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

담당 교수 : 김영재 교수님

조 / 조원 :정성엽(학번 : 20161641)

개발 기간 : 2020/10/15~2020/11/1

1. **개발 목표**

- **해당 프로젝트에서 구현할 내용을 간략히 서술**

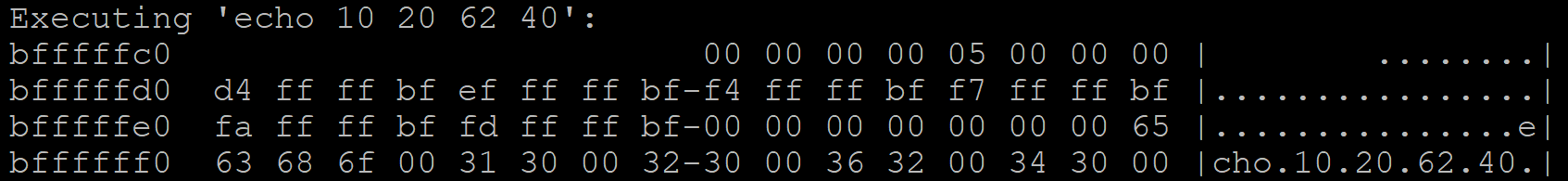
Pintos 운영체제에서 프로그램의 실행이 정상적으로 동작하도록 프로그램의 인자를 stack에 적제하고, 프로그램이 정상적으로 실행되도록 Synchronization을 적용하여 Process의 execute, wait, exit을 구현합니다. 또한 사용자의 프로그램이 운영체제와 상호작용할 수 있도록 System Call의 동작을 구현하는 것이 이번 프로젝트의 목표입니다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

: Process가 생성될 때 인자로 받는 Argument들을 프로세스의 생성과 동작에 사용 가능하도록 인자로 받은 문자열을 공백 기준으로 Parsing하여 이를 Stack에 저장합니다. 구현 결과를 hex\_dump() 함수를 통해 정상적으로 stack에 저장됨을 확인할 수 있습니다.



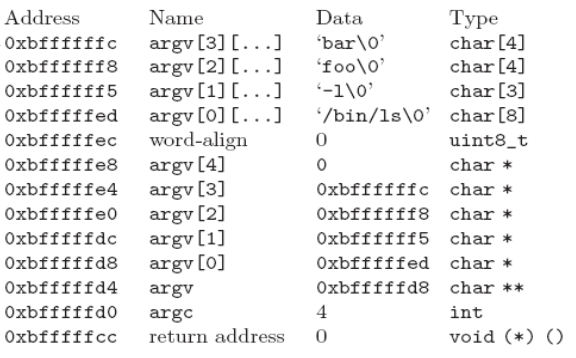
1. User Memory Access

: User Process는 user에게만 할당된 메모리를 참조해야 합니다. 하지만 user에게 할당된 포인터가 User영역이 아닌 Kernel영역 혹은 아예 틀린 주소를 가리키는 경우가 발생할 수 있고 커널의 보호를 위해 이를 미리 방지하여야 합니다. check\_user\_vaddr() 함수를 구현하여 User가 참조할 메모리 영역이 User에게 할당된 영역인지 검사해 줌으로써 이를 해결합니다. 이를 통해 부적절한 명령어로 인해 오류가 발생하여 Page fault가 발생하거나 Pintos가 오작동하는 대신 프로세스가 에러를 표기하며 종료합니다.

1. System Calls

: Pintos상에서의 System Call은 systemcall\_handler() 를 통해 System Call을 호출하면, 이를 인자의 개수마다 각각의 syscal0() 매크로를 통해 Stack에 저장, int $0x30 명령어를 통해 interrupt를 발생, interrupt handler를 통해 system call handler를 호출. 해당 위치로 가서 해당 system call 호출. 과 같은 과정을 통해 이루어집니다. 이러한 구현을 완료하면, 정상적으로 system call이 이루어져 해당하는 system call의 동작 결과가 출력됩니다(혹은 동작합니다).

