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

담당 교수 : 김영재

조 / 조원 : 20xxxxxx

개발 기간 : 2024-09-16 ~ 2024-09-22

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

현재 주어진 pintos의 상태에서는 ‘echo x’와 같은 명령어의 실행 결과를 볼 수 없다. 이는 system call, system call handler, argument passing, user stack이 구현되어 있지 않기 때문이다. 이번 프로젝트의 목표는 위 요소들을 구현하여 명령어의 실행 결과를 볼 수 있도록 하는 것이다. 따라서 사용자는 user program을 정상적으로 사용할 수 있게 된다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

사용자로부터 user program을 실행하라는 명령을 받게 되면, pintos는 커널 영역에서 새로운 process를 생성한다. 그리고 입력 받은 명령어를 parsing하여 실행할 program name과 argument들을 얻는다. 이렇게 얻은 문자열들을 virtual memory의 user stack 영역에 적재한다. 이를 통해 사용자 영역에서 user program의 process가 argument를 정상적으로 사용할 수 있게 된다.

1. User Memory Access

사용자로부터 전달된 pointer가 유효한지를 검증하고, invalid pointer access를 방지할 수 있다. 이를 위해, system call을 통해 전달된 pointer가 user virtual memory 내에 있는지 확인하고, kernel memory에 속한 주소로 접근하려고 하는 경우 이를 차단한다. 잘못된 pointer access는 page fault를 발생시키며, 이를 적절히 처리하기 위해 page fault handler 함수를 적절히 수정한다. 따라서 user program이 invalid pointer access를 시도할 때 커널에 영향을 주지 않고, 안전하게 처리된다.

1. System Call

User program이 system call을 통해 kernel에서 제공하는 다양한 기능을 사용할 수 있다. System call을 호출하면 syscall\_handler() 함수가 system call 번호에 따라 적절한 작업을 수행한다. 여기서 exit, read, write 등의 system call에 대한 동작을 수행하는 함수를 구현하여 user program이 원하는 작업을 kernel 영역에서 안전하게 수행할 수 있게 된다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명
    1. User program에서 전달된 command line을 parsing하여 각 argument로 분리한다. strtok\_k() 함수를 이용하여 공백을 기준으로 잘라낸다. 분리된 각 argument는 argv 배열에 저장된다.
    2. 각 argument를 stack에 역순으로 복사한다. strlcpy() 함수를 사용하여 null-termination을 보장한다. Stack에 복사된 후 해당 포인터 값이 argv 배열의 해당 위치에 저장된다.
    3. Stack의 word alignment를 4byte 단위로 맞추기 위해 null-padding을 추가한다.
    4. Stack에 저장된 각 argument의 주소를 다시 stack에 차례대로 저장한다. 이는 user program이 argument를 참조할 수 있도록 하는 것이다.
    5. Argument의 개수인 argc를 stack에 저장하고, argv 배열의 시작 주소도 stack에 저장하여 user program이 argument를 참조할 수 있도록 한다. 마지막으로 return address를 null로 설정한다.
* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명  
    Pintos에서 invalid memory access란 user program이 잘못된 메모리 주소를 참조하거나 접근 권한이 없는 메모리 영역에 접근하려 할 때 발생하는 문제다. Null-pointer 참조 혹은 kernel 영역에 접근 시도가 이에 해당한다. 이러한 memory access는 page fault를 유발하며, kernel에서 이를 적절하게 처리해야 한다.
  + Invalid memory access를 어떻게 막을 것인지 설명

1. System call을 handling하고 작업을 수행하는 과정에서 interrupt frame에 push되어 전달된 주소가 올바른 user memory 영역에 속하는지 확인하여 만약 kernel memory 영역에 접근하려 한다면 프로세스를 종료시킨다. threads/vaddr.h 파일에 user memory와 kernel memory의 경계를 나타내는 PHYS\_BASE와 주소가 어느 영역인지 체크하는 함수가 정의되어 있으므로 이를 활용한다.
2. 앞서 잘못된 memory access를 시도하면 page fault가 발생한다고 하였다. 이런 상황이 발생하면 page fault handler 함수가 호출되는데 여기서 page fault의 원인을 분석하여 잘못된 memory access가 user program에 의해 발생한 경우 해당 프로세스를 종료시킨다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

User program은 file system 접근, 프로세스 관리, I/O 등의 기능을 직접 처리할 수 없기 때문에 system call을 통해 커널에 해당 작업에 대한 요청을 보내고, 커널에서 이를 수행한다. 이를 통해 user program이 커널의 자원을 안전하게 사용하여 시스템의 안정성이 유지된다.

