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

담당 교수 : 김영재

조 / 조원 : 20161598 손동현

개발 기간 : 2020.10.3 ~ 2020.11.1

1. **개발 목표**

해당 프로젝트의 초기 상태는 크게 Argument Passing, Accessing User Memory Check, System call 이 3가지가 구현되어 있지 않았다. 따라서 이번 프로젝트에서는 해당 부분을 구현하여, process를 load한 후, 구현한 system call을 통해 process가 제시된 기능(halt, wait, exit, execute, read, write, fibonacci, max\_of\_four\_int)을 수행할 수 있는 환경을 만드는 것이 가장 큰 목표이다.

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

User로부터 입력 받은 argument는 빈칸을 기준으로 token으로 쪼갠다. 그 후, ESP가 가리키는 stack에 쌓는 과정을 수행한다. 아래와 같은 순서로 stack에 쌓도록 한다.



1. User Memory Access

User가 Pintos를 실행할 때, 잘못된 메모리에 access하는 경우(Unmapped virtual memory, pointer to kernel address space)에 예외 처리를 하여 프로그램을 종료할 수 있도록 한다.

1. System Calls

Argument Passing을 통해 stack에 쌓여 있는 argument들의 내용을 갖고, 이에 맞는 system call을 수행할 수 있도록 한다.

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

User가 수행하려는 프로그램은 다수의 Argument를 가질 수 있기 때문에 이를 메모리에 80x86 Calling Convention에 맞추어 메모리에 배치해야 프로그램을 제대로 수행할 수 있다. User로부터 입력 받은 argument는 빈칸을 기준으로 token으로 쪼갠다. 그 후, ESP가 가리키는 stack에 쌓는 과정을 수행한다. Stack은 아래로 쌓는 과정으로 구현하게 되는데, 아래와 같은 순서로 쌓는다.



이 때, 주의할 점은 word-alignment를 하는 것이다. 위의 예에서 –l\0은 길이가 3이므로 word-alignment를 해주기 위해서는 모든 token들을 4의 배수로 맞춰 주어야 한다. 따라서 위와 같은 과정을 거친 뒤, NULL pointer sentinel을 쌓고, 해당 token들의 주소값을 쌓아준다. 그리고 argv[0]의 주소를 담은 곳의 주소를 쌓은 뒤, 전체 token의 개수를 쌓고, 마지막으로 return address를 쌓아주면 된다.

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

User program은 invalid pointer를 pass할 수 있다. Invalid pointer에는 3가지 종류가 있는데, 첫 번째로는 NULL pointer로 예를 들어 open(NULL)과 같은 것이 있다. 두 번째로는 mapping되지 않은 virtual memory가 있고, 세 번째로는 kernel address space를 가리키는 pointer이다. 이러한 invalid pointer들은 kernel 또는 다른 실행중인 process에 악영향을 끼칠 수 있으므로 미리 차단되어야 한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

이러한 Invalid pointer를 막기 위해서 경우를 나눠서 막겠다. 첫 번째로는 mapping되지 않은 virtual memory를 막기 위해서는 userprog/pagedir.c에 있는 pagedir\_get\_page()함수를 이용해 파라미터로 현재 thread의 pagedir와 현재 포인터를 넘겨 주어 NULL 값이 return 되면 unmapped되었다는 것이므로 종료시킨다. 두 번째로는 kernel address space를 가리키는 경우를 막는 것인데, 이 때는 threads/vaddr.h에 있는 is\_user\_vaddr()함수를 사용하여 만일 is\_user\_vaddr의 return 값이 false인 경우, 즉 user 공간이 아닌 다른 공간을 가리키는 경우, 종료시키는 방식으로 Invalid memory access를 막을 수 있다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

시스템 콜은 user 모드에서 kernel 모드로 진입하는 동작이다. 다른 관점으로 시스템 콜은 user 공간과 kernel 공간 사이 가상 계층이자 인터페이스라고 볼 수 있다. 만일 해커가 피해를 입히기 위해 악의적으로 시스템 콜을 사용하는 경우나 초보 사용자가 하드웨어 명령어를 잘 몰라서 아무렇게 함수를 호출했을 경우 시스템 전체가 망가질 수 있는 상황이 생길 수 있다. 따라서 시스템 콜은 이러한 상황을 막아줌으로써 시스템 안정성과 보안을 지켜준다.

* + 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() 함수를 호출하여 pintos를 종료하여 준다.

exit() :

현재 수행하고 있는 user program을 종료하고, 해당 process의 status를 kernel에

반환한다.

exec() :

child process를 생성하기 위해 userprog/process.c에 있는 process\_excute() 함수를

호출하여, input으로 받은 명령어에 맞는 thread를 만들고 process를 수행하게 한다.

wait() :

작업이 끝날 때까지 child process를 기다린다. 그리고, child thread의 ID가 유효한지 확인하고 child thread가 죽었을 때, child thread에서 exit status를 가져온다.

read() :

모든 구현은 하지 않고, STDIN으로부터 읽을 수 있게만 구현한다. STDIN의 File Descriptor는 0이므로 이를 보고 판단하면 된다.

write() :

모든 구현은 하지 않고, STDOUT으로부터 읽을 수 있게만 구현한다. STDOUT의 File Descriptor는 1이므로 이를 보고 판단하면 된다.

int fibonacci(int n) :

피보나치 수열의 n번째 값을 return한다.

int max\_of\_four\_int(int a, int b, int c, int d) :

파라미터로 넘어온 a, b, c, d 중 제일 큰 값을 return 한다.

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

User 모드에서 사용자 프로그램을 돌다가 system call api를 호출하게 되면 그 때 OS에게 해당 system handler를 실행해달라고 요청을 하게 된다. 이 때, software trap을 걸고, 이를 통해 user mode에서 kernel 모드로 넘어가게 된다. kernel에서 interrupt handler 에 의해 service routine(system call handler) 해당 routine 찾고 호출하여 해당 rountine을 수행한다. 그 후 return 값을 eax에 받아서 다시 user모드로 돌아간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/03 ~ 10/7 – proj1 동영상 시청 및 명세서 분석   
10/8 ~ 10/12 - Argument passing, Accessing User Memory 구현   
10/13 ~ 10/18 – syscall handler, system call(halt, exit, exec, wait, read, write, Fibonacci, max\_of\_four\_int) 구현  
10/29 ~ 10/31 – process\_wait, process\_exec 구현  
10/31 ~ 11/01 – 보고서 작성

