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

담당 교수 : 김영재

조 / 조원 : 이지인

개발 기간 : 9/12~10/3

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

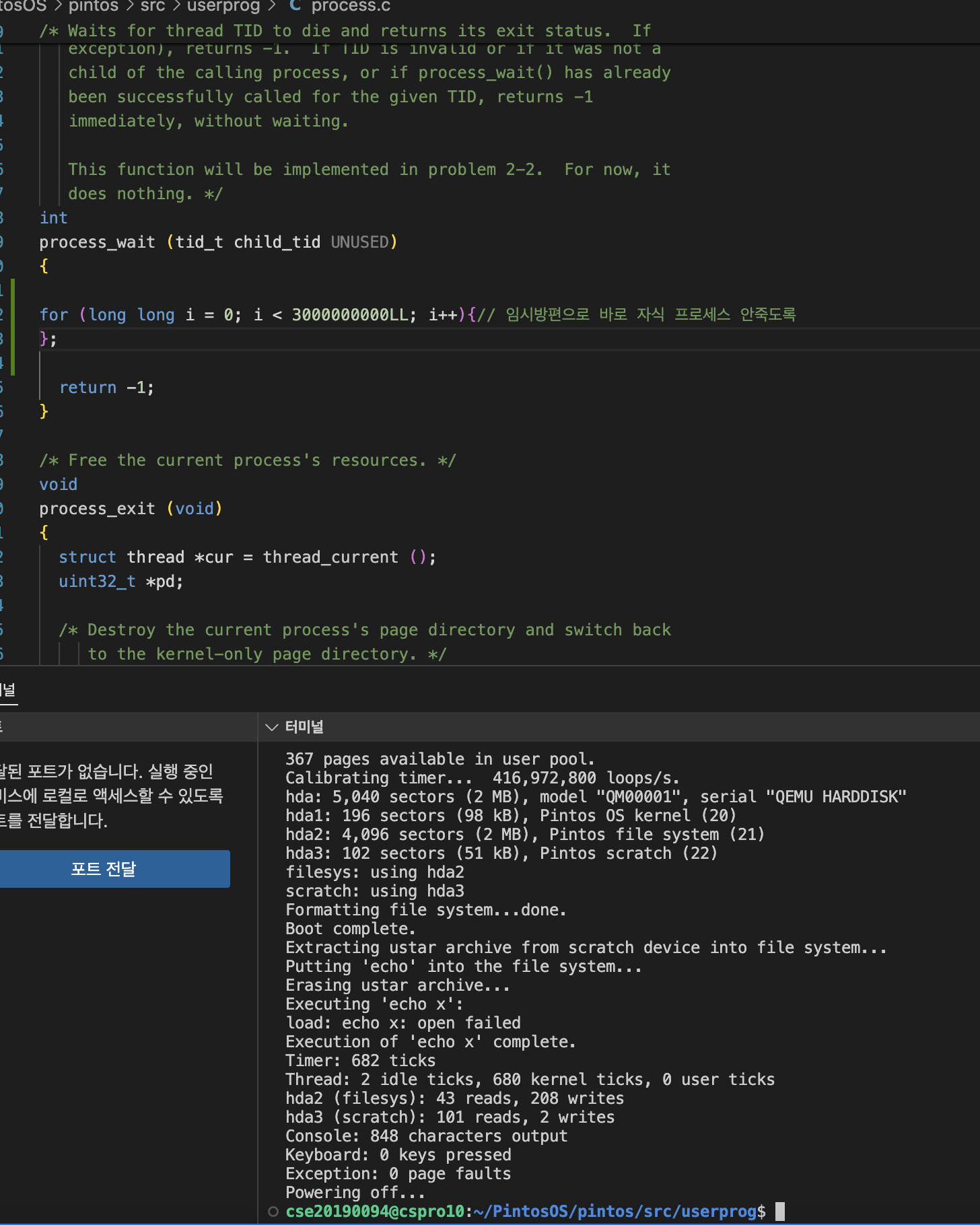
이번 유저 프로그램 과제의 개발 목표는 Pintos OS에서 사용자 프로그램의 실행과 시스템 콜을 구현하는 것이다. 이를 위해 먼저 인자 전달(Argument Passing)을 구현하고, 스택에 인자를 적재하여 시스템 콜 처리 과정에서 활용할 수 있도록 한다. Exit, Wait, Exec 등의 시스템 콜을 구현하여 유저 프로그램이 시스템 콜을 요청하면 커널이 해당 작업을 대신 수행할 수 있어야 한다.특히, 사용자 프로그램이 잘못된 포인터(NULL 포인터, 매핑되지 않은 가상 메모리, 커널 주소 공간의 포인터 등등)를 전달할 수 있으므로, 이러한 유효하지 않은 포인터를 안전하게 처리하여 커널이나 다른 실행 중인 프로세스에 해를 끼치지 않도록 해야 한다. 따라서 메모리 접근 시 접근 가능한 영역만 안전하게 접근할 수 있도록 해야 한다.

**개발 범위 및 내용**

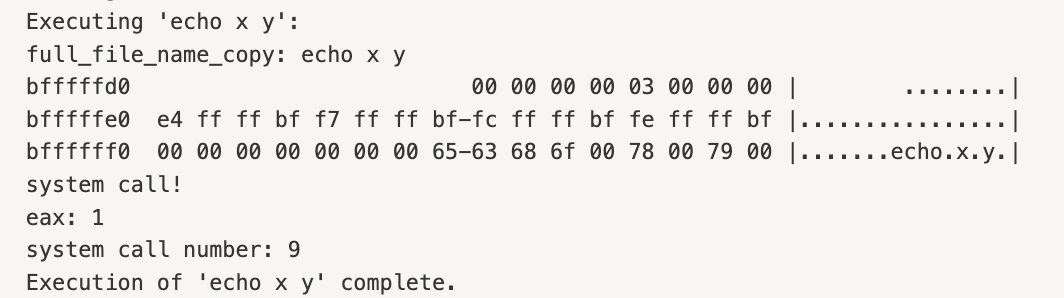
* 1. **개발 범위**
* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

argument passing 기능이 구현되지 않았을 때, 프로그램에서 명령어(예시:echo x)를 실행하려고 하면, 프로그램 이름과 인자 분리 작업이 제대로 이루어지지 않았다. 이로 인해 실행할 프로그램인 echo와 그 뒤의 인자들이 올바르게 처리되지 못하여 다음 사진과 같은 오류가 발생하였다.’



argument passing 기능을 올바르게 구현한 후, 명령어(예시:echo x y)는 정상적으로 처리되었다. 프로그램 실행 시 프로그램 이름인 echo와 인자인 x y를 정확하게 분리하고, 이를 스택에 적절히 할당한 후 프로그램을 실행할 수 있게 되었다. 아래는 hex\_dump()를 통해, 스택에 parsing한 argument가 적절히 쌓였는지 확인해본 터미널 출력 결과이다.

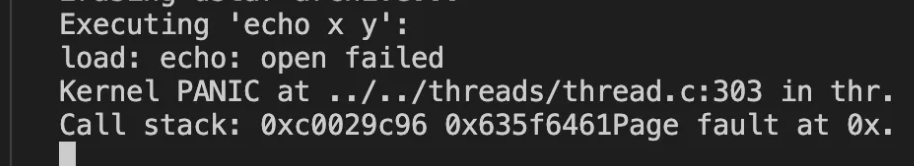


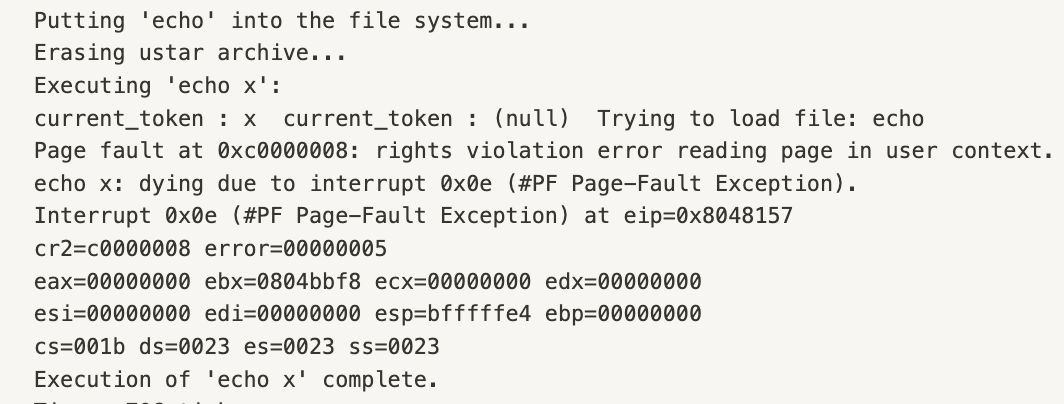
argument passing이 성공적으로 작동하니, 프로그램은 올바르게 실행되어, 9번(write)에 해당하는 적절한 시스템 콜이 호출되었다.

2. User Memory Access

User Memory Access를 구현함으로써 커널 패닉과 페이지 폴트 문제를 해결할 수 있었다. User Memory Access 구현 이전에는 커널이 잘못된 사용자 메모리 접근으로 인해 kernel panic 또는 page fault 문제가 아래와 같이 발생하였다. syscall.c에서는 시스템 콜 인자로 전달되는 메모리 주소에 대한 유효성 검사가 없었고, 잘못된 주소를 그대로 참조하다가 커널 패닉이 발생했다. 또한, process.c에선 함수에서 스택에 인자를 넣는 과정에서 스택 포인터 유효성 확인이 누락되었기 때문에, 잘못된 스택 주소로 인해 메모리 오류가 자주 발생했다. 하지만, User Memory Access를 구현함으로써 커널 패닉과 페이지 폴트 문제를 해결할 수 있었다.

* 기존에 발생했던 오류에 관한 터미널 출력들





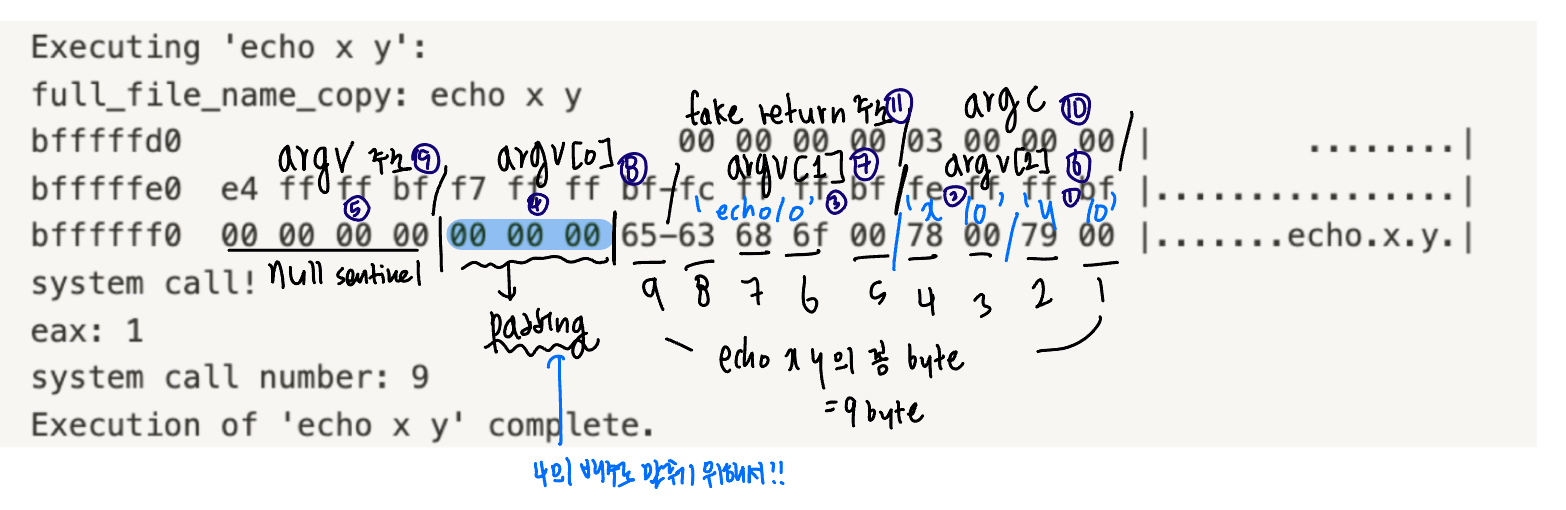
3. System Calls

시스템 콜이 구현되기 전에는 프로그램이 echo x y 명령어를 실행할 때 시스템 콜 번호가 올바르게 표시되었으나, 실제 출력 기능이 작동하지 않아 터미널에 출력되지 않았다. 시스템 콜을 호출하는 부분은 있었지만, 기능적으로 제대로 처리되지 않았기 때문에 출력이 이루어지지 않은 것이였다. 시스템 콜을 적절히 구현한 이후에는 echo x y 명령어를 실행할 때 시스템 콜을 통해 제대로 처리되었고, 터미널에 echo x y가 출력되었다. 시스템 콜이 프로그램 내부에서 제대로 호출되고, 이를 통해 표준 출력 장치로 데이터가 전달되면서 출력 기능이 정상적으로 동작하게 된 것이라 할 수 있다. write 시스템콜을 적절히 구현하였을 때, 위와 같은 결과를 확인할 수 있었고, 이외에 나머지 시스템콜을 구현하고 적절히 작동함은 여러 테스트 케이스에 대한 결과가 pass가 뜨는 것을 확인하여 알 수 있었다.

