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

담당 교수 : 박 성 용 교수님

조 / 조원 : 20181614 김 주 연

개발 기간 : 2020/10/05 ~ 2020/11/01

1. **개발 목표**

user program이 Pintos 위에서 정상적으로 실행되고 종료될 수 있도록 하기 위해서 다음 기능들을 구현한다.

* **Argument Passing**: Parse the arguments and allocate it to memory

according to 80x86 calling convention

* **User Memory Access**: Protect user memory accesses from system calls.
* **System Call Handler**: Implement syscall\_handler() to handle system call.
* **System Call Implementation**: Implement halt, exit, exec, wait, read, write
* **Additional Implementation**: Implement fibonacci(), max\_of\_four\_int().

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

: 입력으로 들어 온 command line에서 argument들을 parsing하여 80x86 calling convention에 맞게 user stack에 쌓는다.

1. User Memory Access

: user program이 kernel 영역을 침범하게 될 경우 invalid pointer로 판단하여 작동하지 못하도록 한다.

1. System Calls

: Pintos manual에 있는 system call 중 halt, exit, exec, wait 기능을 완성하고, read, write에 대해서는 standard input/output이 가능할 정도로 구현한다. 추가적인 system call인 fibonacci, max\_of\_four\_int를 구현한다.

* 1. **개발 내용**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정

: 입력으로 들어 온 command line에서 argument들을 parsing하여 80x86 calling convention에 맞게 user stack에 쌓기 위해 우선 load() 함수에서 argument parsing 작업을 수행한다. strtok\_r() 함수를 이용하여 argument들을 parsing하고, 각 argument는 argv[] 배열에, argument의 개수를 argc에 저장한다. 이후 setup\_stack() 함수가 제대로 수행되었다면 construct\_stack() 함수를 호출하여 해당 함수에서 parsing된 argument들을 stack에 쌓는 작업을 수행하게 된다. 초기 esp 값인 PHYS\_BASE, 즉 0xc0000000부터 거꾸로 argument들을 쌓게 되며, 쌓을 argument에는 argument data, word alignment, null pointer sentinel, argument address, argv, argc, return address를 포함한다. 이때 실제로 반환하는 것이 아니므로 return address에는 fake address로 0을 넣어준다. argument passing의 결과는 hex\_dump()를 사용해서 확인할 수 있는데, 예를 들어 ‘echo x’ 가 command line으로 들어왔을 때 stack에는 정보가 다음과 같이 저장된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이를 명세서에서의 형식대로 나타내면 아래와 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Name | Data | Type |
| 0xbffffffe | argv[1][…] | ‘x\0’ | char[2] |
| 0xbffffff9 | argv[0][…] | ‘echo\n’ | char[5] |
| 0xbffffff8 | word-align | 0 | uint8\_t |
| 0xbffffff4 | argv[2] | 0  (null pointer sentinel) | char\* |
| 0xbffffff0 | argv[1] | 0xbffffffe | char\* |
| 0xbfffffec | argv[0] | 0xbffffff9 | char\* |
| 0xbfffffe8 | argv | 0xbfffffec | char\*\* |
| 0xbfffffe4 | argc | 2 | int |
| 0xbfffffe0 | return address | 0 | void(\*) () |

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념

: Pintos는 PHYS\_BASE를 기준으로 하여 memory를 user memory와 kernel memory의 두 부분으로 나누어 사용하는데, invalid memory access란 user program이 잘못된 포인터 주소로 접근하는 것을 말한다. 잘못된 포인터 주소로는 Null pointer, Unmapped virtual memory, Kernel address space pointer 등이 있으며, 특히 user program이 kernel memory 영역에 접근하는 경우 page fault 등의 오류가 발생하게 되기 때문에 kernel을 침범하지 않도록 처리해 주는 과정이 필요하다.

