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

담당 교수 : 김영재 교수님

조 / 조원 : 20190345 김동현

개발 기간 : 2023.09.20 - 2023.10.04

1. **개발 목표**

Pintos라는 OS 환경에서 프로그램이 실행될 수 있는 환경을 구성한다. 이를 위해 총 3가지 범위에서 개발을 진행한다. 입력으로 들어온 명령어와 인자를 유저 스택에 쌓는 과정, 프로그램 실행 중 잘못된 메모리 주소로의 접근을 막는 과정, 커널과의 소통을 위해 시스템 콜을 호출하는 과정을 구현한다. 뿐만 아니라 추가적인 시스템콜과 프로그램을 구현하여 OS의 명령어 흐름을 익히고자 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing  
   command line으로 들어온 명령어와 인자를 실행하기 위해서는 user virtual memory에 순서대로 적재해야 한다. 올바른 적재 순서는 80x86 calling convention을 따른다. 스택의 가장 높은 주소에서부터 인자를 역순으로 넣는다. 이후 word-align을 맞춰준 뒤, 인자가 저장된 스택의 주소 역시 역순으로 넣는다. 이후 스택에 쌓인 0번째 인자를 가리키는 주소가 담겨있는 스택을 넣은 뒤, 인자의 개수와 return address(fake address)를 0으로 넣는다.  
    이를 위해 다음과 같은 과정을 구현해야 한다. 하나의 문자열로 들어온 command line을 공백을 기준으로 파싱해야 한다. 이후 해당 프로그램의 스택을 구성한 뒤, 스택 포인터를 통해 순서에 맞게 인자를 스택에 넣는다.
3. User Memory Access

유저 프로그램이 valid한 주소에만 접근할 수 있어야 한다. 만약 invalid한 주소에 접근하고자 한다면, 프로그램을 종료시킨다. invalid한 접근의 대표적인 예로는 Null에 접근하거나, kernel memory 영역으로의 접근, mapping 되지 않은 virtual memory로의 접근이 있다.

이를 확인하기 위해서 포인터의 Null값을 확인하거나, threads/vaddr.h 의is\_user\_vaddr( ) 함수를 사용하거나, userprog/exception.c의 page\_fault 함수에 invalid한 접근을 확인하는 statements를 추가한다. 만약 invalid한 접근이 발생한다면 exit 시스템 콜을 호출하여 -1을 전달하여 비정상적인 종료를 알린다.

1. System Calls

user 영역에서 system call을 호출하였을 경우 kernel 영역에서 이를 확인하여 요청에 따른 system call을 실행하고자 한다. 이를 위해 system call handler와 system call function을 구현해야 한다. 이번 프로젝트에서는 halt( ), exit( ), exec( ), wait( ), read( ), write( ), fibonacci( ), max\_of\_four\_int( ) 총 8개의 시스템 콜을 구현한다. read와 write 시스템 콜은 현재는 STD\_IN, STD\_OUT의 기능만 동작하도록 한다.

* 1. **개발 내용**
* Argument Passing  
  커널 내 스택에 argument를 쌓는 과정 설명  
   커널 내 스택에 인자를 쌓기 위해서는 커맨드 라인으로 입력된 인자가 어떠한 flow로 실행되는지 확인해야 한다. 아래 그림은 pintos가 동작하는 과정으로 마지막으로 userprog/process.c의 load( ) 함수에 도달하게 된다.   
  텍스트, 스크린샷, 폰트, 명함이(가) 표시된 사진

  자동 생성된 설명  
  load( ) 함수에서 문자열 파싱과 스택에 쌓는 과정이 진행된다. 파싱된 문자열은 공백을 기준으로 나뉘게 되며, argv[0]은 파일명, argv[1] ~ argv[argc-1]은 해당 파일의 인자값이 된다. 문자열을 스택에 쌓는 과정을 위해 setup\_stack(esp) 함수를 통해 스택을 구성한다. 이후 struct intr\_frame에 있는 esp 포인터를 통해 스택에 접근하며 필요한 인자를 저장한다. 인자의 저장은 다음과 같은 순서로 진행된다.  
  1) 인자를 역순으로(argv[argc-1], argv[argc-2], ... , argv[0])으로 스택에 푸시한다.  
  2) 스택의 word-alignment인 4 바이트를 맞추기 위해, 문자열의 길이에 따라 추가적으로 0을 스택에 푸시한다.  
  3) null pointer sentiel을 스택에 푸시한다.  
  4) 인자를 저장하고 있는 스택의 주소(&argv[argc-1], ... , &argv[0])를 스택에 푸시한다.  
  5) 인자의 주소를 가리키는 스택의 첫번째 주소를 스택에 푸시한다.  
  6) 인자의 개수를 나타내는 4바이트 정수를 스택에 푸시한다.  
  7) fake return address 0을 스택에 푸시한다.  
    
  올바른 스택 구성이 되었는가를 확인하기 위해서는 lib/stdio.c에 있는 hex\_dump( ) 함수를 사용해 스택의 구성을 확인한다. 아래는 'echo x'에 대하여 올바르게 스택이 구성되었음을 hex\_dump( )을 이용하여 확인한 결과이다.  
    
  텍스트, 전자기기, 키보드이(가) 표시된 사진

  자동 생성된 설명

|  |  |
| --- | --- |
| 주소 | 값 |
| 0xbfffffe0 ~ 0xbfffffe3 | 0 (return address) |
| 0xbfffffe4 ~ 0xbfffffe7 | 2 (argc) |
| 0xbfffffe8 ~ 0xbfffffeb | 0xbfffffec (&argv) |
| 0xbfffffec ~ 0xbfffffef | 0xbffffff9 (&argv[0]) |
| 0xbffffff0 ~ 0xbffffff3 | 0xbffffffe (&argv[1]) |
| 0xbffffff4 ~ 0xbffffff9 | 0 (word alignment) |
| 0xbffffff9 ~ 0xbffffffd | echo + null (argv[0]) |
| 0xbffffffe ~ 0xbffffffff | x + null (argv[1]) |