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

커널은 프로그램에 전달된 인자들을 스택에 쌓아 올려 유저 프로그램이 이를 접근할 수 있도록 준비한다. 이때, 인자들을 메모리의 상위 주소부터 역순으로 인자들을 메모리의 상위 주소부터 역순으로 쌓는다. 그후, 각 인자의 주소를 쌓는다. 그 다음으론, 인자의 주소 배열에 해당하는 argv와 인자의 개수 argc도 스택에 저장하여, 유저 프로그램이 해당 인자들을 쉽게 접근할 수 있도록 한다.

아래는 echo x y를 기준으로 유저 스택에 쌓은 예시이다. (아래는 hex\_dump()로 디버깅하여 나온 결과에 해당한다.)



|  |
| --- |
| 최상단 (PYHBASE쪽에 해당) |
| 1. y\0 2. x\0 3. echo\0 4. 패딩 (4의 배수로 맞추기 위함) 5. NULL sentinel 6. argv[2] (y)의 주소 7. argv[1] (x)의 주소 8. argv[0] (echo)의 주소 9. argv 주소 10. argc 11. fake return address |
| 하단 |

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

Pintos에서 invalid memory access란 사용자 프로그램이 접근해서는 안 되는 메모리 영역, 즉 할당되지 않았거나 커널 영역에 있는 메모리에 접근하는 것을 의미한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

위와 같이 사용자 프로그램이 접근해서는 안되는 메모리 영역을 접근하지 못하도록 하기 위해서는, 메모리 접근 시 커널은 is\_user\_vaddr() 함수로 주소가 사용자 영역에 속하는지 확인하고, pagedir\_get\_page()로 해당 주소가 실제로 매핑된 페이지인지를 확인해 유효성을 검사해야 한다.

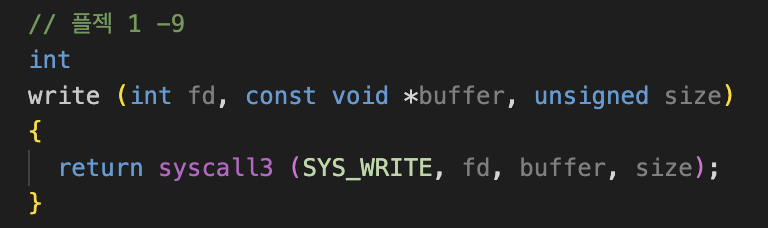
* System Calls
  + **시스템 콜의 필요성에 대한 간략한 설명**

시스템 콜이 필요한 이유는, 하드웨어에 접근해서 특정 작업을 수행하거나, 파일 시스템을 사용하는 것은, 유저 프로그램이 직접적을 수행할 수 없는 일이기 때문이다. 왜냐하면, 만약, 유저 프로그램에게 직접적으로 이러한 권한을 부여할 시에 ,시스템의 안정성과 보안을 해칠 수 있기 때문이다. 따라서, 운영체제는 시스템 콜을 통해 유저 프로그램이 커널 모드로 전환하여 필요한 작업을 요청하고, 유저 프로그램 대신, 커널이 해당 작업을 대신 수행해준다.

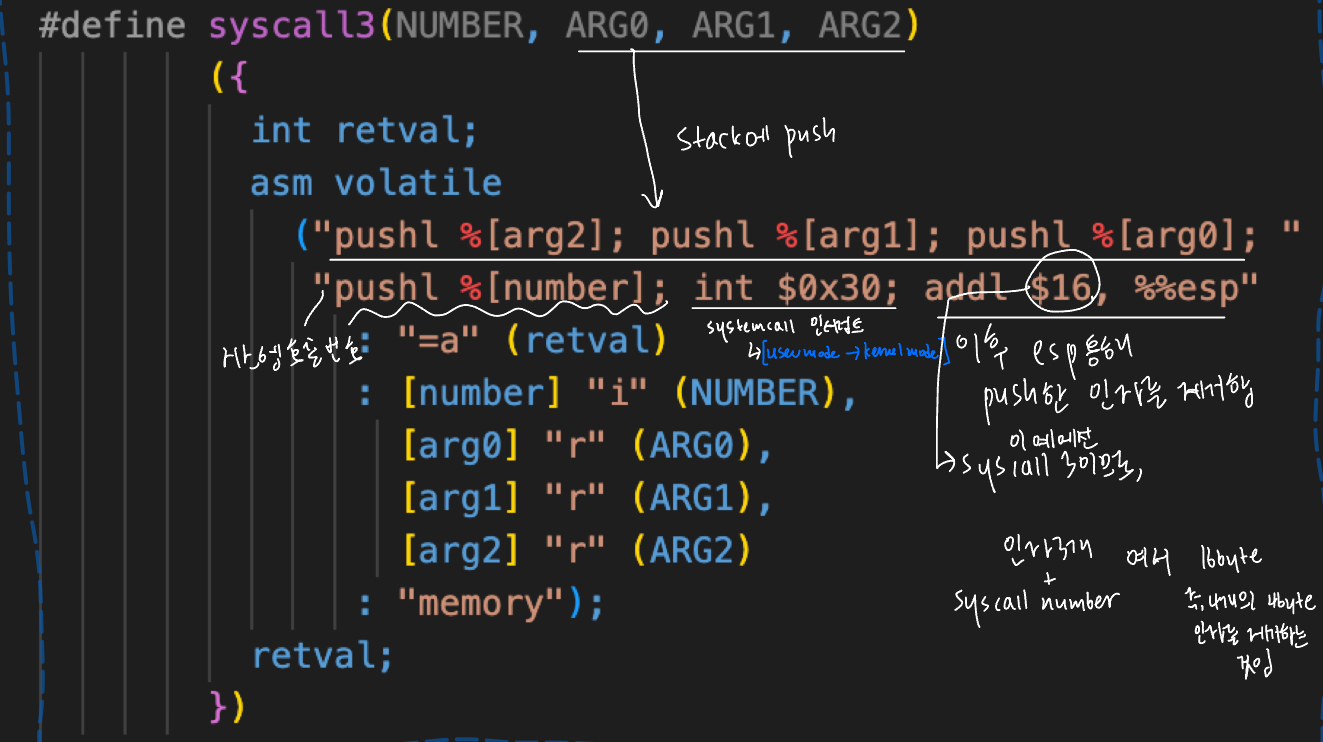
따라서, 유저 프로그램이 시스템콜 인터페이스를 통해, 시스템 콜 번호와 인자를 전달하고, 시스템 콜 인터럽트가 보내지면, 커널 모드로 전환되어 운영체제내의 시스템콜 처리 기능을 통해, 시스템콜 작업이 수행되는 것이다.

* + **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(): 시스템 전체를 종료하는 시스템 콜로, shutdown\_power\_off() 함수를 호출하여 운영체제를 안전하게 종료한다.
2. )exit(): 현재 실행 중인 프로세스를 종료하는 시스템 콜로, 호출 시 해당 프로세스의 종료 상태를 출력한다. 부모 프로세스는 이 상태 값을 통해 자식 프로세스가 정상적으로 종료되었는지 확인할 수 있다. 잘못된 메모리 접근이나 예외 상황에서도 이 시스템 콜을 호출해 프로세스를 안전하게 종료할때도 사용한다.
3. write(): 파일 디스크립터(fd)를 사용하여 데이터를 출력하는 시스템 콜로, 콘솔에 데이터를 기록한다. fd 값에 따라 출력 대상이 달라지며, 콘솔 출력은 fd == 1일 때 수행되는데, 현재 유저 프로그램1에서는, fd==1일때, 콘솔에 출력하는 것만 구현된 상태이다. 버퍼에 있는 데이터를 지정된 길이만큼 출력하며, 출력된 바이트 수를 반환한다.
4. read(): 콘솔에서 데이터를 입력받아 버퍼에 저장하는 시스템 콜로, 주어진 파일 디스크립터로부터 데이터를 읽어온다. fd == 0인 경우 콘솔로부터 입력을 받으며, 지정된 크기만큼 읽어들인다. 입력된 데이터는 버퍼에 저장되며, 읽은 바이트 수를 반환한다.
5. exec(): 새로운 프로그램을 실행하는 시스템 콜로, 주어진 프로그램 파일을 로드하고 실행한다. 실행된 프로그램은 현재 프로세스를 대체하며, 새로운 프로세스의 PID가 반환된다. 실행 실패 시 -1을 반환하며, 프로세스는 실패 상태로 종료된다.
   * **유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명**
6. 유저 모드에서 시스템콜 인터페이스를 호출한다. ex) write()



1. 시스템 콜이 호출되면, int 0x30 인터럽트를 사용하여 유저 모드에서 커널 모드로 전환된다.

* 시스템 호출 매크로 설명
* 인자들과 , 시스템 호출 번호를 스택에 push함
* int $0x30: 소프트웨어 인터럽트를 발생시켜 커널로 진입
* 스택 포인터(esp)를 조정하여 푸시한 인자들을 제거
* 

1. 이 인터럽트는 intr\_handler()를 호출 -> 이때 시스템 콜 번호가 eax 레지스터에서 읽히게 됨
2. 시스템 콜 핸들러인 syscall\_handler()가 호출됨

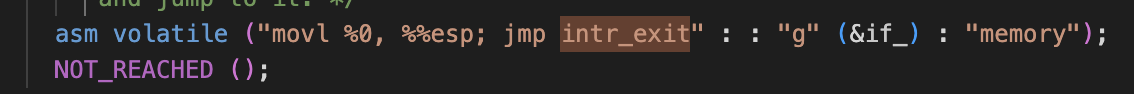
전달된 시스템 콜 번호를 바탕으로 적절한 시스템 콜 처리 함수를 호출 ex) write() 등등

1. 커널에서 작업을 완료한 후, syscall\_handler()는 반환 값을 eax 레지스터에 저장하고, intr\_exit()을 호출하여 인터럽트 리턴 방식으로 유저 모드로 돌아간다. 이때, 시스템 콜의 결과가 유저 프로그램에 반환되며, 유저 프로그램은 해당 결과 값을 받아 계속해서 실행된다.

* 이러한 양식으로 eax 레지스터에 반환값을 저장

image7.png

* start\_process() 하단에 존재



**추진 일정 및 개발 방법**

* 1. **추진 일정**

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

****

1. 9월 12일 : 핀토스 세팅

깃허브 private repository로 작업 가능하게 세팅, 비주얼 스튜디오 코드로 작업 가능하게 remote server로 연결해놓음

1. 9월 20일 : additional system call을 위한 syscall4 메크로함수 먼저 구현
2. ~ 9월 21일

* arguement들을 적절하게 parsing하여 전달하는 것 구현
* additional system call을 구현하기 위해 필요한 시스템콜 인터페이스 추가, 프로토타입 추가 먼저 구현

1. ~ 9월 26일

* argument passing 까지 구현 완료
* write 시스템콜, read 시스템콜,halt 먼저 구현함
* process.c 내 user memory access 구현 완료

-> echo x 출력, stack에 쌓는 부분까지 성공

1. 25~26일 : 부모 자식 프로세스 동기화 작업을 수행할 수 있을지, 계속해서 고민함 -> 처음에는 자식 프로세스 구조체를 별도로 만들어서 동기화를 구현하려 했으나, 실패함
2. 9월 27일 ~28일

* struct thread를 활용해서 부모, 자식 프로세스 동기화 활용할 수 있도록 코드 추가
* 다른 시스템콜 작업하면서 테스트 케이스들을 확인하며, 이 부분 계속적으로 수정하였음

1. 9월 27일 : wait 시스템콜 구현
2. 9월 28일 - 1

* 계속해서 fail이 발생했던 sc-bad-sp,sc-boundary-3 둘다 passt 시킴
* syscall.c와 exeption.c에서 user meomory access 검사 보완 추가

1. 9월 28일 - 2 : exit 시스템콜 구현 완료(관련 테스트케이스 pass)
2. 9월 28일 - 3 : exec 시스템콜 구현 완성 : (exce 관련 테스트 케이스 fail하던 것 pass로)

-> 80개의 테스트 케이스 중, 35개 fail, 45개 pass 달성함

1. 9월 28일 - 4 : additional system call까지 userprog내에서 pintos 명령어로 실행 가능하도록 구현

**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. Argument Passing

구현 파일: pintos/src/userprog/process.c

file name을 오픈하는 것을, strtok\_r을 사용해, 첫번쨰 인자를 파싱해, full file name 대신, 첫번째 인자(ex. echo 프로그램명에 해당)를 넘기도록 수정한다.

setup\_user\_stack(char \*file\_name, void \*\*esp) 함수를 구현하여 사용자 프로그램이 실행될 때 인자들을 스택에 적절히 설정하도록 한다. load() 함수 내에서 구현한 setup\_user\_stack() 함수를 호출한다.

1. User Memory Access

구현 파일

* pintos/src/userprog/syscall.c
* pintos/src/userprog/process.c
* pintos/src/userprog/exception.c

해당 파일들에, is\_user\_vaddr()와 pagedir\_get\_page() 함수를 사용해 유효한 사용자 주소인지 확인하는 로직을 추가한다.

* exception.c의 페이지 폴트가 발생했을 때 호출되는 예외 처리 함수에 해당하는 page\_fault() 내에, pagedir\_get\_page()와 is\_user\_vaddr()을 사용하여 잘못된 사용자 메모리 접근을 감지할 수 있도록 한다.
* process.c에서 pagedir\_get\_page()와 is\_user\_vaddr()을 사용하여 잘못된 사용자 메모리 접근을 감지하는 is\_valid\_stack\_pointer() 함수를 구현하여, 스택을 쌓을 때, esp로 메모리에 접근할 떄마다, 유효한 스택 주소인지 확인할 수 있도록 한다.
* syscall.c에선 시스템 콜 처리 시 , 유저의 포인터의 유효성을 검사하는 check\_valid\_vaddr() 함수를 추가하여, 시스템 콜 핸들러가 호출될 때 사용자 메모리에 대한 접근이 올바른지 확인하고, . check\_valid\_buffer() 함수는 버퍼의(인자가 여러개일경우) 모든 바이트가 유효한지, 각각 인자별로 check\_valid\_vaddr()를 호출하여 확인한다.

