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

담당 교수 : 문의현 교수님

학번 / 이름 : 20181264 / 노영현

개발 기간 : 10/27 ~ 11/02

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

- 넘겨받은 FILE\_NAME 을 띄어쓰기를 기준으로 parsing해서 parsing 한 argument들을 stack에 passing한다. argument만을 passing하는 것이 아닌 인자의 개수, 주소 등 다른 것들도 pass 한다. 자세한 내용은 아래에서 하겠다.

- pointer가 user address를 건드려야 하는데 kernel address를 건드리는 경우, kernel address를 건드려야 하는데 user address를 건드리는 경우 2가지를 검사해 잘못된 memory access를 방지한다.

- halt, exit, exec, wait, write, read 등 system call을 구현하여 user mode에서 kernel mode에서 제공하는 서비스를 사용할 수 있도록 한다.

- N번째 피보나치수를 출력하는 fibonacci() 함수, 4개의 정수 중 가장 큰 값을 출력하는 max\_of\_four\_int() 함수를 구현해 kernel mode에서 실행시킨다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

Argument Passing을 구현하기 위해서는 우선 Argument를 Parsing하는 과정을 거쳐야 한다. Argument Parsing이란 입력 받은 file name을 띄어쓰기를 기준으로 나누는 것을 얘기한다. 예를 들어, "echo ab c d"를 Parsing한다면 "echo", "ab", "c", "d"로 나눠지게 된다. 이렇게 parsing된 구문을 80X86 calling convention에 따라서 메모리에 할당해준다. 메모리에 할당하는 과정은 stack에 push하는 과정으로 이루어진다. 스택에는 정해진 규칙에 따라서 문자열 등이 들어간다. 여기서 말하는 규칙은 스택에 push하는 순서이다. parsing한 문자열, 4바이트의 배수로 들어가기 위한 word\_align, parsing된 문자열이 들어간 주소, parsing된 문자열의 개수, 마지막으로 return adress로 0을 push하는 순서이다.

1. User Memory Access

프로그램에서 사용하는 포인터의 주소가 잘못된 메모리에 접근하는 경우가 발생할 수 있다. 이를 방지하기 위해서 threads폴더 내의 vaddr.h 파일 안에 있는 is\_user\_vaddr 함수와 is\_kernel\_vaddr 함수를 사용한다. is\_user\_vaddr 함수를 이용해서 user virtual address인지 확인하고 is\_kernel\_vaddr함수를 이용해서 kernel virtual address인지 확인한다. 메모리가 변경될 때 해당 함수를 이용해서 변경되는 메모리가 valid한지 확인한다.