아래의 표와 같이 올바른 순서로 인자가 저장된 것을 확인할 수 있다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명  
     pintos상에서 virtual memory는 총 4GB로 구성되어 있다. 0GB ~ 3GB까지는 user의 virtual memory, 3GB ~ 4GB까지는 kernel의 virtual memory 영역이다. 이때 유저 프로그램이 user virtual memory에 access한다면 valid한 접근이지만, kernel virtual memory에 접근하고자 한다면 invalid memory access이다.
  + Invalid memory access를 어떻게 막을 것인지 설명  
     유저 프로그램이 invalid한 주소에 접근하는 경우는 스택을 쌓는 과정에서 커널 영역의 주소를 스택에 푸시하거나, mapping 되지 않은 주소에 접근하여 page fault()를 유발하는 과정에서 발생한다.  
     invalid한 메모리 접근을 막기 위해서 위해서, 해당 메모리 주소를 dereference하여 값을 얻기 전, 해당 주소값이 user 영역인지 kernel영역인지를 확인하여 잘못된 접근을 한 경우에는 exit을 하도록 한다. 또한 page fault()가 발생할 때, fault\_addr의 주소값이 커널 영역에 속하는지 확인하는 과정을 통해 user-provided pointer의 validity를 보장하고자 한다.
* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명  
     운영체제의 실행 모드는 크게 유저 모드과 커널 모드로 구분된다. 유저 프로그램이 실행되면서 프로그램은 IO device가 필요한 상황이 발생한다. 하지만 유저가 IO device와 같은 하드웨어를 직접 컨트롤 하게 된다면, 오류의 가능성은 물론 여러 유저 프로그램이 경쟁하는 상황이 발생한다. 이를 관리하고 보호하기 위해 IO device와 같은 하드웨어 기능을 커널 모드에서 관리한다. 유저 모드에서는 커널 모드에게 하드웨어 디바이스를 사용하고자 하는 요청을 보내야 하는데 이를 위한 기능이 시스템 콜이다. 커널 모드는 시스템 콜을 요청 받고 처리하면서 불안정한 요청을 거부하고, 필요한 자원을 프로그램 간의 충돌 없이 안정적으로 유저 프로그램에 제공하게 된다.
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)  
    - halt( )  
     devices/shutdown.h의 shut\_down\_power\_off( )를 호출하여 pintos를 종료시킨다.  
    - exit( )  
     프로세스가 종료될 때 호출되며, 프로세스가 종료되었음을 나타내는 출력문을 출력하고, 인자로 들어온 종료 상태를 해당 스레드의 종료 상태에 저장하고 thread\_exit( )을 호출한다.  
    - exec( )  
     process\_execute( )를 호출하여, 인자로 들어온 파일 이름을 실행한다. 실행에 성공하면 해당 스레드의 아이디를, 실패한다면 -1을 리턴한다.  
    - wait( )  
     process\_wait( )를 호출하여, 부모 프로세스가 인자로 들어온 스레드 아이디를 가진 자식 프로세스가 종료될 때까지 대기한다. 만약 자식 프로세스의 아이디가 아닐 경우 -1을 리턴한다.  
    - read( )  
     표준 입력으로 들어온 문자열을 input\_getc( ) 함수를 통해 읽어온다. 읽어온 총 바이트 수를 리턴하며, 표준 입력이 아닌 경우 -1을 리턴한다.  
    - write( )  
     표준 출력을 하고자 하는 문자열을 putbuf( ) 함수를 통해 출력한다. 출력한 총 바이트 수를 리턴하며, 표준 출력이 아닌 경우 -1을 리턴한다.  
    - fibonacci( )  
     인자로 들어온 정수 n에 대하여 n번째 피보나치 수를 계산하여 리턴한다.  
    - max\_of\_four\_int( )  
     인자로 들어온 4개의 정수 중 가장 큰 정수를 리턴한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명  
     유저 레벨(lib/user/syscall.c)에서 시스템 콜 API를 호출하게 되면, 해당 시스템 콜의 인자 개수에 맞는 매크로 함수가 실행된다. 이 매크로 함수는 시스템 콜의 번호와 인자를 스택에 푸시한 뒤, interrupt를 걸어 커널 레벨에서 시스템 콜 요청을 알아차리도록 한다.  
     커널 레벨(userprog/syscall.c)에서 시스템 콜 핸들러가 실행되고, 해당 시스템 콜 번호에 맞는 시스템 콜 함수가 실행된다. 이 함수에서 해당 요청에 맞는 instruction이 실행되고, 결과를 반환하여 유저 프로그램에게 전달한다.  
     유저 레벨(lib/user/syscall.c)는 반환값을 통해 시스템 콜의 결과를 확인하며 다음 instruction을 이어나간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2023.09.20 - 2023.09.24 : pintos manual 및 ppt 숙지

2023.09.25 - 2023.09.27 : argument parsing and passing

2023.09.28 - 2023.09.28 : Invalid Memory Access

2023.09.29 - 2023.10.02 : System Call

2023.10.03 - 2023.10.03 : Fibonacci and Max\_of\_four\_int

2023.10.03 - 2023.10.05 : Debugging and Document

* 1. **개발 방법**

- Argument Passing

userprog/process.c load( ) 함수에서 setup\_stack( ) 함수를 통해 스택이 구성된 이후, file\_name을 파싱하고 esp 스택 포인터를 사용하여 스택에 인자를 쌓기 위해서 parseAndInsertArgumentIntoStack( ) 함수를 호출한다.

userprog/process.c parseAndInsertArgumentIntoStack( ) 함수에서 문자열을 빈칸을 기준으로 파싱한 뒤, 80x86 Calling Convention에 따라 문자열을 스택에 적재한다

- Invalid Memory Access

threads/vaddr.h에 선언되어 있는 is\_user\_vaddr( ), is\_kernel\_vaddr( ) 함수를 사용하여 해당 주소가 valid한 주소값인지 주로 확인하게 된다.

userprog/syscall.c syscall\_handler( ) 함수에서 스택에 담겨온 인자들은 유저 프로그램에서 호출한 시스템 콜이므로, 메모리의 유저 영역에 존재해야 한다. 만약 커널 영역에 해당 값이 존재한다면 이는 invalid memory access이다. 스택에 담긴 모든 시스템 콜과 인자들의 주소를 확인해야하므로 check\_user\_address( ) 함수를 구현한다. 만약 커널 영역에 접근하게 된다면 exit(-1)을 호출하여 프로세스를 종료하도록 한다.  
userprog/exception.c page\_fault( ) 함수에서 page\_fault로 들어온 fault\_addr이 valid한지 확인한다. page\_fault가 커널모드에서 발생하거나, 커널 메모리 영역에서 발생한다면, invalid한 메모리 접근이므로 exit(-1)을 호출한다.

- System Call  
 src/userprog/syscall.h에 구현할 시스템 콜의 prototype을 선언한다. sys\_halt, sys\_exit, sys\_exec, sys\_wait, sys\_read, sys\_write, fibonacci, max\_of\_four\_int 총 8개의 시스템 콜 프로토타입을 선언한다.

src/userprog/syscall.c에서 위에서 선언한 프로토타입의 구현과 더불어 syscall\_handler( ) 함수를 구현한다. 스택 포인터를 참조하여, 시스템 콜 번호를 얻어낸 뒤, switch문을 활용하여 요청에 알맞은 시스템 콜을 호출한다. 시스템 콜의 반환값이 있을 경우 intr\_frame 구조체의 eax 인자에 저장하도록 한다.

각각의 시스템 콜은 다음과 같이 구현한다.  
 sys\_halt( )은 src/device/shutdown.h에 정의된 shutdown\_power\_off( ) 함수를 사용하여 핀토스를 종료시킨다.

sys\_exit( )은 src/threads/thread.h에 정의된 struct thread에 스레드의 종료 상태를 저장하기 위한 정수형 변수 exit\_status를 선언한다. 스레드의 현재 종료 상태를 커맨드 창에 띄우는 출력문을 작성한다. 이후 현재 exit 요청을 받은 스레드 구조체의 exit\_status에 인자로 들어온 status를 저장한 뒤, src/userprog/process.c의 thread\_exit( ) 함수를 호출하여 유저 프로그램을 종료한다.

sys\_exec( )은 src/userprog/process.c의 process\_execute( ) 함수를 호출하여 인자로 들어온 프로그램을 실행한다. 이때 스레드의 이름을 명령어로 저장해야하기 때문에, thread\_create( )를 호출하기 전 스레드의 이름을 위해 파일명을 파싱하여 저장해야 한다. 이때 invalid한 파일명이 입력으로 들어온다면 -1을 반환한다. 만약 valid한 파일명이 입력으로 들어오고, 정상적으로 스택이 구성된다면 해당 프로세스의 id를 반환한다. sys\_exec( )은 process\_execute( )의 반환값인 해당 프로세스의 id를 유저 영역으로 반환한다.

