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

담당 교수 : 김영재

조 / 조원 : 한솔

개발 기간 : 2021.09.11~2021.10.02

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

메인 함수에 입력되는 argument를 user program의 kernel stack에 전달하는 기능과 user program에서 kernel space에 접근하지 못하도록 하는 기능, 그리고 user program에서 호출하는 시스템 콜을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

입력되는 argument들을 커널 스택에 쌓게되고 이를 esp라는 포인터가 가리키게 된다. 유저 프로그램의 메인함수에 argument를 전달 할 수 있게 한다.

1. User Memory Access

유저 프로그램에서는 user space만 접근이 가능하고 kernel space는 접근할 수 없게한다. 또한 page\_fault가 발생했을 때, 유저 프로그램에서 발생한 것이 아니라면 -1을 반환하게 된다.

1. System Calls

유저 프로그램에서 시스템 콜을 호출하여 커널에서 특정한 동작을 실행하고 반환값을 받을 수 있게된다.

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

esp가 가리키는 메모리의 주소보다 작은 주소에 argument를 쌓아야 하기 때문에, esp를 저장하려는 메모리의 크기만큼 빼주고, esp의 해당 메모리에 값을 저장한다. 구체적으로는 argument를 역순으로 쌓나서 esp가 가리키는 메모리의 주소가 word-aligned될 수 있도록(word의 크기는 4byte) 0을 필요한 만큼 저장한다. 그리고 4바이트 크기의 0을 저장한 후에, 위에서 저장한 arugment의 메모리를 주소의 주소를 역순으로 쌓는다. 마지막에 쌓인 argv[0]의 주소를 쌓고, argument의 개수를 4바이트에 저장한다. 그리고 4바이트의 0을 저장한다.

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

invalid memory access는 유저 프로그램에서 접근할 수 없는 메모리에 접근하려는 것을 의미한다. 예를 들어 NULL값에 접근하려하거나, 유저 프로그램에서 커널 스페이스의 주소에 접근하려는 것을 말한다. 이를 막기 위해서 is\_user\_vaddr를 통해서 system call 상에서 user virtual address가 아닌 메모리에 접근하는 지를 확인하고 아닌 경우 exit한다. 또한 page\_fault가 발생한 경우에는 원인이 유저가 아닌 경우이거나, 유저이지만 kernel 메모리에 접근하려는 경우 exit한다.

* 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

시스템콜은 유저프로그램에서 운영체제를 통해서만 수행할 수 있는 동작들을 수행하기 위해서 필요하다. 예를 들면 새로운 운영체제가 프로세스를 관리하고 새로운 프로세스를 띄울 수 있으므로, 하나의 프로세스 내에서 새롭게 프로세스를 실행하기 위해서는 운영체제를 통해야만 한다. 이를 위해서 시스템 콜이 존재한다.

이번 프로젝트에서는 halt(), exit(), exec(), wait(), read(), write(), Fibonacci(), max\_of\_four\_int()라는 시스템콜을 개발한다. halt()의 경우에는 운영체제가 pintos를 종료시키는 기능을 한다. exit()은 시스템콜을 호출한 프로세스를 종료하고 반환값을 커널에게 넘겨준다. exec()는 자식프로세스를 생성해서 실행하고, 성공적으로 생성된 경우에는 자식프로세스의 id를. 아닌 경우에는 -1을 반환한다. wat() 시스템콜은 넘겨받은 프로세스id에 해당하는 프로세스가 종료될 때까지 기다리고 종료값을 반환한다. 이번 프로젝트에서 read()는 stdin을 통해서 입력된 값을 읽고, 그 크기를 반환한다. stdin이 아닌 경우에는 -1을 반환한다. write()는 stdout에 값을 입력해주고 입력한 크기를 반환하는데, 전달받은 fd가 stdout이 아닌 경우에는 -1을 반환한다. Fibonacci()는 입력받은 항의 피보나치 수열값을 계산해서 반환한다. max\_of\_four\_int()는 입력받은 4가지 수에서 가장 큰 값을 반환한다.

유저프로그램에서 시스템콜 api를 호출하면 해당하는 argument들을 stack에 push하고나서 interrupt를 발생시킨다. 이는 매크로로 만들어진 syscallN에서 수행한다. 이는 intr\_entry로 들어가서 여러가지 유저프로그램의 레지스터들을 저장한 후에, intr\_handler를 호출한다. intr\_handler는 syscall\_handler를 호출한다. 이때 argument로 intr\_frame이 입력되는데, 이를 이용하여 유저프로그램에서 push했던 argument들에 접근할 수 있고, 값을 반환해줄 수 있다. intr\_frame구조체인 f의 f->esp를 이용하여 커널에서 동작을 수행한뒤 반환값이 있다면 이를 f->eax에 저장한다. 이후 syscall\_handler와 intr\_handler, intr\_entry가 차례로 종료된다. syscallN으로 돌아와서 esp를 원래 위치로 돌려놓고 반환값을 반환한다. 이를 시스템콜 api에서 반환해주면 유저프로그램에서 값을 받게된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

9.12 ~ 9.15 argument passing 구현

9.16 ~ 9.17 user memory access 구현

9.18 ~ 9.20 system call handler 구현

9.21 ~ 9.29 system call implementation 구현

9.30 ~ 10.01 additional implementation 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)**

argument passing을 구현하기 위해서 userprog/process.c 의 load함수에 argument를 parsing하는 부분을 추가하고, esp가 가리키는 메모리에 argument를 넣는 함수인 push\_argument를 구현한다.

user memory access를 구현하기 위해서 userprog/exception.c 의 page\_fault함수에서 page\_fault의 원인을 파악하고 이를 처리해 주는 부분을 추가한다. 또한 userprog/syscall.c 에서 syscall\_handler함수에서 유저프로그램에서 esp가 가리키는 스택에 입력된 주소가 올바른지 확인한다. 이를 위해서 is\_user\_vaddr()를 사용한다.