* + Invalid memory access를 어떻게 막을 것인지

: invalid memory access가 발생하면 kernel이나 다른 process가 제대로 수행될 수 없기 때문에 이를 막아 주어야 하는데, 이 프로젝트에서는 user program pointer가 PHYS\_BASE보다 아래에 접근하고자 하는지 체크하는 방법을 사용한다. pintos/src/thread.vaddr.h 파일에 정의된 is\_kernel\_vaddr() 함수는 해당 주소가 PHYS\_BASE보다 크다면, 즉 kernel virtual address라면 true를, 아니라면 false를 리턴하게 되는데, 이 함수를 사용하여 user program이 kernel 영역에 접근하는지 체크하여 invalid memory access가 수행되지 않도록 구현한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

: 위의 User Memory Access 파트에서 말했듯이 Pintos에서는 memory를 user memory와 kernel memory의 두 부분으로 나누어 사용하는데, user program은 user mode에서만 수행되기 때문에 kernel이 관리하는 memory나 disk에는 직접적으로 접근할 수 없다. 따라서 user level의 process가 kernel의 기능을 사용하고자 할 때는 system call을 통해서 kernel mode로의 전환이 필요하다. 즉, system call은 user program이 kernel mode에 요청할 수 있게 해 주는 interface 역할을 하게 된다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명

- halt : shutdown\_power\_off() 함수를 호출하여 Pintos를 종료한다.

- exit : status를 argument로 받아 현재 process 이름과 status를 콘솔에 출력하고 kernel에 status를 전달하여 해당 user program이 종료되도록 한다. parent가 있다면 parent에게 status를 전달하며, status가 0이 아니라면 비정상적으로 종료된 것이다.

- exec : cmd\_line을 인자로 받아 실행 파일 이름과 argument로 새로운 process를 생성한다. 이후 process가 제대로 만들어졌다면 해당 process의 id를 반환하게 되며, 새로 만들어진 process는 현재 process의 child가 된다.

- wait : pid를 인자로 받아 해당 pid를 갖는 child process가 있다면 해당 child가 종료되기까지 기다리도록 한다. 이때 pid가 유효한지 판단하여 유효하다면 child가 종료될 때 exit status를 받아오게 되며, 유효하지 않은 tid라면 -1을 반환한다. 이 프로젝트에서는 wait 기능을 수행하기 위해 semaphore를 이용하였다.

- read : project 1에서는 standard input을 받을 수 있을 정도로만 read 기능을 구현하였다. file discriptor와 buffer, size를 인자로 받아 file descriptor 값이 0인 경우 STDIN이므로 데이터를 size만큼 읽어서 buffer에 저장하게 된다. input\_getc()함수를 통해 standard input을 받는 방식으로 구현하였다.

- write : project 1에서는 standard output을 할 수 있을 정도로만 write 기능을 구현하였다. file discriptor와 buffer, size를 인자로 받아 file descriptor 값이 1인 경우 STDOUT이므로 buffer의 데이터를 size만큼 출력하게 된다. putbuf() 함수를 통해 standard output을 하는 방식으로 구현하였다.

- fibonacci: project 1에서 추가적으로 만든 system call로, 정수 n을 인자로 받아 n번째 Fibonacci 수열 값을 반환한다.

- max\_of\_four\_int : project 1에서 추가적으로 만든 system call로, 4개의 argument를 인자로 받아서 그 중 최대값을 반환한다.

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

: 일반적인 OS에서 system call을 수행하는 과정을 간단히 하면 다음과 같다.

* + 1. user application에서 함수(API)를 호출한다.
    2. API가 interrupt를 발생시키면서 해당 system call을 가리키는 system call number를 kernel mode로 보낸다.
    3. system call handler는 이 system call number를 받아서 system call interface table로 전달한다.
    4. system call interface table에서 해당 system call number에 해당하는 주소로 이동한다.
    5. 해당 주소에 있는 system call 기능을 수행한다.
    6. 수행이 끝나면 user mode로 돌아간다.