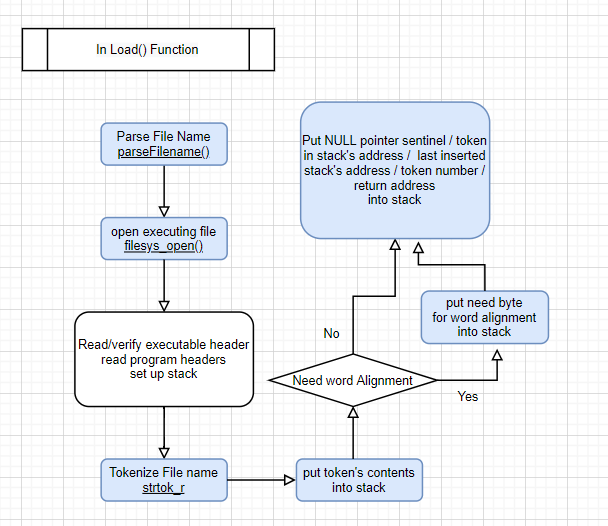
* 1. **개발 방법**

우선 Argument Passing을 구현하기 위해, userprog/process.c에서 fileName에서 빈칸을 기준으로 첫 번째 token을 가져오는 함수에 대한 구현이 필요하다. 해당 함수에 대한 구현은 반복문을 통해 NULL문자 또는 빈칸을 찾은 뒤 그 앞 부분을 가져오는 방법으로 구현하면 된다. 그 후 stack을 쌓는 과정은 process.c의 load 함수 내부에 추가적인 함수 구현 없이 명세서에서 요구하는 순서대로 쌓아주면 될 것이다.

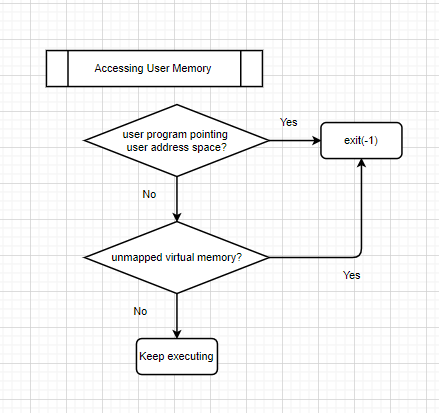
User Memory Access를 구현하기 위해서는 User program의 pointer가 invalid한지 check해야 한다. 이를 구현하기 위해 threads/vaddr.h에 있는 is\_user\_vaddr()함수를 사용할 수 있다. 그리고 다른 경우인 mapping되지 않은 virtual memory를 막기 위해서는 userprog/pagedir.c에 있는 pagedir\_get\_page()함수를 이용한다.

System Call에서 wait를 구현하기 위해 semaphore를 사용하겠다. 따라서 이에 필요한 semaphore , parent thread가 child thread를 가리키기 위해 사용하는 list 변수와 이의 구성 요소인 list\_elem 그리고 마지막으로 child process가 종료된 것인지 아닌지를 나타내는 status를 알려주는 변수를 threads/thread.h의 struct thread인 thread 구조체에 추가하여 semaphore를 사용할 수 있게 한다. 그 후, process.c에 있는 process\_exit, process\_wait 함수에 semaphore와 이에 관한 함수를 이용하여 이들을 잠시 정지등을 할 수 있게 구현한다. userprog/syscall.c에 있는 syscall\_handler 함수를 수정하여 각 system call number에 맞는 함수를 호출할 수 있게 if-else 문으로 구현한 후, halt, wait, exec, exit, read, write, fibonacci, max\_of\_four\_int 함수를 syscall.h에 prototype을 정의하고 각 함수에 대한 내용을 실행할 수 있게 syscall.c에 구현한다.

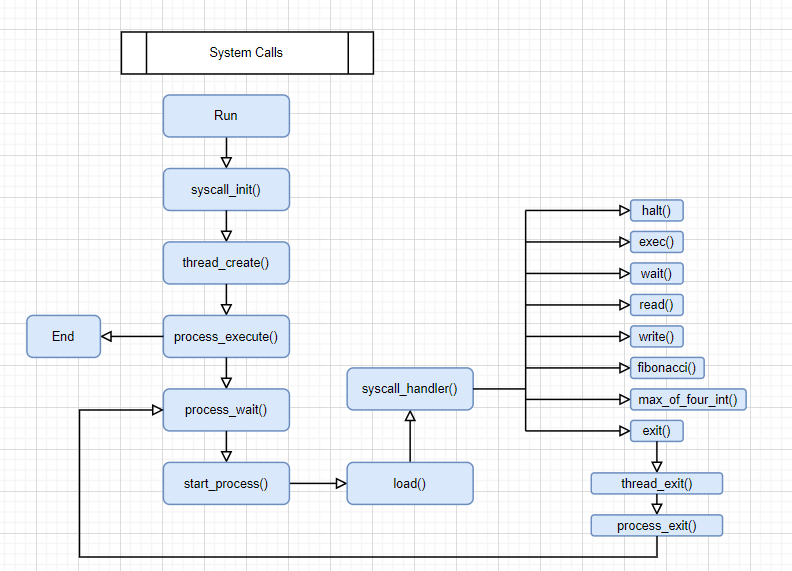
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



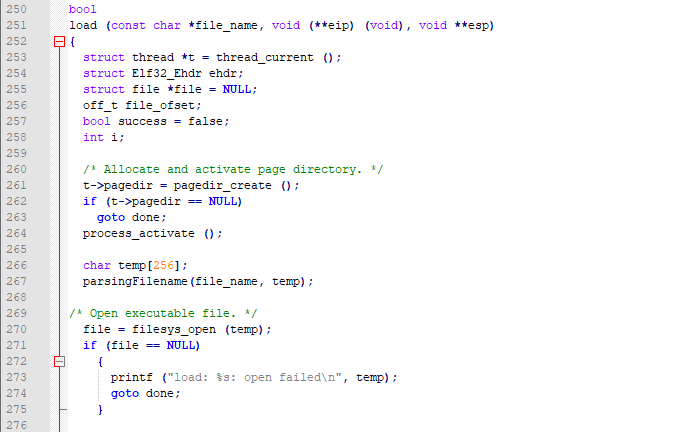
1. System Calls



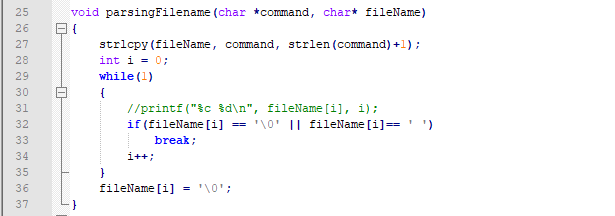
* 1. **제작 내용**

1. Argument Passing

Argument passing은 load 함수 내부에서 일어난다. start\_process() 함수에서 load함수에 input 명령어와, eip, esp 값을 넘겨준다. 그러면 load 함수의 구현은 아래 그림과 같이 하였다. Load 함수가 길기 때문에 수정한 내용에 대해서만 첨부하겠다.



