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

담당 교수 : 문의현 교수님

학번 / 이름 : 20190554 송재현

개발 기간 : 10/05 ~ 10/31

1. **개발 목표**

User program이 pintos에서 정상적으로 작동하도록 Argument Passing, User memory access, system call handler, system call implementation을 구현한다. Argument Passing은 argument를 parsing해서 memory에 allocate하는 것이고 User memory access는 user program이 invalid pointer를 pass하도록 한다. 또한 exit, halt, exec, wait, read, write과 더불어 fibonacci, max\_of\_four\_int의 system call을 추가적으로 구현하고 이를 handle하는 syscall\_handler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

argument를 입력 받아서 이를 parsing하여 80x86 calling invention에 따라서 stack에 push한다.

1. User Memory Access

User program이 Kenrel adreess에 접근하거나 null pointer에 접근할 경우 invalid pointer로 간주하여 pass한다.

1. System Calls

System call halt, exit, exec, wait, read, write의 기능을 구현한다. Read와 write은 각각 standard input, standard output에 대해서만 구현하도록 한다. 또한 additional system call로 fibonacci와 max\_of\_four\_int를 구현한다.

* 1. **개발 내용**
* Argument Passing

입력으로 들어온 command를 strtok\_r 함수를 이용하여 argument들을 나누고 argument의 개수인 argc를 구한다. 공백을 기준으로 나눈 argument들을 이용하여 80x86 calling convention을 바탕으로 구현한 push\_stack 함수를 통해서 stack에 push한다. stack에는 argument data, 그 후에 word alginment 그 후 null pointer를 push한다. 그 후 argument data들의 주소를 Push하고 argv, argc, return address를 push한다. 이 떄 esp 값은 PHYS\_BASE로부터 시작되는데 이를 이용하여 word\_alignment가 얼마가 필요한지를 계산하여 word\_alignment를 해준다.

* User Memory Access

memory는 pintos상에서 PHYS\_BASE를 기준으로 user memory 부분과 kernel memory 부분으로 나누어진다. 이 때 user program에서의 invalid memory access는 NULL pointer, Unmapped virtual memory, kernel memory에 접근하게 될 경우를 의미한다. 따라서 syscall handler에서 syscall을 호출하기 전에 is\_valid\_vaddr를 통해서 invalid memory access를 사전에 방지한다.

* System Calls

System call은 kernel memory에 접근하기 위해서 필요하다. User program만으로는 kernel memory에 접근할 수 없기 때문에 system call을 통해서 kernel memory에 접근하도록 해준다.

이번 프로젝트에서 개발할 시스템 콜은 다음과 같다.

halt: shutdown\_power\_off() 함수를 통해서 pintos를 종료한다.

exit: exit\_status를 인자로 받아서 현재 실행되고 있는 process 이름과 status를 출력한다. 이 때 status가 0이 아니라면 비정상적으로 exit 된 것이다. 또한 kernel에 이를 전달하여 user\_program을 종료한다. parent가 있다면 parent에게 exit\_status를 전달한다.

exec: 인자로 받은 command를 통해서 parse 한 후 파일명과 argument를 이용해 새로운 process를 실행한다. process가 제대로 실행될 경우 pid를 return한다. 이 때 새롭게 만들어진 process는 exec를 실행한 process의 child가 된다.

wait: pid를 인자로 받아 이 pid에 해당하는 child\_process를 찾아서 해당 process가 종료될 때까지 기다린다. 이 때 pid가 유효하다면 종료되는 것을 기다리지만 유효하지 않다면 -1을 return한다.

read, write : project 1에서는 각각 stdin, stdout을 통해서 입력과 출력을 하게된다. File descriptor, buffer, buf\_size를 입력 받아 read는 input\_getc()함수를 통해 stdin을 입력 받고 write은 putbuf()를 통해 stdout에 출력한다.

fibonacci: 정수 n을 인자로 받아 n 번째 피보나치 수열 값을 return한다.

max\_of\_four\_int : 정수 a,b,c,d를 인자로 받아 가장 큰 값을 return한다.

유저레벨에서 system call은 다음의 과정을 거쳐 수행된다.

- User level에서 system call api를 호출

- API에서 interrupt를 통해 system call number를 kernel에 전송

- sys call handler에서 이 number를 system call interface table로 전달

- system call interface table에서 해당 system call에 해당하는 주소로 이동

- system call 수행 후 user mode로 복귀

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

Argument Passing 구현 : 2022/10/10 ~ 2022/10/14

System call 구현 : 2022/10/15 ~ 2022/10/31

Additional system call 구현 : 2022/10/31 ~ 2022/10/31

document 작성 : 2022/10/31 ~ 2022/11/01

* 1. **개발 방법**

Argument Passing

pintos/src/userprog에 있는 process.c 파일에서 load함수를 수정한다. load함수에서 기존에는 file\_name으로 file을 여는데 그 전에 parsing을 통해서 real\_file\_name을 구하고 argument들을 따로 argv배열에 저장한다. 또한 argv의 수를 argc에 저장한다. 이후 setup\_stack함수가 호출 된 뒤 process.c에 새로 구현한 push\_stack()함수를 통해서 stack에 80x86 calling convention을 바탕으로 2-B에서 설명한 방식대로 stack에 push한다.