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

1. halt()  
   시스템을 종료한다.
2. exit(int status)  
   프로세스를 종료하고 exit status를 반환한다.
3. exec(const char \*cmd\_line)  
   새로운 프로그램을 실행한다.
4. wait(pid\_t pid)  
   해당 pid 값을 가진 프로세스가 종료될 때까지 기다린 후 exit status를 반환한다.
5. read(int fd, void \*buffer, unsigned size)  
   File descriptor로부터 최대 size만큼 데이터를 읽어 buffer에 저장한다. 이번 프로젝트에는 STDIN에 대해서만 구현한다.
6. write(int fd, const void \*buffer, unsigned size)  
   Buffer의 데이터를 최대 size만큼 읽고 file descriptor에 데이터를 쓴다. 이번 프로젝트에서는 STDOUT에 대해서만 구현한다.
7. fibonacci(int n)  
   주어신 n에 대한 n번째 피보나치 수열 값을 계산하여 반환한다.
8. max\_of\_four\_int(int a, int b, int c, int d)  
   네 개의 정수 중 가장 큰 값을 반환한다.
   * 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명
9. System call 호출 (user mode)  
   User program에서 system call API가 호출된다. 이때 system call 번호와 인자가 interrupt stack에 저장된다.
10. Interrupt 발생 (switching to kernel mode)  
    “int $0x30” 명령어를 통해 interrupt를 발생시켜 kernel mode로 전환된다. Interrupt handler에서 “0x30” 값을 확인하여 system call handler를 호출하여 제어권을 넘긴다.
11. System call 처리 (kernel mode)  
    전달된 interrupt stack의 내용물을 확인하여 system call 번호 및 인자를 가져와 요청된 작업을 수행한다.
12. 결과 반환 (switching to user mode)  
    System call 수행 결과가 eax register에 저장되고, user mode로 변환된다.
13. **추진 일정 및 개발 방법**
    1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