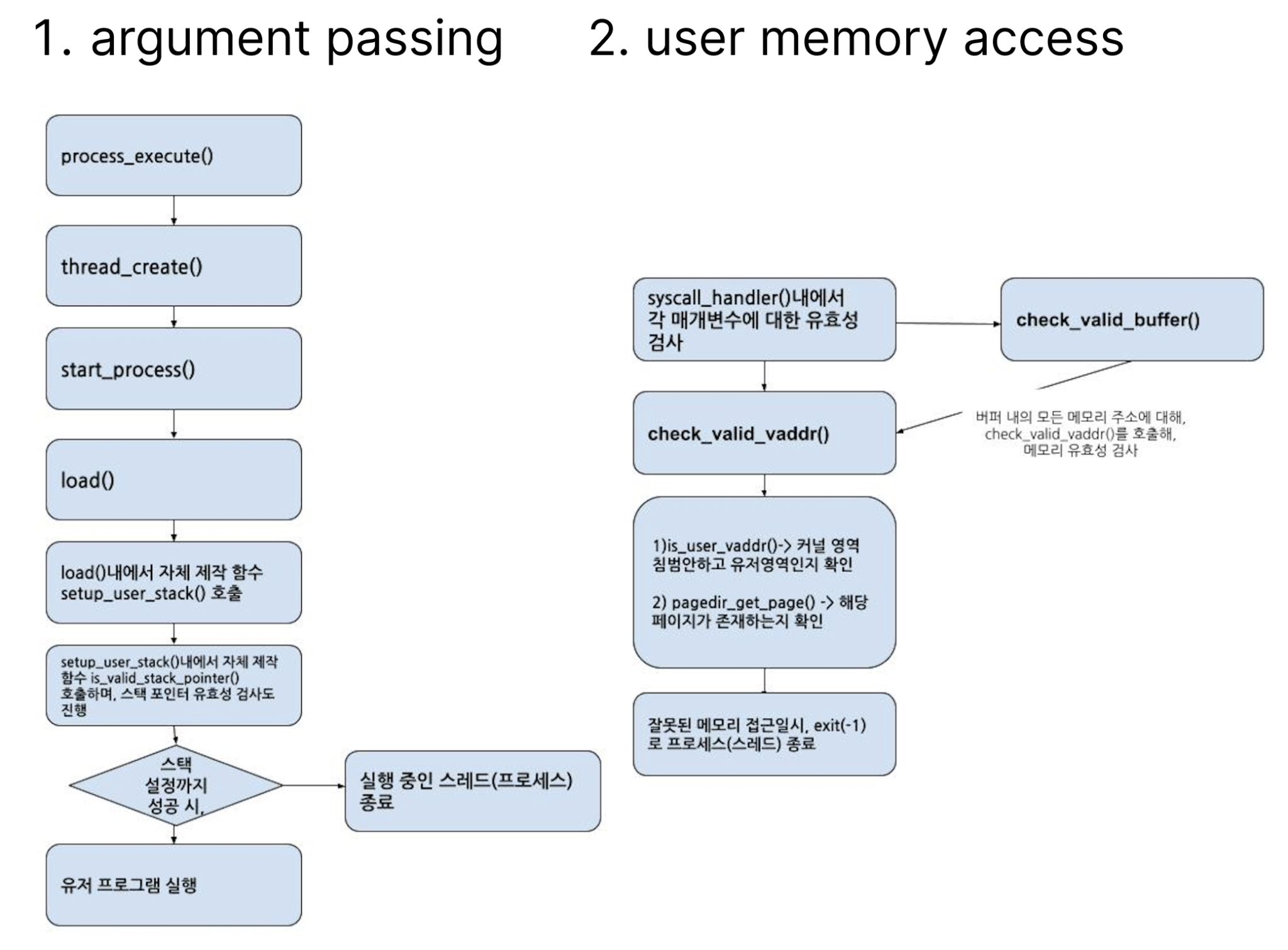
1. System Call

구현 파일

* intos/src/userprog/syscall.c
* pintos/src/userprog/process.c
* pintos/src/userprog/exception.c
* pintos/src/threads/thread.c
* pintos/src/threads/thread.h
* intos/src/userprog/syscall.c에서 syscall\_handler() 함수에서 시스템 콜 번호에 따라 각 시스템 콜 함수를 호출하도록 switch-case문으로 분기를 작성한다. 이때 각각의 시스템콜 번호에 알맞게 호출될 halt(),exit(),write(),read(),exec(),그리고 추가 시스템콜을 구현한다.
* pintos/src/threads/thread.c
  + init\_thread() 함수 수정 : load\_sema, exit\_sema, wait\_sema 세마포어를 초기화
  + child\_list 초기화, 부모-자식 관계를 설정하기 위해 parent 포인터를 설정
* pintos/src/threads/thread.h
  + struct thread에 load\_sema, exit\_sema, wait\_sema 세마포어를 추가하여 부모와 자식 프로세스 간 동기화 구현
  + parent: 부모 스레드를 가리키는 포인터 추가
  + child\_list: 자식 스레드들의 리스트 추가
  + child\_elem: 자식 리스트에 자신을 연결하기 위한 리스트 요소 추가
  + exited, wrong\_exit 플래그 추가
* pintos/src/userprog/process.c에서
  + process\_wait() 함수에서 세마포어를 이용하여 자식 프로세스의 종료를 대기하도록 수정
  + start\_process() 함수에서 자식 프로세스가 로드에 성공하거나 실패했을 때 부모에게 알리기 위해 세마포어 수정
  + get\_child\_thread\_by\_tid() 함수를 추가 -> 현재 스레드의 자식 스레드 중 특정 tid를 가진 스레드를 찾을 수 있도록 함

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

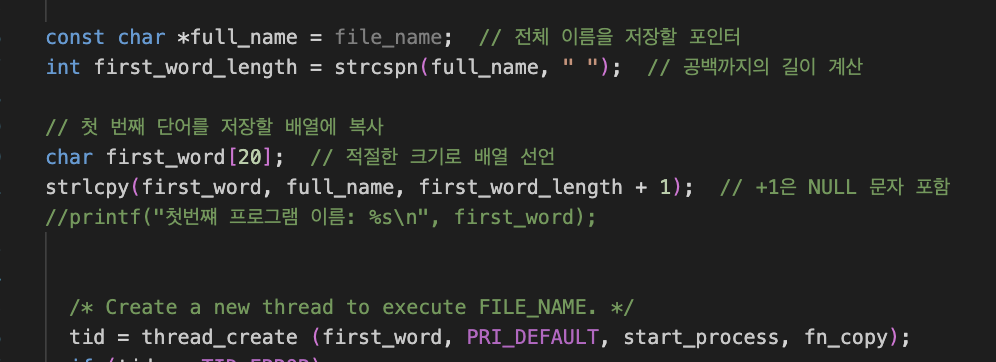




**제작 내용**

* **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
2. process\_execute() 함수 내에, 실행 파일의 이름을 추출하여, 파일 이름에서 프로그램 이름을 정확히 추출하지 못하는 문제를 해결하여, 사용자 프로그램이 제대로 실행될 수 있도록 했다.

이떄, strcspn() 함수를 사용하여 공백까지의 길이를 계산하고, strlcpy()를 사용하여 알맞은 크기의 복사본을 생성하였다.

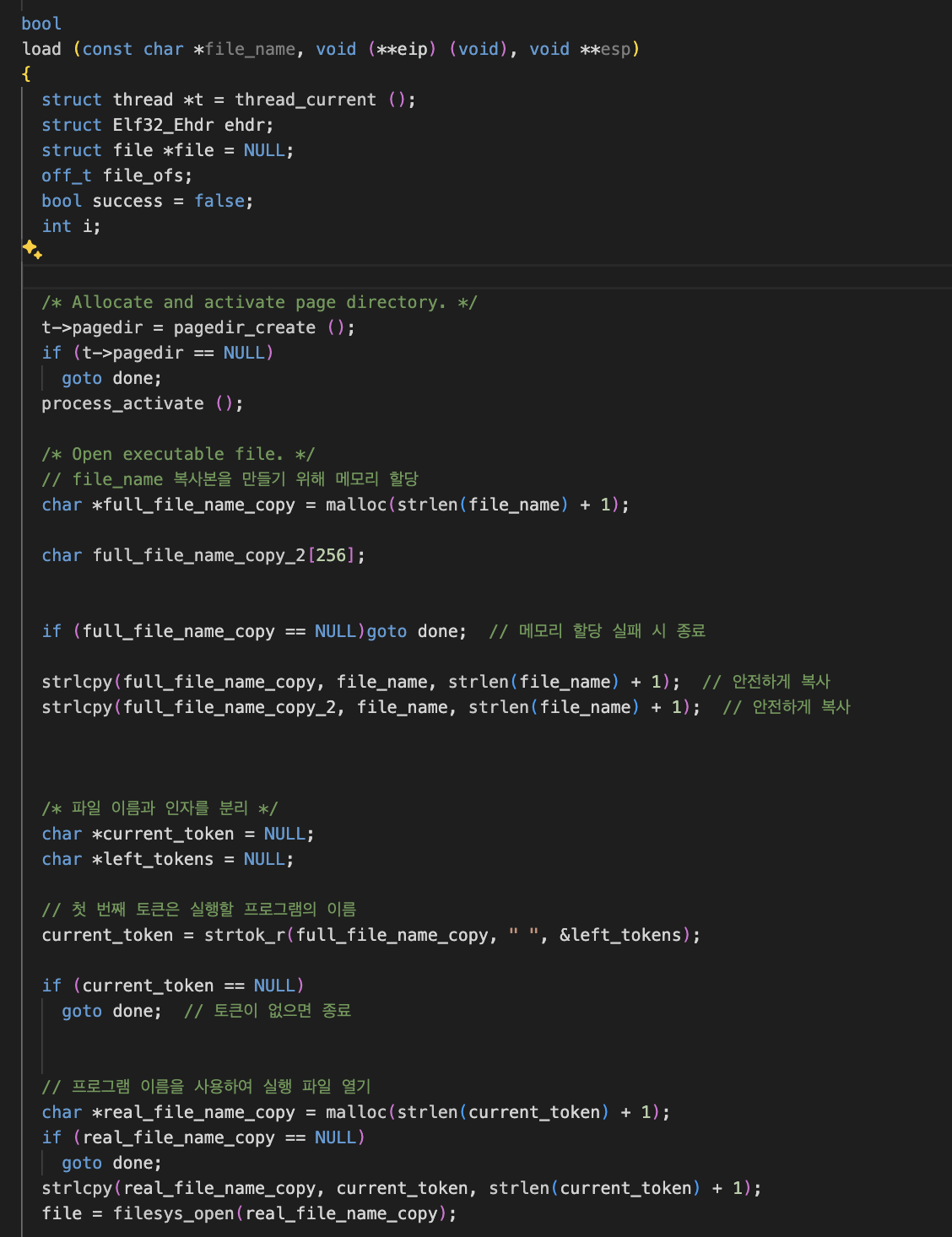
1. load() 함수

load() 함수 내에서 알맞은 실행 파일을 메모리에 로드할 수 있도록 첫번쨰 인자를 추출하였다.

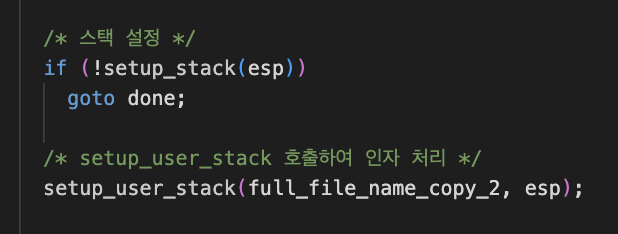
이때 원본 문자열에다 작업하면 훼손되므로, 복사본을 생성하여 작업하였다.

file\_name을 복사하고 strtok\_r()을 사용하여 프로그램 이름과 인자를 분리하고, filesys\_open()으로 알맞은 실행 파일을 로드하였다. (real\_file\_name\_copy에 해당)

또한, 첫번쨰 인자(프로그램명)뿐만 아니라, 스택에 각각의 인자들을 알맞게 쌓을 수 있도록, full\_file\_name\_copy2를 두어, 전체 입력된 문자열의 복사본을 별도로 마련해 두었다. 이때, full\_file\_name\_copy 자체를 사용하면, 조작 과정에서, 첫 번째 인자만 남게 되어, full\_file\_name\_copy 말고, 이외에 full\_file\_name\_copy\_2 별도의 복사본을 생성하여, full name의 훼손을 막을 수 있었다.



또한, load() 함수 내에서 setup\_stack() 이라는 빌트인 함수를 통해 스택 초기화가 성공적으로 이루어진 후에, 스택에 인자들을 알맞게 쌓을 수 있도록 직접 제작한 setupt\_user\_stack 함수를 호출하였다. 이때, 훼손 가능성이 없게끔 안전하게 복사한 full\_file\_name\_copy\_2 를 인자로 넘겨주었다.