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

: pintos의 Argument Passing은 Stack을 사용하여 이루어집니다. 명세서 요구조건으로 userprog/process.c 의 load() 함수 상에 이를 구현해야 합니다. 때문에 문자열로 받아온 argument들을 parsing 하고, 이를 Stack에 차례대로 쌓습니다. 초기 esp = 0xC0000000를 기준으로 ward aliasing을 하면서 각 parsing 결과들을 저장합니다. Memory 상의 stack은 아래로 growth 함으로 실제로 “/bin/ls -l foo bar” 라는 argument가 입력된다면 아래와 같이 저장됩니다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명

user program의 Memory access는 user에게만 할당된 메모리를 사용 가능합니다. 하지만 구현상의 오류나 명령어의 오타 등의 이유, NULL 포인터, 포인터 초기화 문제 등의 이유로 적절하지 않은 메모리를 참조할 수 있습니다. 이러한 경우를 방지하기 위해 user가 메모리를 참조하는 경우 pointer 가 적절한 영역을 참조하는지 검사합니다. 이를 위해 check\_user\_vaddr() 함수를 구현하여 User가 참조할 메모리 영역이 User에게 할당된 영역인지 검사해 줍니다. is\_user\_vaddr() 함수를 사용하여 user 메모리 영역을 참조하는지 검사하고 만약 잘못된 메모리 영역을 참조한다면 오류를 발생시키지 않도록 해당 프로세스를 종료합니다. 또한 Page fault 를 판단하는 함수 /userprog/exception.c 의 page\_fault() 함수에 page fault에 대하여 적절히 동작하도록 함수의 구현을 추가합니다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

user program은 다양한 동작을 수행할 수 있습니다. 하지만 모든 동작이 가능하도록 방치한다면 user의 실수나 악의적인 의도로 인해 운영체제가 정상적으로 동작하지 않을 수 있습니다. 때문에 위험할 수 있는 동작들에 대하여 kernel만이 할 수 있도록 구현되어 있습니다. 만약 User가 kernel의 권한이 있어야 수행가능한 서비스를 사용하고자 한다면 System Call을 통해 이를 사용할 수 있습니다. Kernel은 User의 요청을 검사하고, 미리 오작동을 방지할 수 있습니다.

이번 프로젝트에서 구현한 System Call은 아래와 같습니다.

1) halt(): 시스템을 종료합니다.

2) exit() : process(thread)를 종료하고 종료 결과를 출력합니다.

3) exec() : process(thread)를 실행합니다.

4) wait() : process(thread)를 기다리게 합니다.

5) read() : 이번 프로젝트에서는 STDIO에 대해서만 입력을 받습니다.

6) write() : 이번 프로젝트에서는 STDIO에 대해서만 출력합니다.

7) fibonacci() : 입력받은 N번째 피보나치 수를 출력합니다.

8) max\_of\_four\_int() : 입력 받은 4개의 수 중 가장 큰 1 개를 출력합니다.

pintos의 System Call은 다음과 같이 수행됩니다. User Level에서 System Call API를 호출하면, syscall0 메크로를 통해 system call number와 argument들이 stack에 저장됩니다. 이후 int $0x30 명령어를 통해 interrupt를 발생, interrupt handler를 통해 system call handler를 호출합니다. system call handler는 해당 syscall number에 해당하는 system call을 호출하고, 실행 결과는 eax에 저장되어 현재 과정을 다시 거슬러 올라가 결과를 반환합니다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

1) 2020/10/15 : Pintos 프로젝트 이해 및 코드 분석, 문재 해결방법 구상

2) 2020/10/16 : Argument Pa ssing, System Call 일부 구현

3) 2020/10/17 : System call handler 구현 및 Synchronization 구현

4) 2020/10/18 : Synchronization 디버깅 및 User address 처리

5) 2020/10/30 : 디버깅, Additional 구현

6) 2020/10/31 ~ 2020/11/1 : 디버깅, 보고서 작성

* 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)**
* Argument Passing

: Argument Passing을 구현하기 위해 프로젝트 명세서에 따라, userprog/process.c 의 load() 함수의 시작 부분에서 Parsing 을 수행합니다. 이후, load() 함수의 끝 부분에 parsing 한 결과를 esp를 통해 stack에 저장하는 코드를 추가해야 합니다.