User Memory accessing

따로 함수를 구현하지는 않았으나 userprog에 있는 syscall.c에서 syscall\_handler()를 구현할 때 is\_valid\_vaddr()함수를 통해서 esp를 이용하여 stack에 접근한 후 인자들의 주소를 미리 검사하여 invalid한 주소라면 syscall\_exit(-1)을 호출하여 해당 process를 종료시킨다.

System call

우선 userprog에 있는 syscall.c에서 syscall\_handler()를 구현한다. esp를 통해 stack에 접근하여 각 system call에 필요한 정보들에 접근한다. esp를 통해 system call number를 확인하고 각 number에 해당하는 system call을 호출하기 전 선처리를 해준다. 각 system call에 필요한 인자들의 메모리 유효성을 검사하고 이들을 가져와서 system call을 수행해준다. 만약 return 값이 있는 system call이라면 eax에 return 값을 저장해준다.

halt: shutdown\_power\_off() 함수를 통해서 pintos를 종료한다.

exit: exit\_status를 인자로 받아서 현재 실행되고 있는 process 이름과 status를 출력한다. 이후 thread\_exit()를 호출한다. 이 때 semaphore를 통해서 child\_process가 종료됐다는 것을 알려준다. process\_exit()함수에서 semaphore를 up시켜주고 이를 process\_wait()함수에서 parent에 있는 child\_list에서 제거하게 되면 그 때 exit을 수행한다.

wait: pid를 인자로 받아 process\_wait() 함수를 호출한다. 현재 thread에서 child\_list 중에서 인자로 받은 pid를 갖는 child process를 찾고, 만약 해당하는 pid가 있으면 child가 exit될 때 까지 기다린다. 그렇지 않다면 -1을 반환한다. semaphore를 통해서 child가 죽은 것을 확인할 수 있다. child가 죽으면 Process\_wait()함수에서 얻은 status를 통해 child\_list에서 child를 제거하고 해당 child를 exit한다.

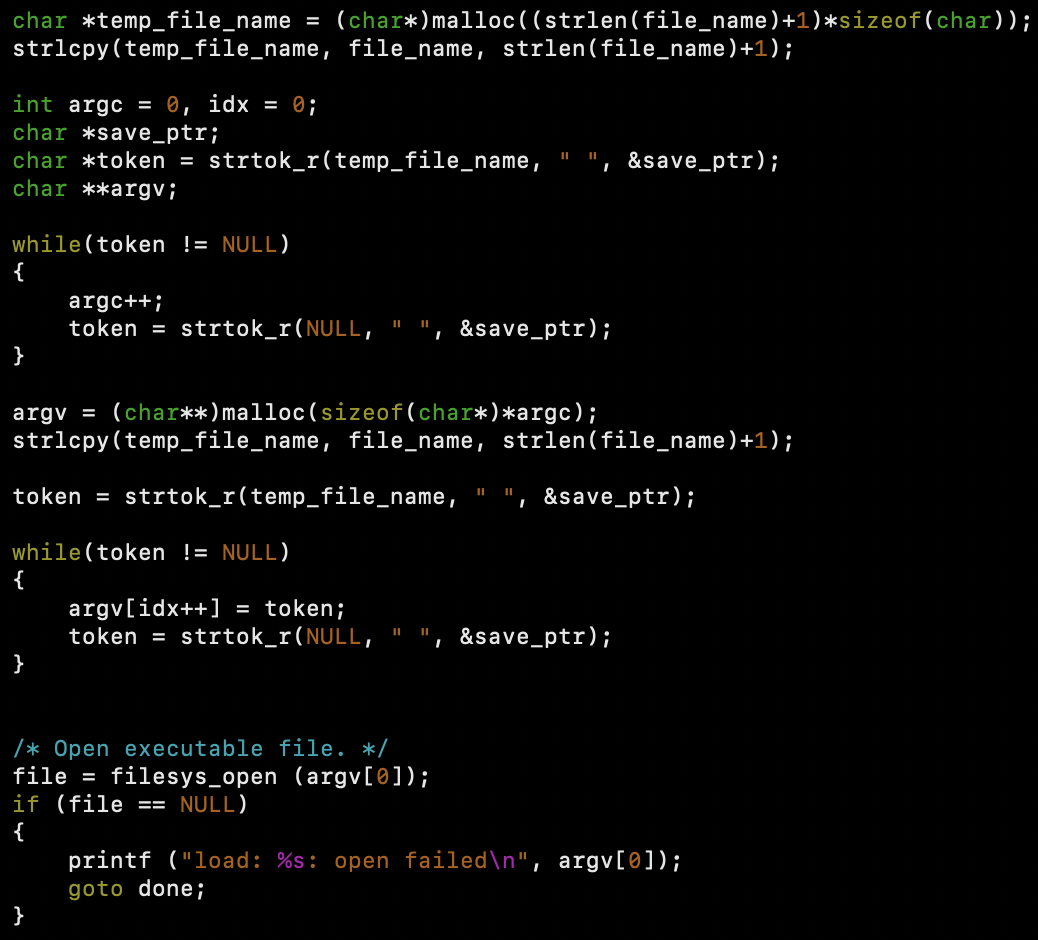
exec: 인자로 받은 command를 통해서 parse 한 후 파일명과 argument를 이용해 process\_execute()함수를 호출한다. process\_execute() 함수에서는 새로운 thread를 thread\_create()함수를 통해 만들게 되고 이 때 새롭게 만들어진 process는 exec를 실행한 process의 child가 된다. 따라서 현재 process의 child\_list에 새롭게 만들어진 process를 추가한다. Process가 정상적으로 만들어졌으면 만들어진 process의 pid를 return한다.