sys\_wati( )은 src/userprog/process.c의 process\_wait( ) 함수를 호출하여 인자로 들어온 프로세스의 아이디를 대기하도록 한다. 개발 초기에는 process\_wait( )는 무한루프를 구현하여 부모 프로세스가 자식 프로세스가 실행되는 동안 wait 상태를 유지하도록 한다. 추후 함수를 완성할 때는 다음과 같은 요소를 구현한다.

src/threads/thread.h에 struct thread에 부모-자식 관계 저장을 위해 자식 스레드를 저장할 연결리스트 자료구조와 본인을 가리키는 연결리스트 원소를 선언한다. 이후 자식 프로세스가 생성될 때마다 부모-자식 관계를 연결리스트에 추가하기 위해 src/threads/thread.c init\_thread( ) 함수에 부모 스레드의 연결 리스트의 자식 스레드를 추가하는 기능을 구현한다. 뿐만 아니라 자식 스레드 역시 부모 스레드가 될 수 있으므로 자식 스레드의 연결리스트를 초기화해준다.

src/userprog/process.c의 process\_wait( )에서는 위와 같은 자료구조를 사용하여 부모가 본인의 자식 프로세스를 담은 연결리스트를 순회하며 wait하고 있는 자식 프로세스를 찾는다. 만약 찾지 못한 경우 -1을 반환하고, 찾은 경우 연결리스트에서 자식 프로세스를 제거하고 해당 자식 프로세스의 종료 상태를 반환한다. 이때 자식 프로세스의 실행이 끝나지 않았음에도 불구하고, 즉 자식 프로세스에서 exit( ) 시스템 콜이 호출되지 않았음에도 불구하고 부모 스레드는 자식 프로세스를 제거하는 경우가 발생할 수 있다. 이를 방지하기 위해 src/threads/synch.h에 있는 세마포어 자료형을 사용한다. process\_exit( )에서 세마포어를 업시켜 프로세스가 끝났음을 알리고, process\_wait( )에서 리스트에서 제거하기 전 세마포어를 다운시켜 두 실행 흐름의 순서를 강제한다. 뿐만 아니라 부모 스레드가 wait. 과정을 실행하는 동안 새로운 자식 프로세스의 exit을 놓치지 않기 위해 새로운 세마포어를 사용한다. 이는 process\_wait( )가 끝나기 전 세마포어를 업 시켜 wait 과정이 끝났음을 알리고, process\_exit( )에서 세마포어를 다운시켜 대기 하고 있는 자식 프로세스의 실행을 종료 시킨다.

sys\_read( )는 현재 프로젝트에서 표준 입력으로부터만 입력을 받아들인다. 인자로 들어온 읽어들일 문자열의 크기만큼 읽어들이거나, 문자열이 끝날 때까지 읽어들읻다. 문자열을 읽는 과정은 src/devices/input.h에 선언된 input\_getc( ) 함수를 사용한다. 이후 읽어들인 문자열의 길이를 반환한다.

sys\_write( )는 현재 프로젝트에서 표준 출력으로만 출력을 한다. 인자로 들어온 버퍼를 putbuf를 사용하여 표준 출력을 수행한뒤, 출력한 문자열의 수를 반환한다.

fibonacci( )는 인자로 들어온 n을 통해 n번째 정수의 피보나치 수열을 반환한다. src/userprog/syscall.c의 fibonacci 함수에 값을 구하는 함수를 구현한 뒤, lib/syscall-nr.h에 해당 시스템 콜 번호를 등록한다. lib/user/syscall/h에 fibonacci( ) 함수를 정의 한 뒤, lib/user/syscall.c에 fibonacci( ) 함수를 구현하여 syscall1( )을 호출하도록 한다.

max\_of\_four\_int( )는 인자로 들어온 4개의 정수 중 가장 큰 정수를 반환한다. src/userprog/syscall.c의 max\_of\_four\_int( ) 함수에 값을 구하는 함수를 구현한 뒤, lib/syscall-nr.h에 해당 시스템 콜 번호를 등록한다. lib/user/syscall/h에 max\_of\_four\_int( ) 함수를 정의 한다. lib/user/syscall.c 인자의 개수가 4개인 max\_of\_four\_int( )를 위해 syscall4( )을 define한 뒤, max\_of\_four\_int( ) 함수를 구현하여 syscall4( )을 호출하도록 한다.

위에서 구현한 fibonacci( ), max\_of\_four\_int( ) 시스템 콜을 구현하기 위해 src/examples/additional.c 프로그램을 생성한다. 두 시스템 콜을 호출하여 값을 출력한다. 이를 make 명령어로 컴파일 하기 위해 src/examples/Makefile에 additional.c에 관한 명령어를 추가한다.

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

텍스트, 스크린샷, 폰트, 그래픽 디자인이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access  
   텍스트, 스크린샷, 폰트, 그래픽이(가) 표시된 사진

   자동 생성된 설명
2. System Calls  
   스크린샷, 텍스트, 디자인이(가) 표시된 사진

   자동 생성된 설명   
   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

src/userprog/process.c의 load( ) 함수 내에서 파싱 및 스택으로의 인자를 쌓기 위한 parseAndInsertArgumentIntoStack( ) 함수 호출한다. 기존에는 파싱과 패싱을 다른 함수를 통해서 구현하고자 하였으나, 구현 과정에서 파싱된 인자값이 함수 인자로의 전달과정에서 사라지는 이슈가 발생하였다. 이를 해결하기 위해 하나의 함수에서 파싱과 패싱을 진행하였다.



위 그림는 file\_name을 파싱하는 과정을 구현한 코드이다. 우선 문자열의 가장 앞부분에 빈칸이 들어올 수 있는 경우를 대비해 빈칸을 제거하였다. 이후 문자열이 null문자를 만나거나, 빈칸을 만나는 경우에 해당 인덱스를 기준으로 문자열을 파싱하였다. 구분된 문자열은 2차원 배열 parsed\_command 안에 담기게 되며, 문자열의 마지막과 배열의 마지막은 null 문자로 표시하였다

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

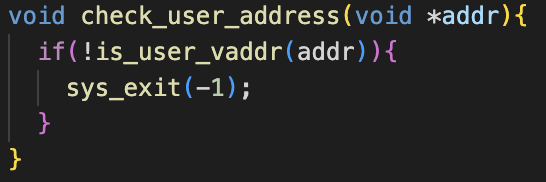
자동 생성된 설명

위 그림는 문자열을 스택에 쌓는 과정을 구현한 코드이다. 문자열을 역순으로 스택에 쌓게되는데, 이때 word-alignment를 맞추기 위해 쌓이는 문자열의 길이를 len 변수에 저장하였다. 또한 addr\_list에 argv가 스택에 쌓이는 주소를 저장하였다. 인자를 저장하며 계산한 len 변수를 이용하여 word-alignment를 맞춰주기 위해 스택 포인터를 4의 배수형태로 줄여주었다. 이후 저장된 스택 주소를 다시 스택에 쌓아주었으며, 이에 대한 첫번째 주소를 가리키는 주소와 인자의 개수, 마지막으로 fake address값 0까지 스택에 쌓아주었다.

또한 test 상에서 multi-recurse 테스트가 FAIL이 되는 이슈가 존재하였다. 이는 기존 파싱된 문자열을 저장하기 위한 배열의 크기와 나머지 자료형의 크기가 4KB를 넘어 page\_fault를 유발하였기 때문이라고 판단하여, 파싱된 문자열을 저장하는 배열의 크기를 감소하여 해결하였다.