* User Memory Access

: /userprog/syscall.c 에 check\_user\_vaddr() 함수를 구현하여 syscall\_handler() 함수에서 User메모리 참조가 있을 때 마다, User가 참조할 메모리 영역이 User에게 할당된 영역인지 검사해 줍니다. is\_user\_vaddr() 함수를 사용하여 검사 함수 check\_user\_vaddr()을 만들어, user 메모리 영역을 참조하는지 검사하고 만약 잘못된 메모리 영역을 참조한다면 오류를 발생시키지 않도록 해당 프로세스를 종료합니다(exit(-1)). 또한 Page fault를 판단하는 함수 /userprog/exception.c 의 page\_fault() 함수에 page fault에 대하여 적절히 동작하도록 함수의 구현을 추가합니다. (page 가 없을 경우 exit(-1) 호출)

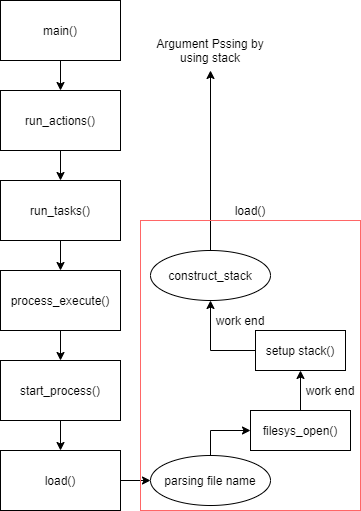
* System Calls

: system call의 정상작동을 위해 /userprog/syscall.c 에 syscall\_handler() 함수를 각 시스템 호출에 해당하는 enum 타입의 수에 맞게끔 동작하도록 switch문을 사용하여 각 경우를 분류한다. /userprog/syscall.c 상에 호출할 halt(), exit(), exec(), wait(), read(), write() 각 system call 함수를 구현한다.

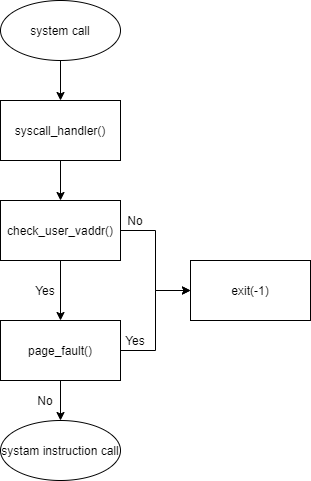
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

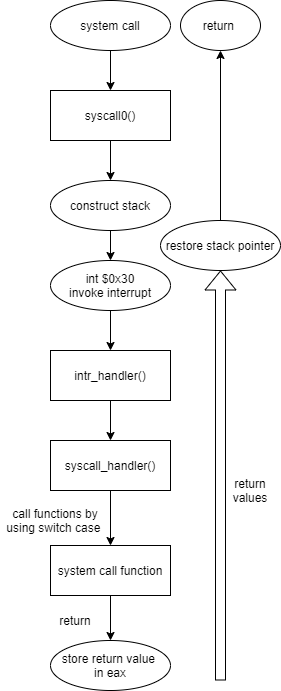
1. Argument Passing



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

: userprog/process.c 의 load() 함수에서 strtok\_r() 함수를 사용하여 file\_name의 parsing을 수행하였다. strsok\_r() 함수는 pintos에 내장된 함수인데, 이 함수는 문자열의 parsing 기준 문자를 지정하여 해당 문자를 기준으로 문자열을 분리하여 각 문자열의 시작 pointer를 반환한다. 이 함수의 장점은 연속된 공백이 있는 경우, 연속된 공백문자 모두를 ‘\0’으로 대치하고 다음 문자열의 시작 주소를 반환하여, 연속된 공백문자 처리가 용이하다는 장점이 있다. 초기 개발과정에서 strtok\_r() 함수가 아니라, for문과 같은 반복문의 사용을 통해 문자열 parsing을 하고자 하였으나, 연속된 공백문자 처리가 어려웠다. 하지만 이를 strtok\_r() 함수를 통해 해결하였다.