read, write : project 1에서는 각각 stdin, stdout을 통해서 입력과 출력을 하게된다. File descriptor, buffer, buf\_size를 입력 받아 read는 input\_getc()함수를 통해 stdin을 입력 받고 write은 putbuf()를 통해 stdout에 출력한다.

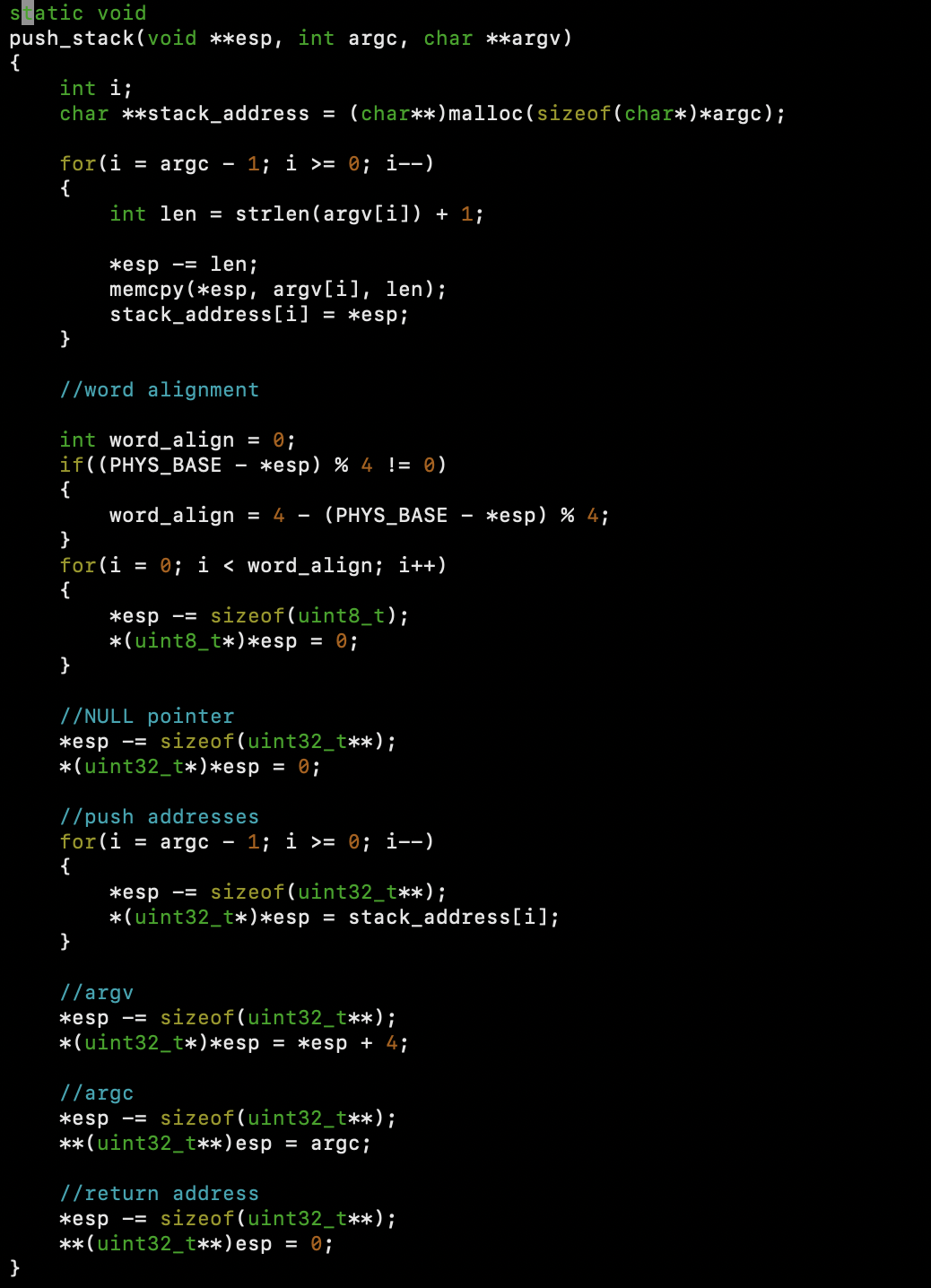
fibonacci: 정수 n을 인자로 받아 n 번째 피보나치 수열 값을 return한다.