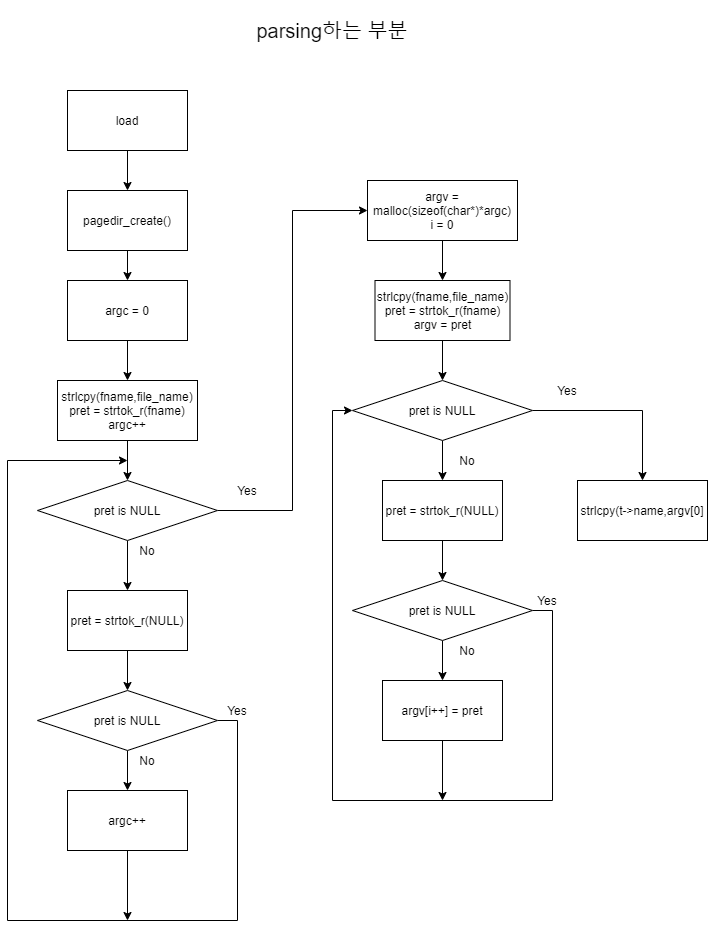
system call을 구현하기 위해서 userprog/syscall.c 의 syscall\_handler함수에서 argument로 입력되는 struct intr\_frame \*f를 통해서 어떤 시스템콜인지를 파악하고 이에 맞는 처리를 구현한다. exit()을 처리하기 위해서 kernel\_exit이라는 함수를 구현한다. exec()를 위해서는 userprog/process.c의 process\_execute함수를 사용하고, wait()를 위해서는 process\_wait함수를 사용한다. process\_wait에서는 실행되는 프로세스가 종료되기를 기다리는 부분을 추가한다.

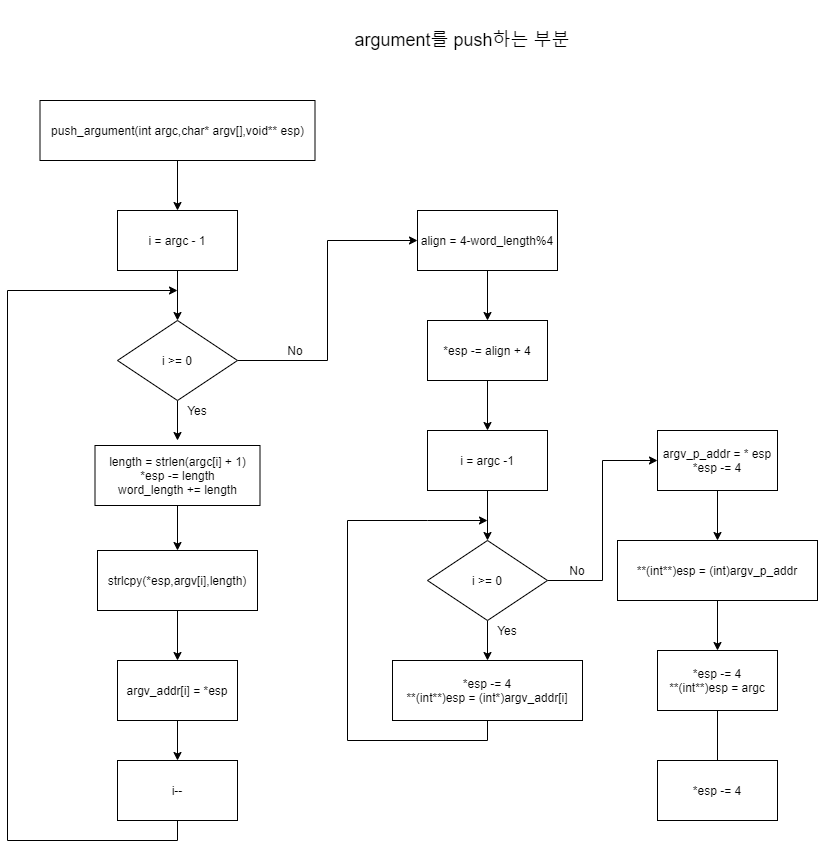
struct thread 구조체에 종료 값 저장하기 위해 exit\_number를 선언하고, 자식 프로세스를 저장하기 위해 struct list, struct list\_elem를 선언한다. 또 부모 프로세스를 저장하기 위해서 struct list, struct list\_elem를 선언한다.

