정**
   2. **개발 방법**
4. **연구 결과**
   1. **Flow Chart**
   2. **제작 내용**
   3. **시험 및 평가 내용**
5. **개발 목표**

* 이번 프로젝트의 큰 흐름은 pintos 입력으로 받은 명령어와 인자들을 저장하고, 저장한 정보를 읽어서 대응하는 명령어를 수행하는 것이다. 이 과정을 하나하나 살펴보면, 우선 입력 받은 정보를 저장하는 ①Argument passing이 필요하다. 그 다음으로는 입력한 system call들을 읽어서 대응하는 system call들을 호출할 ③System call handler가 필요하다. 다음, 대응하는 system call의 실질적인 행동을 실행해주는 ④System call을 구현해주어야 된다. 이 과정에서 user memory stack등을 사용하게 되는데 이 주소가 kernel memory를 침범하지 않게 하기위한 ②Memory protection도 필요하다. 마지막으로 추가적으로 ⑤Fibonacci, max of four int 등의 함수를 구현하게 도니다.

따라서 이번 프로젝트에서 구현하게 된 내용을 5가지로 요약하면 다음과 같다.

1. Argument passing
2. Memory protection
3. System call handler
4. System call implementation
5. Additional implementation
6. **개발 범위 및 내용**
   1. **개발 범위**
7. **Argument Passing**

실시하는 명령에서 system call의 이름을 포함한 인자들이 user stack에 저장이 된다. 이는 이후에 system call을 실행할 때 저장된 인자들을 포인터를 이용하여 접근하여 요청된 일을 수행한다.

1. **User Memory Access**

Pintos에는 kernel virtual memory와 user virtual memory가 있다. 각 프로세스마다 가상 메모리가 할당되어 있으며 Argument Passing을 통해서 읽어온 값들도 여기에 저장된다. 또한 system call을 위한 argument, system call number도 여기에 저장된다. 이후에 이를 가리키기 위해서, 또는 새로운 virtual memory를 할당 받기 위해서 주소를 읽어올 때 이 부분이 kernel virtual memory, unmapped virtual memory, NULL이어서는 안 된다. 만약 그럴 경우 적절한 exit\_status(=-1)을 출력하고 종료한다.

1. **System Calls**

System call을 호출하였을 때 요청한 작업을 올바르게 수행하게 한다. 프로젝트1의 범위에 속하는 system call은, halt, exit, exec, wait, read(stdin), write(stdout), 추가적으로 구현한 Fibonacci, max\_of\_four\_int등 이 있다.

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

들어온 argument를 공백을 기준으로 parsing을 한다. Userprog/process.c의 load함수에서 process의 stack을 생성한 뒤, argument의 처음부터 stack에 push를 한다. Argument를 다 push한 이후, word alignment를 위해 필요시 공간을 띄운 뒤 각 argument의 시작 주소를 저장한다. 마지막으로 argument의 수와 return address를 push한다.

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

Invalid memory access란 invalid pointer로 메모리를 접근할 시 발생하는 오류이며, invalid pointer에는 NULL, unmapped virtual memory, pointer to kernel address space가 있다. 각 포인터가 왜 오류가 생기는지 간단하게 설명하면, 우선 NULL 포인터와 unmapped virtual memory는 접근하고자 하는 메모리가 없거나 쓰레기 값을 가지고 있거나 할당 받은 가상 메모리 범위를 초과하여 오류가 발생한다. Kernel memory는 interrupt vector, interrupt handler등의 중요한 내용들을 담고 있다. 따라서 User program에 의해서 이 영역으로의 access는 금지되어야 한다.

* + **Invalid memory access를 어떻게 막을 것인지 설명**

Pintos 내부에서 제공한 함수와 비교 연산자를 사용하여 invalid memory access를 막았다. Invalid memory access를 발생시키는 3가지 pointer에 대하여 하나씩 설명하면 다음과 같다.

NULL: 포인터의 값이 NULL이면 오류를 감지하였다.

Unmapped virtual memory: pagedir\_get\_page()함수를 통하여 반환 값이 NULL이라면, 또는 포인터가 0x08048000보다 크거나 같다면 오류를 감지하였다.