09/18 ~ 09/19 : Argument passing 구현

09/20 ~ 09/21 : User memory access, system call handling 구현

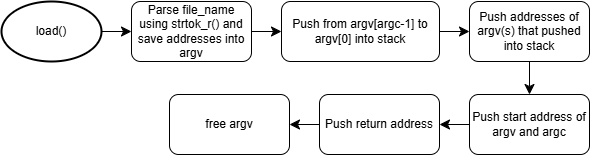
09/22 : Additional 구현

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

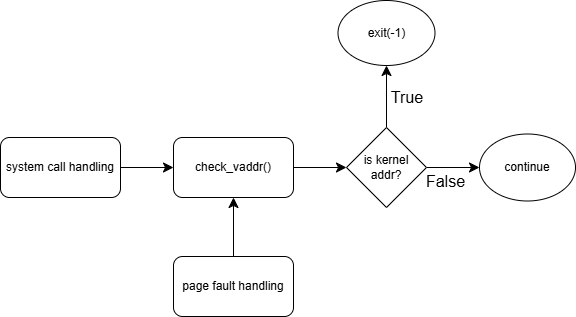
1. src/lib/user/syscall.c  
   4개의 인자를 받는 system call을 invoke하는 macro 함수를 정의한다. 그리고 추가적인 system call 호출을 위한 wrapper 함수를 만들고 헤더 파일에 prototype을 선언한다.
2. src/threads/thread.h  
   자식 프로세스의 상태 관리를 위해 pintos의 list 자료구조를 이용하여 thread 구조체에 child list와 child element 변수를 추가한다. 그리고 프로세스 종료 상태를 저장하는 변수를 추가한다. 프로그램이 정상적으로 load되었는지 알리는 Boolean 변수를 추가한다. 마지막으로 부모 프로세스가 자식 프로세스의 load 확인과 reaping 과정에서 synchronization을 위한 semaphore 변수들을 추가한다.
3. src/threads/thread.c  
   init\_thread() 함수에 semaphore 초기화, 자식 list 초기화, 부모의 자식 list에 삽입 과정을 추가한다.
4. src/userprog/exception.c  
   page\_fault() 함수에서 커널 영역에 접근하려는 시도가 발생한 경우 프로세스를 강제 종료 시키는 코드를 추가한다.
5. src/userprog/process.c  
   - load() 함수에서 file\_name을 parsing하고 stack에 적재하는 과정을 추가한다.  
   - process\_execute() 함수에서 자식 프로세스의 load semaphore를 기다린 후 실패하면 -1를 반환하는 코드를 추가한다.  
   - start\_process() 함수에서 load 성공 여부를 저장하고 semaphore를 올린다. 그리고 실패하면 프로세스를 종료시킨다.  
   - process\_wait() 함수와 process\_exit() 함수에서 프로세스 종료 과정이 정상적으로 진행되도록 semaphore 제어 과정을 추가한다.
6. src/userprog/syscall.c  
   각 system call에 대한 동작을 수행하는 함수를 정의한다.
7. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

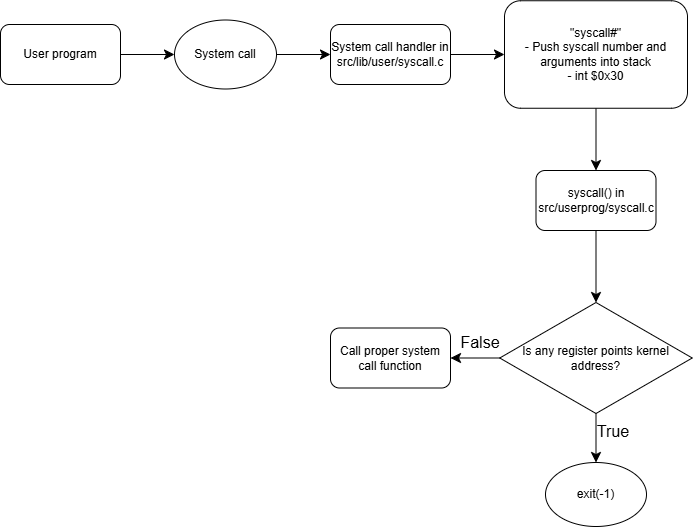
1. Argument Passing



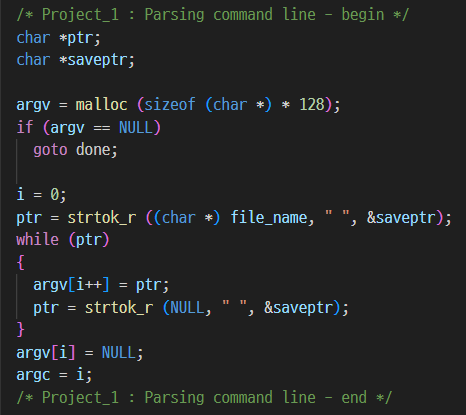
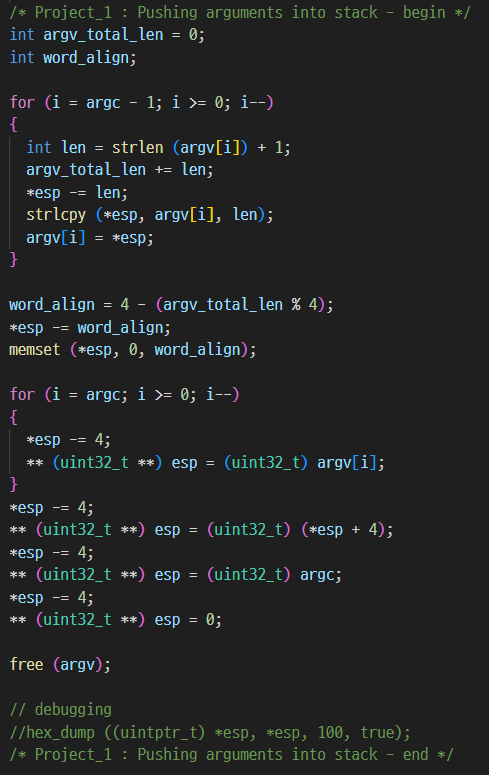
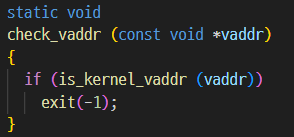
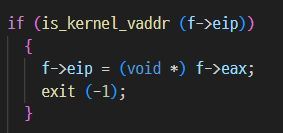
1. User Memory Access