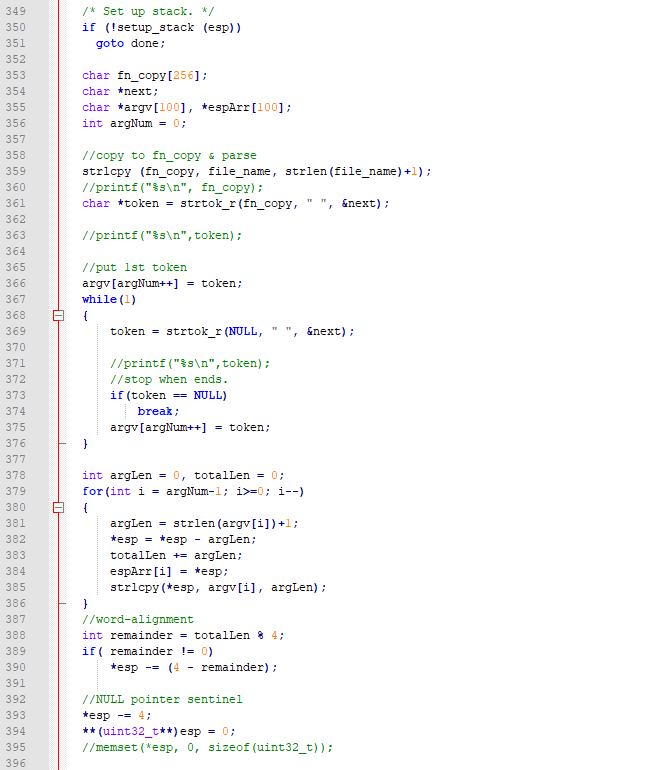
먼저 위의 266 줄에서 temp라는 char 배열을 선언하였다. 입력 받은 file\_name에서 file\_name의 첫 번째 token을 filesys\_open 함수의 파라미터로 넘겨주어야 하므로 file\_name에서 첫 번째 token을 뽑아내는 parsingFilename 함수를 정의 후, temp에 첫 번째 token을 저장하게 하였다. parsingFileName 함수의 구현은 아래와 같이 하였다.

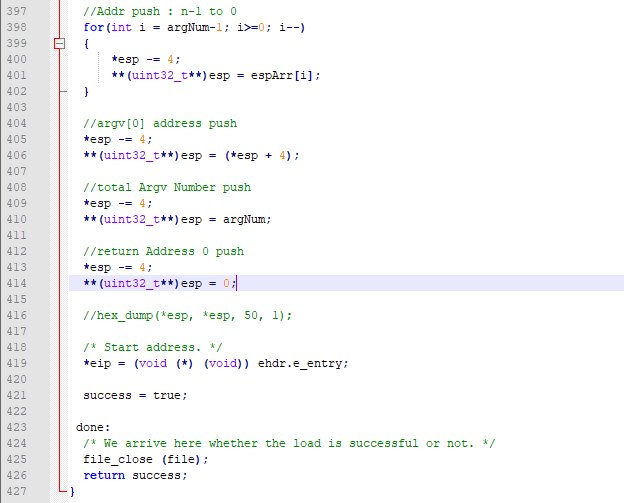


parsingFilename에서 command는 입력 받은 input이고, fileName은 command의 첫 번째 token이 저장될 배열을 의미한다. 처음으로 ‘\0’ 또는 ‘ ’이 나오면 멈추고, 마지막 지점에 NULL 문자를 삽입하여 filename 변수에 첫 번째 token이 저장되게 한다.

그 후, 다시 load로 돌아가서, 267 줄을 수행하고 나면 현재 temp에 file\_name의 첫 번째 token이 저장되어 있으므로, 이를 filesys\_open 함수의 파라미터로 넘겨준다.

아래는 load 함수 내의 그 다음 수정 사항이다. 실제 stack에 쌓는 과정에 대한 코드이다.

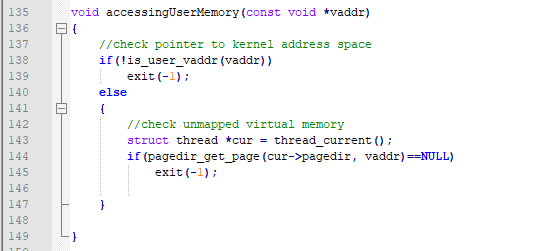




우선 입력 받은 file\_name input을 빈칸을 기준으로 parsing 하기 위해서는 strtok\_r 함수를 사용하였고, 이 함수를 이용하여 file\_name을 나눈 뒤, 각 token들은 argv 배열에 저장하였다. Stack에 쌓을 때는 아래로 쌓는 과정이므로 esp는 계속 감소하게 된다. 우선 제일 각 token들이 갖고 있는 값을 stack에 쌓아주고, 쌓은 주소는 espArr 배열에 저장하여 준다. 여기서 주의할 점은 word-alignment이다. 모든 token들을 전부 stack에 쌓았을 때, 쌓은 token들의 길이 합이 4의 배수가 아니라면, word-alignment가 필요하므로 4의 배수가 되기 위해 필요한 만큼을 추가한다. 그 후, NULL pointer sentinel을 추가한 후, 아까 위에서 token의 내용을 stack에 쌓았던 주소들을 갖고 있는 espArr배열의 내용들을 stack에 다시 한 번 쌓아준 후, espArr[0]을 쌓은 주소를 그 아래에 쌓고, token들의 총 개수를 쌓은 뒤, 마지막으로 return address을 쌓아주면 stack 쌓아주는 것이 완료된다.

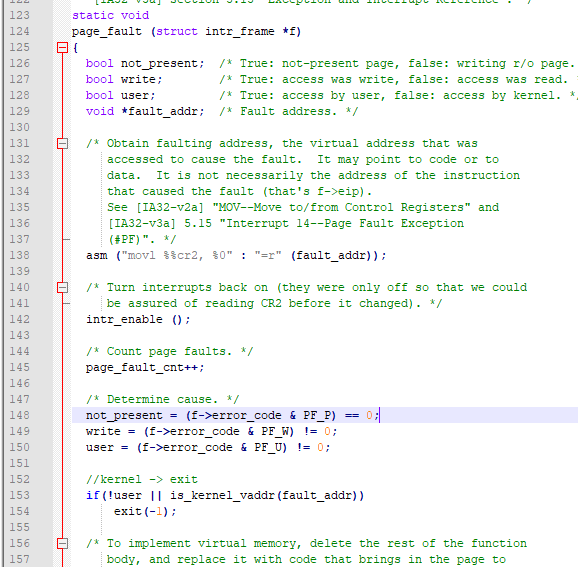
1. User Memory Access

Pintos에서 user program이 kernel 영역의 메모리에 접근을 시도하면 pagefault가 발생하게 된다. 따라서 이를 막기 위해 userprog/syscall.c에 void accessingUserMemory라는 함수를 정의하여 check할 수 있게 구현하였다.