Kernel virtual memory: is\_user\_vaddr()함수를 사용하여 오류를 감지하였다.

* System Calls
  + **시스템 콜의 필요성에 대한 간략한 설명**

잘못된, 또는 악의적인 User program이 함부로 시스템을 해칠 수 없게 하기 위해서 OS는 kernel mode와 user mode 두가지를 제공한다. Kernel mode에서만 수행이 가능한 privileged instruction을 사용하여 이러한 protection을 제공한다. System call은 user mode에서 kernel mode로의 전환을 시켜주는 역할을 한다. 대부분 OS는 system call을 바로 사용하지 않고 user library등에 있는 API를 통해서 system call을 사용한다.

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

Shutdown\_power\_off()를 호출하여 pintos를 종료한다.

* + - Exit

화면에 종료 상태(status)를 출력하고 현재 돌고 있는 프로그램을 종료한다. 만약 부모가 wait()를 하고 있다면 부모에게 이 status를 반환한다. 정상적인 종료 시 0을 반환한다.

* + - Exec

자식 프로세스를 생성하고 실행시켜준다. 이에 실패했을 시에 -1을 반환하고, 성공했으면 자식 프로세스의 pid를 반환한다. 부모는 자식 프로세스의 프로그램이 잘 생성되었는지 확인하기 전까지 exec에서 return을 하면 안됨으로 적절한 동기화(synchronization) 방식이 필요하다.

* + - Wait

모든 자식 프로세스가 종료될 때까지 기다렸다가, 그 반환 값(exit status)를 받는다. 반환 값을 가지고 자식 프로세스가 정상적으로 종료되었는지 판단할 수 있다.

* + - Read

현재 파일에서 지정된 byte 수만큼 읽어온다. 프로젝트 1에서는 standard input (fd=0)인 경우만 고려한다. 실제로 읽어온 byte수를 return한다.

* + - Write

현재 buffer에 있는 내용을 지정된 byte만큼 파일에 기록한다. 프로젝트 1에서는 standard output (fd=1)인 경우만 고려한다. 실제로 쓴 byte 수를 return 한다.

* + - Fibonacci

입력된 인자 4개 중 첫 인자에 해당하는 Fibonacci 수열의 수를 출력한다.

* + - Max\_of\_four\_int

입력된 인자 4개 중 가장 큰 인자에 해당하는 수를 출력한다.

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

유저 레벨에서 시스템 콜 API를 호출하면, src/lib/user/syscall.c에 있는 해당하는 system call의 함수가 호출이 된다. 여기서 user stack에 전달받은 인자와 system call number등을 push한다. 그 다음 interrupt handler에서 interrupt를 발생시켜 system call에 해당하는 interrupt number(0x30)을 interrupt vector table에서 찾고, 알맞은 interrupt handler(src/userprog.syscall.c에서 구현한)으로 넘어간다. 여기부터는 kernel mode에 해당한다. User stack에 저장한 system call number와 인자에 해당하는 system call을 수행한다. 수행한 결과를 eax레지스터에 저장을 한다. 이후 시스템 콜 API에 이 값을 반환한다. 이에 따라서 user mode로 돌아오게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 10.3~10.10 | Argument Passing |
| 10.10~10.15 | User Memory Access |
| 10.15~10.25 | System Calls |
| 10.25~10.30 | Additional System Calls |

* 1. **개발 방법**
* **Argument Passing**
* Src/userprog/process.c

전달받은 filename을 공백을 기준으로, 수행할 program의 이름, 뒤의 인자들을 parsing한다. parsing하여 배열에 저장하고 이를 생성한 user virtual memory stack에 형식에 맞게 넣는다. 이 과정에서 strtok\_r()을 사용한다.

* **User Memory Access**
* Src/userprog/syscall.c

System call별로 user stack에 저장된 인자를 추출하는 함수를 구현할 것이다. 또한, invalid memory access가 발생하는지 검사하는 함수를 구현할 것이다. 이과정에서 이미 구현이 되어있는 pagedir\_get\_page, is\_user\_vaddr와 같은 함수들을 사용한다.