직접 제작한 setupt\_user\_stack 함수 내에서는,

|  |
| --- |
| 최상단 (PYHBASE쪽에 해당) |
| 1. y\0 2. x\0 3. echo\0 4. 패딩 (4의 배수로 맞추기 위함) 5. NULL sentinel 6. argv[2] (y)의 주소 7. argv[1] (x)의 주소 8. argv[0] (echo)의 주소 9. argv 주소 10. argc 11. fake return address |
| 하단 |

위에서도 언급했던, 이와 같은 순서로 스택을 쌓는 로직을 구현했다. 이때, esp 포인터가 접근 가능한 메모리 영역을 가리키고 있는지 포인터를 이동할때마다, 직접 제작한 is\_valider\_stack\_Pointer 함수를 통해 esp를 검사하고 만일 잘못되었으면, thread\_exit()을 호출할 수 있도록 구현하였다.

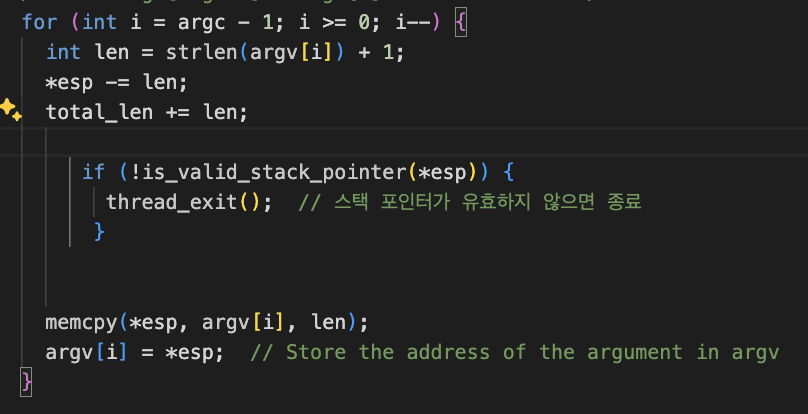
1. 프로그램 이름과 인자 파싱



strtok\_r() 함수를 사용해 file\_name 문자열을 공백을 기준으로 토큰화를 진행하였다.

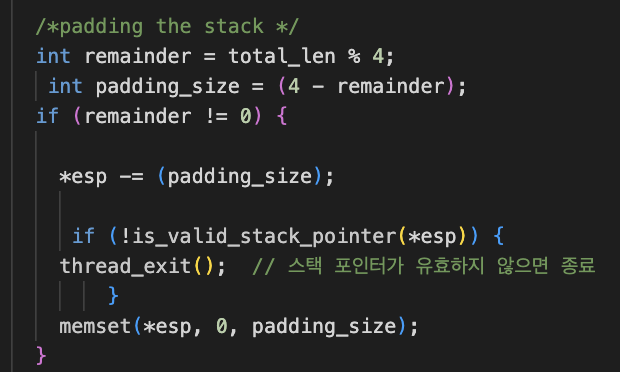
각 토큰을 argv 배열에 순서대로 저장하고 argc도 증가시켜놓았다.

1. 인자들을 역순으로 스택에 쌓기



argv[i]에 저장해놓은 인자들을 역순으로 스택에 쌓았다. 핀토스에서 스택은 상단에서 하단으로 쌓아지므로, 마지막 인자부터 저장했다. 이를 위해, 각 인자의 길이를 계산하고, 그만큼 esp를 감소시켜 공간을 확보했다. 그리고 스택에 인자에 해당하는 문자열을 복사하기 위해선, memcpy() 함수를 사용하였다. 그 후, 인자가 스택에 저장된 주소를 argv[i]에 다시 저장하여 나중에 주소 값을 사용할 수 있게 하였다.

1. 스택 패딩을 통해 4바이트 배수로 정렬



전체 인자 문자열의 길이가 4의 배수가 아닐 경우, 스택의 워드 정렬을 위해 패딩을 추가하였다.

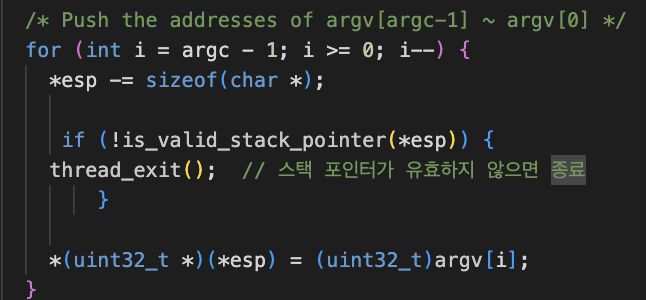
만약, echo/0 x/0 y/0와 같이, 문자열의 길이가 4+1+1+1+1+1=9바이트라면, remainder = 9 % 4 = 1이고, padding\_size = 4 - 1 = 3이 되었다. 이때에는, 3바이트의 패딩을 추가하여, 4의 배수가 될 수 있도록 로직을 구성하였다.

1. NULL Sentinel 추가

인자 주소 배열의 끝을 나타내는 NULL 포인터를 스택에 추가하였다.



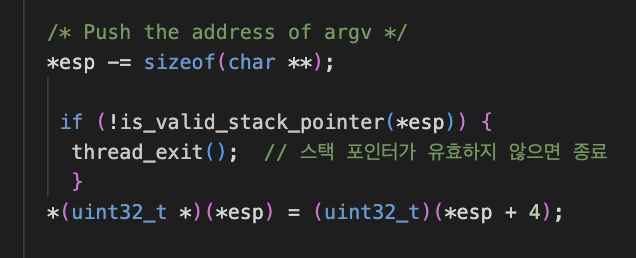
1. 각 인자의 주소를 스택에 저장



스택에 저장된 인자들의 주소를 스택에 저장하였다.

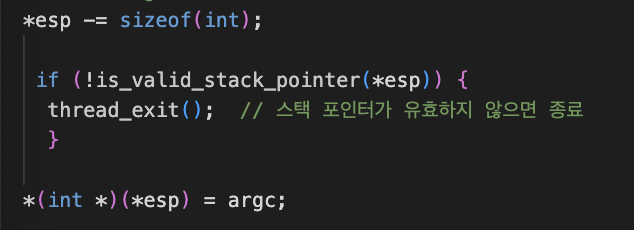
1. argv의 주소 저장

argv 자체의 주소를 스택에 저장하였다.

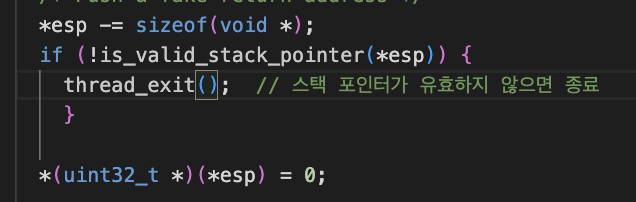


1. argc 저장

인자의 개수에 해당하는 argc를 스택에 저장하였다.



1. 가짜 리턴 주소 저장

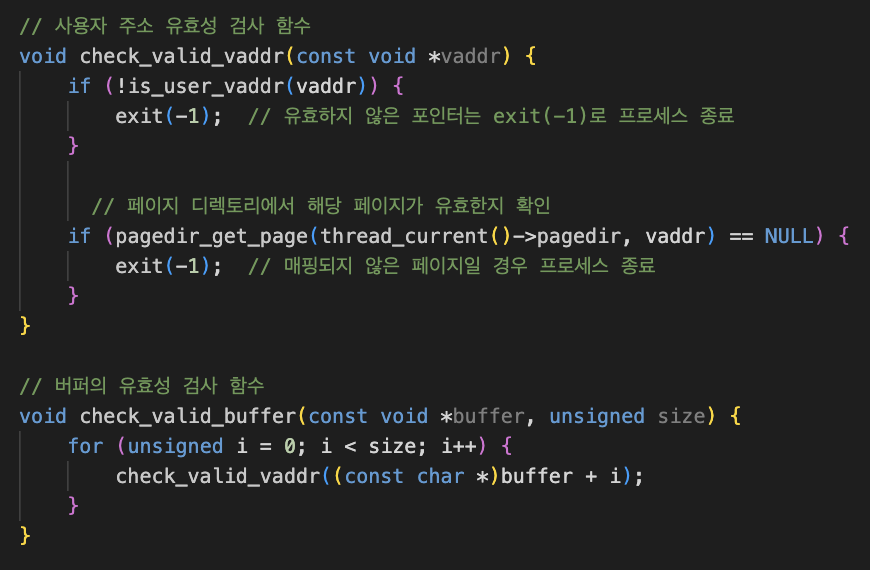


스택 프레임의 형태를 맞추기 위해, 가짜 리턴 주소를 저장하였다.

이를 구현하면서, 스택에 데이터를 저장할 때 esp가 유효한 사용자 메모리 영역을 벗어나면 커널 패닉이 발생했을 때가 있었다. 원래는 스택 포인터 유효성을 검사하는 부분을 추가하지 않았는데, 페이지 디렉토리에 매핑되어 있는지 확인하도록 is\_valid\_stack\_pointer() 함수를 통해, 스택 포인터를 변경할때마다 검사하여 메모리 관련 오류들이 사라질 수 있게 되었다

.

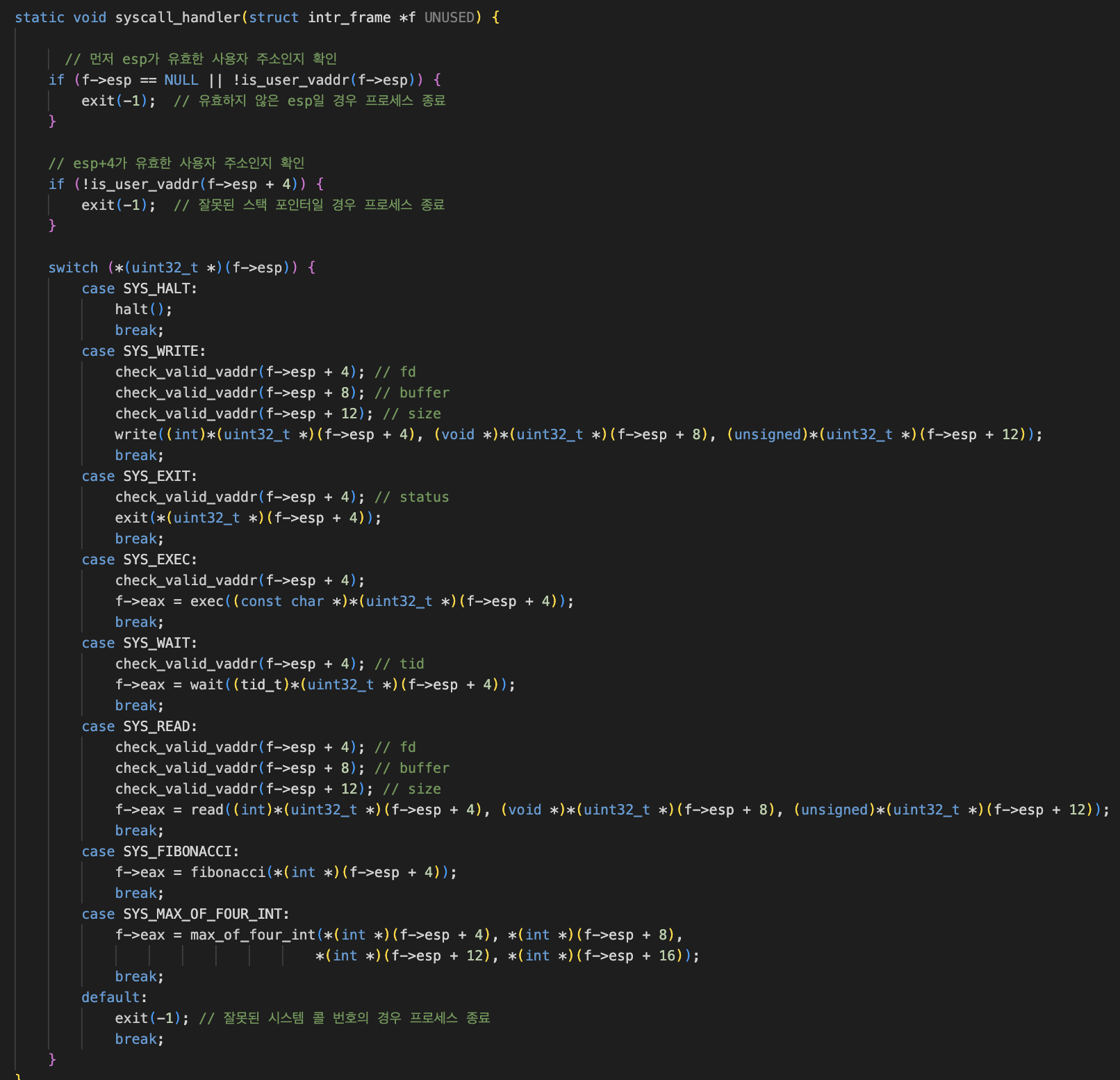
2. User Memory Access



핀토스의 빌트인 함수인, is\_user\_vaddr()를 통해, 주소가 사용자 영역에 속하는지 확인하였다. 또한, pagedir\_get\_page() 함수를 통해, 페이지 디렉토리에서 해당 주소가 유효한지 확인하였다. 이 두 빌트인 함수를 이용하여, check\_valid\_vaddr()와 check\_valid\_buffer() 유효성 검사 함수를 구현하였다. 이를 syscall.c 파일에서, 시스템 콜에서 사용자로부터 전달받은 포인터나 버퍼에 접근해야 하는데, 이때 해당 메모리 주소가 유효한 사용자 영역의 주소인지 확인하는 용도로 사용하였다. 즉, 이를 이용해 각 시스템 콜 핸들러에서 사용자로부터 전달받은 포인터나 버퍼에 대해 유효성 검사를 수행한 것이다. 그리고 만일, 유효하지 않은 포인터일 경우에는 프로세스가 종료될 수 있도록 exit(-1)을 호출하였다.