1. System Calls

syscall.c 파일에서 syscall\_handler함수를 수정해 system call을 구현한다. 구현하고자 하는 system call은 halt, exit, exec, wait, read, write, fibonacci, max\_of\_four\_int이다. syscall.c내에서 현재 intr\_frame \*f가 가리키는 esp주소 값에 따라서 그에 해당하는 system call을 호출한다. fibonacci와 max\_of\_four\_int는 새로 정의해줘야 하는 함수이므로 src/lib/user 폴더 내의 syscall.c에서 새로 syscall함수를 써서 정의해주어야한다. 특히 max\_of\_four\_int의 경우 인자가 4개이므로 syscall4를 새로 define해주어야한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + file\_name의 입력으로 띄어쓰기가 포함된 문자열이 들어온다. 우리의 목표는 이를 띄어쓰기를 기준으로 parsing해서 parsing한 argument를 stack에 넣어주는 것이다. 우선 parsing하기 위해서 strtok\_r 함수를 사용하였다. strtok\_r함수를 사용하여 array에 parsing한 argument를 넣어주었다. 이제 array에서 argument를 빼서 stack에 push한다. stack에 push할 때에는 정해진 규칙에 따라서 push한다. 여기서 말하는 규칙은 스택에 push하는 순서이다. parsing한 문자열, 4바이트의 배수로 들어가기 위한 word\_align, parsing된 문자열이 들어간 주소, parsing된 문자열의 개수, 마지막으로 return adress로 0을 push하는 순서이다. stack에 push하는 과정에서 esp값을 감소시키면서 스택에 넣어준다. 즉, esp를 감소시키는 것이 stack을 늘리는 것이다.
* User Memory Access
  + invalid memory access란, pintos상에서 메모리가 건드리지 말아야 할 부분을 건드리는 것을 말한다. kernel address를 건드려야 하는 point가 user memory를 건드리는 것, user adrdress를 건드려야 하는 point가 pintos adress를 건드리는 것 모두 pintos 상에서 memory access를 의미한다.
  + 이러한 Invalid memory access를 막기위해서 vaddr.h 파일 안에 있는 is\_user\_vaddr 함수와 is\_kernel\_vaddr 함수를 사용한다. 각각은 현재 메모리가 user address를 건드리는 지, kernel address를 건드리는 지 확인해준다. 메모리가 변경될 때마다 각 함수들을 필요에 맞게 사용해 pintos상에서 invalid memory access를 방지할 수 있다.
* System Calls
  + 우리가 사용하는 대부분의 프로그램은 '응용프로그램'이다. 또한 이러한 응용프로그램들의 상당수는 kernel mode에서만 사용되도록 프로그래밍 되어있다. pintos프로그램에서 처음 user가 call을 해도 아무런 일이 일어나지 않는다. 이는 User mode에서 실행되지 않기 때문이다. 따라서 이를 해결하기 위해서 system call을 사용해 kernel mode에서 실행될 수 있도록 해야한다.
  + halt의 경우, pintos를 종료시킨다. 이를 위해서 Pintos 내부에 있는 shutdown\_power\_off() 함수를 이용해서 pintos를 종료시킨다.
  + exit의 경우, 현재 프로세스를 종료시키는 system call로, 현재 thread를 exit시켜 프로세스를 종료시킨다.
  + exec의 경우, 현재 프로세스를 실행시키는 system call로 process.c에 구현되어있는 process\_execute함수를 사용한다.
  + wait은 현재 프로세스를 wait시키는 system call이다. 이를 위해서 process.c.에서 process\_wait함수를 직접 구현하였다. process\_wait에 대해서는 아래에서 더 자세히 다루겠다.
  + read는 주어지는 문자열을 buffer에 저장하는 system call이다. 각 문자를 읽어들이는 input\_getc()함수를 사용한다. 각 문자를 읽어들여 buffer에 저장해주었다.
  + write는 buffer에 저장되어 있는 문자열을 모니터에 출력하는 system call이다. buffer에 저장되어 있는 문자열을 출력하는 putbuf함수를 사용하여 출력하였다.
  + fibonacci는 n번째 피보나치 수를 반환하는 함수를 직접 구현했다. max\_of\_four\_int는 주어지는 4개의 정수 중 가장 큰 수를 반환하는 함수를 직접 구현하였다. 이 두 함수에 대해서는 아래에서 더 자세히 설명하겠다.
  + 유저 레벨에서 system call을 호출하면 우선 하드웨어가 mode bit를 바꿔 kernel mode에 진입시킨다. 각각의 system call을 숫자로 구별되는데 system call interface가 이 숫자에 따라 매핑된 테이블을 가지고 있다. 이 테이블을 바탕으로 user가 어떤 system call을 요청했는지 kernel mode에서 확인한다. system call interface는 OS 커널에서 의도된 시스템 콜을 발생시킨다. 시스템 콜이 완료되면 다시 mode bit를 바꿔 user mode에 진입 시켜 user level 로 돌아간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

10/27~10/28 argument passing 구현

10/29~10/31 user memory access, system call 구현

11/01 additional implementation(fibonacci, max\_of\_four\_int) 구현

11/02 오류 수정

* 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)**

- thread.h 내의 struct thread구조체에 int exit\_status, struct list child\_list, struct list\_elem child\_element, struct semaphore child\_semaphore, struct semaphore parent\_semaphore 를 추가한다. 이 인자들을 이용해서 process\_wait을 구현한다.

- argument passing을 하기 위해 process.c 파일 내부의 load함수에서 parsing하는 코드와, parsing한 arguments들을 stack에 passing 하는 코드를 추가한다.