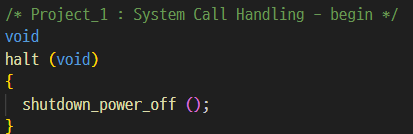


1. System Calls



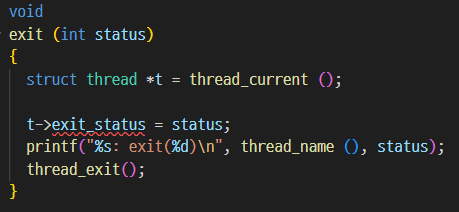
* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing  
   1-1. Parsing command line  
     
   char pointer형을 담는 argv배열을 동적 할당한다. 그리고 file\_name에 대해 strtok\_r()을 수행하여 argv 배열에 주소를 담는다. strtok\_r() 함수는 strtok과 기능은 같지만 thread safe하다는 차이가 있다.  
   1-2. Pushing arguments into stack  
     
   argv와 argv의 주소 및 argc를 stack에 적재하는 코드다.
2. User Memory Access  
   2-1. Memory access from system calls  
     
   src/threads/vaddr.h에 정의되어 있는 is\_kernel\_vaddr() 함수를 이용하여 system call에서 접근하는 메모리 영역이 kernel 영역인 경우 exit(-1)을 통해 프로세스를 종료시키도록 구현하였다.  
   2-2. Page fault handling  
     
   Page fault가 발생하면 fault가 발생한 instruction의 주소를 확인하여 커널 영역에 접근하려는 시도가 확인되면 프로세스를 강제종료 시킨다.
3. System Calls

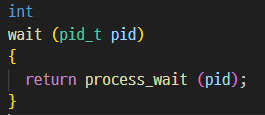
* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**  
  3-1. halt()  
  

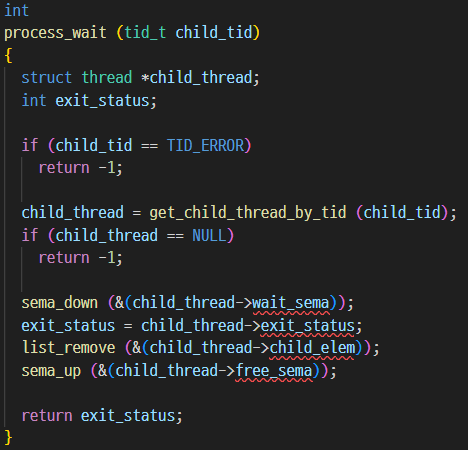
pintos에서 제공하는 시스템을 종료하는 함수인 shutdown\_power\_off() 함수를 호출한다.