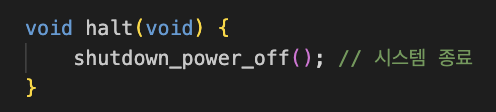
1. System Calls

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



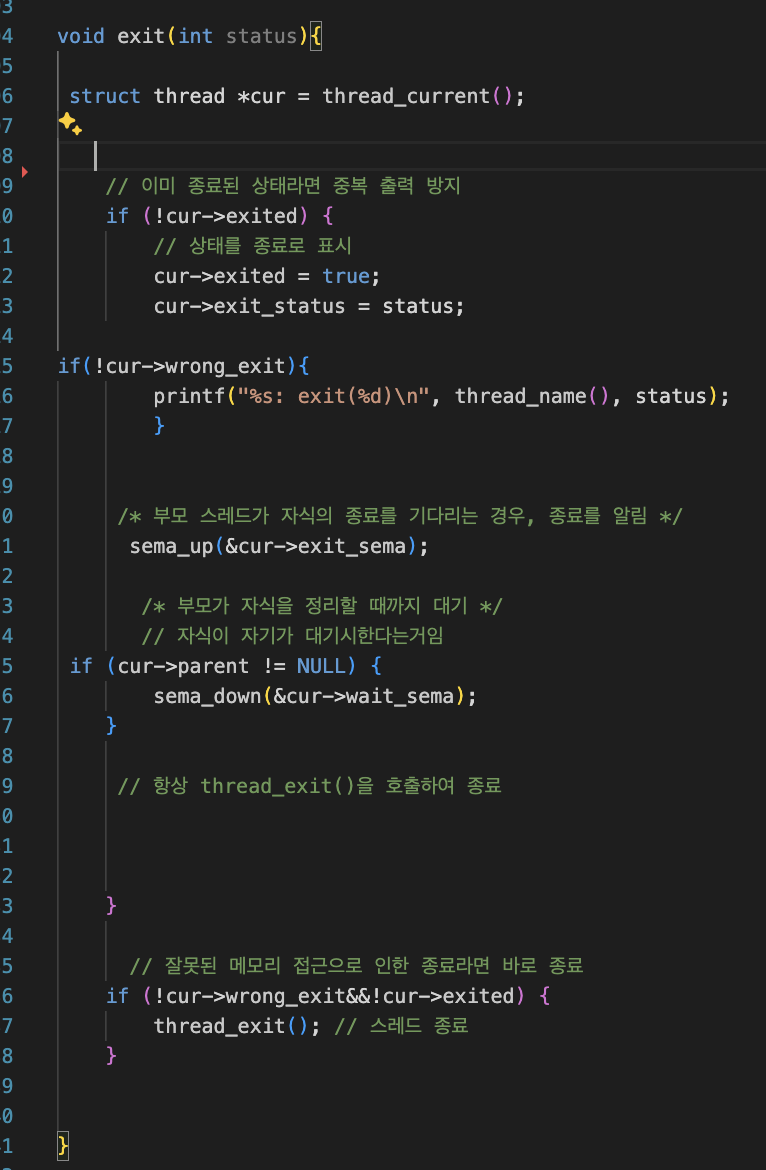
우선 위와 같이 syscall\_handler() 함수에서 시스템 콜 번호에 따라 각 시스템 콜 함수를 호출하도록 switch - case문을 통해 구현하였다. 이때, 사용자로부터 전달된 포인터의 유효성을 검사를 각각의 콜별로 호출하여, 잘못된 메모리 접근으로부터 커널을 보호할 수 있도록 하였다.

1. halt() 시스템 콜



핀토스에서 시스템을 종료하기 위해 제공되는 빌트인 함수인 shutdown\_power\_off() 함수를 호출하여 halt 시스템 콜을 구현하였다.

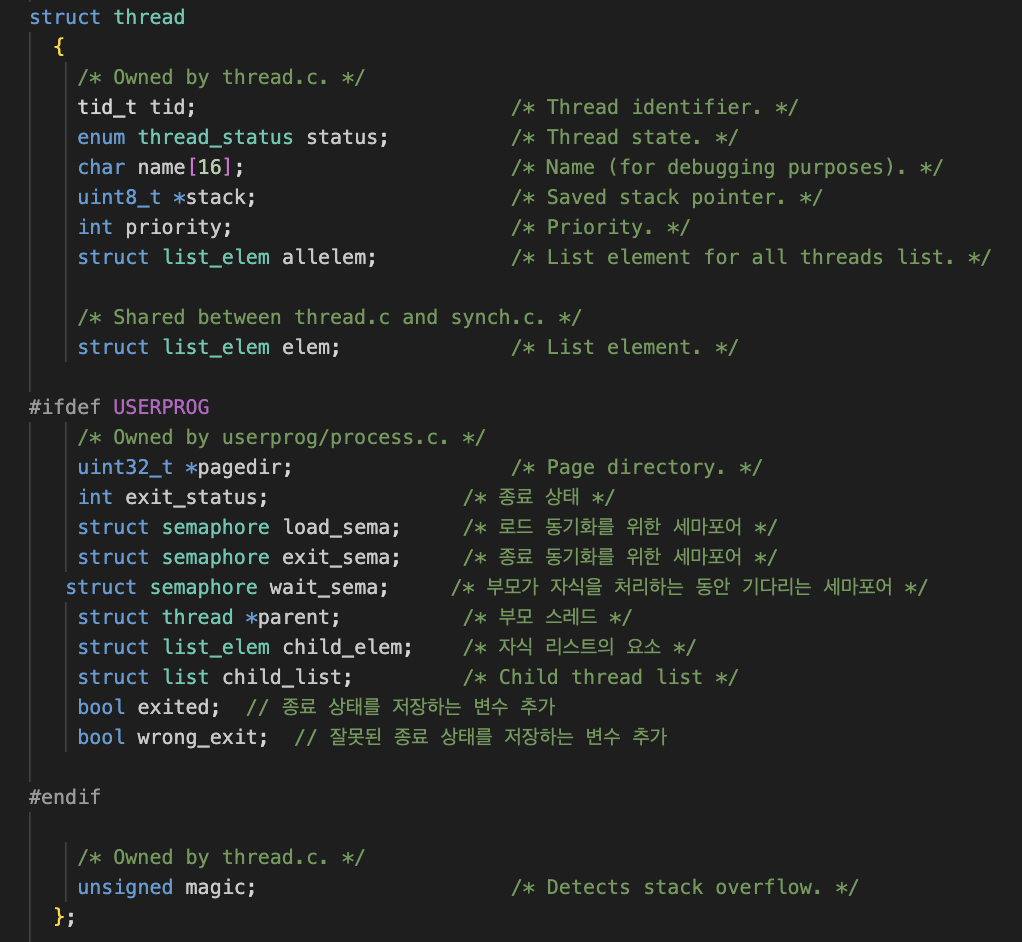
1. exit() 시스템 콜



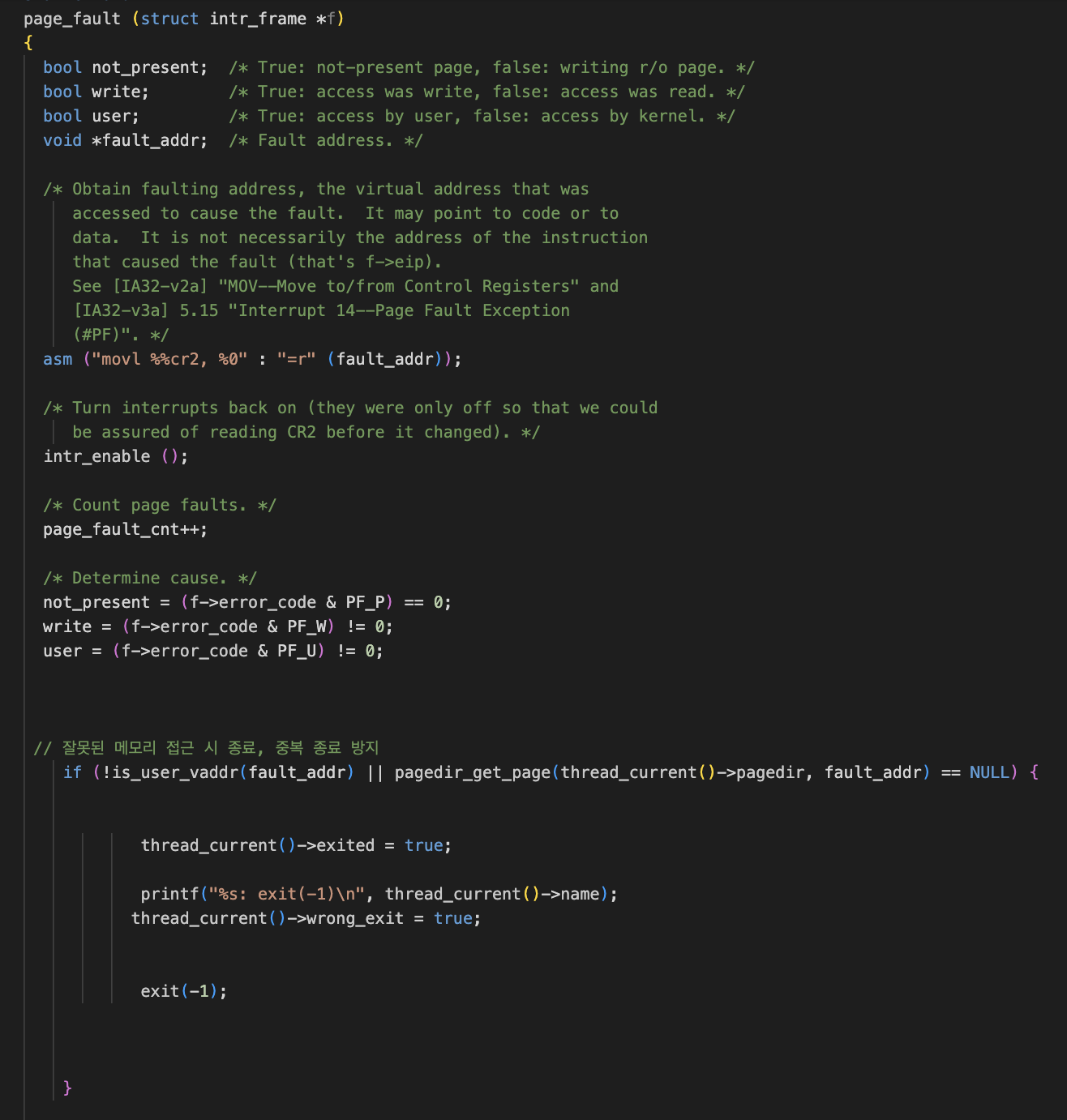
현재 프로세스를 종료하고, 종료 상태를 부모 프로세스에게 전달할 수 있도록 exit 시스템콜을 구현하였다.

부모 프로세스가 자식의 종료를 대기하고 있을 수 있으므로, exit\_sema 세마포어를 통해 부모에게 종료를 알릴 수 있도록 했다. 그런데 이를 개발하는 과정에서, 중복된 종료 메시지가 출력되는 문제가 발생하였었다. 그리고, 종료 시차로 인해, 잘못된 메모리 접근으로 인한 커널 패닉의 문제도 발생하였었다.

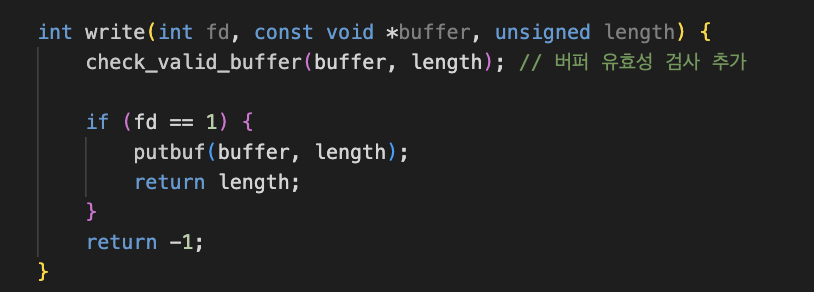
이를 해결하기 위해, thread 구조체에 exited 및 wrong\_exit 변수를 추가하여 상태를 관리하였다.



이미 종료되었는데 시간차로 인해 또 종료시키려고 하여 커널 패닉도 발생하지 않게끔 싱크를 맞추는 작업이 중요했는데, 이때에도, 잘못된 메모리 접근 시 커널 패닉이 발생하지 않도록, page\_fault() 핸들러에서 exit(-1)을 호출하도록 수정하였다. page\_fault()내에서, 이미 exit(-1)의 종료 메시지를 출력하였으므로, 이 경우에는, exit 시스템콜 내에서 중복 종료 메시지가 출력되지 않게끔 할 필요가 있었다. 때문에, page\_fault()내에서 잘못된 메모리 접근을 원인으로 종료시킬 경우에는, thread\_current()->exited = true;로 하고, thread\_current()->wrong\_exit = true;로 하여, exit 시스템콜과 싱크를 맞출 수 있도록 하였다. 즉, exit 시스템콜 내에서, wrong\_exit이 true인 경우에는 이미 execption.c의 page\_fault 함수 내에서 종료 메시지가 출력되었을 것이므로, wrong\_exit이 false인 경우에만, 종료 메시지가 출력되도록 하여 이 문제를 해결할 수 있었다.