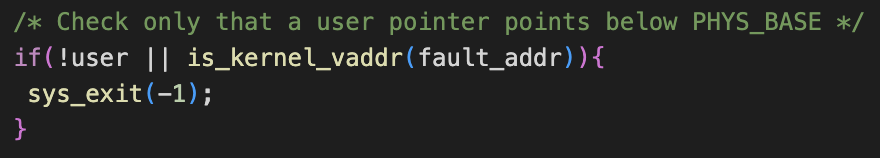
1. User Memory Access

src/userprog/syscall.c 에서 스택에 위치한 시스템 콜 및 인자가 유저 영역에 올바르게 위치했는지 확인하는 과정이 필요했다. 이를 위해 check\_user\_address( ) 함수를 구현하였다.



위 그림은 현재 포인터가 가르키는 주소 영역이 유저 영역을 가리키고 있는지 확인하는 함수이다. threads/vaddr.h에 선언된 is\_user\_vaddr( ) 함수를 이용함으로써 현재의 가상 메모리 주소가 유저영역인지 확인한다. 만약 커널 영역이라면 잘못된 영역을 참고하고 있으므로 exit(-1) 시스템 콜을 호출하여 종료한다. 이를 시스템콜 번호, 시스템 콜 인자 모두에 대해 확인하여 잘못된 메모리 접근을 막는다.

src/userprog/exception.c에서 page\_fault( )의 인자로 들어오는 fault\_addr에 대하여 유저의 잘못된 메모리 참조를 검사하였다.



위 그림은 page\_fault가 발생할 경우 호출되는 page\_fault( )함수로, user가 false이거나 인자로 들어온 주소가 커널 영역인지를 확인해주는 is\_kernel\_vaddr( )이 true인 경우 유저가 커널영역을 침범하였음을 알 수 있다. 이럴 경우 exit(-1) 시스템 콜을 호출하여 프로그램을 종료시킨다.

1. System Calls

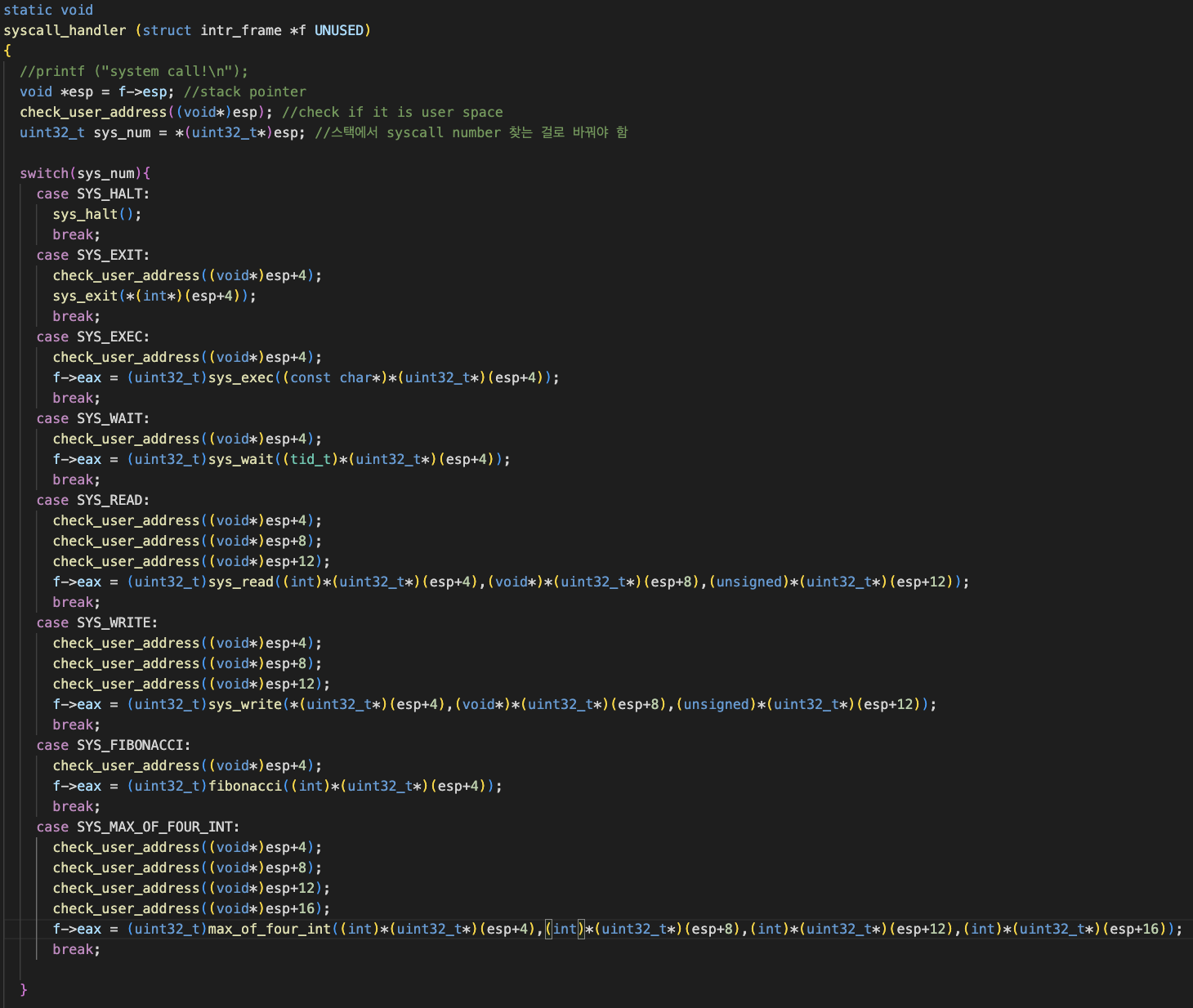
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

시스템 콜 함수 중 exit( ), wait( ), exec( )에 필요한 자료구조를 threads/thread.h에 있는 struct thread에 추가하였다.

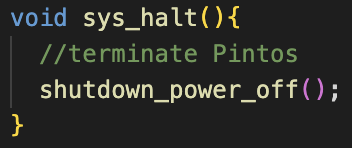
- src/userprog/syscall.c

유저 프로그램이 시스템 콜을 호출하게 되면, 스택에 시스템콜 번호와 인자가 담겨 시스템콜 핸들러가 호출된다. 시스템 콜 핸들러는 스택에서 값을 pop하여 시스템 콜 번호를 확인하고 스위치 문을 통해 이에 맞는 커널 영역에서의 시스템 콜 함수를 호출한다.

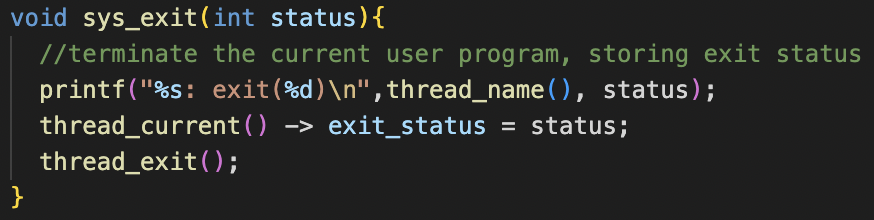


위 그림은 시스템 콜 핸들러를 구현한 코드로, 시스템 콜 함수를 스택에서 확인하여 알맞은 함수를 호출하는 것을 알 수 있다. 또한 시스템 콜이 반환값을 가질 경우 스택의 eax 인자에 값을 저장한다.

시스템 콜 함수의 각각의 구현을 살펴보면 다음과 같다.



halt( )는 devices/shudown.h에 선언된 shutdown\_power\_off( ) 함수를 호출함으로서 QEMU위에서 구동되고 있는 핀토스를 종료시킨다.