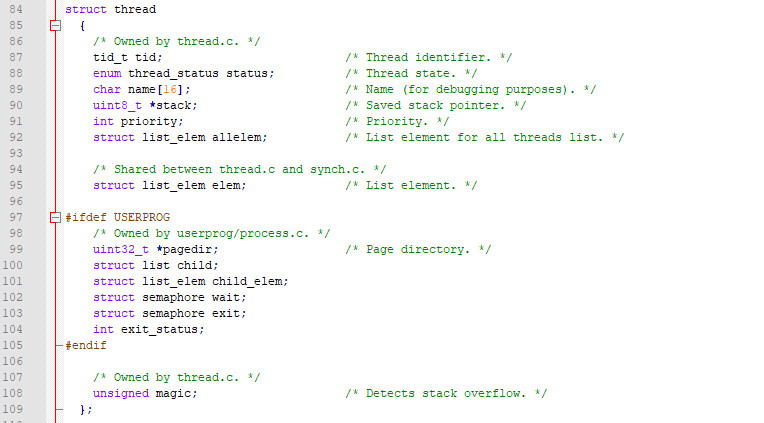


User program은 invalid pointer를 pass할 수 있다. 이러한 Invalid pointer를 막기 위해서 첫 번째로는 kernel address space를 가리키는 경우를 막는 것인데, 이 때는 threads/vaddr.h에 있는 is\_user\_vaddr()함수를 사용하여 만일 is\_user\_vaddr의 return 값이 false인 경우, 즉 user 공간이 아닌 다른 공간을 가리키는 경우, 종료시킨다. 두 번째로는 mapping되지 않은 virtual memory를 막기 위해서는 userprog/pagedir.c에 있는 pagedir\_get\_page()함수를 이용해 파라미터로 현재 thread의 pagedir와 현재 포인터를 넘겨 주어 NULL 값이 return 되면 unmapped되었다는 것이므로 종료시킨다. 위와 같은 방법으로 Invalid memory access를 막을 수 있게 구현하였다.

또한 만일 page\_fault()가 실행된 경우를 대비하여, userprog/exception.c에 있는page\_fault() 함수 내부에도 kernel에 접근한 경우, exit 해주는 아래와 같은 코드를 추가해 주었다.

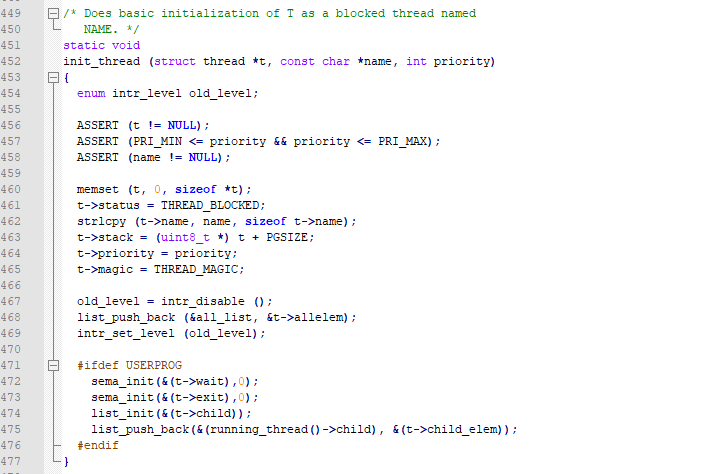


1. System Calls

우선 wait()로부터 return 전에 process가 종료되는 것을 막기 위해 semaphore를 사용하였다. 따라서 이에 대한 semaphore를 추가하기 위해 threads/thread.h의 thread 구조체 내부 #ifdef USERPROG #endif 사이에 아래와 같은 코드를 추가해주었다. 

우선 struct list child는 parent thread가 child thread를 가리키기 위해 사용되는 변수이고, project0-2에서와 마찬가지로 list 구조에서 list안의 구성 요소인 list\_elem 또한 선언해주었다. 그 후, semaphore wait와 exit을 선언하여 주었는데, semaphore wait는 parent process가 child process가 실행될 때 잠시 멈추고 있는 지시를 받기 위한 것이고, semaphore exit는 child process가 종료되었을 때, 이를 parent process로 알리기 위한 것이다. 마지막으로 exit\_status는 child process가 종료된 것인지 아닌지를 나타내는 status를 알려주는 변수이다.

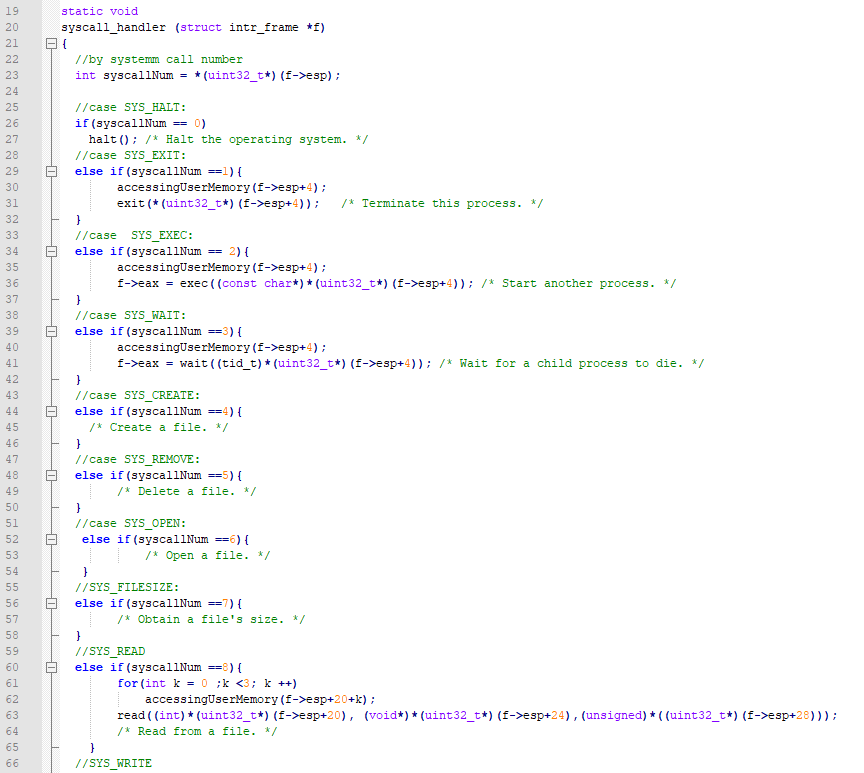
threads/thread.h에서 새로운 변수들을 정의하여 주었으므로 threads/thread.c에서 이에 관한 코드를 추가하였다. 수정한 부분은 thread.c의 inti\_thread 함수이다.