* **System Calls**
* Src/userprog/syscall.c & syscall.h

System call handle 내부에 system call number을 가지고 알맞은 system call을 수행하는 switch문을 작성할 것이고, 각 system call의 경우에 따라서 stack에서 인자를 읽어와 해당하는 system call의 함수를 호출할 것이다. 이 함수들의 기능과 코드는 밑에서 제작 내용에 포함시키도록 하겠다.

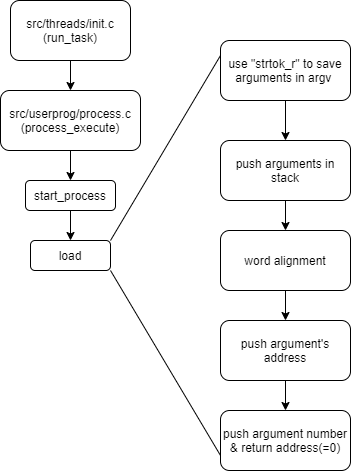
* Src/userprog/process.c

Process\_wait함수를 손볼 것이다. Parent에게 아직 실행이 덜 끝난 child가 있다면 그 child를 wait해야 된다. 이를 위하여 thread.c에서 구현한 linked list를 사용할 것이다.

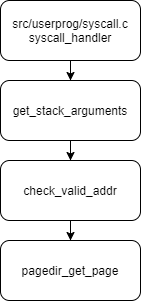
* Src/threads/thread.c & syscall.h

Wait를 구현하기 위해서 semaphore을 사용할 것이다. Parent는 모든 child에 대하여 wait를 해야 되기 때문에, child process들을 저장할 수 있는 구조체도 만들 것이다. 또한 이 구조체들을 연결시켜 linked list로 유지할 것이다. thread가 처음 실행될 때 semaphore를 초기화 해주는 과정도 추가할 것이다.

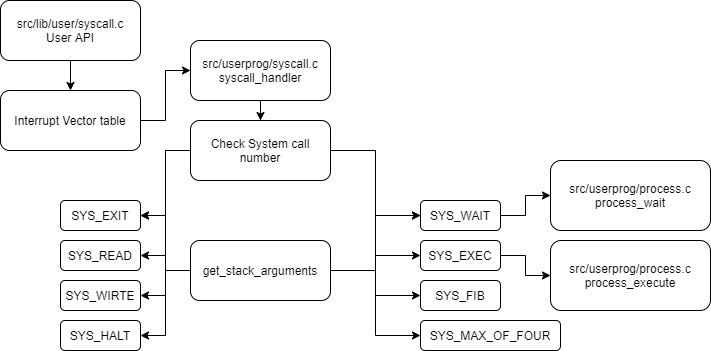
1. **연구 결과**
   1. **Flow Chart**
2. **Argument Passing**



1. **User Memory Access**



1. **System Calls**



* 1. **제작 내용**

1. **Argument Passing**

Parsing, stack에 저장을 하는 코드는 모두 “process.c”의 load함수 내에 작성을 하였다. 파싱을 할 때 사용한 함수는 strtok\_r로 다음과 같은 형식으로 공백을 기준으로 parsing하였다.



이렇게 parsing한 각 argument를 “char \*argv[128]”이라는 포인터 배열에 저장하였다. Test program은 argument에 최대 128byte를 할당한다고 하였기 때문에 최대 개수를 128개로 잡았다. 다음 argc에 총 argument의 수를 저장한다. 이후 첫번째 argument에 해당하는 파일을 열고 만약 실패하면 오류를 출력한다.