이 과정이 Pintos에서 어떻게 구현되어 있는지를 open()을 예시로 하여 살펴보면 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

16번째 줄에서 user program이 open()을 호출하게 되면, 아래의 pintos/lib/user/syscall.c 파일의 system call API의 코드에서는 open()에 해당하는 system call number와 argument를 인자로 하여 매크로 함수인 syscall1을 호출한다.



syscall1에서는 open()의 system call number와 argument를 stack에 push하는 작업을 수행하게 되며, ‘int $0x30’이라는 어셈블리 명령어를 통해 interrupt handler를 직접 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

intr\_handler에서는 아래와 같이 system call handler를 호출하게 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

syscall\_hadler()에서는 인자로 넘어온 intr\_frame 구조체 f의 esp를 이용하여 해당 system call의 정보가 담긴 stack에 접근한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그 후 stack에 저장된 system call number를 통해 해당하는 system call을 알아내고, system call의 인자 수에 맞는 argument를 stack에서 가져온다. 이후 해당 포인터들의 유효성을 체크하여 유효한 포인터라면 해당 system call을 호출하게 된다. system call이 수행되고 나서 반환값이 있다면 intr\_frame 구조체 f의 eax에 저장된다. system call의 수행이 끝나면 다시 원래의 코드로 돌아간다.

Pintos에서는 이와 같은 과정을 통해 system call이 수행된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

전체 프로젝트 기간 : 2020/10/05 ~ 2020/11/01

- 2020/10/05 ~ 2020/10/09 : 강의자료, 프로젝트 설명,

Pintos manual 등을 통한 프로젝트 계획 수립

- 2020/10/10 ~ 2020/10/16 : Argument passing 구현

- 2020/10/17 ~ 2020/10/27 : User Memory Access, System call 구현

- 2020/10/28 ~ 2020/10/30 : Additional Implementation 구현

- 2020/10/31 ~ 2020/11/01 : 보고서 작성

* 1. **개발 방법**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정을 구현하기 위해 우선 pintos/src/userprog/process.c 파일의 load() 함수에서 file을 열기 전 argument parsing 작업을 하는 코드를 작성한다. argument parsing 후 공백을 기준으로 하여 각 argument는 argv[]배열에, argument 개수는 argc에 저장된다. 이후 마찬가지로 load() 함수에서 setup\_stack() 함수를 호출한 뒤 만들어진 stack에 argument passing을 위한 함수인 construct\_stack() 함수를 호출해 준다. construct\_stack()에서는 위의 2-B에서 보였던 것처럼 스택에 argument를 거꾸로 쌓는 작업을 수행하게 되며, 초기에 PHYS\_BASE에 위치해 있는 esp를 감소시키며 해당 부분에 argument를 채워 나가게 된다.
* User Memory Access
  + Invalid memory access를 막기 위하여, syscall.c 파일에 void accessing\_user\_memory(const void\* vaddr) 함수를 작성한다. 이 함수는 pintos/src/thread.vaddr.h 파일에 정의된 is\_user\_vaddr() 함수를 사용해서 해당 주소가 유효한 주소인지 판단하게 되며, 만약 유효하지 않은 주소라면 -1을 exit system call에 보내 해당 프로그램이 종료되도록 하는 기능을 수행한다. system call을 수행하기 전 각 argument의 주소에 대해 accessing\_user\_memory() 함수로 유효성을 체크하여 invalid memory access가 일어나지 않도록 한다.

System Calls

* system call을 구현하기 위해 pintos/src/userprog/syscall.c의 syscall\_handler() 함수를 완성한다. syscall\_hadler()에서는 인자로 넘어온 intr\_frame 구조체의 esp를 이용하여 해당 system call의 정보가 담긴 stack에 접근한다. 그 후 stack에 저장된 system call number를 통해 해당하는 system call을 알아내고, system call의 인자 수에 맞는 argument를 stack에서 가져온다. 이후 해당 포인터들의 유효성을 accessing\_user\_memory() 함수를 통해 체크하여 유효한 포인터라면 해당 system call을 호출한다. system call이 수행되고 나서 반환값이 있다면 intr\_frame 구조체의 eax에 저장한다.