1. write() 시스템 콜



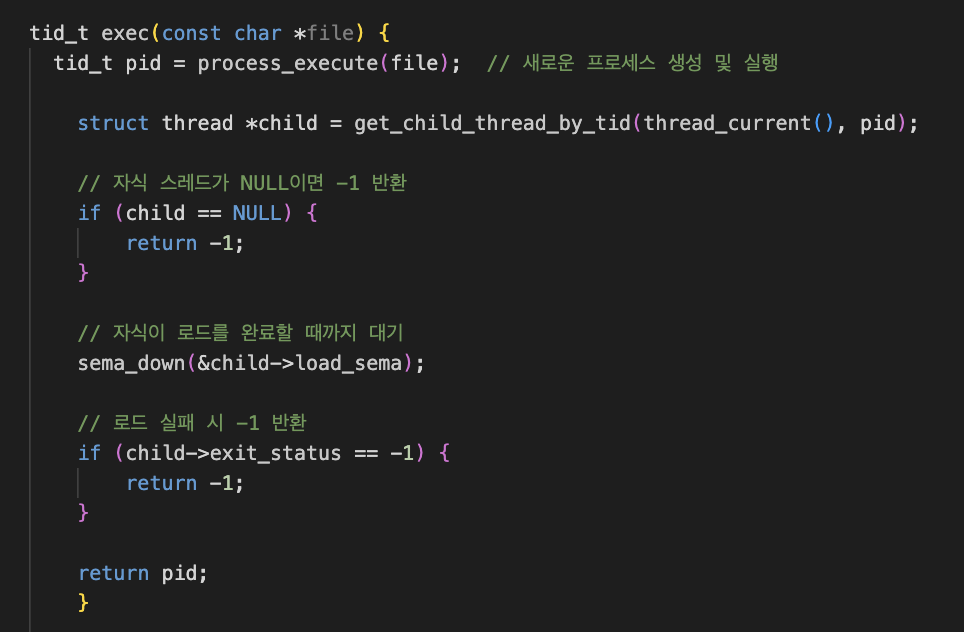
파일 디스크립터가 1인 경우, 즉 표준 출력인 경우에 버퍼의 내용을 화면에 출력하는 것을 구현하였다. 이때, 빌트인 핀토스 함수인 버퍼의 내용을 콘솔에 출력하는 putbuf()를 사용하였다. 또한, 자체 제작 함수를 통해 버퍼의 유효성을 검사하여 잘못된 메모리 접근을 방지할 수 있도록 하였다.

1. read() 시스템 콜

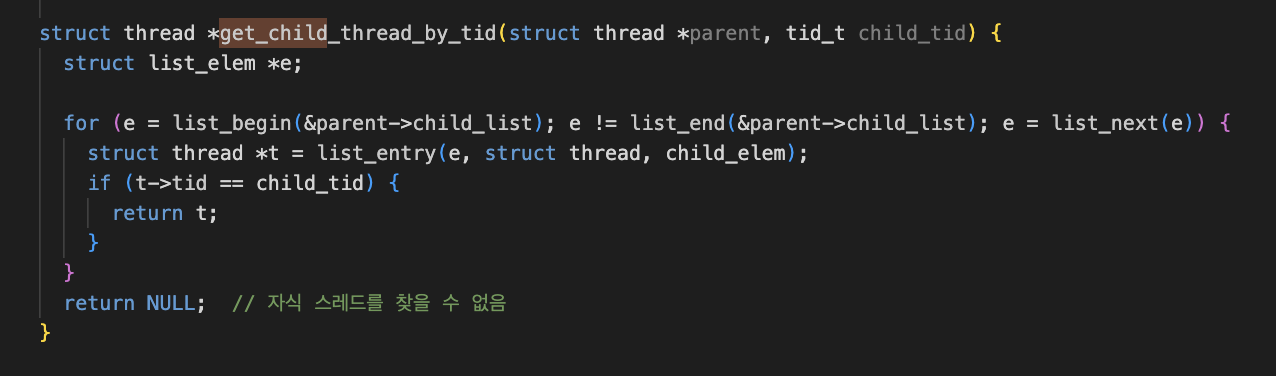


파일 디스크립터가 0인 경우, 즉 콘솔 경우에 입력을 받아 버퍼에 저장하는 부분을 구현하였다. 이때, 핀토스의 빌트인 함수인, input\_getc()를 사용하였다.

5) exec() 시스템 콜



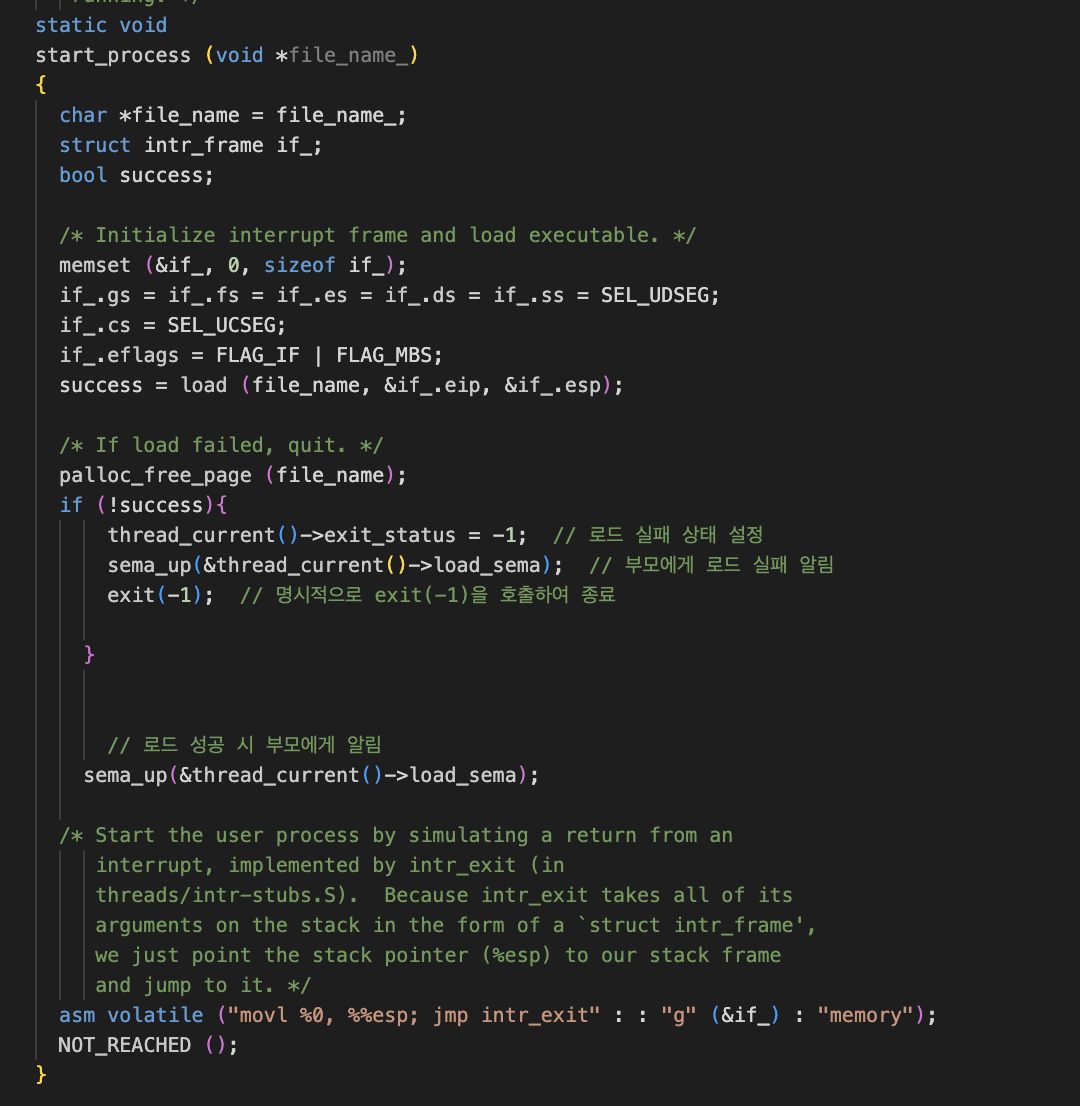
process\_execute() 함수를 호출하여 새로운 프로세스를 생성할 수 있도록 했다. 자식 프로세스가 로드될 때까지 부모 프로세스는 대기해야 했기 때문에, 자식 스레드의 load\_sema 세마포어를 사용하였다. 이때, 생성된 자식 스레드가 무엇인지 알 필요가 있었기 때문에, 자식 스레드를 찾기 위한 함수인, get\_child\_thread\_by\_tid()를 구현하였다.



이 함수는, 부모 스레드(parent)의 자식 스레드 리스트(child\_list)를 순회하해, 특정 TID(child\_tid)를 가진 자식 스레드를 찾아 반환하는 역할을 한다. 이를 위해, 핀토스의 리스트 관련 내장 함수들과 구조체를 사용하였다. 리스트 순회를 위한 포인터에 해당하는 e가 리스트의 끝(list\_end(&parent->child\_list))에 도달할 때까지 순회하며, 각각의 순회마다, list\_entry(e, struct thread, child\_elem)를 통해, 리스트 요소 e로부터 자식 스레드를 얻었다. 이를 통해, 해당 t가 찾고자 하는 TID(child\_tid)와 일치하는지 비교할 수 있도록 했다. 만약 일치한다면, 해당 자식 스레드의 포인터를 반환하고, 만약, 자식 스레드를 리스트에서 발견하지 못하였을 때는, NULL을 반환할 수 있도록 하였다.

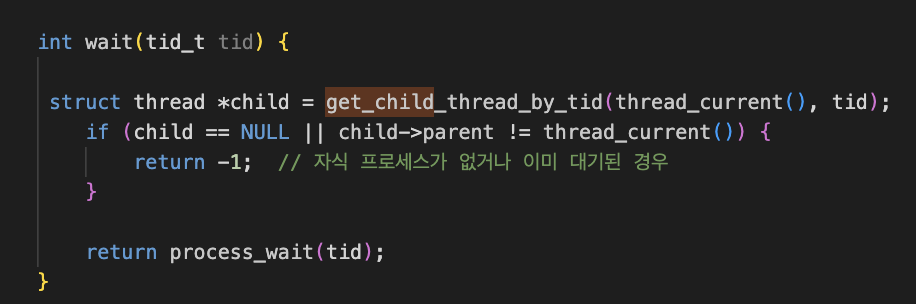
다시 exec() 시스템콜로 돌아오면, 만약, exec()에서 process\_execute() 함수를 호출하여 자식 스레드가 정상적으로 생성되었다면, get\_child\_thread\_by\_tid()는 생성된 자식 스레드의 포인터를 반환해야 할 것이다. 하지만, child가 NULL이면, 자식 스레드 생성에 문제가 발생한 것이므로 -1을 반환하도록 했다. 만약 정상적으로 생성되었다면, 이렇게 반환받은 포인터를 가지고, sema\_down을 통해 자식이 로드를 완료할 때까지 부모 스레드로써 대기할 수 있도록 했다. 만약, 자식 스레드가 로드를 완료하면 sema\_up()을 호출하여 세마포어를 업할 것이기 때문에, 부모 스레드의 대기가 해제되고 이것 이후의 코드가 실행될 것이다. 그럼, 그 이후에, 자식 스레드의 exit\_status가 -1인지 확인하여, 로드가 성공했는지의 여부를 확인했다. 그래서, 로드까지 성공했다면, exec의 모든 과정이 성공적으로 완료된 것이므로, 생성된 프로세스의 PID(pid)를 반환할 수 있도록 했다.만약 로드가 실패했다면, 자식 스레드의 포인터가 null인 경우에 -1을 반환한 것과 마찬가지로 -1을 리턴할 수 있도록 했다.

추가적으로 자식 프로세스의 로드 실패와 성공 알림은 아래와 같이, start\_process 내에서 수행되었다.



6) wait 시스템 콜

wait 시스템 콜은 부모 프로세스가 자식 프로세스의 종료를 대기하고, 자식 프로세스의 종료 상태를 반환받기 위해 사용되는 시스템 콜이다.