1. **연구 결과**
   1. **Flow Chart**

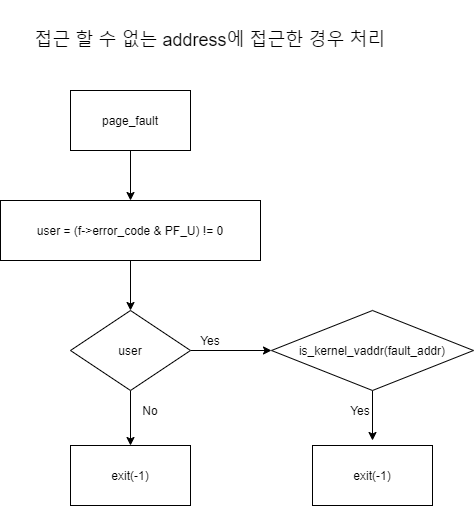
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

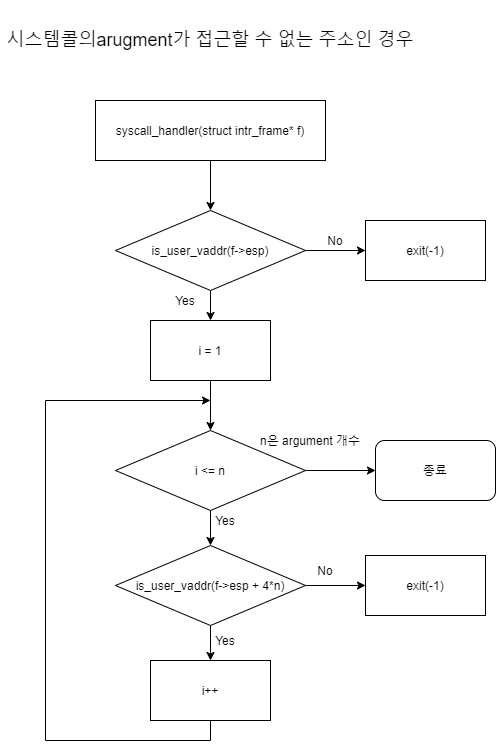
1. Argument Passing



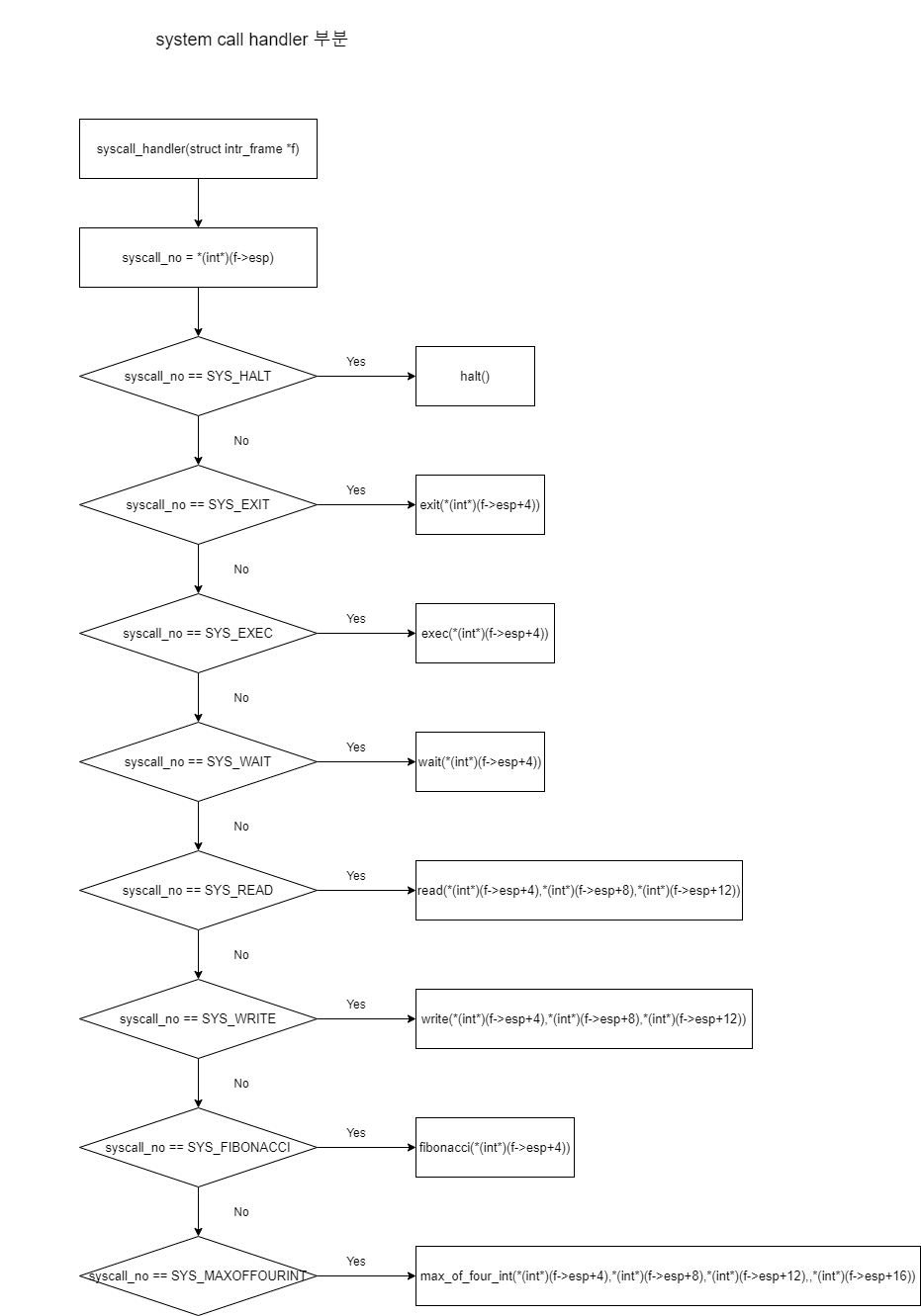


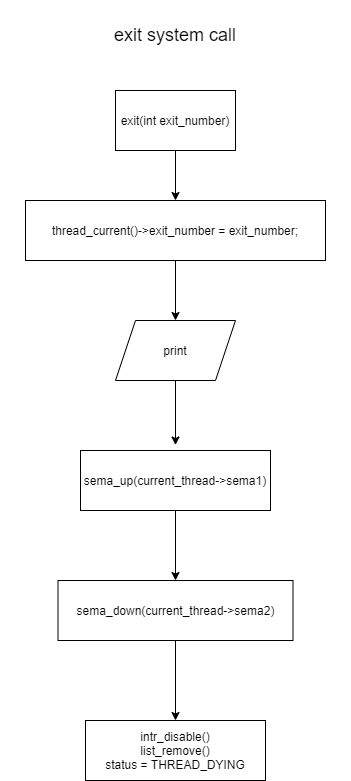
1. User Memory Access