다음 정상적으로 process가 loading된 이후(setup\_stack 이후)에 argv에 있는 argument들을 넣으면 된다. 초기에 esp는 초기에 (PHYS\_BASE)를 가리키고 있다. Argument를 넣어주고 뒤에 각 argument의 주소를 넣어줘야 되기 때문에 이 주소를 저장하는 “uint32\_t \*addrs[128]”이라는 배열을 생성한다. 이제 첫 argument부터 esp가 가리키는 곳에 저장을 하는데, stack은 커지는 방향이 아닌 줄어드는 방향으로 주소가 진행되기 때문에 공간을 먼저 할당하고 저장을 해야 된다. 예를 들어서 3byte의 문자열을 저장하기 위해서 우선 esp-(3+1)을 한 뒤 memcpy함수를 사용하여 esp의 위치에 argument를 넣는다. 1을 더하는 이유는 각 argument마다 뒤에 ‘\0’문자가 필요하기 때문이다. 이때 addrs[]배열도 update를 같이 해준다.

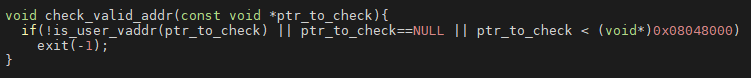
이제 argument를 다 넣었으니 word algin을 해준다. 정수 3과 “esp”를 &연산하여 0이 아닌 수가 나오면 “\*esp = \*esp – 1”연산을 수행하여 주소 값이 4의 배수가 되도록 한다.

다음은 앞에서 저장한 addrs[]의 값들을 하나씩 넣는다. 정상적인 주소는 4byte만큼의 공간을 차지할 것이다.

마지막으로 총 argmuent의 수(argc)를 저장하고 return address(=0)을 저장한다.

1. **User Memory Access**

User program에서 제공하는 pointer가 invalid한지 검사하는 부분을 코드에 추가하였다. 이러한 pointer들은 대부분 syscall.c에서 처리를 하였다. Syscall에 다음과 같은 함수를 만들었다.



포인터를 받아와서 pintos에 미리 정의되어 있는 is\_user\_vaddr함수를 사용하여 user virtual address인지 확인한다. 이는 주소의 값이 PHYS\_BASE보다 작은가를 확인한다. 하지만 pintos에서 사용하는 user virtual address의 시작점인 0x08048000보다 작은 주소가 들어와도 여기서 true가 나올 수 있기 때문에 뒤에서 이 또한 검사를 해준다. 다음 invalid pointer 중 하나였던 NULL의 값이 들어오면 오류를 출력하게 하였다.

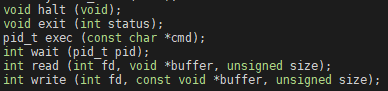
다음은 unmapped user virtual address인지 검사하기 위해서 핀토스에서 제공한 pagedir\_get\_page함수를 사용하였다. 이 함수는 해당하는 주소가 mapping되어있지 않다면 NULL을 return한다.



위와 같이 사용하여 확인하였다. 현재 돌고 있는 thread의 pagedir에서 args[0]의 주소가 매핑 되어 있는지 아닌지를 확인하고 NULL이면 exit(-1)을 한다. 이렇게 확인해야 되는 경우는 인자들에 값이 아닌 주소가 저장된 경우이다. 즉, 저장된 주소가 check\_valid\_addr로 check를 하였지만, mapping되어있는지도 확인해야 되기 때문이다. 여기서 exit(-1)은 밑에 system call을 구현하면서 만든 함수로 exit status를 출력하고 thread\_exit을 하는 함수이다.

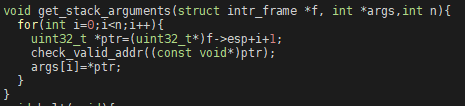
1. **System Calls**

User API를 거치고, interrupt가 발생하여 userprog/syscall.c의 system call handler로 오게 되면 우선 어떤 system call인지 구분을 해주어야 된다. Intr\_frame의 구조체에 저장된 user stack을 ‘f->exp’을 통해서 접근할 수 있다. System call을 호출하는 동시에 stack을 생성할 때 가장 마지막으로 system call number을 push했기 때문에 확인할 때는 가장 위의 값을 확인하면 된다. 이 값을 switch문에 넣어주어 system call을 구분하였다. SYS\_HALT와 같은 #define은 “lib/user/syscall.h”를 include하여 그대로 사용하여 가독성을 높여주었다. 또한, 각 system call을 system call handler 내부에서 처리를 하지 않고 함수를 호출하는 형식으로 구현하였다.