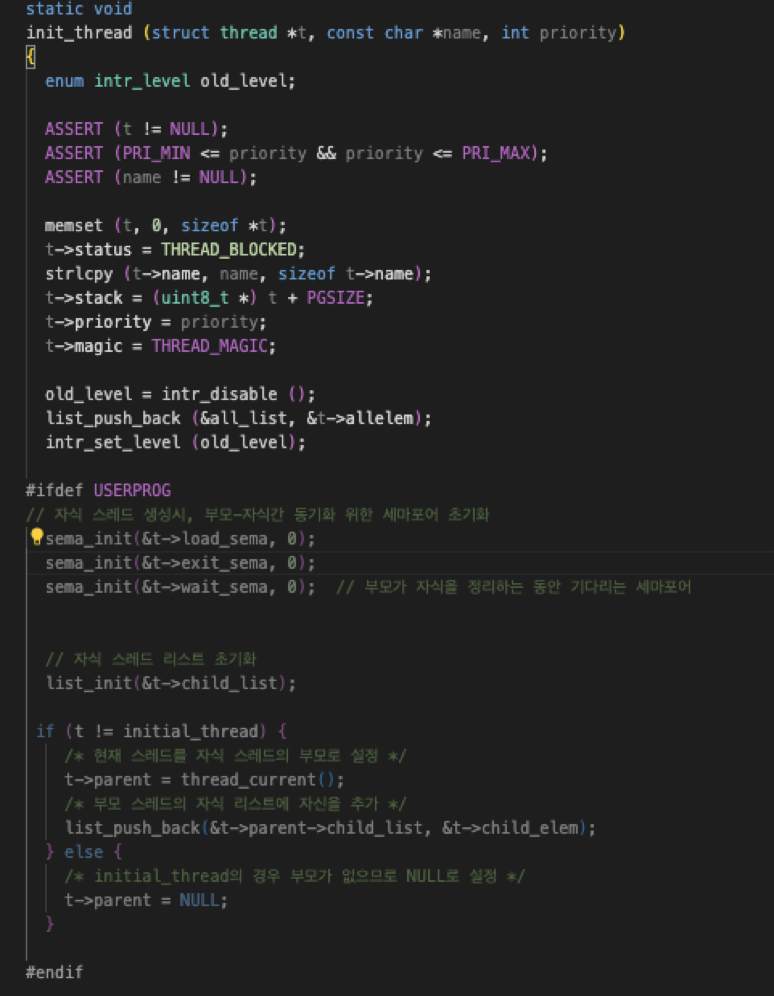


먼저 get\_child\_thread\_by\_tid() 함수를 호출하여 현재 스레드(thread\_current())의 자식 스레드 리스트에서 주어진 TID(tid)를 가진 자식 스레드를 찾고, 이를 반환한다. 그리고, 반환된 자식 스레드 포인터 child가 NULL이거나, child->parent가 현재 스레드가 아닌 경우에는, 해당 TID를 가진 자식 프로세스가 없거나, 이미 대기 중인 경우를 의미하기 때문에, -1을 반환하도록 했다. 이러한 예외상황이 아닌 경우에는, process\_wait(tid) 함수를 호출하여 자식 프로세스의 종료를 대기할 수 있도록 했다. 적절히 대기가 가능하도록 이전에는 무한루프를 통해 자식 스레드를 대기하도록 임시방편으로 구현했던 process\_wait을 아래와 같이 수정하였다.



process\_wait 함수는 특정 자식 프로세스가 종료될 때까지 현재 스레드(부모 프로세스)를 대기시키고, 자식 프로세스의 종료 상태를 반환하는 역할을 하는 함수이다. wait 시스템콜에서, process\_wait 함수를 호출하면, process\_wait 함수는 get\_thread\_by\_tid 함수를 통해, 자식 스레드의 유효성을 검사한 후, 유효할 경우에, 자식 프로세스의 종료를 sema\_down()을 통해 대기시킨다. 그리고 자식 프로세스가 exit 시스템콜에서 부모에게, 종료를 알리고, exit 시스템콜에서 자식이 exit에서 wait\_sema를 통해, 부모가 자식을 정리할 때까지 기다리게 sema\_down을 호출한다. 그러면, 자식은 부모가 종료상태를 정리할 때까지 기다리게 되는데, 이때 부모는 process\_wait에서 list\_remove를 통해, 자식 리스트에서 종료될 자식을 제거한 후에, sema\_up을 통해, 자식에게 완전히 종료해도 된다고 알린다. 그럼, 자식 프로세스는, sema\_down(&thread\_current()->wait\_sema);를 통해 대기하고 있다가 부모로부터 신호를 받으면 종료한다.

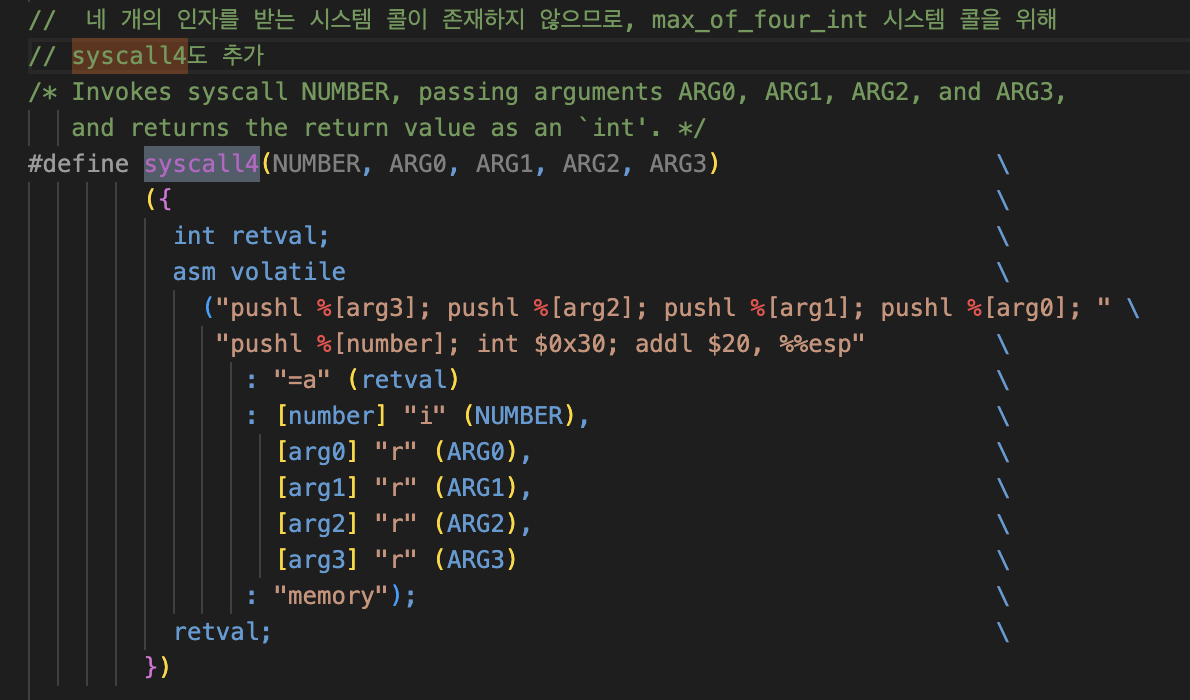
추가적으로 자식과 부모의 synchronization을 위해 세마포어 등의 초기화는 init\_thread() 내에서 이루어졌다.



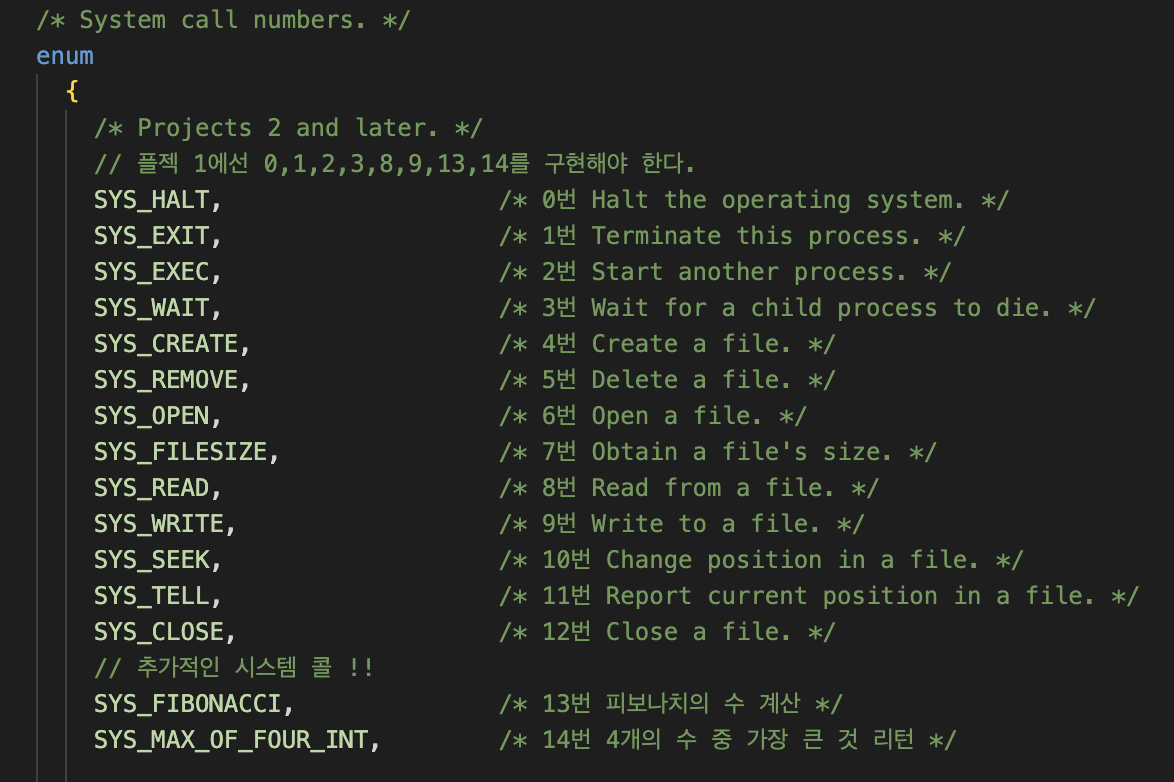
Additional System calls

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

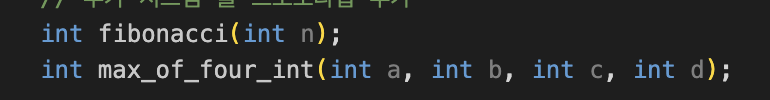
[lib/user/syscall.c] syscall4 추가



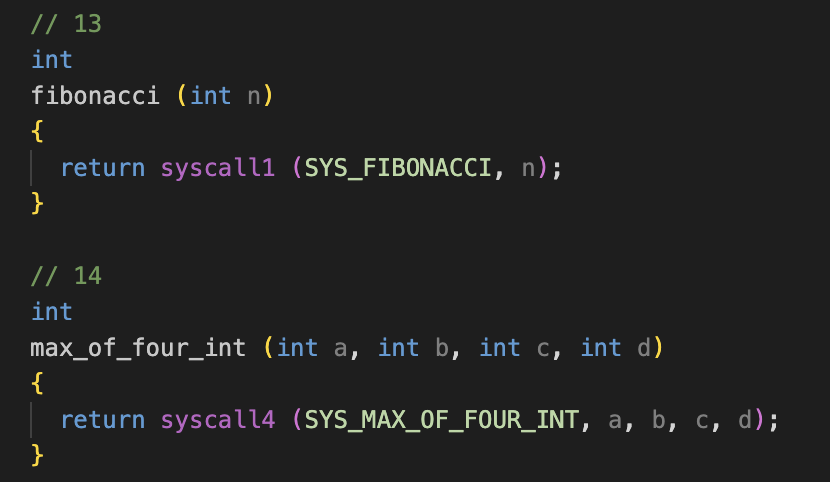
[pintos/src/lib/syscall-nr.h] : enum에 추가 시스템콜 번호 추가,



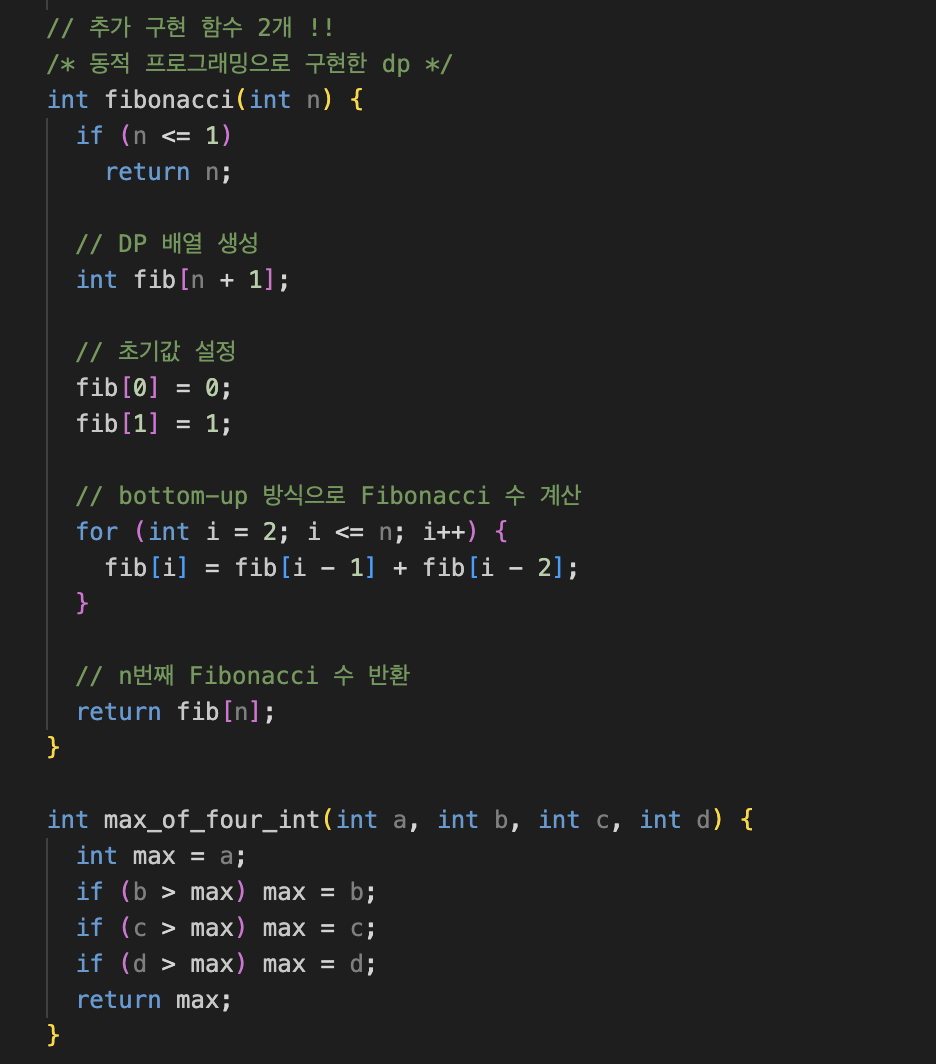
[pintos/src/lib/user/syscall.h] : 추가 시스템콜 프로토타입 추가