parsing이 완료된 후, parsing된 문자열을 esp를 기준으로 stack에 차례대로 저장합니다. stack은 growth downward임으로 각 인자들은 \0을 기준으로 argv의 높은 index부터 차례대로 저장됩니다. 인자의 저장이 끝나고, word alignment를 위해 esp가 4의 배수가 되도록 0을 추가합니다. 이후 0을 입력하여 구간을 구분해주고, 아까 저장한 인자들의 stack 저장 위치를 차례대로 저장합니다. 마지막으로 인자들의 개수 argc를 저장하고 return address를 저장함으로써 parameter를 stack에 저장하는 과정이 완료됩니다.

1. User Memory Access

: is\_user\_vaddr() 함수를 사용하여 /userprog/syscall.c 에 check\_user\_vaddr() 함수를 구현하여 syscall\_handler() 함수에서 User메모리 참조가 있을 때 마다, User가 참조할 메모리 영역이 User에게 할당된 영역인지 검사해 줍니다. user 메모리 영역을 참조하는지 검사하고 만약 잘못된 메모리 영역을 참조한다면 오류를 발생시키지 않도록 해당 프로세스를 종료합니다(exit(-1)). 또한 Page fault를 판단하는 함수 /userprog/exception.c 의 page\_fault() 함수에 page fault에 대하여 적절히 동작하도록 함수의 구현을 추가합니다. user가 아니거나, 만약 fault\_addr이 kernel영역을 침범한다면 오류를 발생시키지 않도록 프로세스를 종료합니다. (exit(-1))

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

: system call을 수행하기 위해서는 system call handler와 각 system call 함수의 구현, 그리고 process(thread)의 정상작동을 위한 Synchronization의 구현이 필요합니다. 우선 syscall\_handler() 함수의 경우, esp를 통해 stack에 저장된 argument의 값을 읽어 각 함수를 수행합니다. 이를 구현하기 위해 각 system call 종류에 따라(enum) switch-case문을 사용해 각 system call 종류에 따라 parameter가 user 메모리를 참조하는지 검사하고 사용할 system 함수를 호출합니다. 함수의 결과는 eax를 사용해 저장되고 반환됩니다.

이번 프로젝트에서 구현한 System Call의 구현방법은 아래와 같습니다.

1) halt(): shutdown\_power\_off() 함수를 통해 시스템을 종료합니다.

2) exit() : thread의 exit\_status를 저장하고, thread\_name()을 사용해 종료된 thread의 이름과, 종료 결과를 출력합니다. thread\_exit()의 호출을 통해 thread를 종료합니다.

3) exec() : process\_execute() 함수를 통해 process(thread)를 실행합니다.

4) wait() : process\_wait() 함수를 통해 process(thread)를 기다리게 합니다.

5) read() : fd를 확인하여 STDIO에 대해서만 buffer를 읽어, 입력을 받습니다.

6) write() : fd를 확인하여 STDIO에 대해서만 buffer에 putbuf() 함수를 사용해 출력합니다.

이때 process의 동기화를 위해 semaphore를 사용해 process\_wait(), process\_exit의 처리를 해주었습니다.

1. Additional System calls

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

새로운 system call을 구현하기 위해 /src/lib/syscall-nr.h의 enum 선언부에 추가할 system call에 해당하는 번호를 추가합니다. (SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INTEGERS 추가) 또한 max\_of\_four\_int() 함수는 4개의 인자를 입력으로 받는데, pintos에서 기본으로 제공되는 syscall 메크로는 최대 3개 까지의 인자만을 전달함으로 4개의 인자를 전달하는 syscall4 메크로를 만들었습니다.

