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

담당 교수 : 문의현

학번 / 이름 : 20191251/ 박효리

개발 기간 : 10/25~10/31

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

pintos에서는 system call, system call handler, argument passing, user stack이 모두 구현되어 있지 않다. 우리는 이번 프로젝트에서 이것들과 몇 가지 함수를 구현할 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

user program의 명령어를 parsing하여 stack에 알맞은 형식으로 저장한다. 이를 통해 로그램이 정상적으로 arguments를 넘겨 받게 된다.

1. User Memory Access

pintos는 user memory와 kernel memory로 memory를 나눈다. 이때 process는 kernel memory에 침범해선 안된다. 이를 위해 process가 invalid한 memory에 접근하려고 하면 process의 접근을 막는다.

1. System Calls

halt, exit, exec, wait , read(stdin), write(stdout)의 system call 함수들과 추가적으로 Fibonacci, max\_of\_four\_int 함수들을 구현하여 process를 처리할 수 있게끔 한다.

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

우선 argument를 단어별로 나누는, 즉 parsing하는 과정이 필요하다. parsing한 argument들을 argv 배열에 저장한다. 그 다음 setup\_stack함수를 통해 stack을 만들고 각 단어들+’\0’을 차례대로 저장한다. 그리고 word align number와 총 문자의 수가 4의 배수가 되게끔 word align number을 설정하여 그만큼 0을 stack에 쌓는다. 그 다음, boundary를 위해 4byte의 데이터를 0으로 하여 stack에 쌓고 단어들이 stack에 쌓인 주소를 다시 stack에 차례대로 쌓아준다. 그리고 argument의 개수와 return address도 쌓아준다.

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

pintos에서는 memory를 user memory와 kernel memory로 영역을 나눈다. 이때 process가 memory영역을 direct하게 접근하게 되면 여러 가지 문제가 발생하는데, process끼리의 memory영역을 침범하거나 더 큰 문제인 kernel code영역을 훼손시키는 등의 문제가 발생한다. 그리하여 os는 virtual memory를 통해 process가 이처럼 process가 자신만의 memory 영역을 갖게 하여 관리한다. virtual memory역시 user virtual memory와 kernel virtual memory로 구분이 된다. 그런데 이러한 상황에서 process가 wrong memory에 접근함에 있어 invalid 한 memory access가 발생할 수 있다.

* + Invalid memory access를 어떻게 막을 것인지 설명

pintos에서 제공하는 함수들을 사용하여 user provided pointer의 유효성을 확인한다. is\_user\_vaddr()와 is\_kernel\_vaddr()함수를 통해 이를 check하여 잘못된 memory access를 방지한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

앞서 언급했듯, pintos에서는 virtual memory를 두 영역으로 나눈다. 따라서 user program은 kernel memory영역에 접근할 수 없게 되는데, 여러 주요 기능들을 사용하기 위해서 system call이 중간다리의 역할을 해주게 된다.

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

syscall\_halt는 pintos를 shutdown\_power\_off함수를 사용하여 pinots를 종료시킨다. syscall\_exit은 현재 thread의 이름과 exit status를 형식에 맞게 출력하고 thread\_exit을 통해 thread를 중지시킨다. syscall\_exec는 cmd\_line을 입력받아 이에 따른 process\_execute함수를 호출한다. syscall\_wait은 자식 thread의 id를 입력 받아 이에 따른 process\_wait함수를 호출한다. syscall\_read는 입력받은 file descriptor값이 0이면, 즉 표준 입력인 경우 input\_getc함수를 사용해서 한 문자씩 입력받는다. syscall\_write은 file descriptor값이 1이면, 즉 표준 출력이면 입력 받은 buffer에 입력 받은 size만큼 putbuf함수를 사용해서 출력한다. syscall\_fibonacci는 정수 n을 입력 받아서 n번째 피보나치수를 반환한다. syscall\_max\_of\_four\_int는 4개의 정수를 입력 받아 이 중 가장 큰 정수를 반환한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

main함수에서 run\_actions(argv)를 호출한다. run\_action(argv)는 a->function(argv)를 호출하고 이는 다시 run\_task함수를 호출한다. run\_task함수는 다시 process\_wait(process\_excute(task))를 호출하고 이는 thread\_create함수를 호출하여 user program을 실행한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

10/25: pintos코드 및 과제 내용 이해하기

10/25~10/27: argument passing, system call, accessing memory 구현

10/28: additional system calls 구현

10/29~10/31: 디버깅 및 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)**

1) 먼저 argument passing을 위해서는 load함수에 명령어를 parsing하고 이를 stack에 쌓는 코드를 추가적으로 작성한다. load함수의 parameter인 file\_name이 명령어 문자열을 담고 있으므로 이를 parsing하여 배열에 저장하고, esp parameter를 사용해서 stack pointer에 접근하여 각 내용들을 stack에 쌓아준다.