각 system call은 별도의 함수를 작성하여 구현한다.

- halt 함수에서는 shutdown\_power\_off() 함수를 호출한다.

- exec 함수에서는 cmd\_line을 인자로 받아 process\_execute() 함수를 호출한다. process\_execute() 함수에서는 thread\_create() 함수를 통해 새로운 thread를 만들게 되고, 이때 새로 만들어진 process는 현재 process의 child가 된다. 이와 같은 parent-child 관계를 나타내기 위해 pintos/src/threads/thread.h의 thread 구조체에 child\_list와 child\_elem을 추가한다. 또한 thread\_create() 함수에서 child가 parent의 thread\_list에 들어가도록 설정하여 parent-child 관계가 제대로 연결되도록 한다. thread\_create() 함수로 process가 제대로 만들어졌다면 해당 process의 id가 반환된다.

- read 함수에서는 file discriptor와 buffer, size를 인자로 받아서 file descriptor 값이 0인 경우 STDIN이므로 데이터를 size만큼 읽어 buffer에 저장하는 기능을 수행한다. input\_getc()함수를 통해 STDIN을 받는 방식으로 구현한다.

- write 함수에서는 file discriptor와 buffer, size를 인자로 받아서 file descriptor 값이 1인 경우 STDOUT이므로 buffer의 데이터를 size만큼 출력하는 기능을 수행한다. putbuf() 함수를 통해 STDOUT을 하는 방식으로 구현한다.

- fibonacci 함수에서는 정수 n을 인자로 받아 n번째 Fibonacci 수열 값을 반환한다.

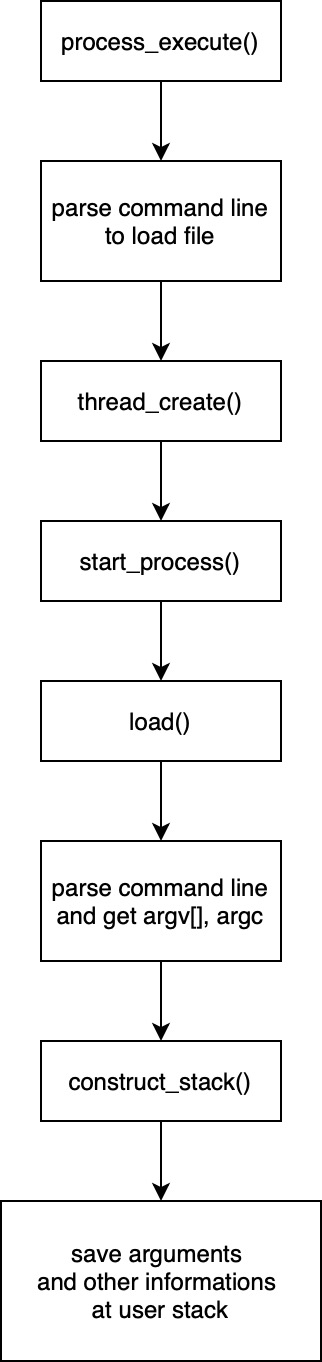
- max\_of\_four\_int 함수에서는 4개의 argument를 인자로 받아서 그 중 최대값을 반환한다.

이 프로젝트에서는 wait과 exit 기능을 구현하기 위해서 semaphore 방식을 선택하였다. 따라서 pintos/src/threads/thread.h의 thread 구조체에 child가 죽었음을 알리기 위한 semaphore 구조체 변수를 추가한다.