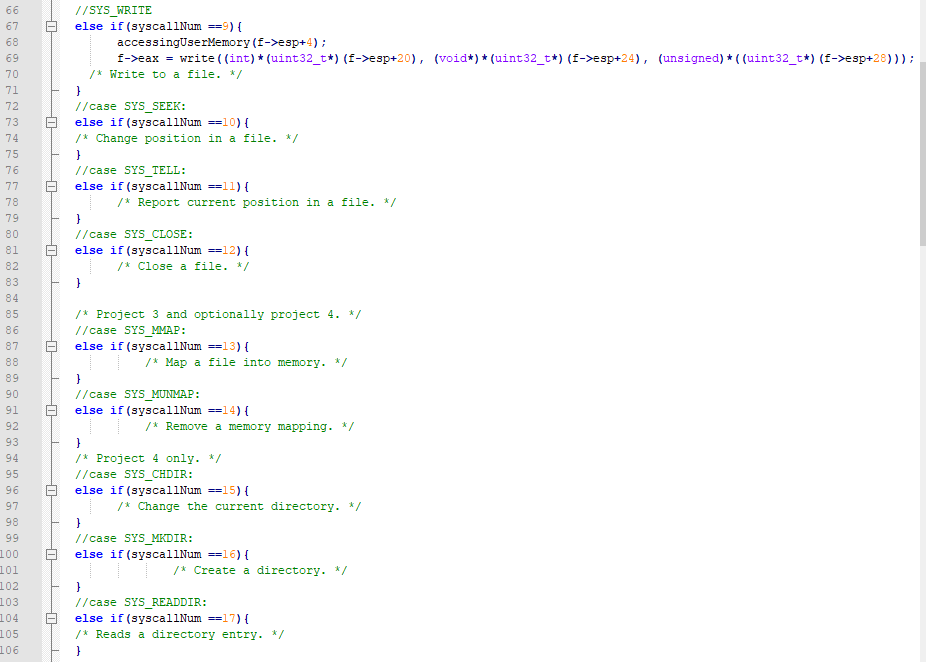


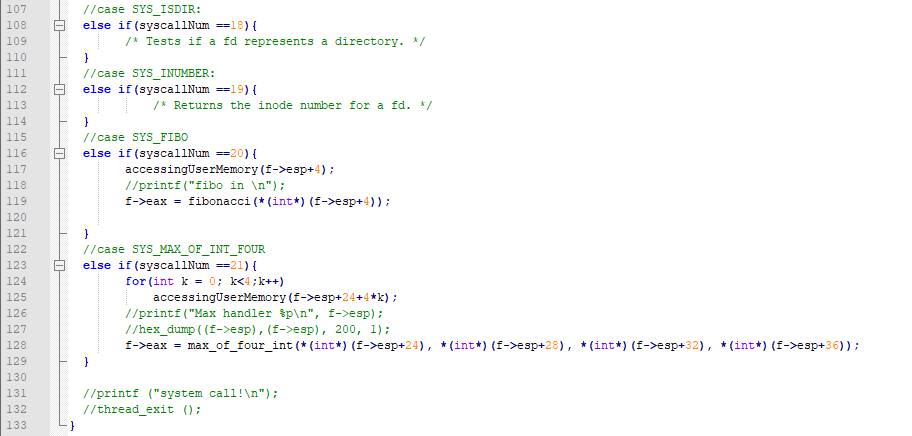
Init\_thread 함수에서 #ifdef USERPROG #endif 부분을 추가해주었는데, 해당 부분에서는 threads/synch.h에 정의되어 있는 sema\_init 함수를 사용하여 thread 내부의 semaphore wait와 exit을 초기화하여 준다. 또한 child 리스트를 초기화한 후, 현재 실행되고 있는 parent thread의 child\_thread를 list\_push\_back하여 주었다.

Syscall\_handler() :

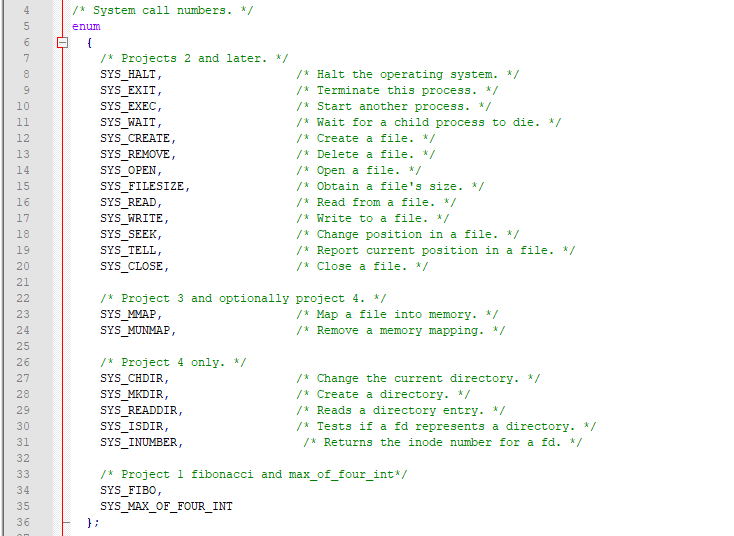
userprog/syscall.c에 있고 이를 수정하였다.







Syscall\_handler는 파라미터로 struct intr\_frame 타입 포인터 f를 받는다. 해당 파라미터의 구성요소인 esp가 lib/syscall-nr.h에 정의된 아래와 같은 syscall에 대한 int 값을 갖고 있게 된다.

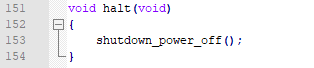


Syscall-nr.h에 정의된 system call numbers에 따라 해당 system call에 맞는 함수를 호출할 수 있도록 if-else 문으로 나눠서 구현하였다.

우선 project1 요구 사항이 halt, exit, exect, wait, read, write, Fibonacci, max\_of\_four\_int 이렇게 이므로 해당 사항에 관한 코드들만 구현하고 나머지 system call들에 대해서는 if문만 구현을 해 놓았다. 여기서 주의할 점은, 위에서 access User Memory를 위해 구현한 accessingUserMemory 함수를 사용하여 각각의 if/else if문에 들어오게 된 경우, stack pointer가 invalid pointer가 아닌지 check한 후, 만일 invalid pointer이면 accessingUserMemory 함수 내부에서 exit(-1)이 되고, 그게 아니라면 해당 내용에 맞는 system call 함수를 실행하게 된다.

SYS\_HALT :

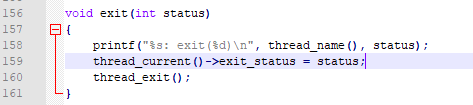
userprog/syscall.c에 구현하였다.



devices/shutdown.h에 정의되어 있는 shutdown\_power\_off() 함수를 사용하여 pintos program이 종료되게 한다.