2) user memory access는 pintos/src/userprog의 syscall.c과 process.c, 그리고 exception.c를 수정할 것이다. process.c에서는 process\_execute함수 내에서 filesys\_open함수가 NULL을 리턴하는지를 확인한다. syscall.c에서는 invalid memory access를 check하는 check\_valid\_addr함수를 만들어 syscall\_handler에서 사용하면서 invalid memory access의 경우 exit(-1)을 해준다. exception.c에서는 page\_fault함수에서 is\_kernel\_vaddr()을 사용하여 구현한다.

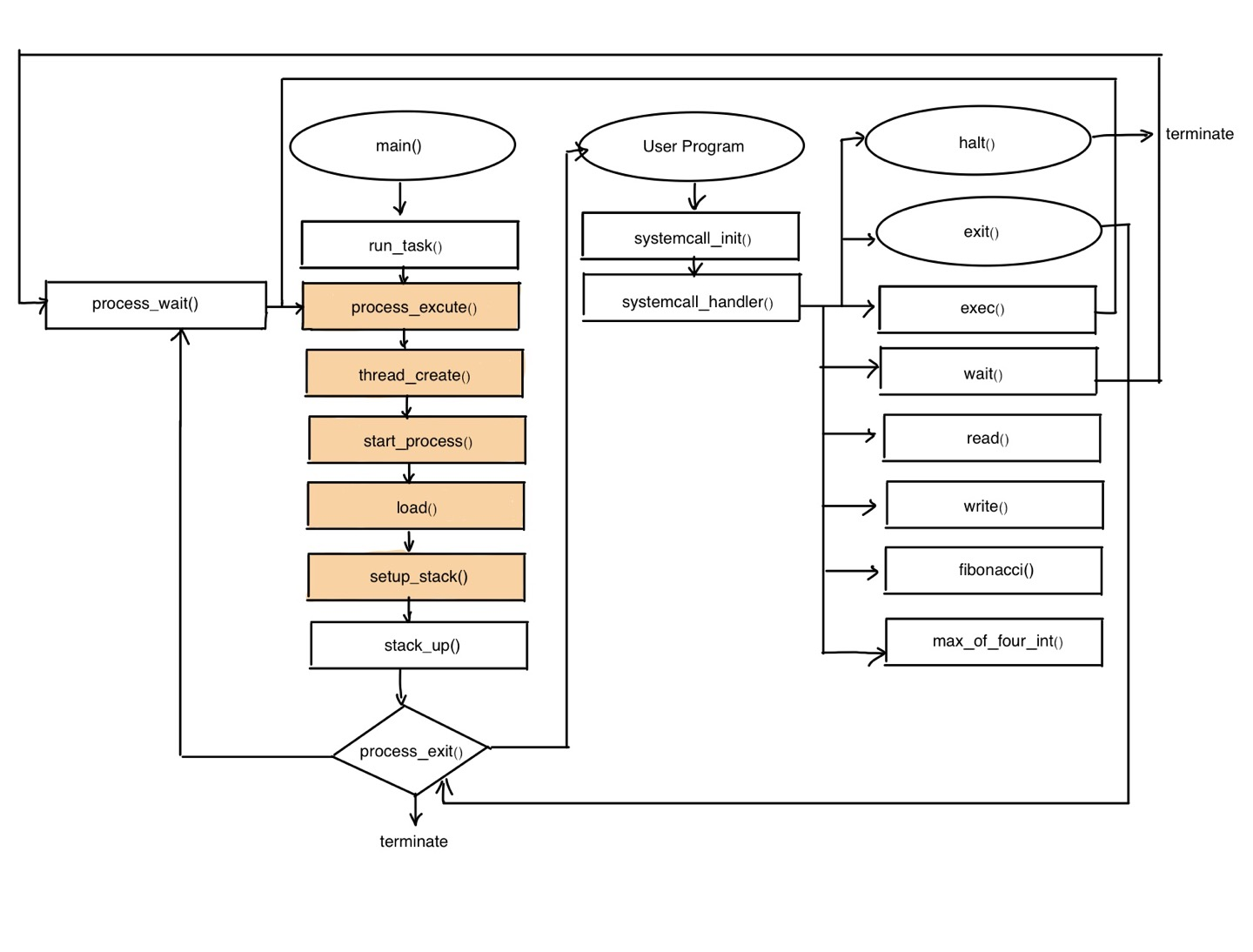
3) system call은 pintos/src/userprog/syscall.c와 syscall.h를 수정한다. syscall.c에서 각 syscall함수들을 구현하고, 이것들을 사용하여 syscall\_handler 함수 안에서 각 system call number에 맞는 동작을 수행하게 한다. 이때 각 명령의 결과값이 생기는 경우에는 이 값을 f->eax에 저장한다. 각 system call number에 대해서는 lib/syscall-nr.h를 참고한다.