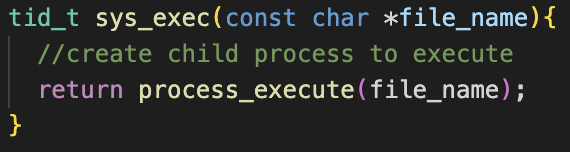


exit( )은 커맨드 라인으로 현재 스레드의 이름과 종료 상태와 함께 종료되었음을 알리고, 현재 스레드의 종료 상태를 저장한 뒤, thread\_exit( )을 호출하여 스레드를 종료한다. 이때 사용된 핀토스 API 함수들은 모두 threads/thread.h에 선언된 함수로, 현재 스레드의 이름을 반환하는 함수, 현재 스레드 구조체를 반환하는 함수, process\_exit( ) 함수를 호출하여 스레드를 종료시키는 함수이다.

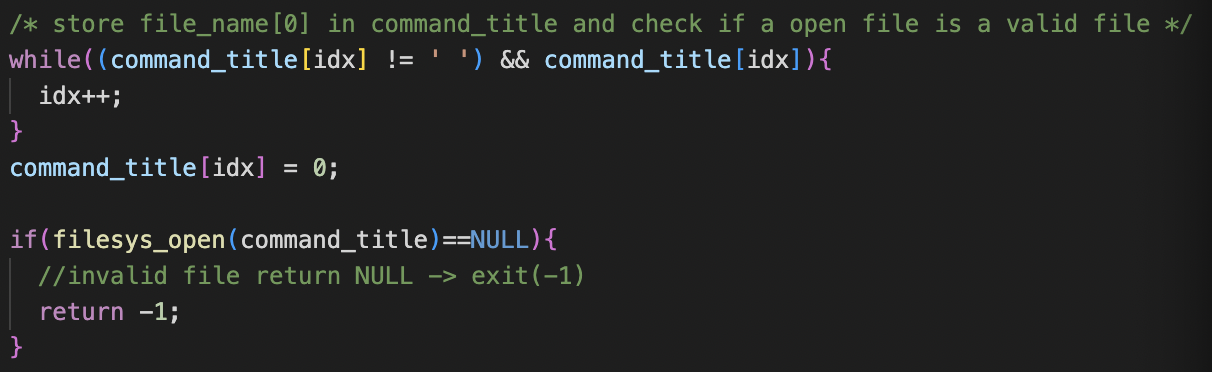
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

process\_exit( ) 역시 exit( ) 시스템 콜을 위해 다음과 같은 코드를 추가하였다. 현재 스레드(자식 스레드) 관점에서 child\_exit\_sema 세마포어를 업시키면서 자식 프로세스가 종료되었음을 부모 프로세스에 알린다. 부모 프로세스는 해당 세마포어가 0으로 블록킹되어 있는 상태에서 1로 바뀌어 자식 프로세스를 Reaping하는 과정을 진행할 수 있게 된다. 또한 child\_wait\_sema 세마포어는 부모 프로세스가 wait( ) 함수를 끝마치기를 기다리며 블로킹되어 있다. 만약 부모가 wait( )를 마치고 해당 세마포어를 업해 준다면 자식이 exit( )하게 되어 자식이 orphan 상태가 되지 않는다.



exec( ) 시스템 콜은 userprog/process.c의 process\_execute( ) 함수를 호출하여 인자로 들어온 파일을 실행하도록 한다. process\_execute( )는 인자로 들어온 실행파일을 확인하고 메모리 영역을 만들어 준 뒤, 새로운 스레드를 만드는 thread\_create( ) 함수를 호출하여 새로운 실행흐름을 만든다.

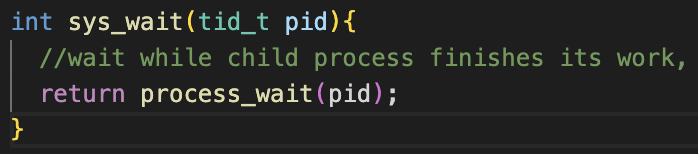


위 그림은 올바른 실행파일명을 얻기 위해 파싱을 진행한 함수를 추가한 코드이다.

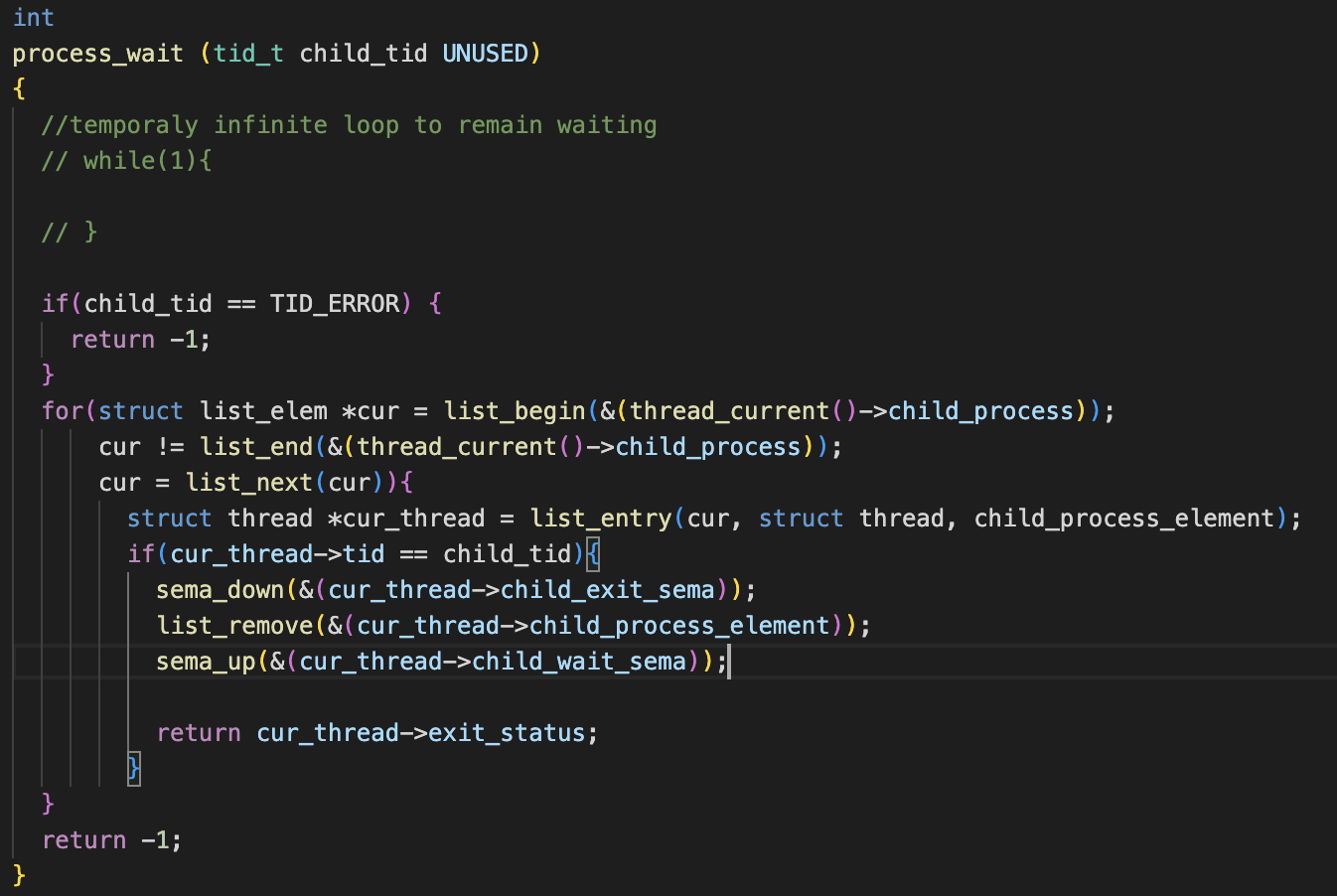
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