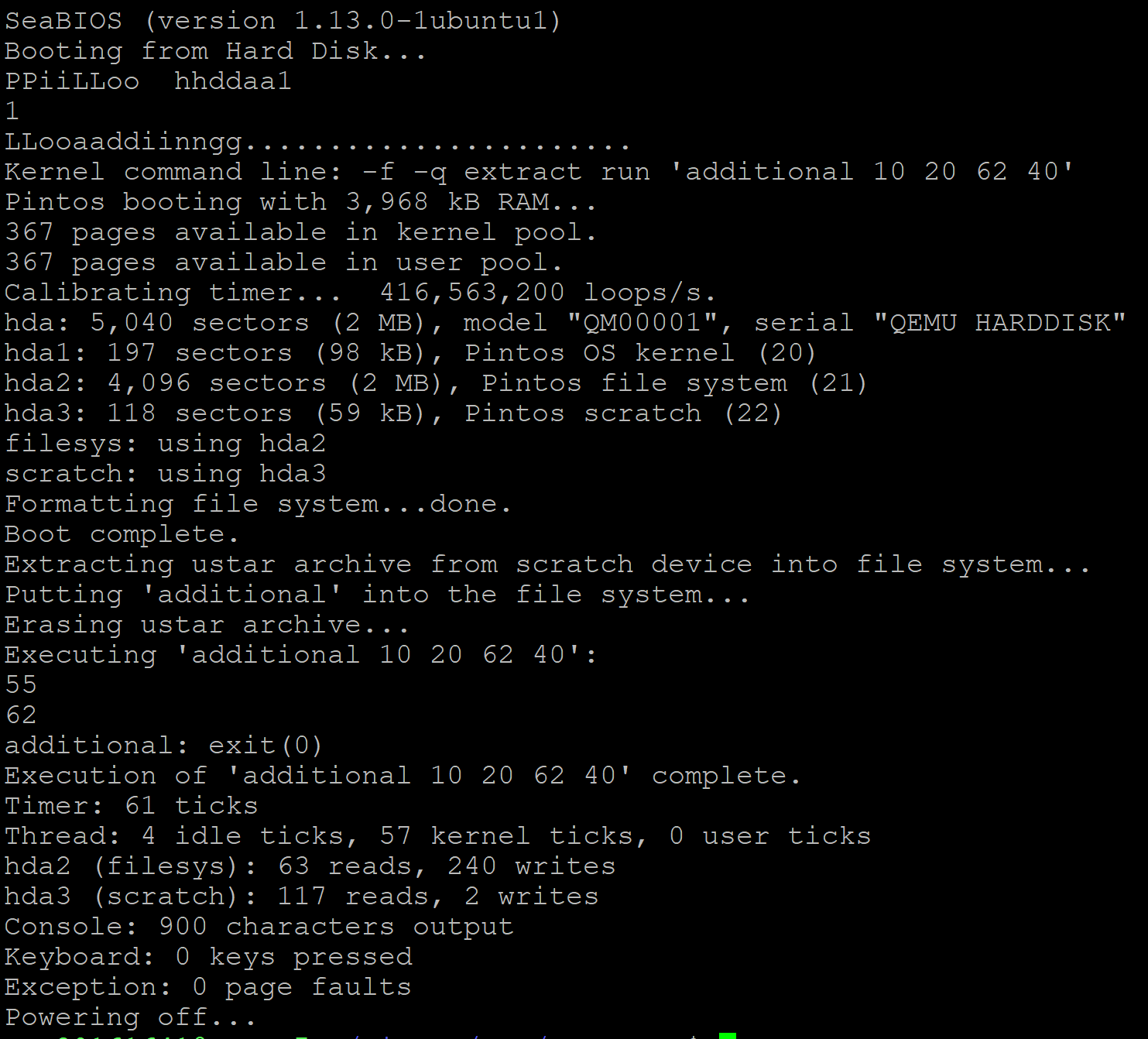
이후 /src/examples/additional.c에 system call을 호출하고 반환 값을 출력하는 main() 함수를 작성합니다. 그리고 Makefile을 수정하여 additional.d, additional.o 파일을 생성합니다. 마지막으로 /src/userprog/syscall.c 파일에 해당하는 아래의 system call 함수를 만들어줍니다.

1) fibonacci() : 반복문을 통해 N번째 피보나치 수를 출력합니다.