4) 추가적으로 구현해야 하는 fibonacci와 max\_of\_four\_int를 위해 syscall.c에서 함수를 구현하고, pintos/src/examples에 additional.c를 만들어 Makefile에서 이 실행파일을 만들게끔 수정한다. 이 두 system call number를 추가하기 위해 syscall-nr.h도 수정한다. 뿐만 아니라 process\_wait도 제대로 구현되어 있지 않기 때문에 parent process가 child process의 exit status를 기다리는 것을 semaphore 구조체를 활용하여 구현한다.

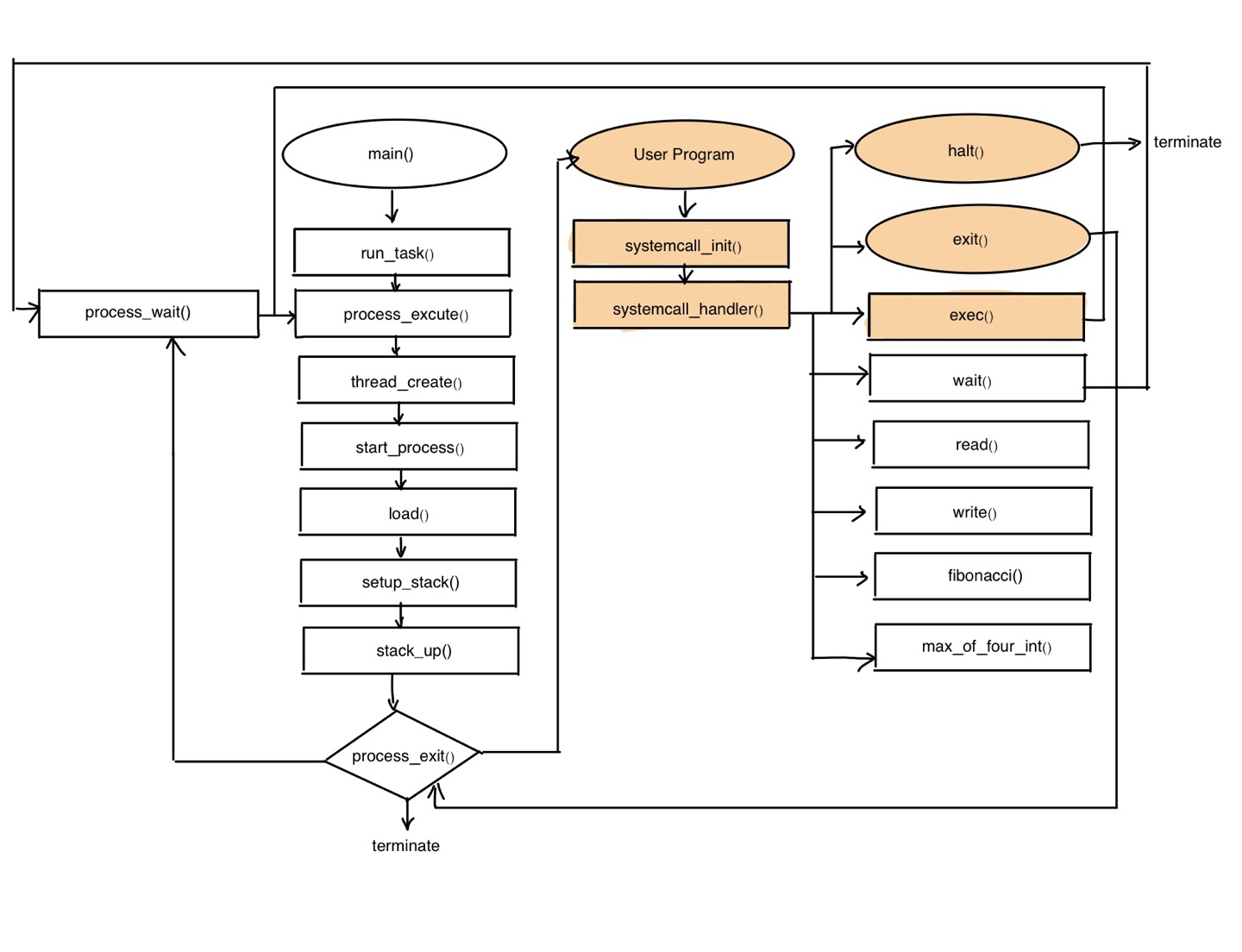
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