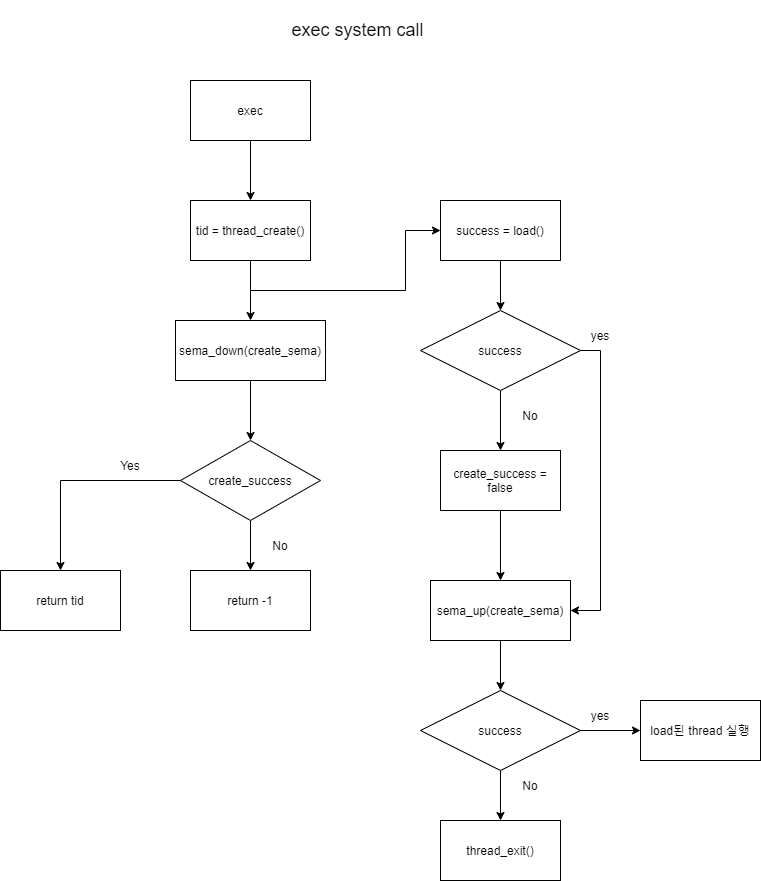


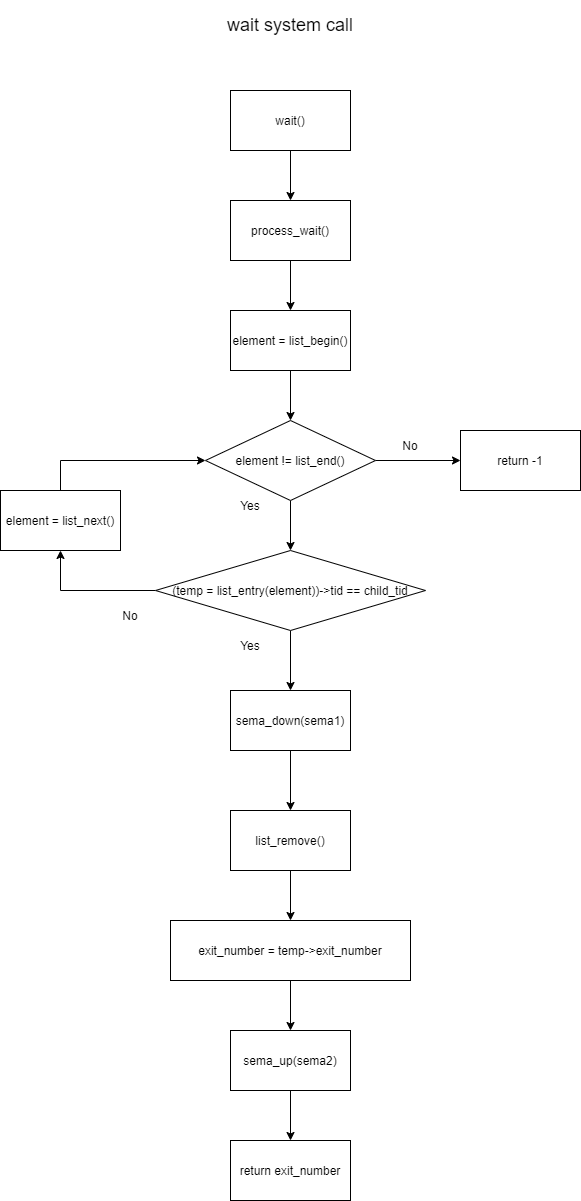


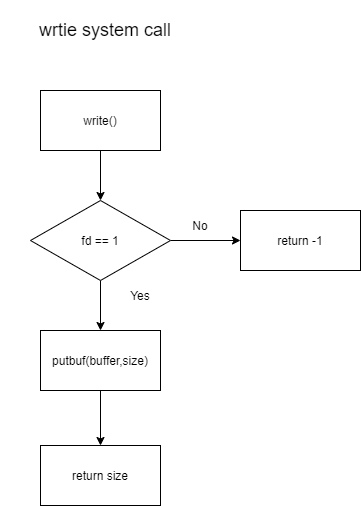
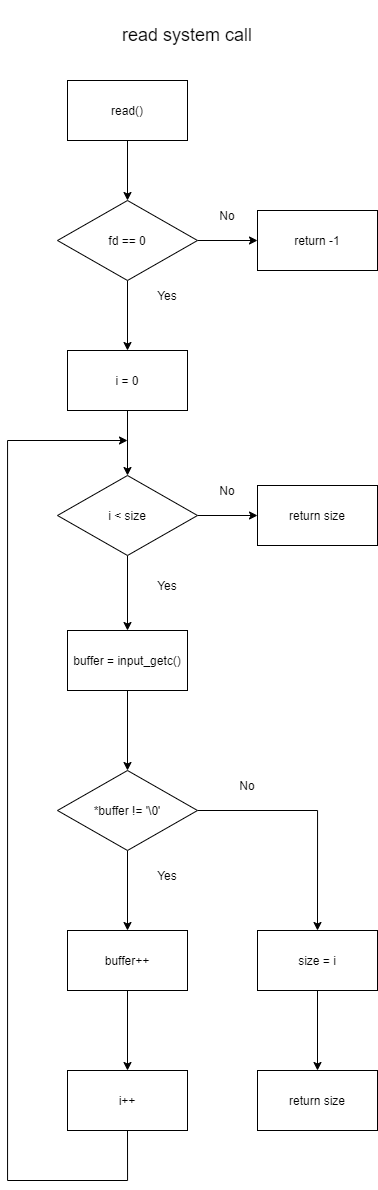
1. System Calls





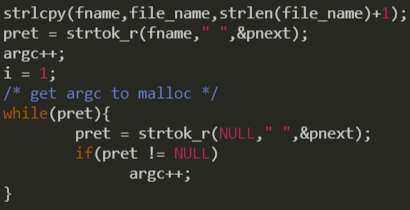




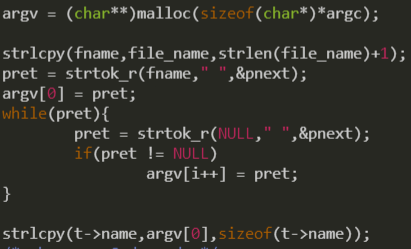


* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

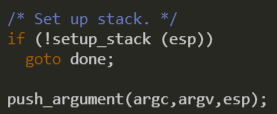
1. Argument Passing



userprog/process.c 의 load()함수에서 filesys\_open을 실행하기 전에 file\_name에 대해서 파싱을 해야한다. 파싱한 문자열을 저장할 배열이 필요하고 이의 크기를 결정하기 위해서 먼저 strtok\_r를 통해서 argc의 값을 구한다. file\_name문자열을 fname에 복사해두고 strtok\_r의 반환값이 NULL이 될때까지 반복하여 argc를 구할 수 있다. 이렇게 구한 argc를 통해서 argv에 malloc으로 메모리를 할당한다.