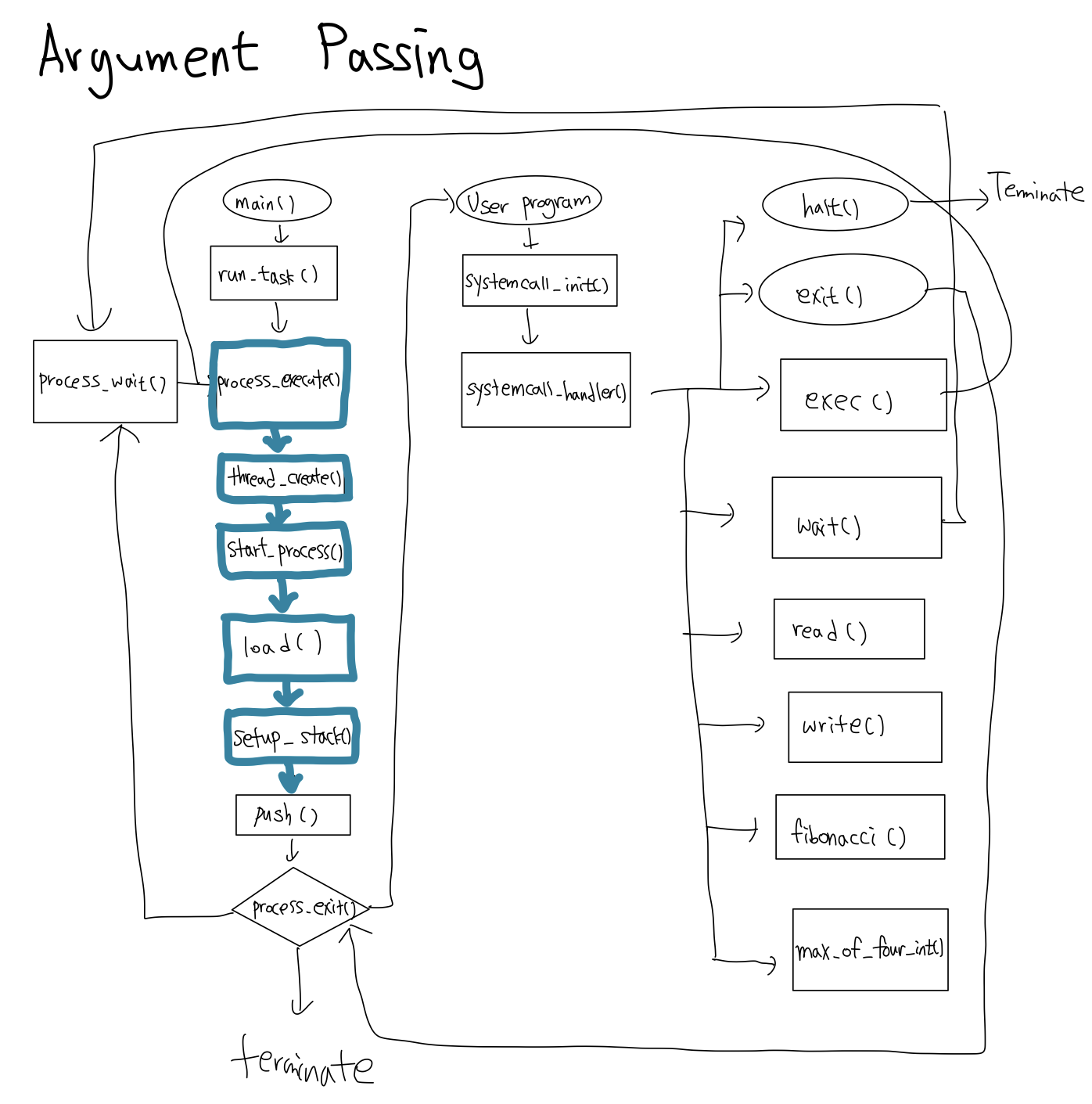
- user memory access를 구현하기 위해 vaddr.h 에 구현되어 있는 is\_user\_vaddr함수와 is\_kernel\_vaddr 함수를 syscall.c 에서 사용한다.