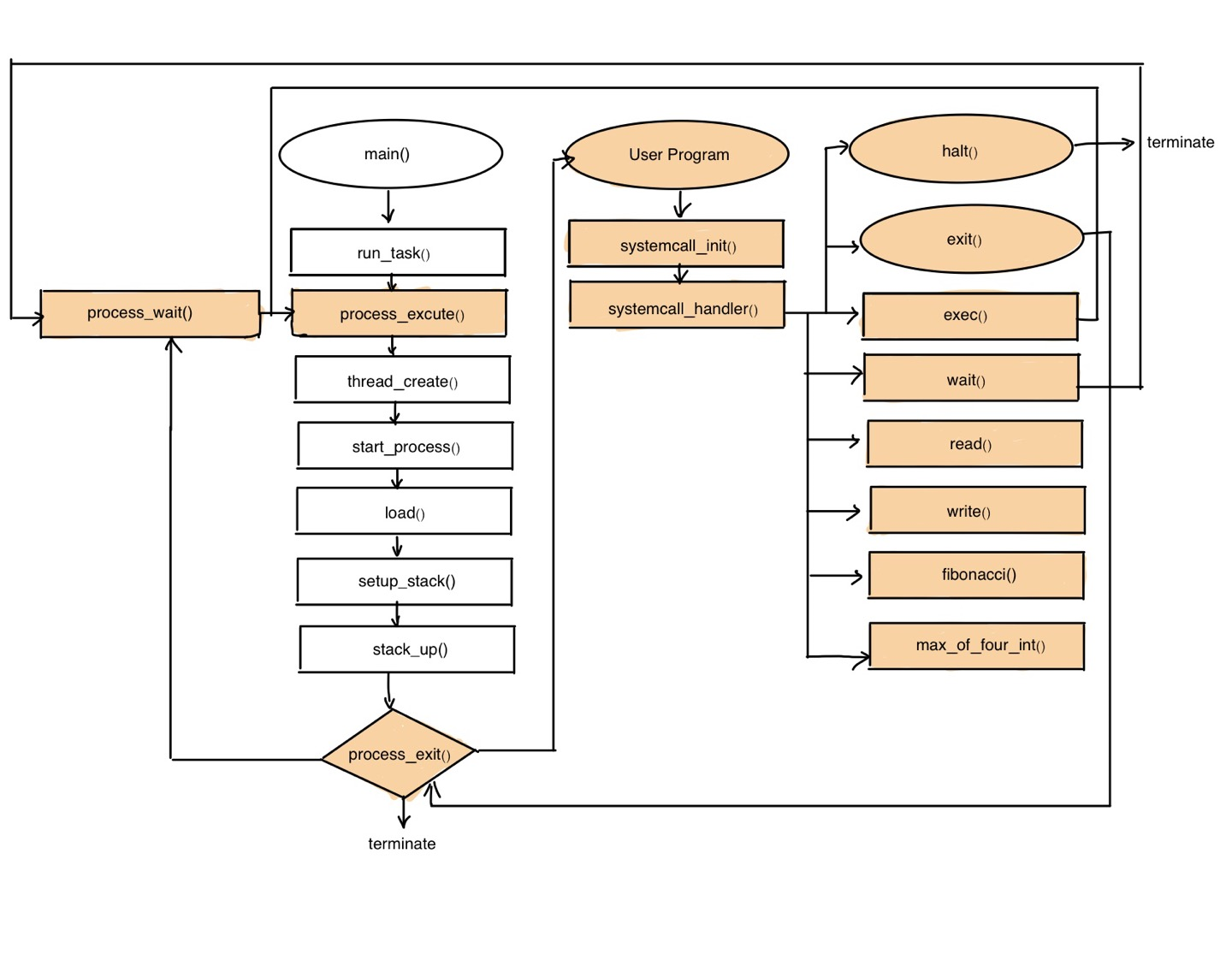
1. Argument Passing



1. User Memory Access

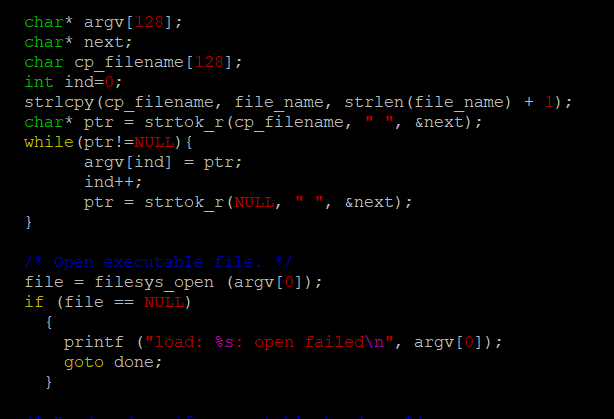


1. System Calls



* 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

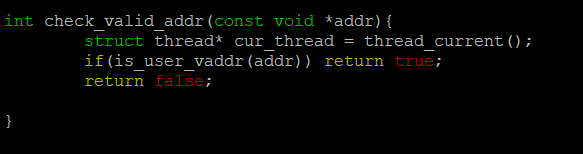


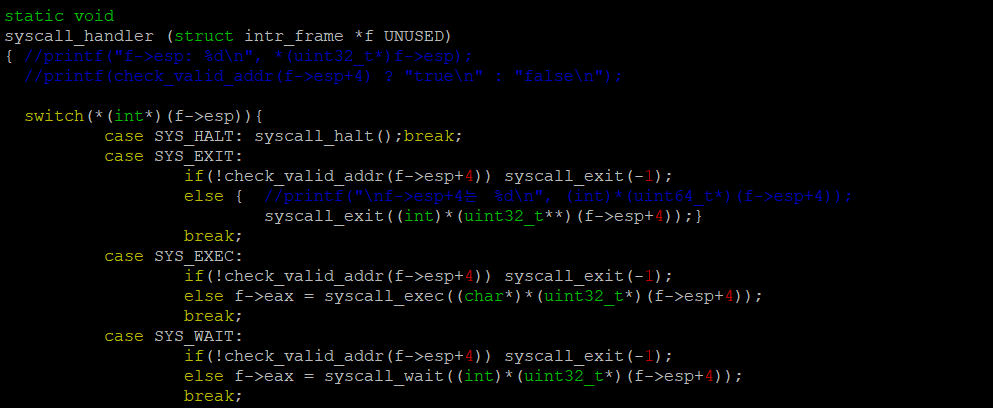
pintos/src/userprog/process.c의 load함수 내부에서 parsing을 구현한 부분이다. load함수의 parameter이며 명령어를 담고 있는 file\_name의 내용을 cp\_filename에 옮긴 다음, strtok\_r함수를 사용해서 각 단어를 나누어 argv배열에 저장하였다. 또한 parsing이 끝난 후 filesys\_open에 argv의 첫 번째 원소를 넘겨주어 파일을 열게끔 하였다.



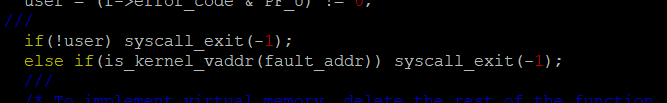
이 또한 load함수 내에서 구현한 코드로, stack을 set up한 이후 esp를 통해 데이터가 추가될 적절한 stack 주소를 찾아 데이터를 넣어주었다.