max\_of\_four\_int : 정수 a,b,c,d를 인자로 받아 가장 큰 값을 return한다.

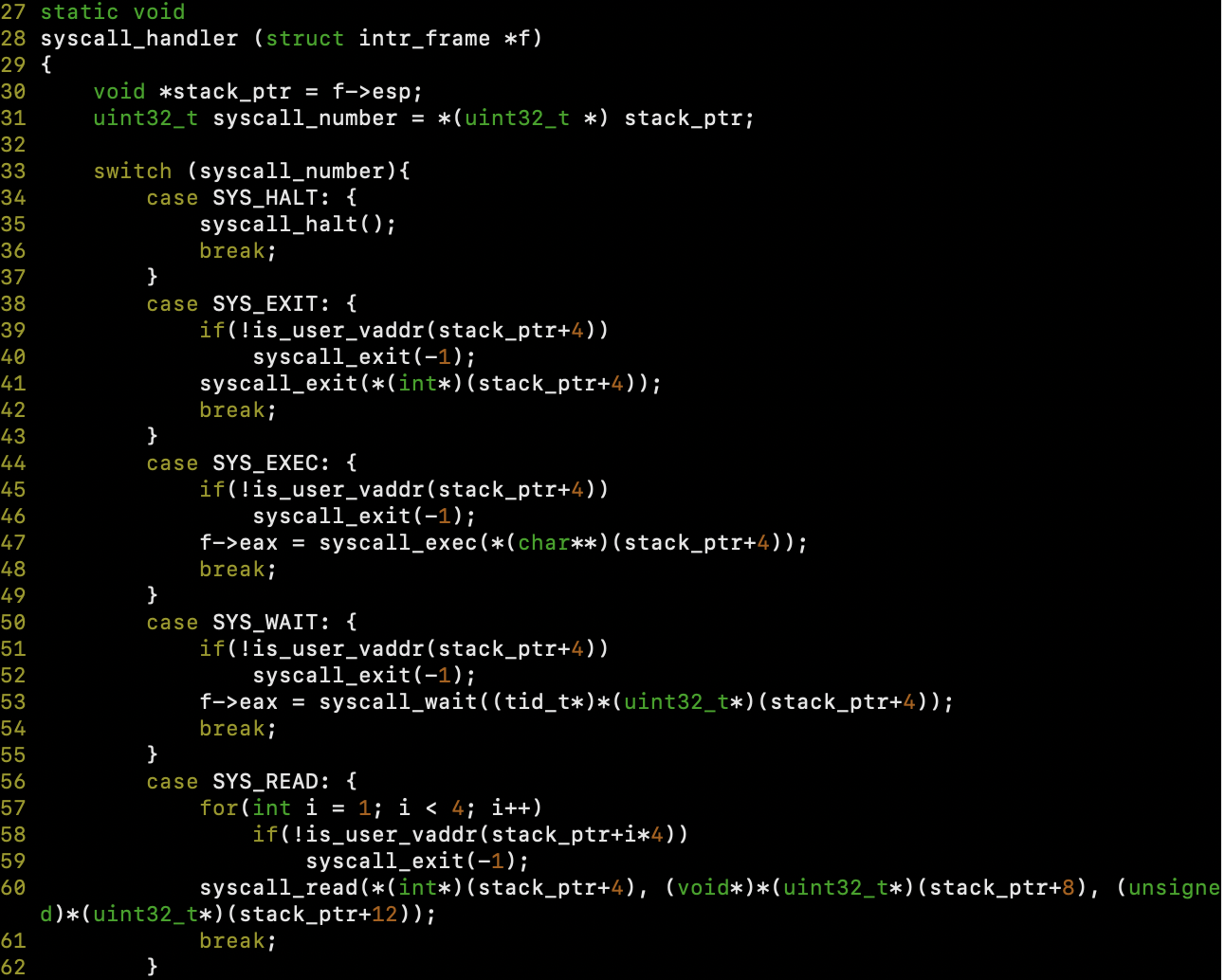
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing
3. User Memory Access
4. System Calls
   1. **제작 내용**