각 switch문에서 인자를 뽑은 뒤 해당하는 system call의 함수를 호출하게 된다. (userprog/syscall.h)

이때 각 system call마다 인자가 필요한데, 공통적으로 get\_stack\_arguments라는 함수를 사용하여 인자를 추출하였다. Get\_stack\_arguments를 살펴보면 다음과 같다.



각 system call마다 읽어와야 하는 인자의 수를 알고 있기 때문에 이를 n이라고 하고, f는 초기에 system call handler로 들어온 intr\_Frame을 사용하고, args[]는 읽어오는 argument를 저장할 공간으로 사용된다. 코드의 흐름을 살펴보면, 총 n개 만큼 user stack(f->esp)에서 uint32\_t(4byte)씩 읽어온 뒤, valid한 주소에 저장되어 있는 것인지 확인하고 이 값을 args에 하나씩 넣는다. 이렇게 읽어온 args를 가지고 각 system call마다 함수를 호출하게 된다.

이제 각 system call에 대하여 살펴보면 다음과 같다.

* + - Halt

Halt는 특별한 인자를 필요로 하지 않는 system call임으로 get\_stack\_arguments도 사용하지 않는다. Halt는 pintos 내부에 구현되어 있는 shutdown\_power\_off()라는 함수를 그대로 사용하여 구현하였다. 다른 작업은 추가로 필요하지 않았다.

* + - Exit

User stack에서 인자를 읽어온 뒤 exit()함수를 호출한다. 이때 인자는 (int status)를 뜻한다. 정상적은 exit이라면 이 값은 0을 나타낼 것이고 아니라면 그 외의 숫자를 나타낸다. exit함수 내에서는 이 exit status를 형식에 맞게 출력하고 현재 돌고 있는 thread의 exit\_status를 update한다. 그 뒤 thread\_exit을 통하여 thread(process)를 종료한다. 만약 이 process의 부모가 wait를 하고 있다면, 이 exit status를 return해 준다.

* + - Read

Get\_stack\_argument를 사용하여 인자를 3개 읽어온다. 어떤 파일인지 나타내는 정수형 숫자(fd), 읽은 내용을 저장할 buffer, 총 읽을 byte의 수를 의미한다. 다음 read함수를 호출하게 되는데, 이번 프로젝트 1에서는 stdin의 read만 고려하면 되기 때문에 fd==0인 경우만 고려하고 그 외의 경우에는 -1을 return한다. Stdin의 read임으로 input\_getc()라는 pintos내부 함수를 사용하여 buffer에 한 byte씩 넣는다. 끝난 뒤 실제로 읽은 총 byte 수를 return한다.

* + - Write

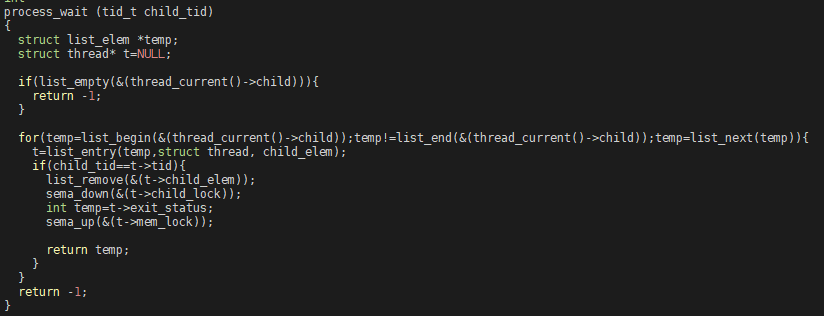
Write도 read와 비슷한 과정을 거친다. 우선 인자를 3개 읽어온다. Fd, buffer, byte의 수(n). 이번에는 stdout만 고려하면 되기 때문에 fd==1인 경우만 고려한다. 그 외의 경우에는 -1을 return 한다. putbuf라는 pintos내부 함수를 사용하여 buffer의 내용을 n만큼 출력한다. 그 뒤 실제로 출력한 크기만큼 return한다.