1. User Memory Access





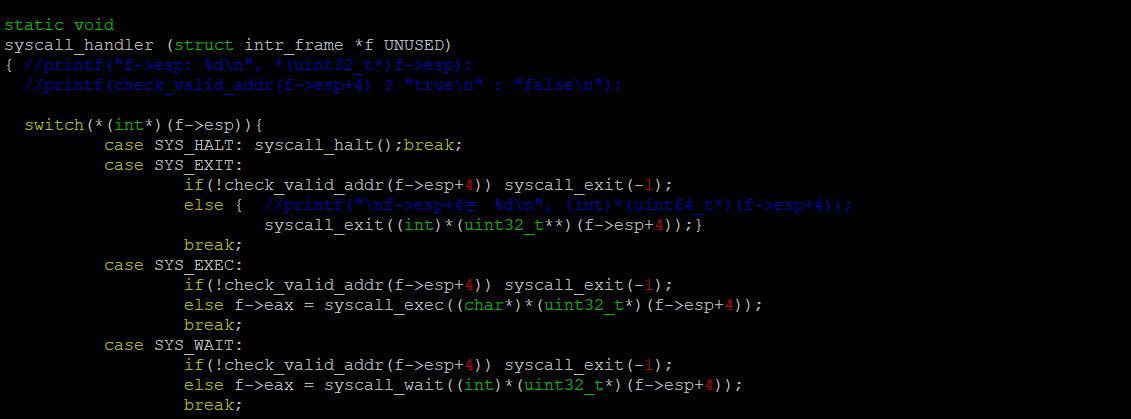
먼저 pintos/src/userprog/syscall.c에서 is\_user\_vaddr함수를 사용한 check\_valid\_addr함수를 구현해준다. 그리고 syscall\_handler내부에서 이 check\_valid\_addr이 false일 경우 syscall\_exit(-1)을, true일 경우 명령에 맞는 수행을 하게끔 코드를 작성하였다. 명령의 결과값이 생기는 경우 eax register에 저장하였다.

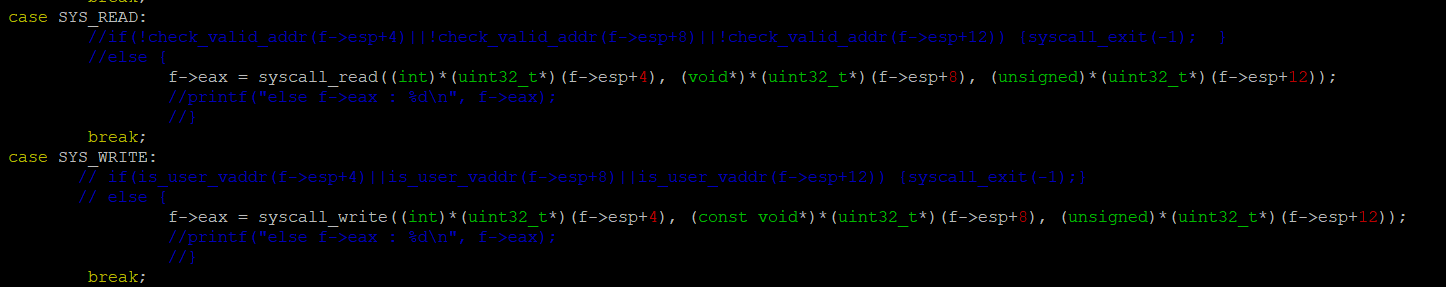


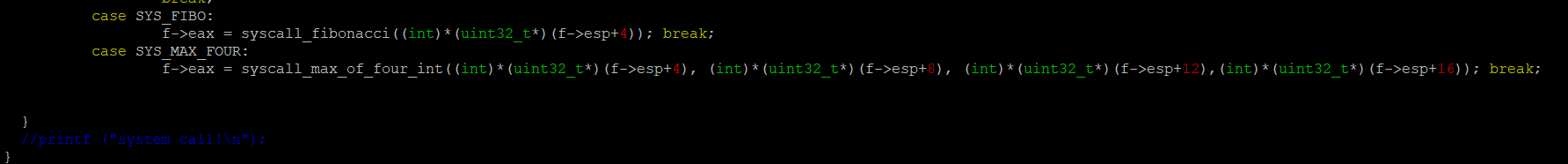
이 부분은 exception.c에서 page\_fault함수 내부에서 작성된 코드로 user==NULL일 때 syscall\_exit(-1)을 하거나 is\_kernel\_vaddr 함수를 사용해서 user memory access를 check하였다.