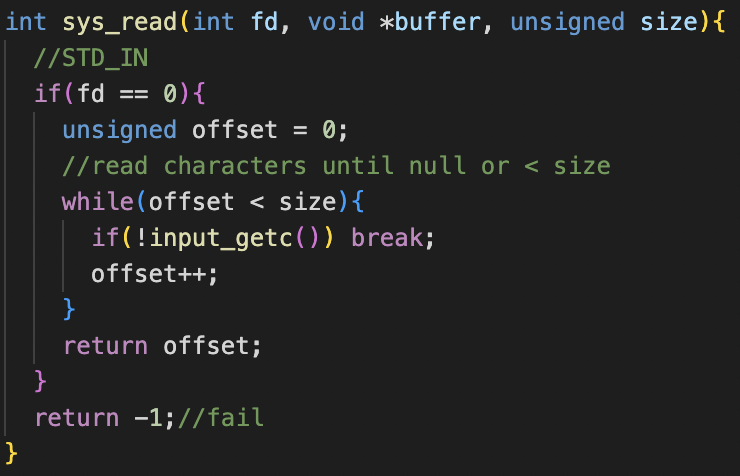
또한 프로세스가 실행되었을 경우 thread 구조체가 생성되는데, 이에 필요한 자료구조를 업데이트 하였다. sema\_init( )은 세마포어를 초기화하는 함수이며, list\_init( )은 연결리스트를 초기화하는 함수이다. list\_push\_back( )은 연결리스트에 마지막에 인자를 추가하는 함수이다.



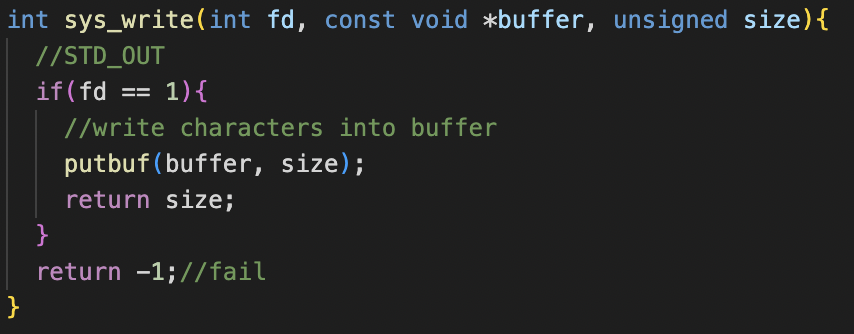
wait( ) 시스템 콜은 부모 프로세스가 자식 프로세스를 실행하고 난 뒤, 자식 프로세스를 reaping하기 위해 대기하는 시스템 콜이다. 이를 위해 userprog/process.c의 process\_wait( ) 함수를 호출한다.



process\_wait( )함수는 현재 구동중인 프로세스인, 즉 부모 프로세스의 구조체에 저장된 연결리스트를 순회한다. 이 연결리스트는 자식 프로세스를 담은 리스트이다. 이를 순회하며 reaping하고자 하는 자식 프로세스의 id와 일치하게 되면, 해당 프로세스를 연결리스트에서 제거하는 list\_remove( ) 함수를 호출하여 reaping을 완료한다. 이때, 자식 프로세스가 종료된 뒤, 리스트에서 제거해야하는 순서를 강제하기 위해 child\_exit\_sema 세마포어를 사용하였고, 다수의 자식 프로세스가 reaping을 기다리게 되어 고아가 되는 것을 막기 위해 child\_wait\_sema 세마포어를 사용하여 순서를 강제하였다. 정상적으로 자식 프로세스의 reaping을 완료한다면, 자식 프로세스의 종료 상태를 리턴하고, 그렇지 않은 경우에는 -1을 리턴하도록 한다.



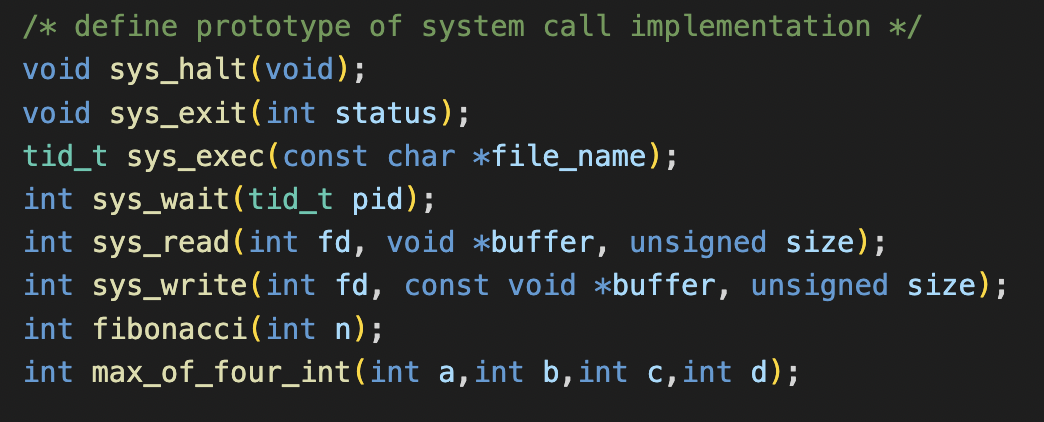
read( ) 시스템 콜은 지정된 파일 디스크립터로부터 입력을 읽어오는 시스템 콜이다. 본 프로젝트에서는 표준 입력만 처리하였으며, 그 외의 경우는 -1을 반환하였다. devices/input.h에 정의되어 있는 input\_getc( )를 통해 문자를 읽어왔다. 버퍼가 다 차거나 문자열을 읽는 중 null을 만나기 전까지 입력을 받아 버퍼에 저장한다. 입력 받은 크기를 알기 위해 정수 offset을 문자 하나를 읽어올 때마다 offset을 증가시켜 문자열의 길이를 측정하여 반환한다.



write( ) 시스템 콜은 지정된 파일 디스크립터로의 출력을 하는 시스템 콜이다. 본 프로젝트에서는 표준 출력만 처리하였으며, 그 외의 경우에는 -1을 반환하였다. lib/kernel/console.h에 정의된 putchar( ) 함수를 통해 버퍼를 size만큼 출력하였다. 콘솔창에 사이즈만큼 atomic하게 문자열을 위치시킨 뒤, 출력하게 된다. 반환값은 출력한 문자열의 사이즈이다.