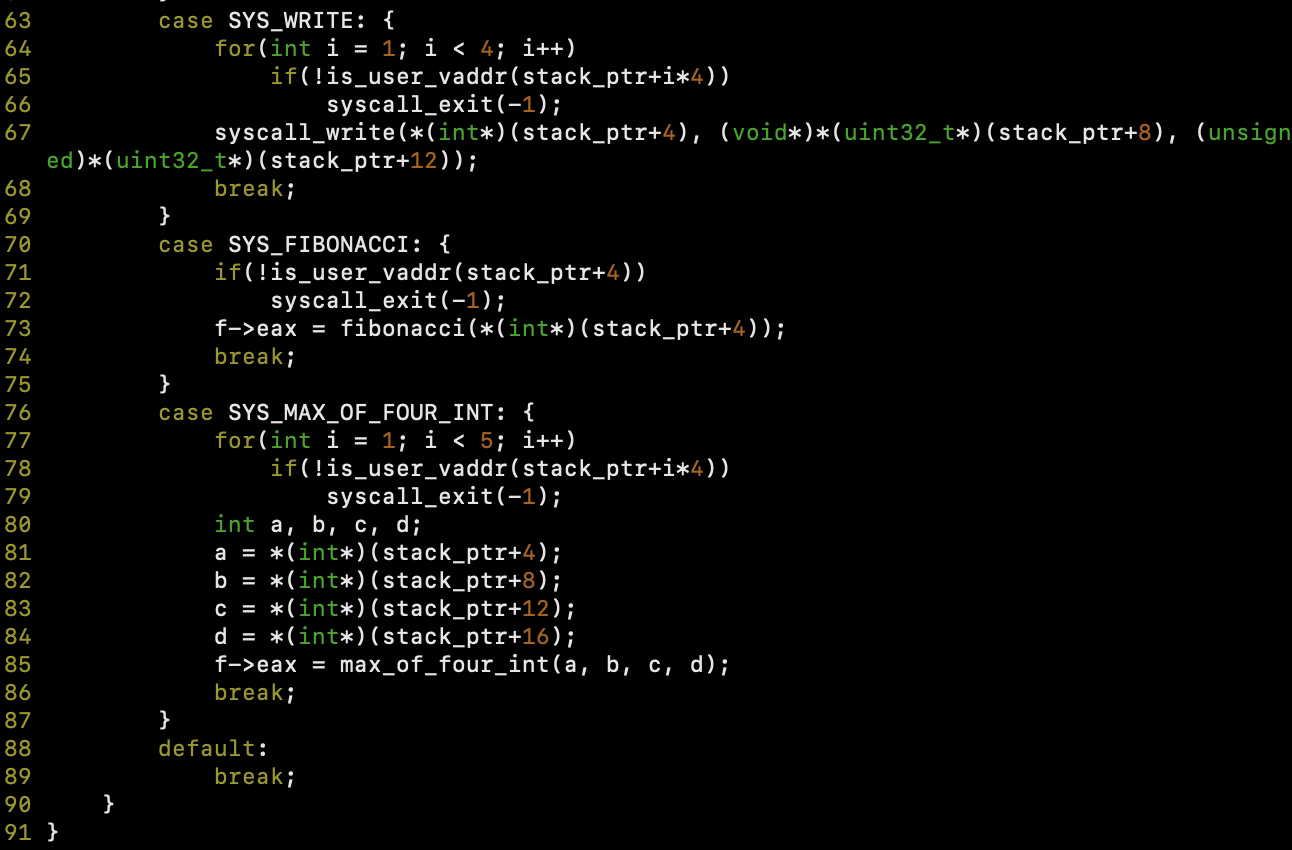
**Argument Passing**

process.c의 load 함수에서 주어진 command line을 parsing하였다. strtok함수가 아닌 thread safe한 strtok\_r함수를 사용하여 공백을 기준으로 Parsing을 진행하였다. 처음 parsing을 통해 argc를 구하고 이를 바탕으로 argv를 동적할당하여 argv에 해당 값들을 할당하였다. 이후 이를 이용하여 push\_stack함수를 호출해 stack을 구성하였다.