- system call 을 구현하기 위해서 process.c 내부의 process\_exit 함수를 수정하고, process\_wait 함수를 구현한다. syscall.c 내부에서 각 system call에 대응되는 함수를 작성한다. 추가로 구현해야 하는 fibonacci함수와 max\_of\_four\_int함수를 syscall.c 에서 구현하고 이를 사용하기 위해서 src/examples 내부에서 additional.c 파일을 만들어준다. 또한, lib/user/ 내부에 syscall.c 파일에서 함수를 구현하고 max\_of\_four\_int 함수를 위해서 syscall4 함수를 define 해준다.

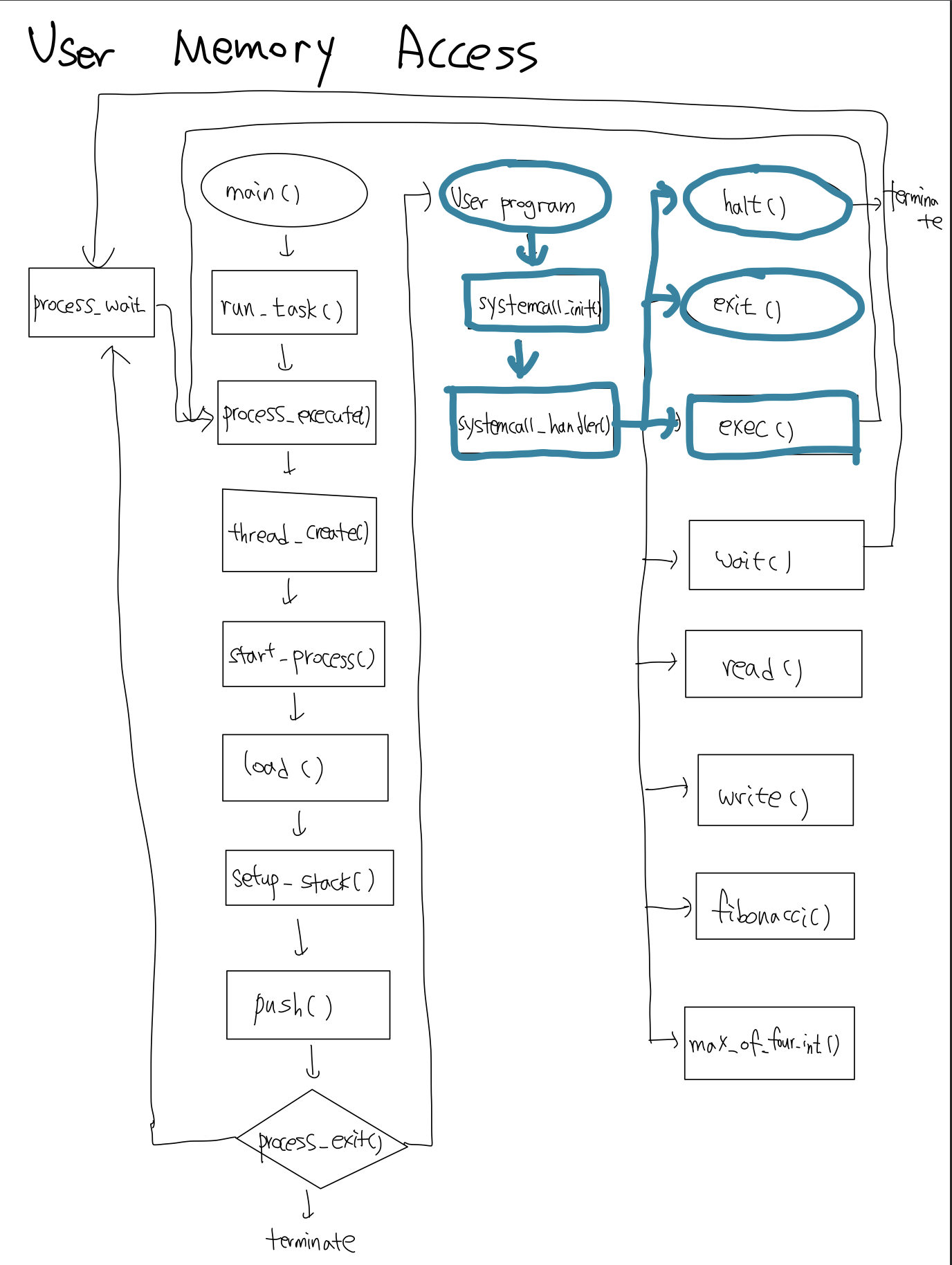
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