- src/userprog/syscall.h

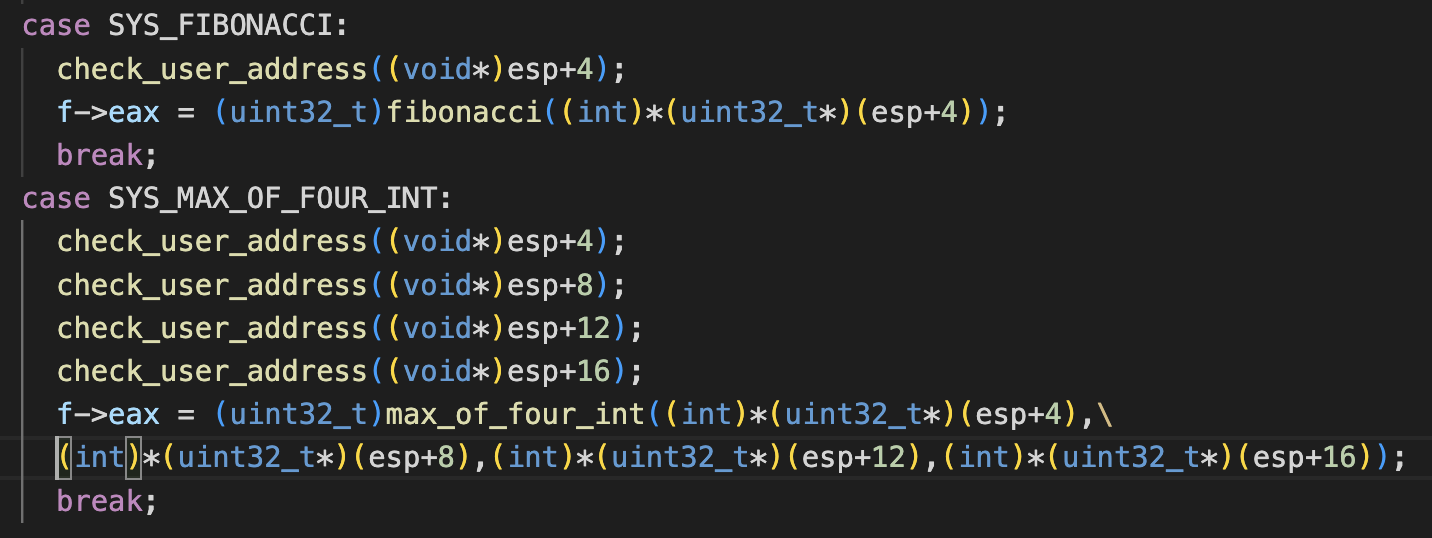
시스템 콜 함수의 프로토 타입을 정의하였다.



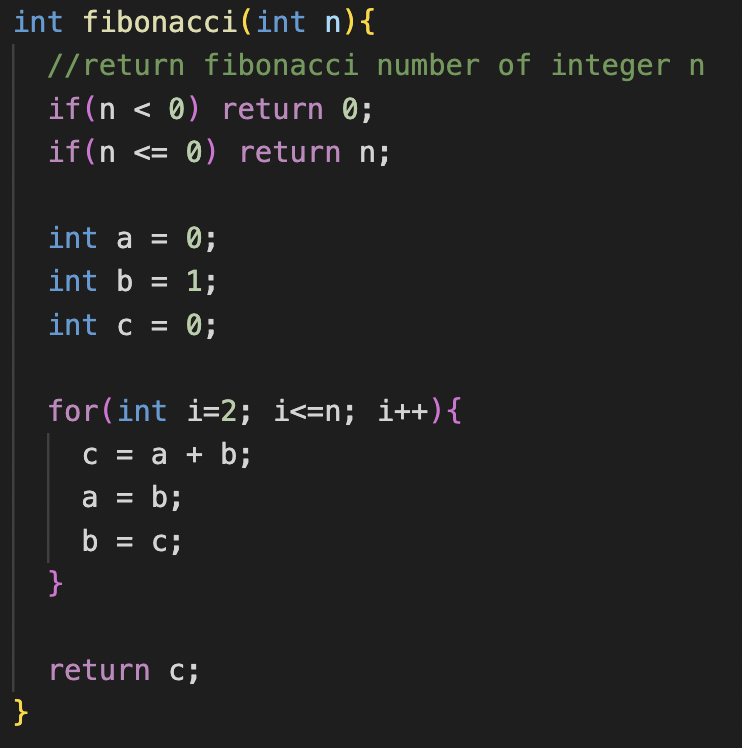
1. Additional System calls

- src/userprog/syscall.c

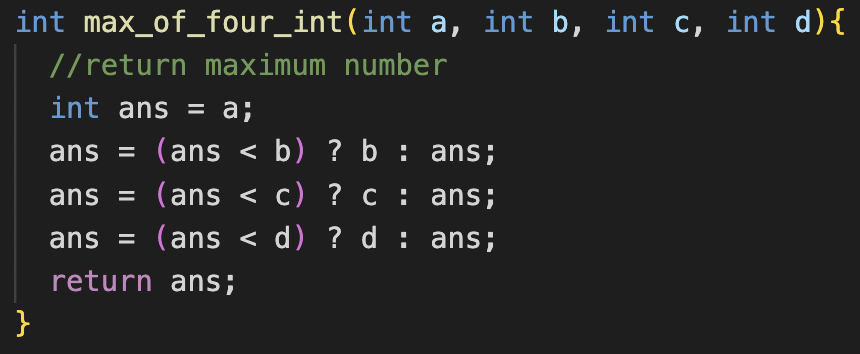
syscall\_handler( )에서는 스위치문을 통해 입력으로 들어온 시스템 콜을 호출한다. 새로운 시스템 콜인 fibonacci와 max\_of\_four\_int 를 스위치 문에 추가한다. 두 시스템 콜 역시 check\_user\_address( ) 함수를 통해 두 시스템 콜 인자가 저장된 영역이 유저 메모리 영역에 정상적으로 저장되어 있는지 확인한다.



int fibonacci(int n)을 구현하였다. 입력으로 들어온 n을 통해 n번째 피보나치 수를 구한다. 피보나치 수의 일반항은 이므로 일차원 for문을 활용하여 값을 구할 수 있다. 이를 통해 구해진 n번째 피보나치 수를 반환한다.

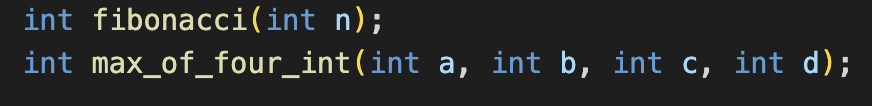


int max\_of\_four\_in(int a, int b, int c, int d)을 구현하였다. 입력으로 들어온 4가지 수 중 가장 큰 수를 반환하도록 한다.



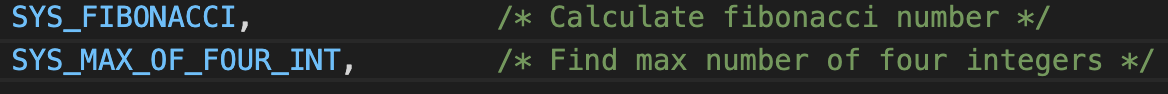
- src/userprog/syscall.h

새롭게 추가된 시스템 콜에 대한 함수 프로토 타입을 정의하였다.