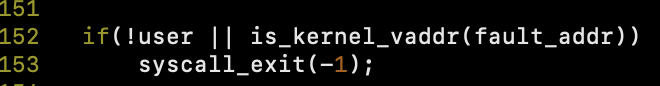


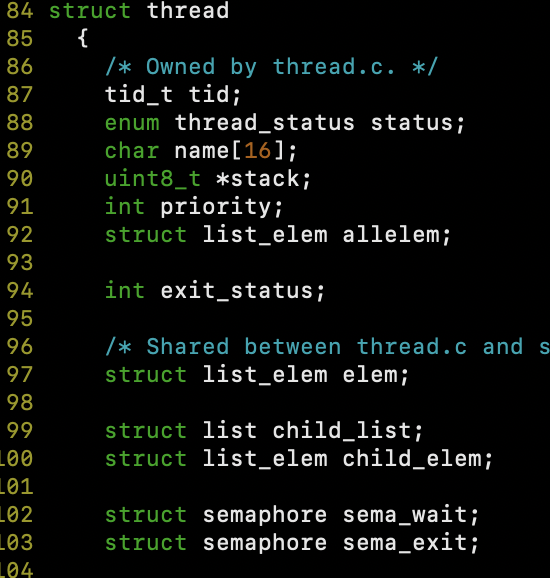
이후 setup\_stack 함수를 호출한 뒤에 push\_stack 함수를 호출하여 stack을 구성하였는데 stack\_address 배열을 argc만큼 동적할당하여 각 argv의 주소 값을 저장해주었다. 2-B에서 서술한대로 esp값을 감소 시킨 후에 필요한 값들을 memcpy함수를 통해 저장해주었다. argv를 모두 저장 후에 word\_alignment를 PHYSBASE와 현재 esp 값을 바탕으로 해주었고 이후 NULL pointer를 저장한 후 stack\_address에 있는 argv의 주소값들을 저장해주었다. 이후 argv, argc, return address를 차례로 저장해주었다.

**User Memory Accessing, System call handler**



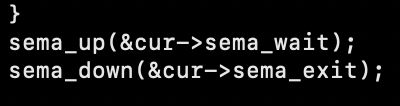
User Memory Accessing에 대해서는 따로 함수를 구현하지 않았기 때문에 System call handler에서 어떻게 사용되었는지를 기술하였다. 각 system call을 호출할 때 is\_user\_vaddr()함수를 통해서 argument들의 주소가 유효한 주소인지 확인하였고 그렇지 않다면 syscall\_exit(-1)을 통해서 process를 종료하였다. 이 때 userprog에 있는 exception.c 파일에 kernel 영역을 참조하게 될 경우 오류가 발생하여 이를 pass시켜주는 code를 작성하였다.

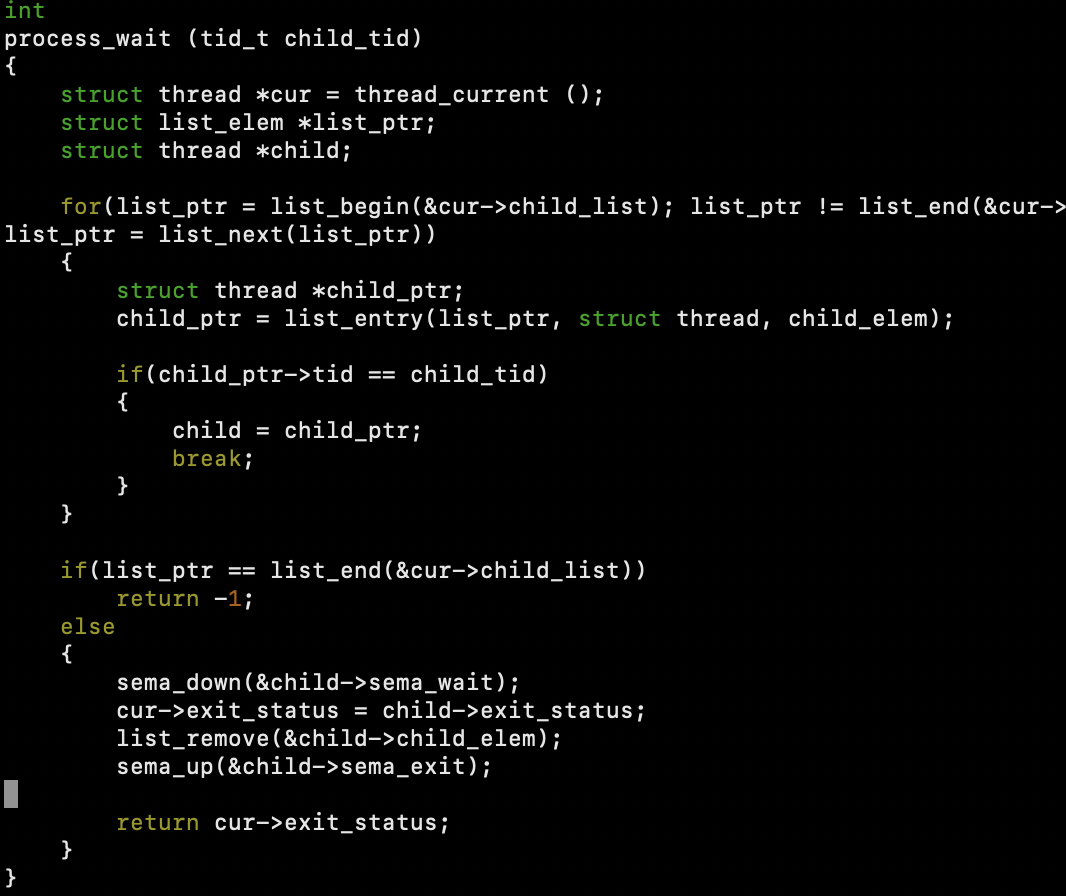
System call handler에서 각 system call은 system call number에 따라서 실행되며 구현은 3-B에서 서술한대로 구현하였다. 여기서 각각의 system call에 대해서 함수를 새로 선언하여 구현하였다.