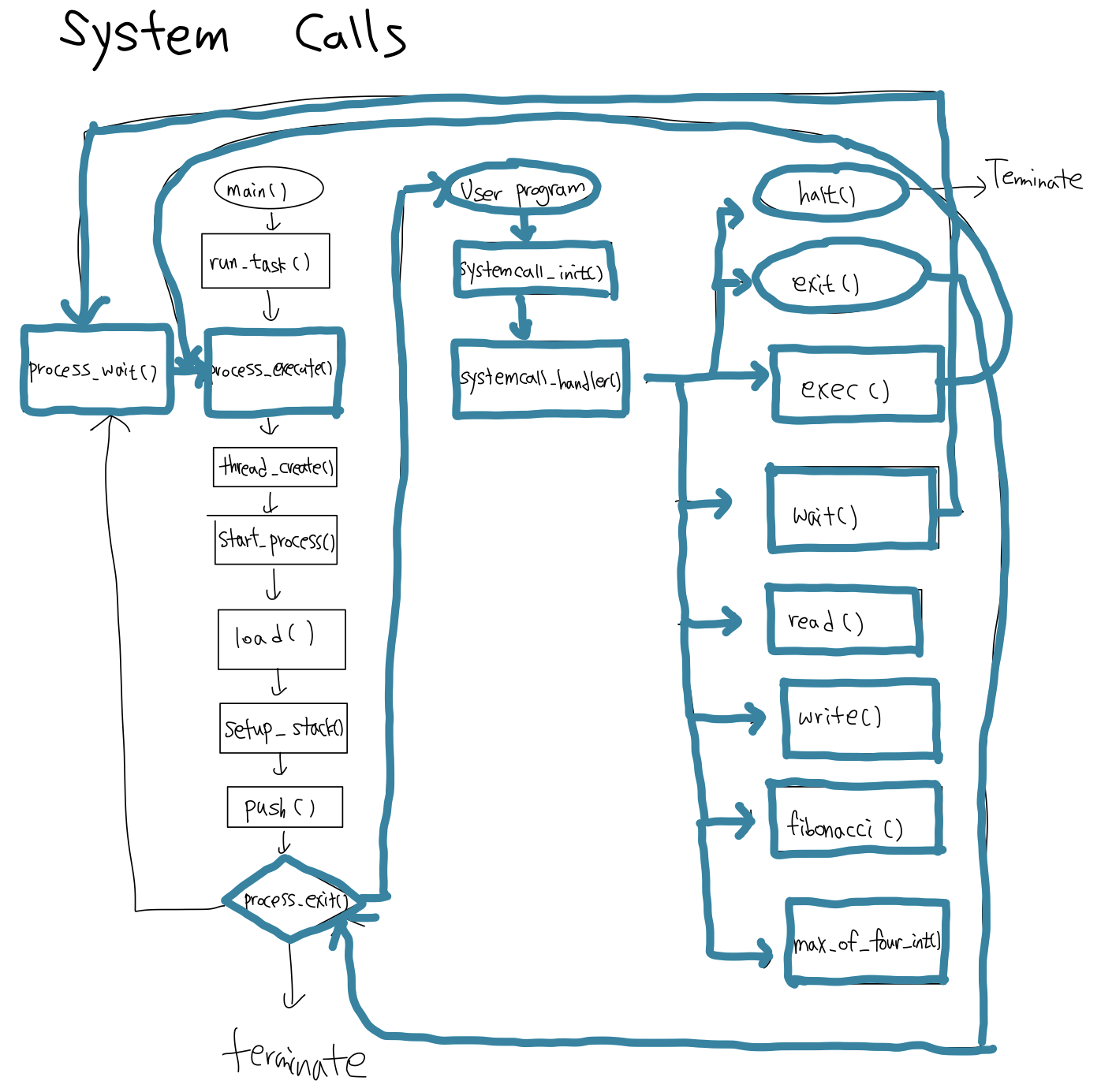
1. Argument Passing



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

우선 argument parsing하는 코드를 설명하겠다. strtok\_r 함수를 사용한다. parsing 한 arguments들을 저장할 char \*array[150]을 만들어줬다. 잘리는 argument를 tok에 저장한다. 현재 File\_name이 const char\*이므로 strtok\_r 함수를 못써주므로 copyary를 만들어서 File\_name을 복사해줬다. tok = strtok\_r(copyary, " ", &next) 함수로 첫 번째 parsing한 argument를 tok에 저장해줬다. 이후 while(tok)문을 통해서 tok이 Null이 될 때 까지 tok = strtok\_r(NULL, " ", &next) 함수로 tok을 업데이트 해준다. 이렇게 만든 tok을 array[idx++]로 array에 넣어준다.

이렇게 parsing한 array를 이제 stack에 넣어준다. stack에 넣어주는 과정을 push 함수를 통해 stack에 넣어준다. esp를 줄여서 stack에 넣어준다. 우선 ary에 들어있는 문자열들을 하나하나 스택에 넣어준다. 스택에는 esp를 1씩 변경시키면 각 문자를 넣는다. 