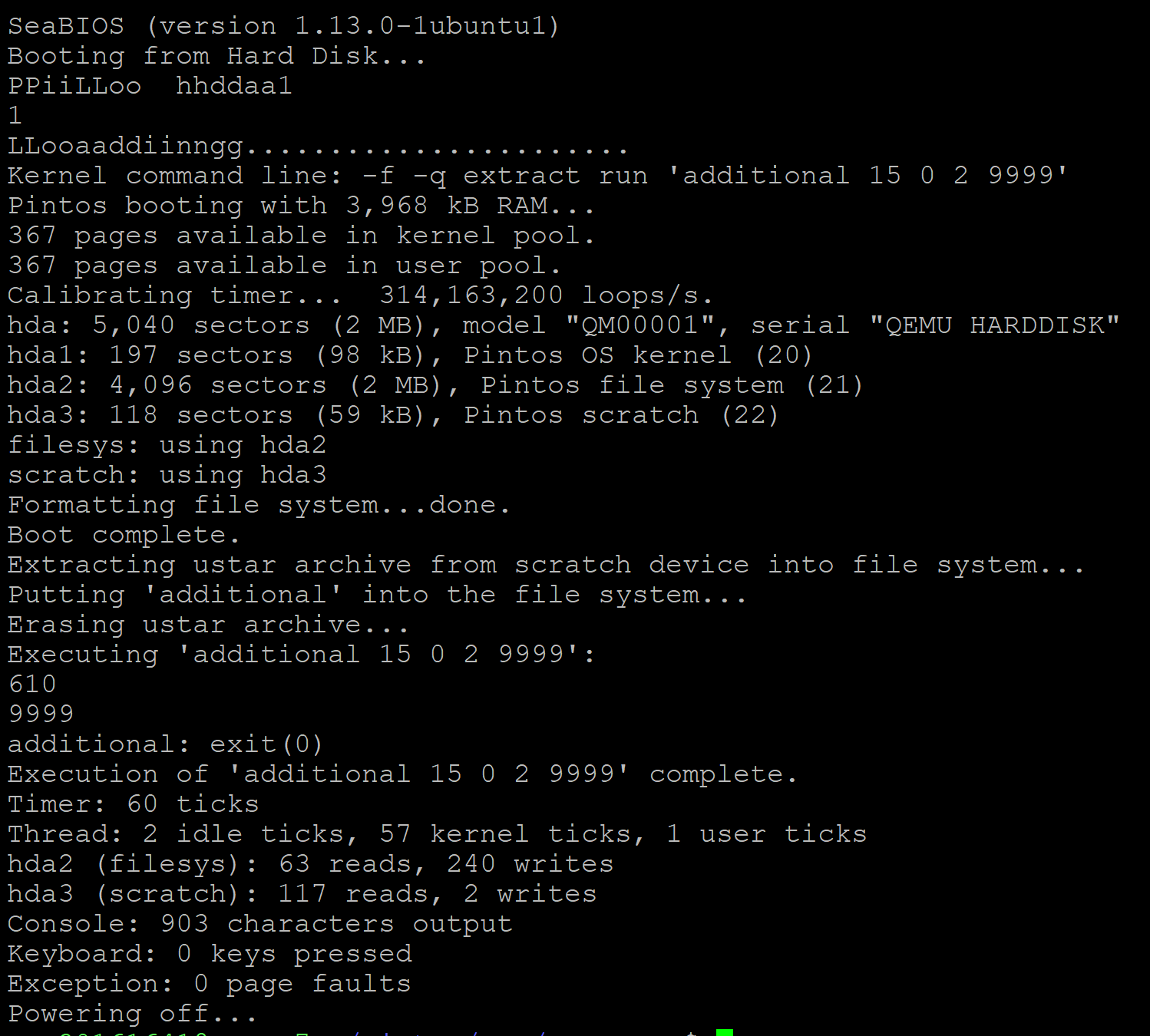
2) max\_of\_four\_int() : 비교연산을 통해 입력 받은 4개의 수 중 가장 큰 1 개를 출력합니다.

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

수행결과 1)

****

수행결과 2

****