1. System Calls

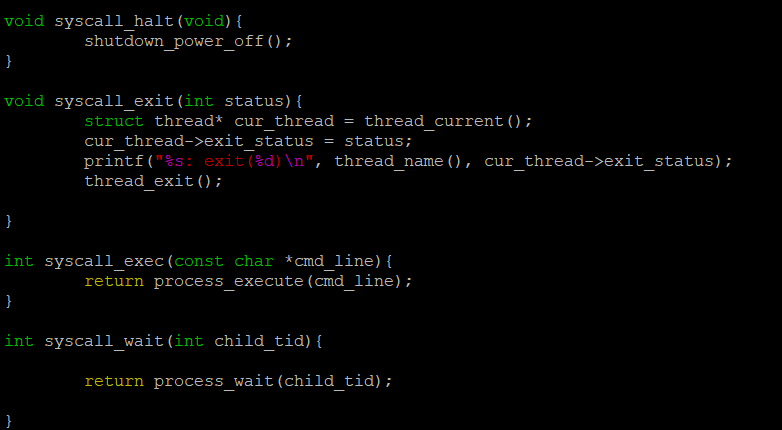
* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

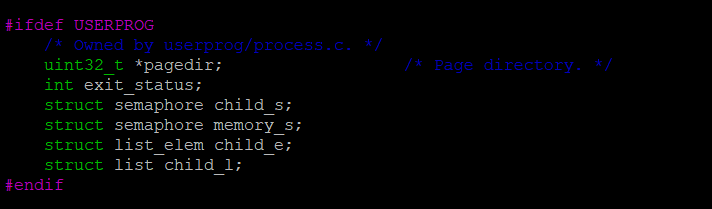


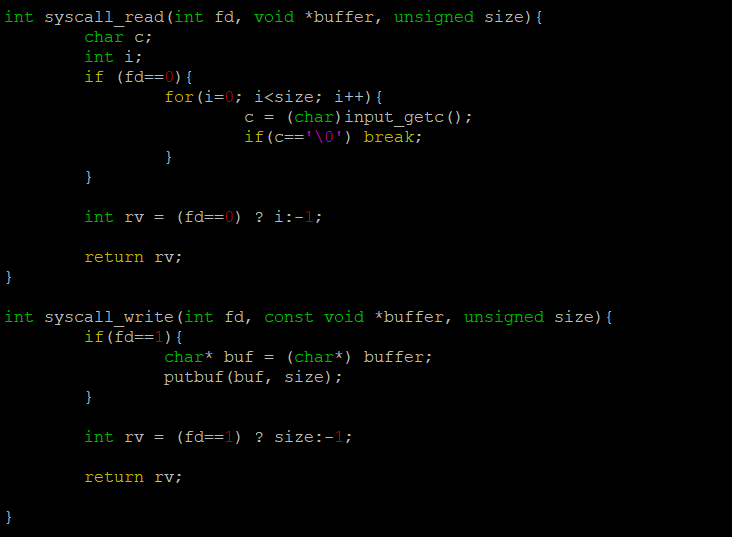




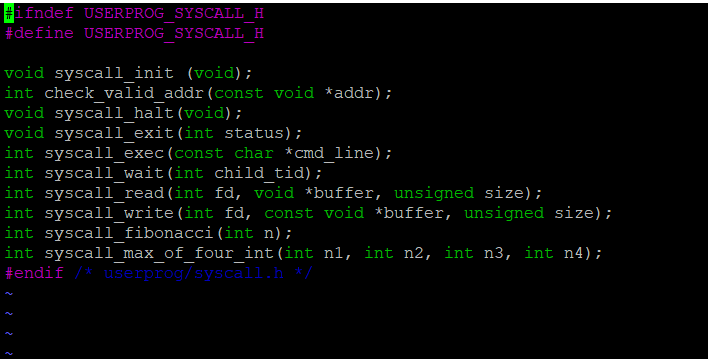
우선 syscall.c에서 syscall\_handler함수를 위와 같이 설계하였다. 앞서 언급했듯이 명령에 따른 수행을 하게끔 switch문을 사용하여 명령에 따른 함수를 호출하고 명령의 결과값은 eax에 저장하였다. 또 syscall.c에서 아래와 같이 각 함수들을 설계하였다.



syscall\_halt함수는 shutdown\_power\_off함수를 호출하여 pintos를 종료한다. syscall\_exit함수는 thread\_current함수를 통해 현재 thread를 불러오고, 이것의 name과 exit\_status를 출력하고 thread\_exit()로 마무리 하였다. 이를 위해 pintos/src/threads/thread.h에서 thread구조체에 exit\_status변수를 추가해주었다. 