* + - Exec

Exec는 user stack에서 하나의 인자만을 읽어온다. 이 인자는 command line을 가리키는 주소를 담고 있다. 이 command line을 이용하여 process.c의 process\_execute()함수를 호출하여 새로운 process를 실행한다.

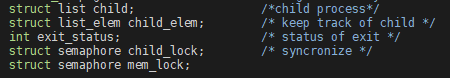
* + - Wait

User stack에서 인자를 하나 읽어온다. 이는 어떤 process의 pid값을 읽어온 것이다. wait한수는 이 인자를 읽어서 process.c에서 구현한 process\_wait함수를 호출한다. Process\_wait함수의 역할은 인자로 받아온 pid의 값을 가진 process가 종료되기를 기다리는 것이다. 이를 구현하기 위한 코드는 다음과 같다.



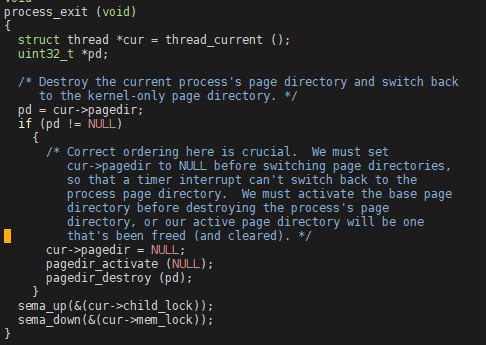
(process\_wait함수는 이번 프로젝트에서 가장 중요한 부분이라고 생각하여 코드를 다 추가하였다)

Parent는 child가 죽기 전까지 wait를 해주어야 된다. 이를 구현하기 위해서 수업시간에 배운 semaphore를 사용하였다. pinots내부에서 구현 되어있는 semaphore를 사용하였다. Wait를 하는 sema\_down, signal을 하는 sema\_up 함수도 사용하였다. 각 thread마다 이 semaphore값을 넣어서 init할 시 semaphore도 init해주었다. 또한 부모가 모든 자식들을 추적하기 위하여 linked list를 thread안에 구현하였다. 새로운 thread가 생성되었을 때 현재 돌고 있는 thread의 child list뒤에 push를 해주는 함수도 init\_threaed에 추가하여 주었다.



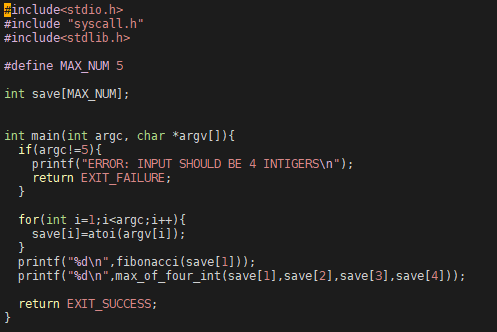
(thread.h에 thread내부에 추가한 자료구조이다)

이제 위의 process\_Wait코드를 살펴보면, 인자로 들어온 child\_tid의 값과 같은 thread id를 가진 child를 찾기 위해 child linked list를 탐색한다. 찾은 뒤, list에서 제거하고 sema\_down, 즉 wait를 한다. Child가 죽을 때, 어디선가 signal을 보내주어야 되는데, 그 부분이 바로 process\_Exit에 구현하였다.

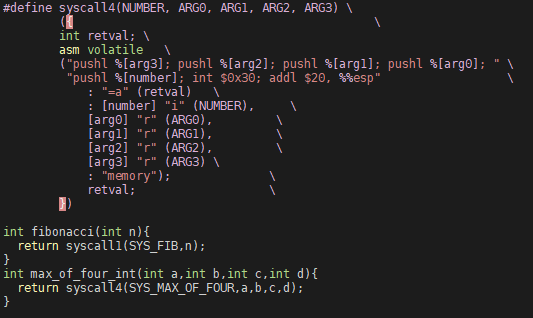
  
여기서 semap\_up을 하는 순간 process\_wait에서 sema\_down되어 있던 부분을 통과하게 되면서 종료된 child thread의 exit status를 return하게 된다.