우선 ary[i]의 길이만큼 esp를 빼줘서 스택의 길이를 늘려준다. 그리고 esp를 1씩 더해주면서 ary[i]에 저장되어 있는 문자 하나하나씩 넣어주었다. 그리고 parsing된 문자열 하나를 다 넣으면 '\0' 널 문자를 넣어준다. 여기서 문자 하나를 넣어줄 때 마다 sum값을 1씩 늘려준다. 마지막에 sum이 4의 배수가 아니라면 이를 4의 배수로 맞춰 주어야 한다. 4바이트 기준으로 스택에 넣어주어야 OS의 속도가 빨라지기 때문이다. 위에서 parsing된 문자열을 하나 넣을 때마다 해당 esp의 주소를 save라는 배열에 넣어주었다. 이제 save배열에 저장되어 있는 주소값들을 esp를 4씩 줄여가며 stack에 넣어준다. 그리고 esp를 4 줄이고 현재 esp+4를 더해주었다. 또 esp를 4 줄이고 parsing한 argument의 개수를 push해준다. 마지막으로 esp를 4 줄여 0을 push해주는 것으로 argument passing을 마무리한다.

1. User Memory Access

syscall.c에서 bool validcheck(const void\* addr)함수를 만들어주었다. valid check 함수에서는 두 가지를 확인한다. is\_user\_vaddr(addr)의 반환값이 false이거나 addr 이 NULL인 경우 false를 반환한다. 이게 아닌 경우 true를 반환한다. 이는 만약 주소가 비어있거나 user address가 아닌 곳을 건드리는 것을 의미한다.