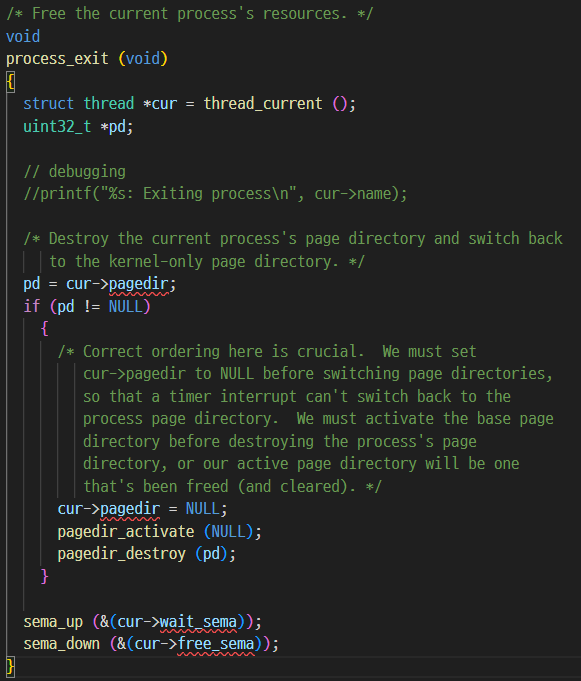
3-2. exit(), wait()



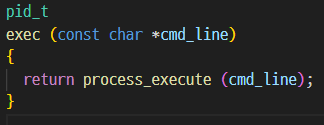
Exit system call의 경우 해당 system call을 호출한 thread의 exit\_status를 인자로 받아온 status로 변경한다. 그리고 프로세스가 종료된다는 것을 알리기 위해 명세서의 형식에 맞게 출력한다. 마지막으로 thread\_exit() 함수를 호출하여 thread를 종료하고 자원을 회수한다.