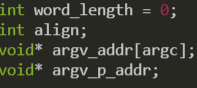


file\_name을 다시 fname에 카피하고 위에서 한 것과 같은 과정을 반복한다. strtok\_r의 결과를 argv에 순서대로 저장한다. 마지막으로 생성된 thread의 name에 file\_name의 가장 첫번째 부분을 저장한다. 그 후 filesys\_open을 통해서 파일을 오픈하게된다.

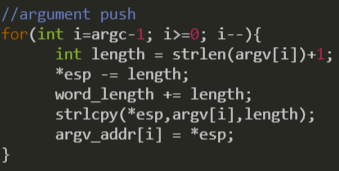


stack을 셋업하고 나서 push\_argument라는 자체 제작한 함수를 이용하여 esp가 가리키는 메모리에 파싱했던 문자열을 넣는다.

push\_argument함수는 다음과 같이 구성된다.



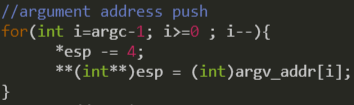
우선 위와 같이 변수를 선언한다.



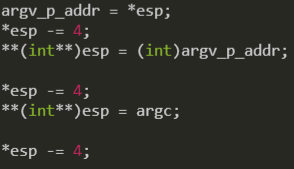
그 후 가장 마지막 argument부터 스택에 쌓게 되는데 낮은 주소로 쌓이기 때문에 argument 문자열의 크기만큼 esp값을 빼주고 esp에 문자열을 복사한다. argv\_addr은 이후 각 argv가 저장된 스택상의 주소를 스택에 집어넣기 위한 용도이고, word\_legnth는 word\_align을 계산하기 위한 용도이다.



주소가 4로 나누어 떨어져야하므로 집어넣었던 argument 문자열의 총 길이를 4로 나누어서 그 값을 4에서 뺀 만큼 0을 집어넣는다. 그리고 4바이트의 0을 집어넣어야 하므로 esp에 align값과 4를 빼준다.

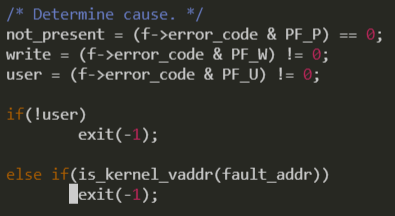


집어넣은 argv의 주소를 위에서 저장했던 argv\_addr배열을 이용하여 esp에 집어넣는다.



그리고 나서 argv[0]의 주소를 저장하는 부분을 다시 스택에 집어넣고, argc값을 4바이트 크기로 집어넣는다. 마지막으로 0을 4바이트 만큼 집어넣기 위해 esp에서 4를 빼주게된다.

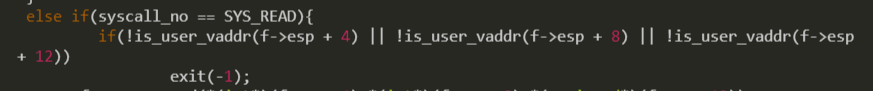
1. User Memory Access



userprog/exception.c 에서 page\_fault()함수에 위와 같이 page\_fault가 발생한 원인을 찾으면 그 원인이 user인지를 확인한다. user가 아닌 경우에 exit 시스템콜을 이용해서 -1을 넘겨주고, user인 경우에는 user가 커널 주소에 접근했는지를 is\_kernel\_vaddr를 이용해서 확인한다. 그런 경우에는 -1를 exit을 사용하여 넘겨준다.



userprog/syscall.c 의 syscall\_handler에서 위와 같이 유저가 esp를 통해서 넘겨준 주소가 user address가 맞는지를 확인하고 아닌경우에 -1를 exit에 넘겨준다. 각 시스템콜의 argument 개수에 따라서 위와 같은 작업을 추가로 진행한다. 예를 들어 n개의 argument가 esp에 쌓여있다면, f->esp + 4\*n까지 is\_user\_vaddr를 통해 확인해야한다.

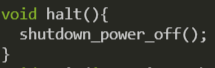


1. System Calls

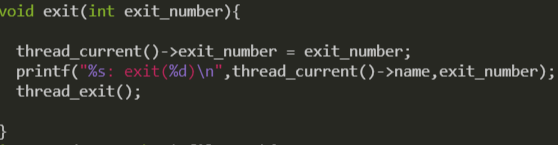
* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**



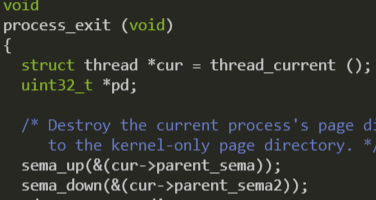
userprog/syscall.c 의 syscall\_handler함수에서 syscall\_no 변수에 \*(int\*)(f->esp)값을 저장한 후에 이 값에 따라서 시스템콜을 호출한다. 시스템콜에 입력되는 argument는 유저 프로그램에서 esp에 쌓여서 전달되었기 때문에 n번째 argument에 접근하기 우해서는 위와같이 f->esp+4\*n을 이용한다. 반환값이 있는 경우에는 f->eax에 저장한다.



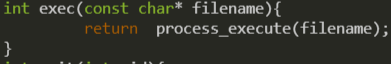
우선 halt()는 위와 같이 shutdown\_power\_off()를 호출하기만 하면 된다.