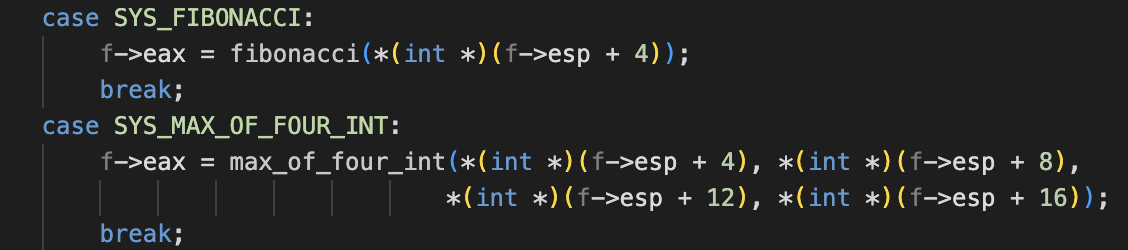
[pintos/src/lib/user/syscall.c] : 추가 시스템콜 인터페이스 추가



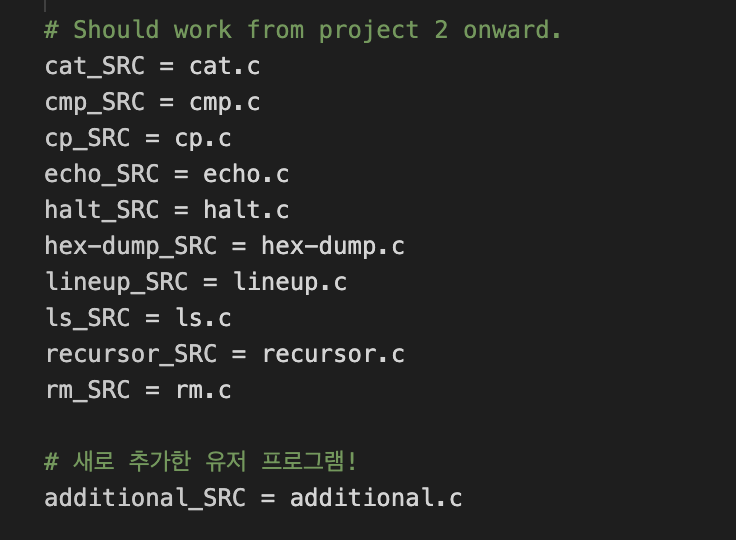
[pintos/src/userprog/syscall.c] : 피보나치, max값 구하는 시스템콜 구현



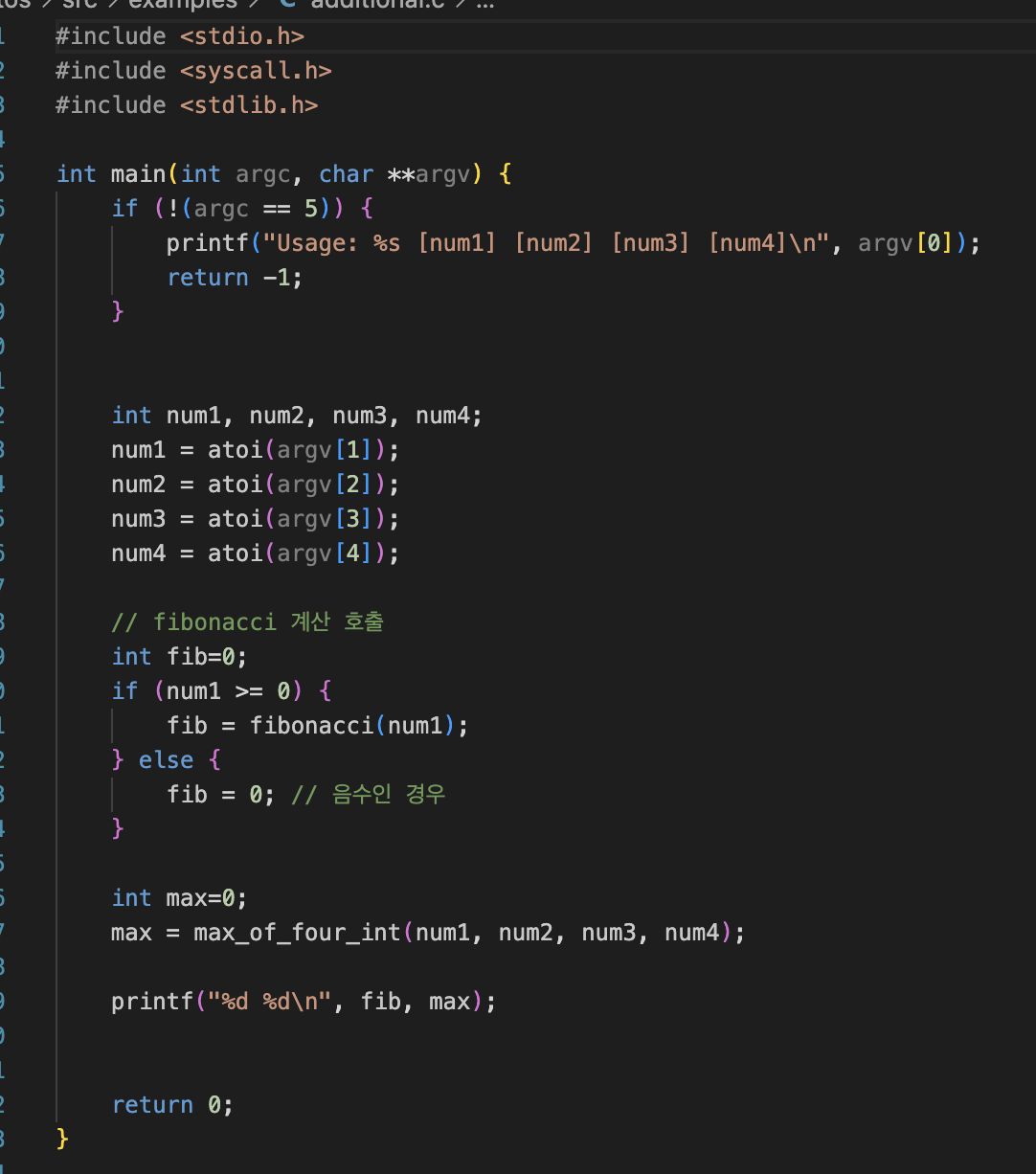
[pintos/src/userprog/syscall.c] : 피보나치, max값 구하는 시스템콜 호출 시스템콜 핸들러에 추가



[pintos/src/examples/Makefile]: additional 유저 프로그램 실행 위한 Makefile 수정

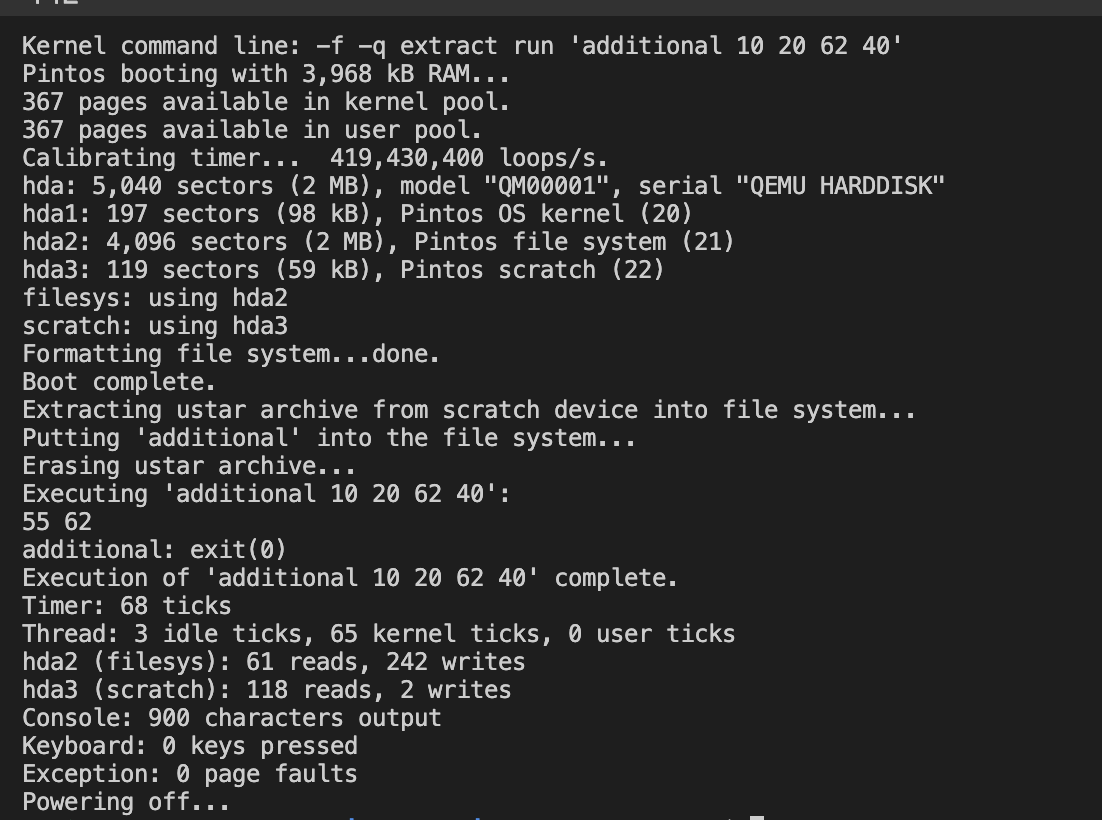


[pintos/src/examples/additional] : additional 유저 프로그램 구현

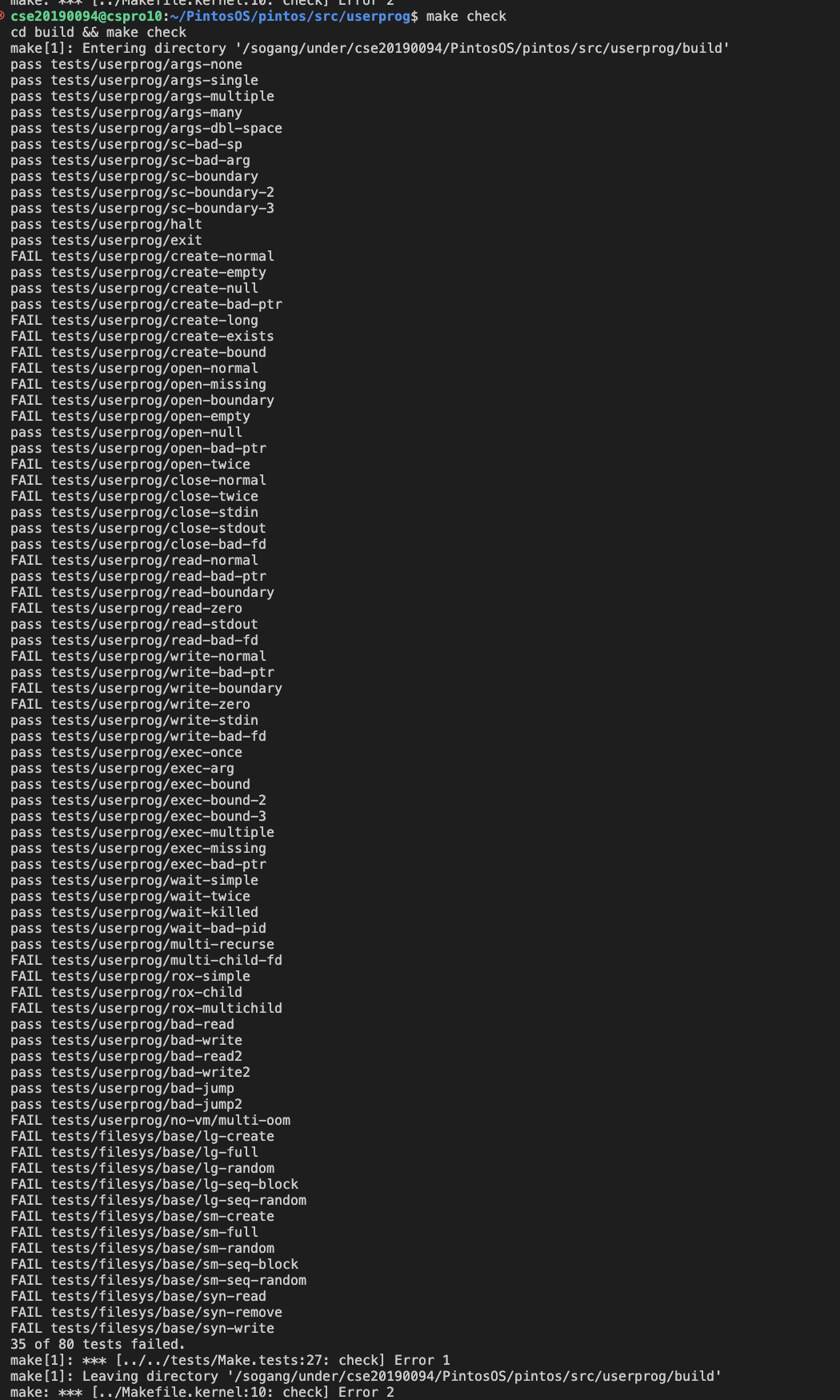


* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

**image1.png**

****

* **추가) 테스트 케이스들 실행 결과**

****