System call handler 이외에 수정을 한 부분은 다음과 같다.

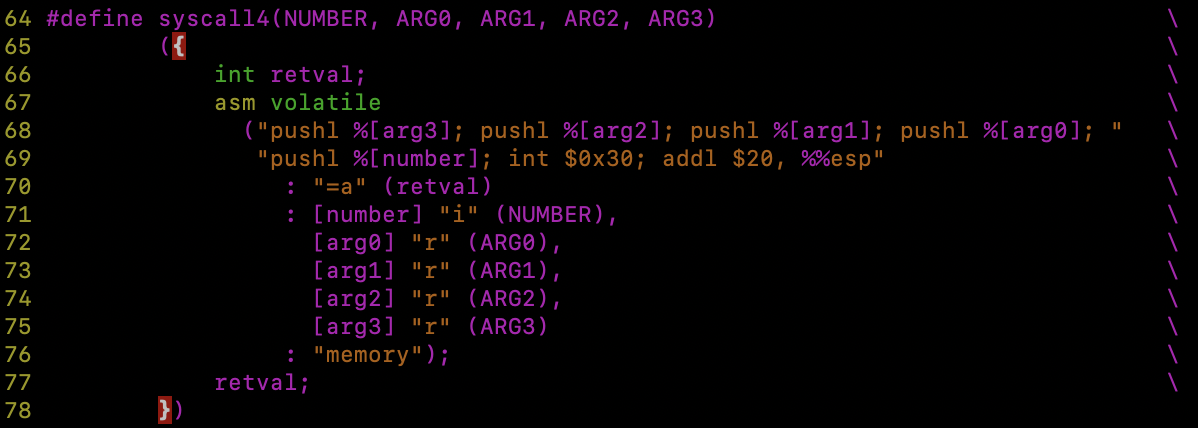
우선 Exec 함수에서 process\_execute를 호출하는데 이 때 thread\_create()을 통해서 새로운 thread가 생성되고 이 process는 현재 process의 child가 된다. src/threads/thread.h에 thread 구조체가 선언돼있는데 child\_list, child\_elem을 추가해주었다. process의 parent, child 구조를 명확하게 하기 위해 child\_list로 이를 관리해준다. 또한 위에서 추가해준 child\_list를 thread.c에 init\_thread 함수에서 list\_push\_back을 통해 child가 parent list에 들어가도록 한다.

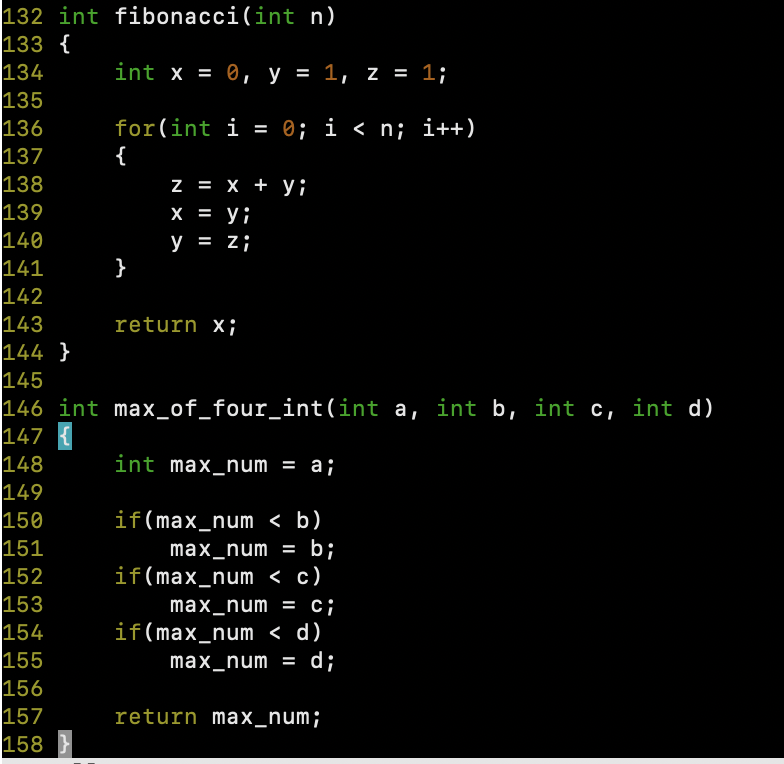
Wait, exit 함수를 구현하는데에는 src/threads/synch.h에 선언되어 있는 semaphore를 이용하였다. 두 개의 semaphore를 이용하였는데 sema\_wait을 통해서는 child가 죽은 것을 확인할 수 있고 sema\_exit을 통해서는 child가 죽기 전에 parent에서 child\_list에서 제거하기 위해 선언하였다. 구현되어있는 sema\_up, sema\_down 함수를 통해서 semaphore의 값을 변경하였으며 semaphore 값들은 init\_thread 함수에서 0으로 초기화시켜주었다.