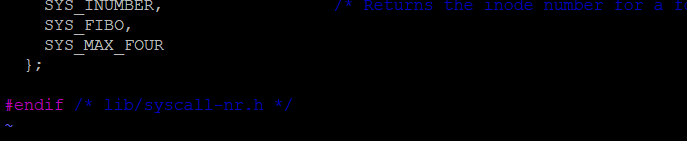


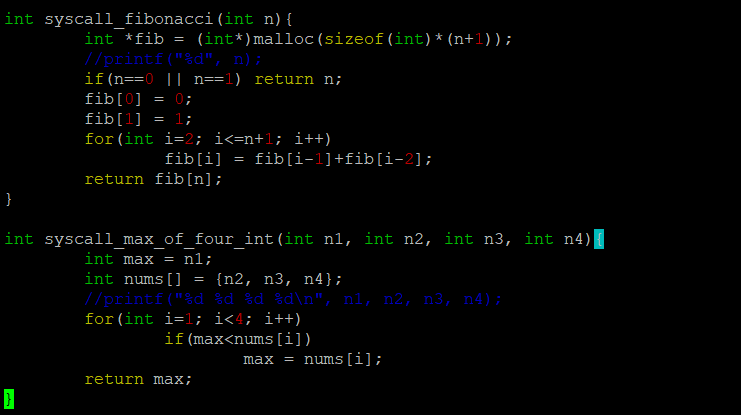
syscall\_read는 는 입력받은 file descriptor값이 0이면, 즉 표준 입력인 경우 input\_getc함수를 사용해서 한 문자씩 입력받는다. syscall\_write은 file descriptor값이 1이면, 즉 표준 출력이면 입력 받은 buffer에 입력 받은 size만큼 putbuf함수를 사용해서 출력한다.



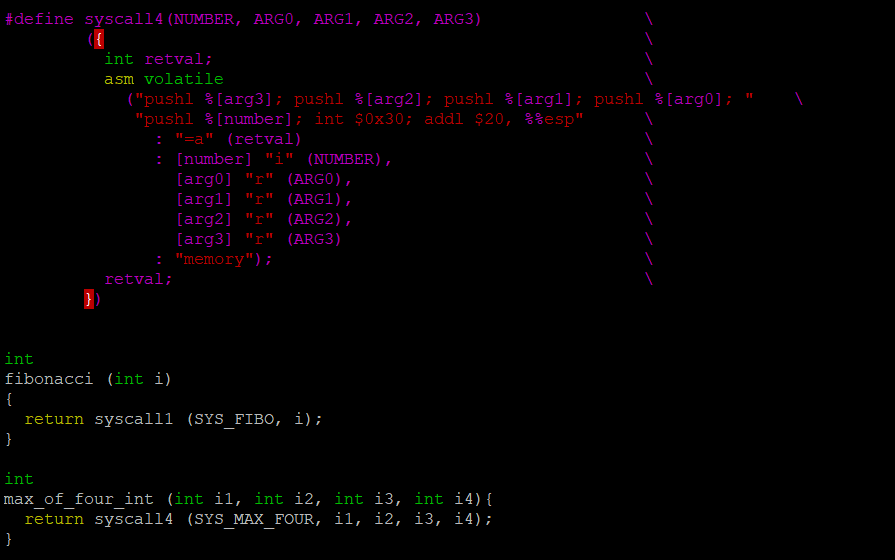
syscall.h에 syscall.c의 수정 사항을 반영해준다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
* 
* lib/syscall-nr.h에 SYS\_FIBO와 SYS\_MAX\_FOUR을 추가해준다.



syscall\_fibonacci는 정수 n을 입력 받아서 n번째 피보나치수를 반환한다. syscall\_max\_of\_four\_int는 4개의 정수를 입력 받아 이 중 가장 큰 정수를 반환한다.



src/lib/user/syscall.c에서 syscall4를 새롭게 정의해주고 fibonacci함수와 max\_of\_four\_int의 system call API를 정의해주었다.

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