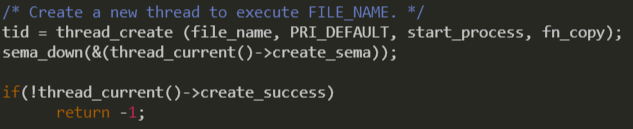
exit은 위와 같이 입력받은 값을 현재 thread의 exit\_number라는 struct thread 구조체에 새로 추가한 변수에 저장한다. 이는 wait함수에서 종료된 자식 프로세스의 상태를 전달받을 때 사용된다. 그 후 종료상태를 출력하고 thread\_exit함수를 사용하여 종료한다. thread\_exit함수에서는 process\_exit함수가 실행되는데, 여기서는 현재 thread의 parent\_sema라는 세마포어를 sema\_up해주게 된다.



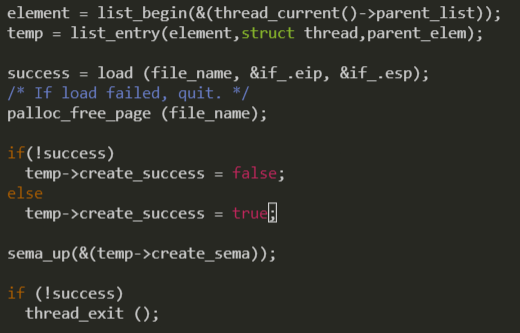
위와 같이 sema\_up을 해주면 wait에서는 sema\_down되어있기 때문에 다음 흐름으로 진행된다. 한편 sema\_up을 해주고 나서 바로 parent\_sema2를 이용해서 sema\_down을 해준다. 이는 wait에서 종료 상태를 저장하는 등의 과정과 synchronization하기 위함이다.



exec에서는 userprog/process.c에 존재하는 process\_execute함수를 실행한다.



process\_execute함수에서 thread\_create를 실행하면 thread가 새로 생성되고 그 thread의 id가 반환되는데, 제대로 load되는지와 상관없이 항상 thread의 id를 반환하게 된다. 따라서 thread가 제대로 로드되었지를 확인하기 위해서 sema\_down을 통해서 동기화한다. 이때 semaphore로는 thread 구조체에 새로 추가한 create\_sema를 이용한다. 제대로 로드가 되었는 지 여부를 thread구조체에 추가한 create\_success 변수를 이용해서 확인하고, 제대로 로드되지 않은 경우 -1을 반환한다.



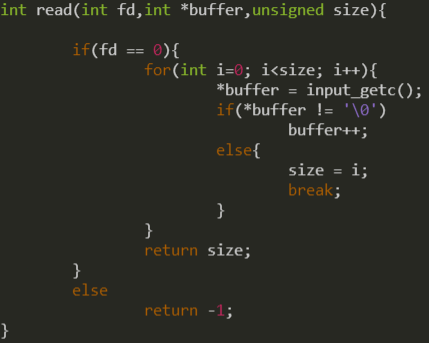
새로 생성된 프로세스에서는 start\_process함수가 실행되는데 여기서 위의 process\_execute와 동기화 해야한다. 이를 위해서 부모 프로세스를 확인하기 위해서 parent\_list와 parent\_elem를 thread구조체에 추가했다. 자식프로세스의 parent\_list의를 이용해서 부모프로세스를 찾고 load가 성공하지 않은 경우에 create\_success값을 false로 성공한 경우에는 true로 설정하고 sema\_up을 한다. 이를 통해서 process\_execute에서는 다시 흐름이 시작된다. 그 후 로드가 success하지 않은 경우에는 thread를 종료한다.



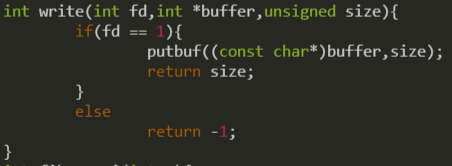
wait 시스템콜에서는 process\_wait() 함수를 실행한다.



process\_wait함수에서는 thread 구조체에 추가한 child\_list와 child\_elem을 이용해서 입력받은 thread id와 일치하는 자식프로세스가 있는 지 확인한다. thread\_current()의 child\_list의 처음부터 이동하며 if문을 통해서 child\_tid와 비교한다. 없는 경우에는 -1을 반환하고, 있는 경우에는 sema\_down을 통해서 block한다. 이는 자식프로세스에서 exit함수를 사용하는 경우에 다시 running할 수 있게 된다. 그 후 리스트에서 자식프로세스를 제거하고, exit\_number라는 변수에 자식의 종료상태를 저장한다. 이는 자식이 종료되고 나면 자식 thread의 exit\_number 멤버변수에 접근할 수 없기 때문이다. 그 후 sema\_up을 해서 block된 자식이 exit함수를 마저 진행할 수 있게하고, exit\_number에 저장된 종료 상태를 반환한다.



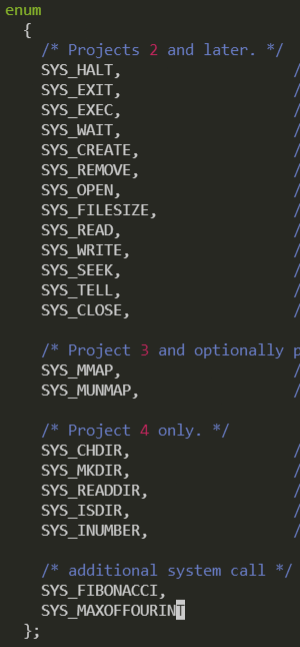
이번 프로젝트의 read()함수에서는 stdin에 대해서만 읽기 때문에 fd가 0이 아닌 경우에는 -1을 반환한다. 0인 경우에는 입력된 size만큼 input\_getc를 사용하여 바이트를 읽고 이를 buffer에 저장한다. buffer는 ‘\0’이 입력되기 전까지 값을 1씩 더해주며 다음에 저장할 위치를 가리킨다. ‘\0’이 입력된 경우 또는 size만큼 읽었을 경우에 총 입력된 사이즈를 반환한다.