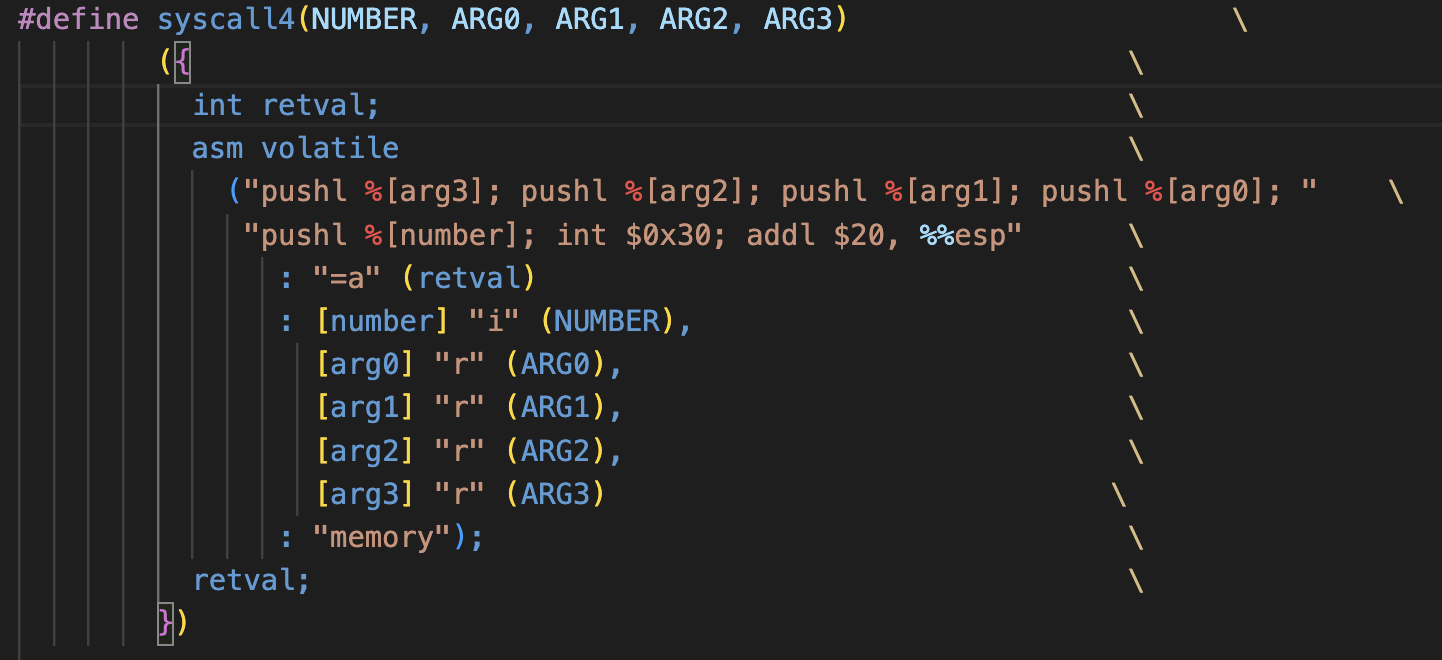
- lib/syscall-nr.h

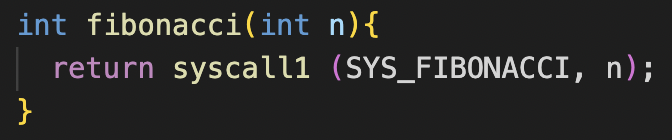
시스템 콜 넘버를 SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INT과 같이 지정하였다. 시스템콜을 가리키는 메크로로 시스템 콜을 호출할 때 다음과 같이 표현하기로 한다.



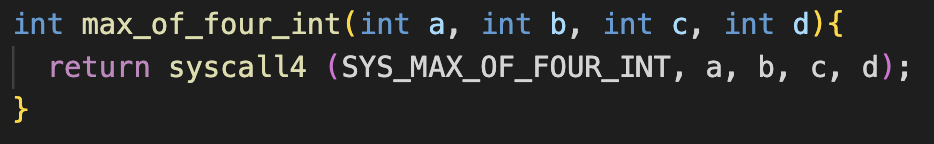
- lib/user/syscall.c

4개의 인자를 받는 시스템 콜인 max\_of\_four\_int( )를 위해 #define syscall4( )을 작성하였다. 시스템콜 번호와 4개의 인자를 받아 스택에 push한다. 이후 0x30의 interrupt를 저장한 뒤, 스택 포인터를 더해준다. 이후 return 값을 retval로 받아와 유저 프로그램으로 돌려준다.



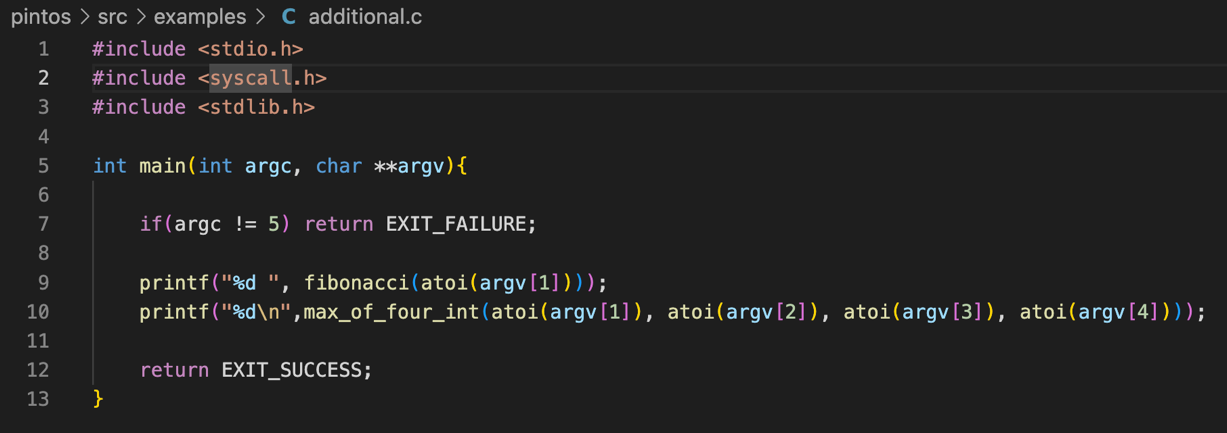
int fibonacci(int n)는 하나의 인자만을 가지므로 syscall1( )을 호출하게 된다.  


int max\_of\_four\_int(int a, int b, int c, int d)은 4개의 인자를 가지므로 새롭게 정의한 syscall4( )를 호출하게 된다.



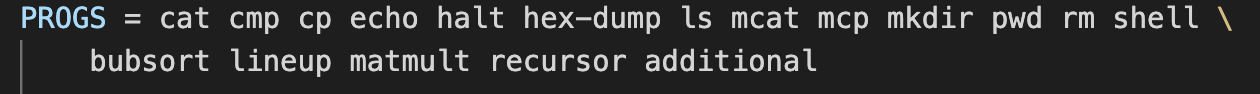
- src/examples/additional.c

2개의 시스템 콜을 호출하는 프로그램을 만들기 위해 새로운 c파일 additional.c를 생성하였다. 인자는 총 5개(./addtional 1 2 3 4)이어야 하며, 그렇지 않을 경우 EXIT\_FAILURE를 반환한다. 그렇지 않을 경우 fibonacci( ) 시스템 콜과 max\_of\_four\_int( ) 시스템 콜을 호출하여 반환값을 출력한다.



- src/examples/Makefile

make 명령어를 통한 컴파일을 위해 다음과 같이 수정하였다.



생성될 프로그램 이름을 지정하는 메크로로 addtional을 추가하였다.



addtional 프로그램을 생성하는 소스파일로 additional.c를 지정하였다.

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부. 텍스트, 스크린샷, 폰트이(가) 표시된 사진

  자동 생성된 설명** 'pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run 'additional 10 20 62 40'을 실행한 결과이다. 출력값으로 피보나치 수열의 10번째 값인 55와 인자 중 가장 큰 수인 62가 출력된 것을 확인할 수 있다.
* **make check 결과  
  텍스트, 폰트, 스크린샷, 흑백이(가) 표시된 사진

  자동 생성된 설명텍스트, 폰트, 스크린샷, 흑백이(가) 표시된 사진

  자동 생성된 설명** make check를 통해 총 21개의 test가 pass한 것을 확인할 수 있다.