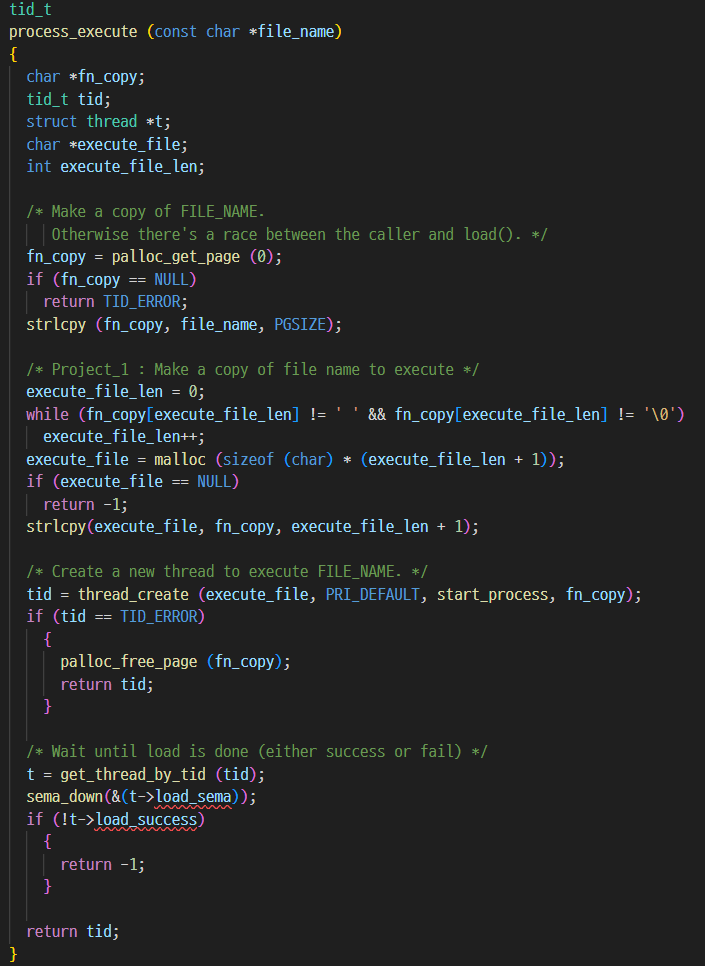


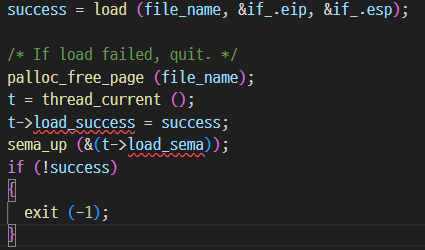


  
wait system call의 경우 process\_wait() 함수를 구현하여 이를 호출하도록 구성하였다. process\_wait() 함수의 구현 내용은 우선 tid가 잘못되었거나 현재 프로세스의 자식이 아닌 경우 -1를 반환하도록 한다. 그리고 앞서 exit에서 호출하는 thread\_exit() 함수의 내부를 살펴보면 process\_exit() 함수를 호출하게 되는데, 프로세스의 자원을 정리한 후 그 다음에 바로 process\_wait() 함수에서 프로세스의 exit\_status를 가져오면서 자식 list에서 삭제하는 작업을 수행하도록 하기 위해 semaphore를 제어하는 과정을 구현하였다.

3-3. exec()

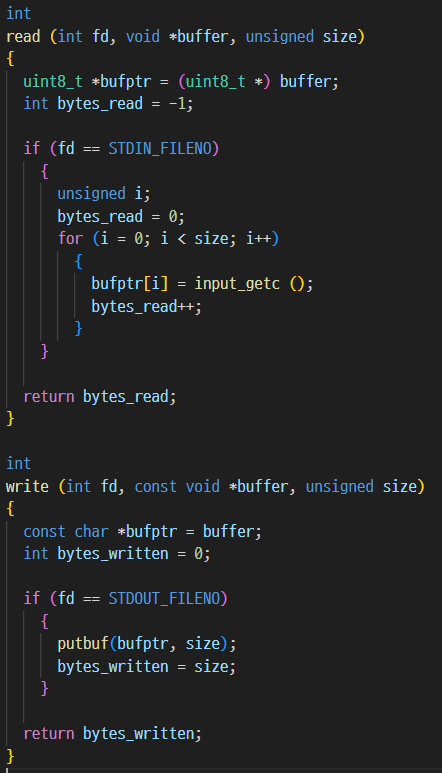


  
exec system call의 경우 process\_execute() 함수를 호출하도록 구성하였다. 그리고 process\_execute() 함수에서 부모 프로세스가 자식의 load가 정상적으로 수행되었는지 확인하기 작업, 그리고 이 과정에서 synchronization을 위한 semaphore 제어를 구현하였다. 또한 thread\_create() 함수에 실행 파일 이름을 넘겨주기 위해 parsing 작업이 필요했는데 file\_name에 그대로 parsing을 수행하면 Kernel panic이 발생하여 복사본을 만든 후 실행 파일 이름을 parsing하는 과정을 구현했다.



마지막으로 start\_process() 함수에서 load의 성공 여부를 저장하고 semaphore를 제어하는 과정, 그리고 load에 실패하면 exit status -1로 프로세스를 종료하는 작업을 구현하였다.

3-4. read(), write()

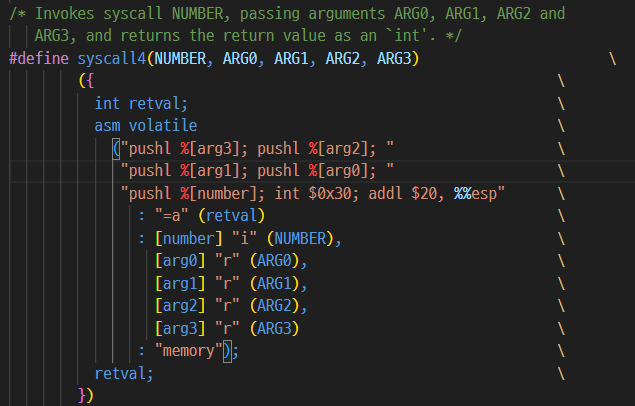


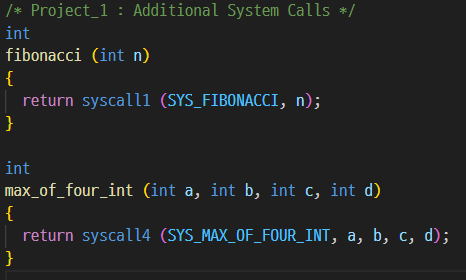
Read와 write system call의 경우 표준입력과 표준출력에 대해서만 구현하였다. pintos에서 제공하는 input\_getc() 함수와 putbuf() 함수를 이용하였다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