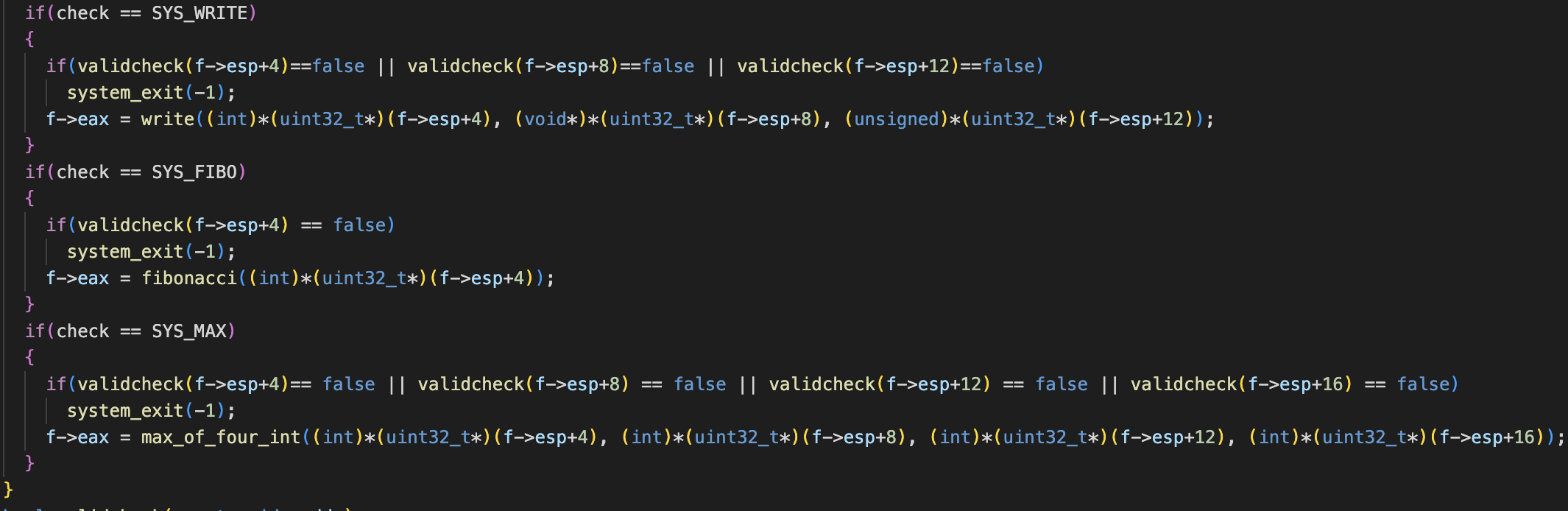
exception.c 파일에서 page\_fault 함수에서 is\_kernel\_vadder(fault\_addr)이 false인 경우 kernel address가 아닌 곳을 건드리고 있는 것이므로 system\_exit(-1)함수를 사용해 exit 시켰다.

1. System Calls

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

텍스트이(가) 표시된 사진

자동 생성된 설명



현재 system call number를 나타내는 \*(uint32\_t \*)(f->esp) 값을 check에 저장하고, check 값에 따라 그에 대응되는 system call을 호출한다. 각 system call에 해당하는

(1) halt()

pintos프로그램을 종료하는 system call로 pintos상에서 구현되어 있는 shutdown\_power\_off()함수를 호출해 pintos프로그램을 종료시킨다.

(2) exit()

thread\_current()의 exit\_status를 f->esp+4 를 정수형으로 바꾼 값으로 바꿔준다. 이후 thread\_exit()함수로 thread를 exit()해 프로세스를 exit()해준다.

(3) exec()

f->esp+4를 const char형으로 바꾸고 이를 인자로 Process.c에 구현 되어 있는 process\_execute 함수를 실행시킨다. process\_execute의 반환값을 f->eax에 저장한다.

(4) wait()

f->esp+4를 tid\_t 자료형으로 바꾼 뒤 process.c에 존재하는 processs\_wait()함수에 인자로 넣어준다. process\_wait()는 직접 구현하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

parent process는 child\_process가 종료될 때까지 wait 해야 하고, 종료되면 child process의 exit\_status를 return 받는다. 따라서 sema\_down을 통해 child\_process가 종료될 때 까지 기다리고 sema\_up을 통해 메모리를 남겨둔다.

우선 현재 thread의 child\_list를 children이라는 배열에 저장한다. 이후 iterator 변수를 사용해서 children 배열을 돈다. iterator변수는 list\_elem 자료형을 가진다. 나는 iterator변수의 thread identity number를 알고 싶으므로 list\_entry 함수를 이용해서 iterator를 가지는 struct thread를 알아낸다. 이 struct thread를 child\_thread라는 변수에 저장한다. 이제 process\_wait의 인자로 받은 child\_tid와 child\_thread가 가지는 tid값이 같은 경우, child\_thread의 child\_semaphore변수를 sema\_down시킨다. 이는 child process가 종료될 때 까지 기다리는 것을 의미한다. 이후 child\_thread의 child\_element를 list에서 삭제한다. 이후 wait이 끝나면 sema\_up을 통해서 메모리를 남겨둔다. child\_thread의 exit\_status를 반환 값으로 함수를 종료한다.