SYS\_EXIT:

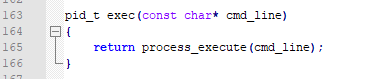
userprog/syscall.c에 구현하였다.



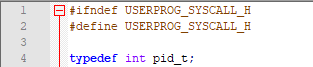
종료할 thread에 대한 exit message를 출력하는 printf()문을 구현하였다. 그리고 현재 수행중인 thread를 thread\_current()함수를 통해 알아내어 해당 thread의 exit\_status에 파라미터로 입력 받은 status를 대입하여 준다. 해당 과정을 구현함으로써 process\_wait에서 child thread가 종료되었다는 것을 status 값의 변화를 통해 알릴 수 있다. 그 후, thread\_exit()를 호출하여 current thread를 deschedule하고 destory해준다. thread\_exit(), thread\_current()함수는 threads/thread.h 파일에 있다.

SYS\_EXEC :

userprog/syscall.c에 구현하였다.



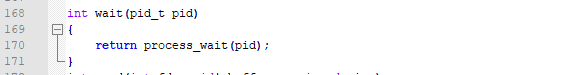
우선 userprog/syscall.h에 아래와 같이 process의 thread id를 뜻하는 pid\_t를 typedef 하여 주었다.



그리고 userprog/process.h에 있는 process\_execute 함수를 호출하여 새로운 thread를 시작할 수 있게 하고, 해당 함수로부터 새로운 process의 thread id를 return 받거나 아니면 thread가 만들어지지 않은 경우, TID\_ERROR를 return 받고 이를 다시 exec 함수가 return 하여 준다.

SYS\_WAIT :

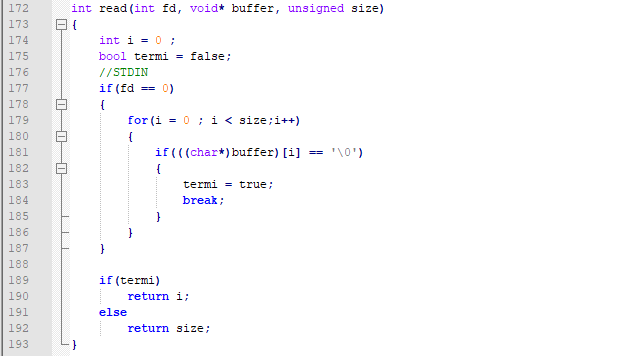
userprog/syscall.c에 구현하였다.



Userprog/process.h에 있는 process\_wait 함수에 thread id를 넘겨주어서 해당 thread id가 죽을 때까지 기다렸다가 그것의 exit status를 return 받은 후, 받은 exit status를 return한다.

SYS\_READ :

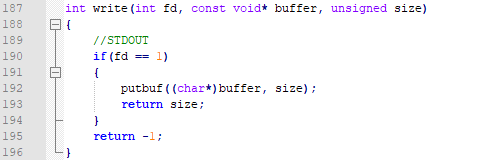
userprog/syscall.c에 구현하였다.



모든 구현은 하지 않고, STDIN으로부터 읽을 수 있게만 구현하였다. STDIN의 File Descriptor는 0이므로 이를 보고 판단하였다. 만일 STDIN이면 들어온 size만큼 for문을 반복하여 buffer에 한 글자씩 넣어주게 되고, 만일 for문이 size만큼 반복되기 전에 NULL 문자를 만나는 경우, 해당 지점까지의 크기를 return하여 준다. for문이 기존 size만큼 반복 된다면, size가 return된다.

SYS\_WRITE :

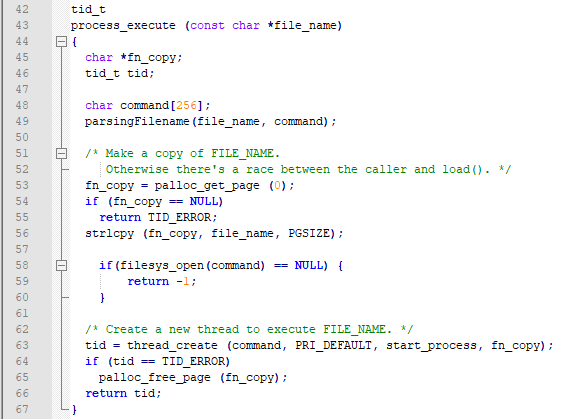
userprog/syscall.c에 구현하였다.



모든 구현은 하지 않고, STDOUT으로부터 읽을 수 있게만 구현한다. STDOUT의 File Descriptor는 1이므로 이를 보고 판단하였다. 만일 STDOUT이면 putbuf 함수를 사용하여 buffer에 저장된 data를 size만큼 가져온 뒤, 해당 size를 return한다.

process\_execute() :

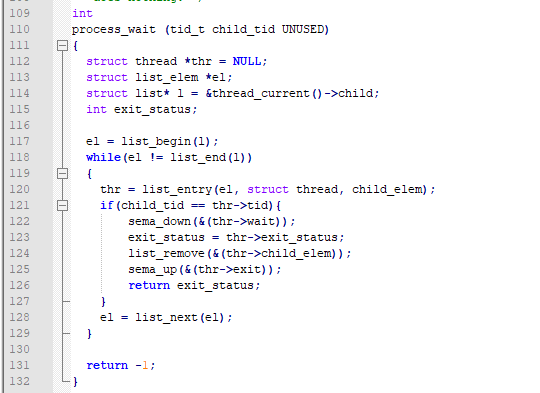
userprog/process.c 에 있고 이를 수정하였다.



파라미터로 넘어온 file\_name의 첫 번째 token을 parsingFilename 함수를 호출해서 얻은 뒤, 해당 tokend을 thread\_create 함수의 인자로 넘겨주어 새로운 thread를 생성하게 한다.

process\_wait() :

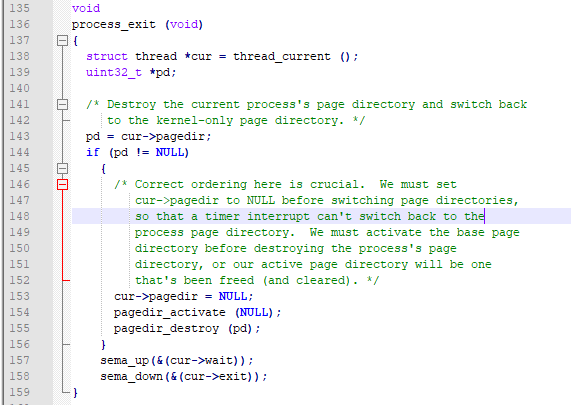
userprog/process.c 에 있고 이를 수정하였다.