wait, exit 함수를 구현하는데 있어서 process\_exit에서는 sema\_wait을 증가시키고 sema\_exit을 감소시킴으로써 process가 exit이 된다는 signal을 보낸다. 이후 process\_wait에서 이 signal을 받은 뒤에 child\_list에서 child를 제거하게 되면 sema\_exit이 sema\_up으로 인해 증가하게 되어 process\_exit에서 sema\_down을 통해 child의 exit이 마무리된다.

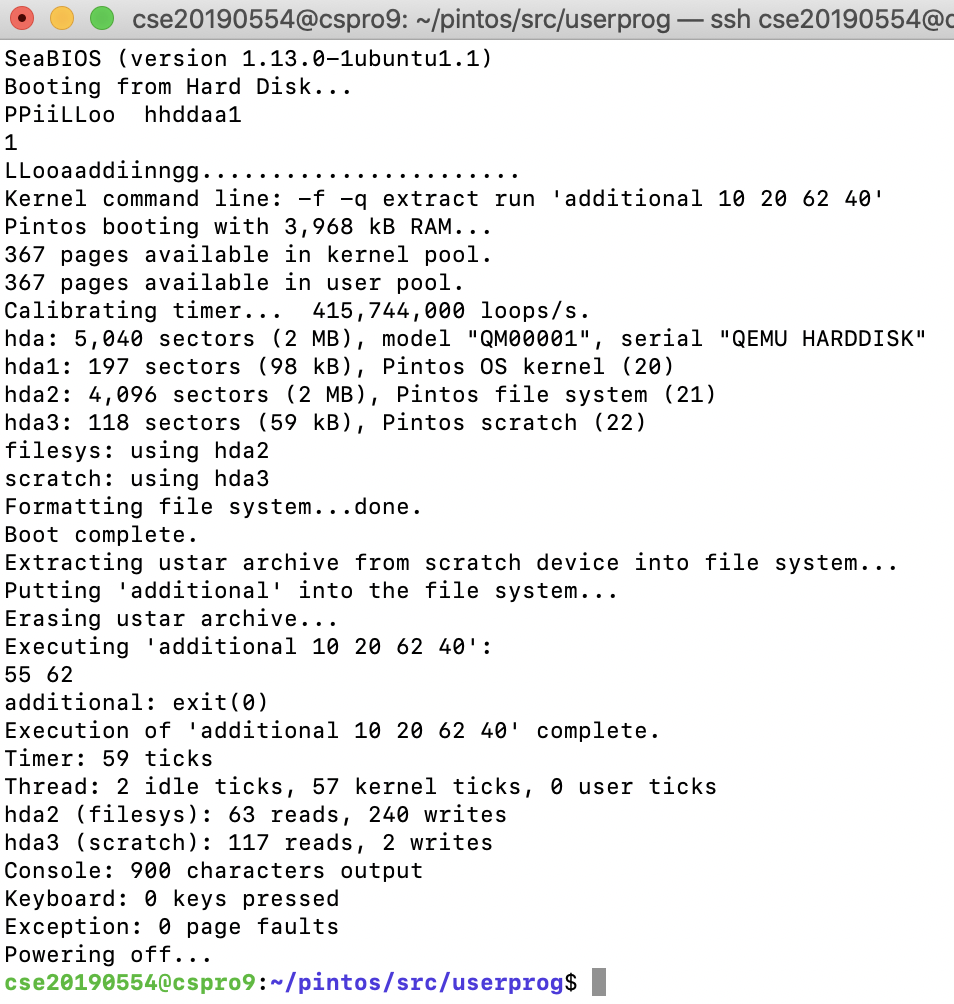


**Additional System Calls**

새롭게 구현한 fibonacci와 max\_of\_four\_int 함수는 userprog/syscall.h에 선언하였다. max\_of\_four\_int 전의 system call들은 인자를 최대 3개를 받았는데 max\_of\_four\_int는 인자가 4개가 필요하므로 새롭게 lib/user/syscall.c에 syscall4를 작성하였다.

또한 userprog/syscall.c 파일에 다른 system call과 마찬가지로 system call API를 작성해주었다.

다른 system call들은 이미 syscall-nr.h 파일에 system call number가 enum으로 정의돼있지만 fibonacci와 max\_of\_four\_int는 정의되어 있지 않으므로 새롭게 정의해주었다.

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