(5) read

read는 인자로 fd, buffer, size를 받는다. fd 값이 0인 경우 read 함을 의미한다. 0이 아닌 경우 -1을 반환하고 함수를 종료하고, 0인 경우 size만큼 반복문을 돌며 input\_getc() 로 문자 하나를 받아 이를 buffer에 넣어준다. 그리고 넣어준 개수만큼 return 해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

syscall\_handler 함수에서는 f->esp+4, f->esp+8 f->esp+12 를 각각 int, void\*, unsigned로 자료형을 바꿔서 인자로 넣어준다.

(6) write

fd가 1인 경우만 buffer에 저장되어 있는 문자를 출력한다. fd가 1이 아닌 경우 -1을 return 하고 함수를 종료한다. write함수에서 인자로 받은 buffer와 size를 putbuf함수에 넣어 출력할 수 있다. 출력한 길이만큼을 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

read와 마찬가지로 syscall\_handler에서 f->esp+4, f->esp+8, f->esp+12를 각각 int, void\*, unsigned로 자료형을 바꿔서 인자에 넣어준다.

fibonacci함수와 max\_of\_four\_int함수는 아래 4번에서 자세히 설명하겠다.

1. Additional System calls

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

텍스트이(가) 표시된 사진

자동 생성된 설명

int fibonacci(int n) 함수는 n번 째 피보나치 수를 반환하는 함수이다. 이를 구현하기 위해 동적계획법을 (Dynamic programming)을 사용하였다. 기본값으로 함수 인자에 1이나 2가 들어오는 경우 1을 반환했다. 1이나 2가 아닌 경우, n-2번 만큼 반복문을 돌며 피보나치 수를 계산한다. 앞 2개의 수 before와 after에 각각 저장한다. 이제 after 를 before로 바꾸고 원래 앞 2개의 수를 더한 값을 after에 저장한다. 이 과정에서 save변수를 활용한다. n-2번 만큼의 반복문을 다 돌았으면 after값을 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

max\_of\_four\_int함수는 4개의 인자를 받아 가장 큰 정수 값을 반환하는 비교적 쉬운 함수이다. ret값을 a로 초기화하고 b, c, d와 비교해 ret이 더 작을 때 마다 ret을 바꿔준다. 이 ret값을 반환한다.

위에서 만든 fibonacci함수와 max\_of\_four\_int 함수를 system\_call로 실행하기 위해서 lib/users의 syscall-nr.h파일에서 SYS\_FIBO와 SYS\_MAX를 enum해준다. 또한 lib/users의 syscall.c 파일에서 int fibonacci(int n)함수와 int max\_of\_four\_int(int a, int b, int c, int d)함수를 만들어준다. 각 함수에서는 아래와 같이 syscall함수를 호출해야한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

syscall(n) 함수는 인자가 n개인 system call을 호출하는 함수다. syscall4의 경우 원래 정의되어 있지 않으므로 새로 작성해줬다. syscall3 함수에서 인자를 하나 늘리는 식으로 구현하므로 어렵지 않게 구현할 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

마지막으로 src/examples/lib/ 에 additional.c 파일을 만들어준다. additional.c 파일에서는 int main함수에서 char\*\*argv를 인자로 받아 argv에 들어있는 숫자들을 fibonacci함수와 max\_of\_four\_int 함수에 넣어 결과값을 출력하는 코드를 작성한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

src/examples/lib 에 있는 makefile에 additional.c를 추가하면 이제 fibonacci함수와 max\_of\_four\_int 함수를 system call 을 사용해서 출력할 수 있다.

사용하는 방법은 src/userprog의 syscall.h 파일의 syscall\_handler함수에서 f->esp+4를 fibonacci함수에 넣어주거나 f->esp+4, f->esp+8, f->esp+12, f->esp+16을 max\_of\_four\_int 함수에 넣어준다.

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

  자동 생성된 설명**