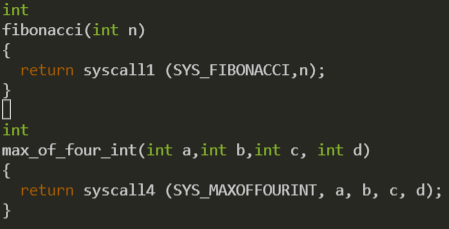
이번 프로젝트의 write()함수에서는 stdout에 대해서만 구현하기 때문에 fd가 1이 아닌 경우에는 -1을 반환한다. 1인 경우에는 putbuf()함수를 사용해서 buffer에 저장된 값을 size만큼 입력한다. 그리고 size를 반환한다.

1. Additional System calls

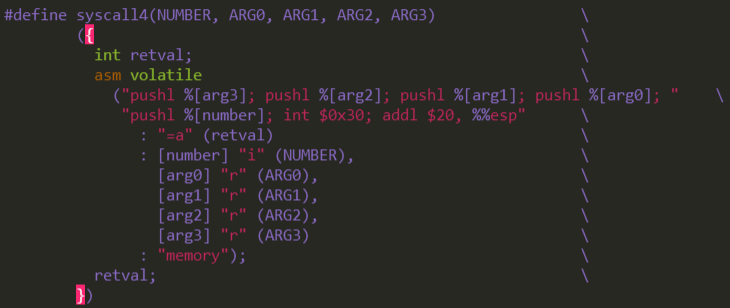
* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**



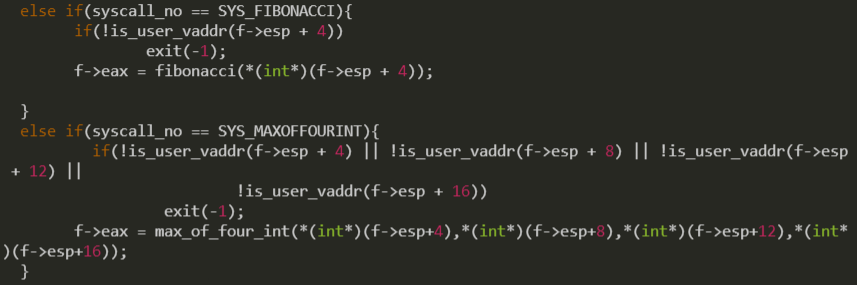
lib/syscall-nr.h에서 위와 같이 SYS\_FIBONACCI, SYS\_MAXOFFOURINT를 추가해준다. 이를 통해서 fibonacci와 max\_of\_four\_int를 시스템콜 넘버를 지정한다.



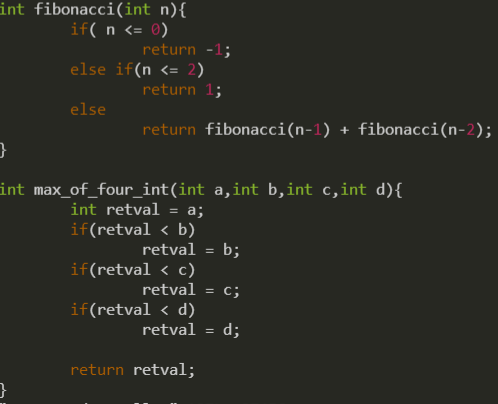
lib/user/syscall.c에서 시스템콜 api를 위와 같이 추가해준다. 이렇게 함으로써 유저 프로그램에서 Fibonacci()와 max\_of\_four\_int 시스템콜을 호출할 수 있게해준다.



4가지 arugment를 사용하는 시스템 콜을 사용하기 위해 lib/user/syscall.c에 위와 같은 매크로를 추가해 주었다. 마지막 argu3부터 argu0까지 4가지를 차례로 push한 후에 인터럽트를 발생시킨다.

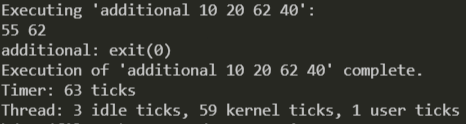


userprog/syscall.c의 syscall\_handler()함수에서 syscall\_no가 위에서 새로 추가한 SYS\_FIBONACCI 또는 SYS\_MAXOFFOURINT인 경우 fibonacci, max\_of\_four\_int라는 새로 추가한 시스템콜을 호출하도록 한다.



userprog/syscall.c에서 위와 같이 시스템콜을 추가한다. Fibonacci()는 피보나치 수열의 입력받은 수에 해당하는 항을 재귀적으로 반환한다. max\_of\_four\_int()는 retval에 a 값을 저장하고 이를 b와 비교해서 큰 값을 retval에 이를 c와 d에 대해서도 실행하여 최종적으로 a,b,c,d중 가장 큰 값을 retval에 저장하여 반환한다.

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

****