여기서 semaphore을 하나가 아닌 2개를 계속 사용하고 있다. 이는 마지막에 t의 exit status를 받기 위함이다. Process\_exit에서 sema\_up을 하는 순간 t가 삭제되면 t에 저장되어 있는 exit\_status를 읽을 수 없게 된다. 따라서 temp에 exit status를 저장한 뒤, 계속 삭제를 진행하라고 지시하는 memory lock을 의미하는 mem\_lock semaphore를 하나 추가로 사용하였다.

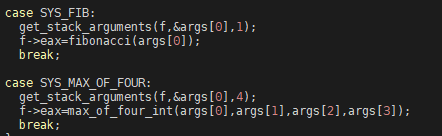
1. Additional System calls
2. 우선 additional.c라는 파일을 만들어주었다. 여기서는 User API를 호출하는 단계로 다음과 같다.



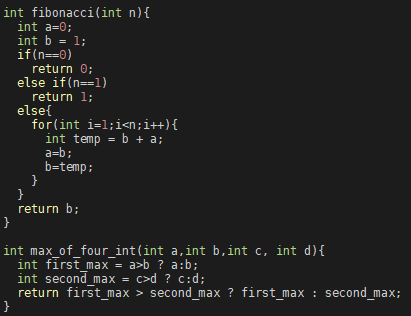
1. 다음으로 src/lib/user/syscall.c에 해당하는 user api가 호출된다. 따라서 syscall.c와 syscall.h에 Fibonacci, max\_of\_four\_int함수를 추가해주었고 인자가 4개임으로 새로운 매크로도 추가해주었다.



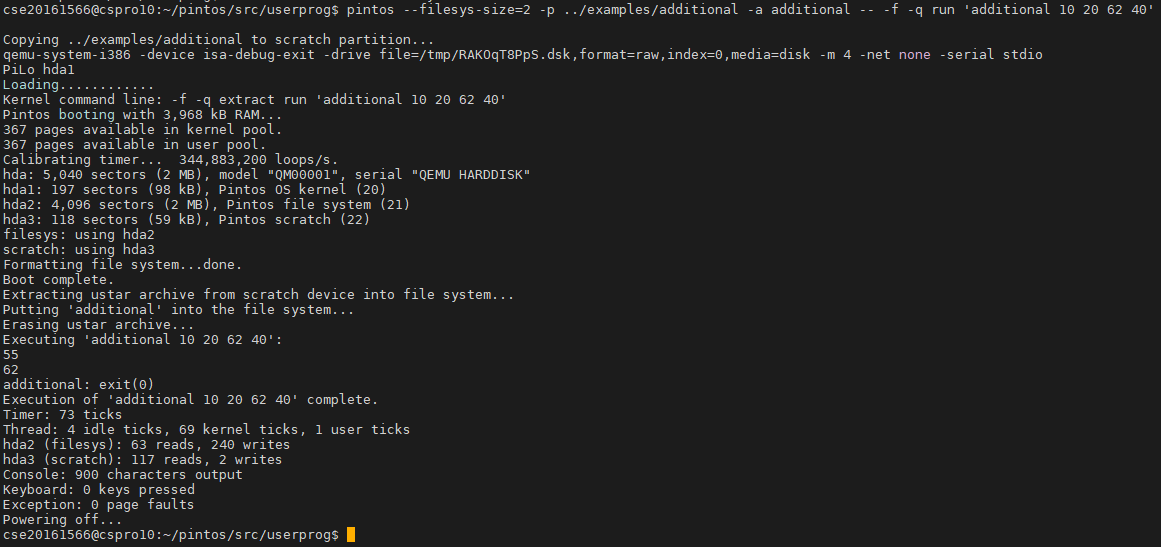
1. 이제 interrupt handler에 의해 system call handler로 넘어오게 된 뒤 userprog/syscall.c의 switch문에 해당하는 system call이 호출된다. 앞에서 구현한 get\_stack\_argument로 인자를 추출한 뒤 해당하는 함수를 부른다.



1. 해당하는 함수는 각각 정해진 일을 하고 출력을 한 뒤 그 값을 return하여 eax레지스터에 저장한다.



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



**-감사합니다-**