4-1. src/lib/syscall.c

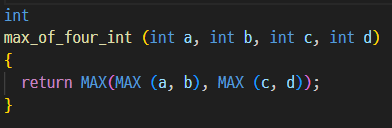
  
syscall을 invoke하는 매크로 함수가 최대 3개의 인자를 받는 것만 구현되어 있었기 때문에 4개의 인자를 받을 수 있는 매크로 함수 syscall4를 추가하였다.



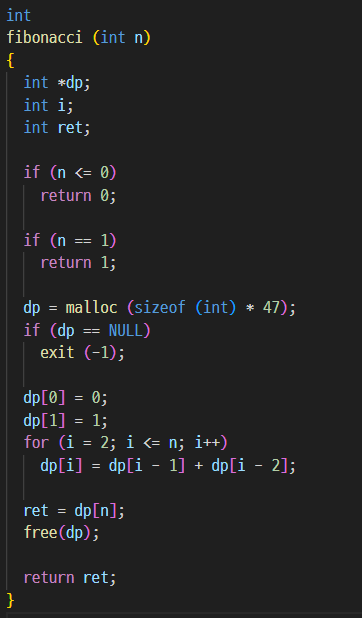
그리고 syscall invoke 함수를 호출해주는 함수를 구현하였다.

4-2. src/userprog/syscall.c

System call에 대한 동작을 수행하는 함수를 구현하였다.

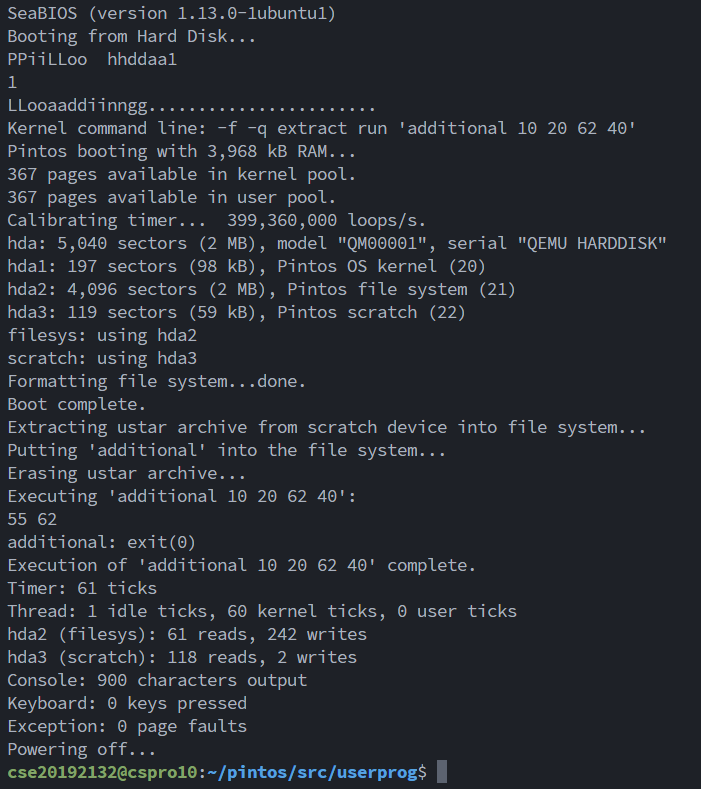


4개의 정수에서 가장 큰 값을 반환하는 max\_of\_four\_int의 경우 두 정수 중 더 큰 값을 반환하는 MAX 매크로 함수를 정의, 3번 사용하여 가장 큰 값을 반환하도록 구현하였다.



n번째 피보나치 수열의 값을 찾기 위해 dynamic programming 알고리즘을 사용하였다. 47번째 이상의 경우 int형의 범위를 벗어나기 때문에 46번째 값 까지만 계산하도록 구현하였다.

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

****