Thread TID가 죽기를 기다리고, 그것의 exit status를 return 하는 함수이다. 만일 이것이 kernel에 의해 종료되었다면 -1을 return하게 된다. 위의 while 반복문은 현재 실행중인 current를 thread\_current()함수를 통해 찾은 뒤, 해당 thread의 child들을 하나씩 확인하기 위한 반복문이다. 해당 반복문을 통해 parent thread의 child thread를 비교하며 해당 thread의 tid가 파라미터로 입력 받은 child\_tid와 같다면, 이것은 현재 실행할 child thread와 동일하다는 것을 의미하고, 따라서 앞서 thread.h에 선언한 semaphore를 이용하여 현재 thread의 parent thread를 sema\_down을 활용해 잠시 정지시킨다. 그 후, 프로그램이 진행되다가 system call인 SYS\_EXIT이 실행되면 exit\_status의 값이 -1이 들어오게 될 것이고, 그 후 현재 thread에서 사용된 list를 list\_remove 함수를 통해 삭제한다. 그리고, parent thread에게 child thread가 종료되었다는 것을 알리기 위해 sema\_up을 이용하여 알린 뒤, exit\_status(-1) 값을 return하게 된다.

process\_exit() :

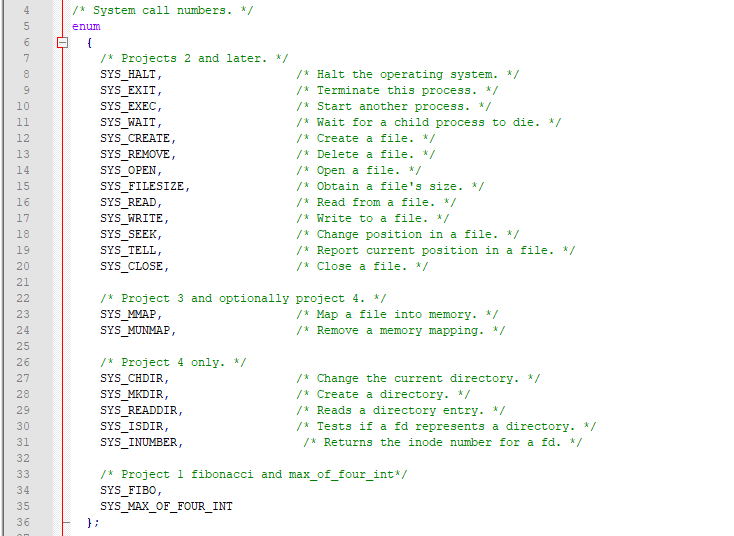
userprog/process.c 에 있고 이를 수정하였다.



process\_wait() 함수에서 sema\_down()을 호출하게 되면 parent process가 잠시 정지되게 된다. 그리고 child process가 진행되고 난 후, 종료될 때, process\_exit() 함수가 호출되게 되는데, 이것이 호출되기 위해서는 system call인 SYS\_EXIT가 호출될 것이다. 이 때, exit() 함수가 실행되면서 status 값을 -1로 호출하고 process\_exit 함수 내부에서는 sema\_up을 통해 child thread가 종료된 것을 알린다. 그리고 list를 삭제하기 위해 잠시 sema\_down을 하여 parent process를 정지시키고, process\_wait() 함수에서 child\_thread의 list가 제거되면 다시 sema\_up을 호출하여 완전히 종료되었음을 parent thread에게 알린다.

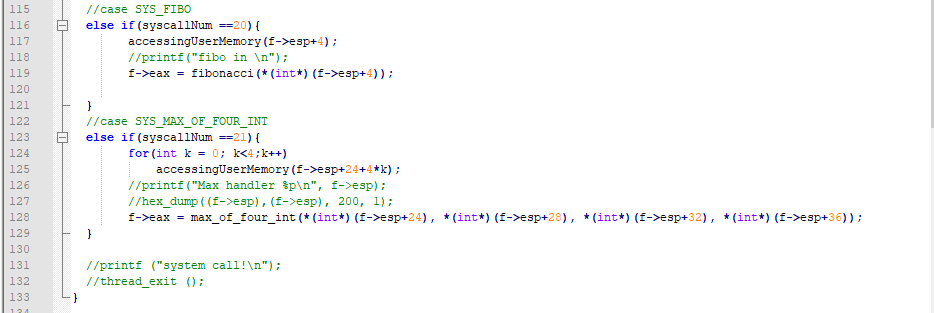
1. Additional System calls

lib/syscall-nr.h에 정의된 아래와 같은 system call number에 새로운 시스템 콜에 대한

system number인 SYS\_FIBO, SYS\_MAX\_OF\_FOUR\_INT를 추가해주었다. 

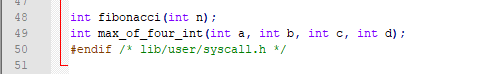
Syscall-nr.h에 정의된 system call numbers에 따라 해당 system call에 맞는 함수를 호출할 수 있도록 userprog/syscall.c에 있는 systcall\_handler 함수에 fibonacci와 max\_of\_four\_int 함수를 호출하는 if-else 문을 추가해주었다.





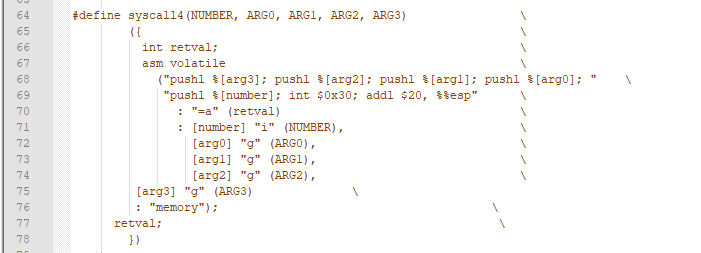
이 때 주의할 점은 해당 함수들도 호출하기 전에 accessingUserMemory 함수를 통해 invalid pointer를 갖고 있지 않은지 check한 후, 함수를 호출하여 주었다. Fibonacci의 경우, additional implementation을 수행할 때, 파라미터로 f->esp+4를 하니 input 값이 제대로 들어 갔으나, max\_of\_four\_int 함수의 경우 f->esp+4 ~ f->esp+16을 파라미터로 넘기니 값이 제대로 들어가지 않아 어려움을 겪었다. 해당 부분은 hex\_dump() 함수로 들어간 값들을 출력하여 이에 맞게 f->esp+24~f->esp+36을 하여주니 올바른 값이 파라미터로 들어가게 되었다. 그리고 각 함수들을 호출한 후 return 값을 f->eax에 대입하여 주었다.

lib/user/syscall.h에

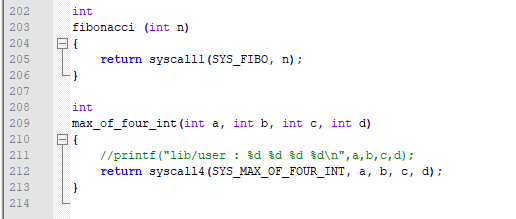


와 같이 fibonacci와 max\_of\_four\_int 함수에 대한 prototype을 정의하였다.

lib/user/syscall.c

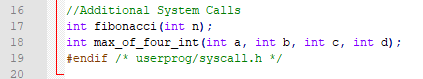


위와 같이 syscall.c에 max\_of\_four\_int를 위해 파라미터를 4개로 받는 syscall4를 정의하여 준다. 이 는 앞에 있는 syscall1~syscall3 정의 부분을 참고하여 작성하였고, arg3를 추가한 것과, stack pointer를 syscall3 보다 4byte 더 추가한 20으로 작성하였다.



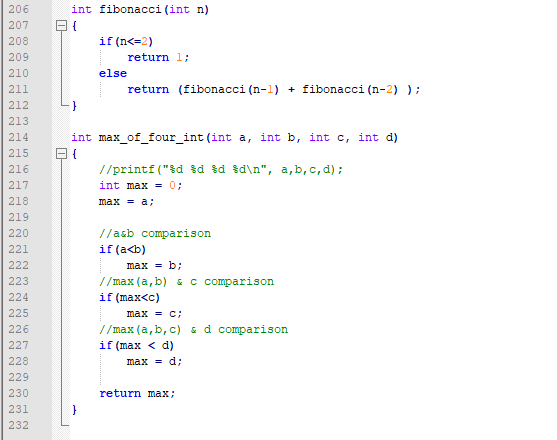
위와 같이 fibonacci 함수는 syscall1에, max\_of\_four\_int는 syscall4에 연결하여 system call이 실행될 수 있게 구현하였다.

userprog/syscall.h에

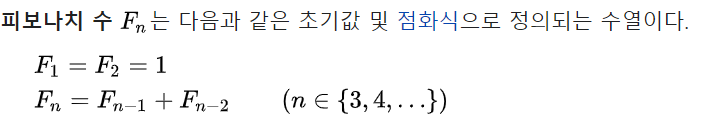


와 같이 fibonacci와 max\_of\_four\_int 함수에 대한 prototype을 정의하였다.

lib/user/syscall.c에



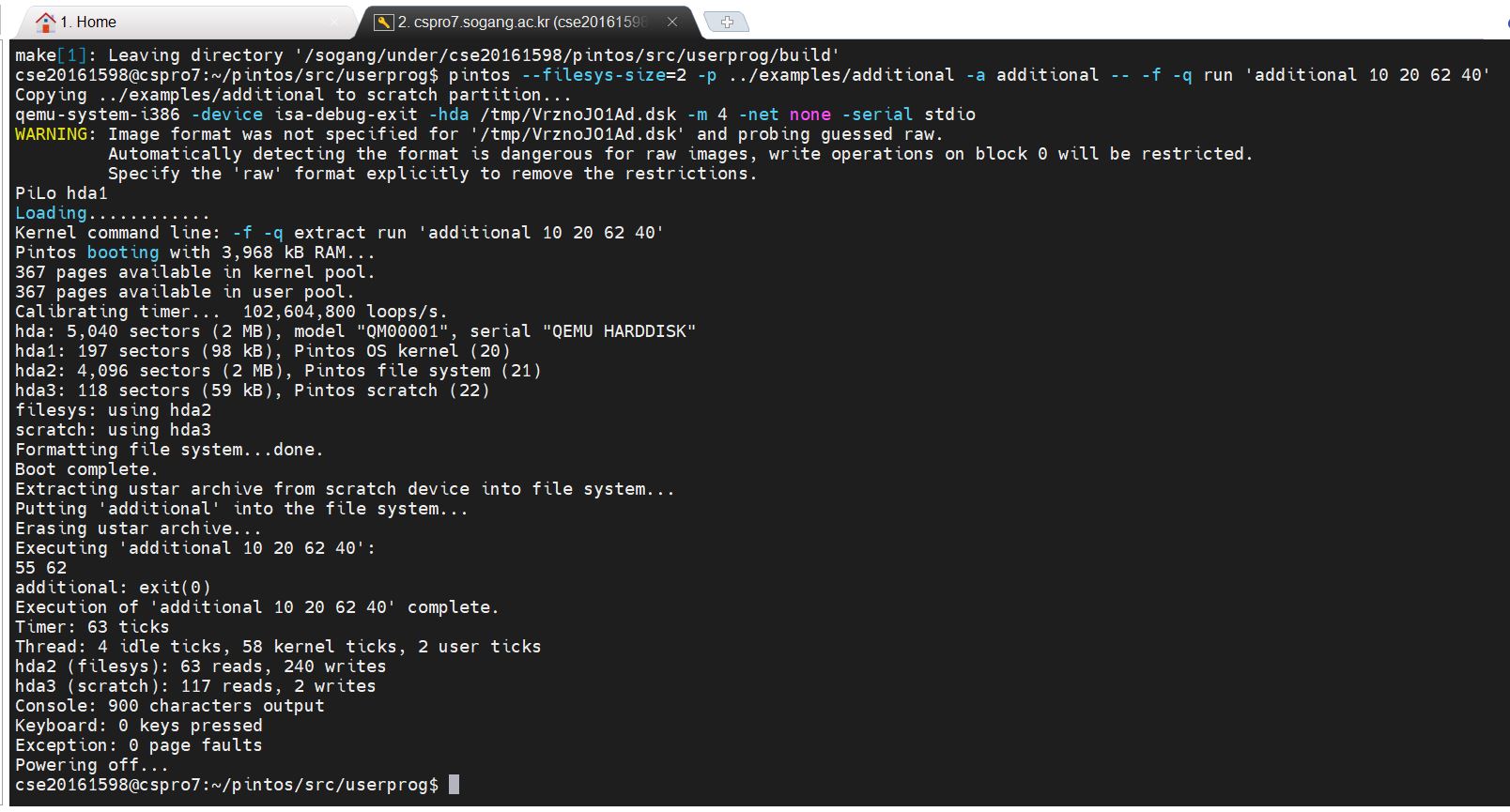
fibonacci와 max\_of\_four\_int 함수의 요구사항을 구현하였다. 우선 Fibonacci 수열은 아래와 같이 정의된다.



따라서 이를 코드로 구현하되, 재귀함수를 이용하여 구현하였다. 이렇게 하면 피보나치 수열의 n번째 값을 return하게 된다.

그 다음 max\_of\_four\_int 함수는 파라미터로 전달 받은 4개의 수 중에서 제일 큰 수를 반환하는 함수이다. 먼저 max 라는 가장 큰 값을 저장할 int 변수를 정의한 뒤, a와 b의 비교 값 중 큰 수를 max에 저장하고, max와 c의 큰 값을 max에 그리고 max와 d의 큰 값을 max에 저장하여 가장 큰 값을 return하게 된다.

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

****