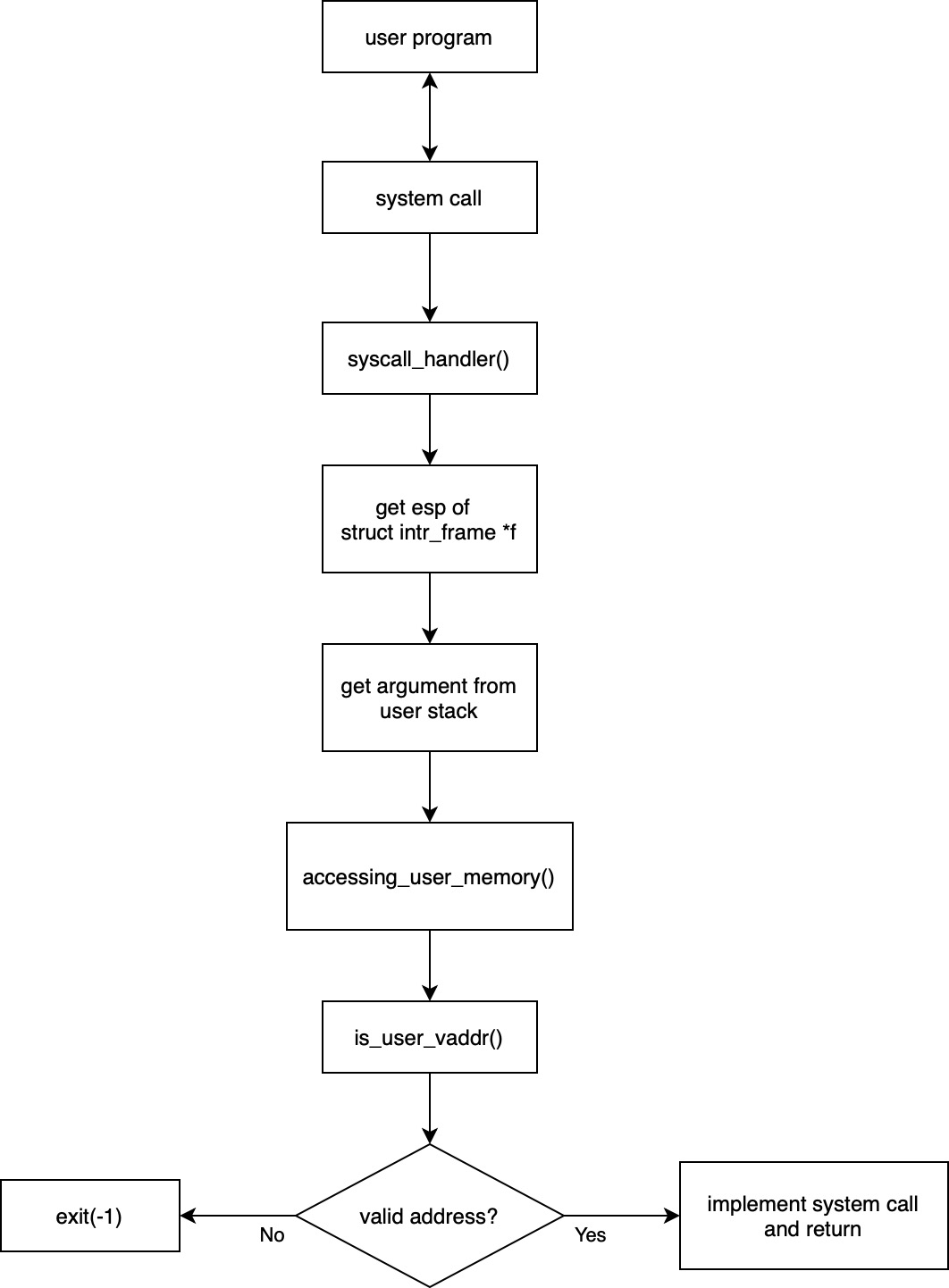
- wait 함수에서는 pid를 인자로 받아 process\_wait() 함수를 호출한다. process\_wait() 함수에서는 현재 thread의 child\_list를 순회하면서 해당 pid를 갖는 child process가 있는지 검색하게 되고, 만약 해당하는 child가 있다면 child가 종료되기까지 기다리도록 한다. child가 존재하지 않는다면 -1을 반환한다. 해당하는 child가 죽었음은 semaphore 변수의 값 변화를 통해 알 수 있으며, child가 죽었다면 process\_wait() 함수에서 해당 child의 status를 얻고 parent의 child\_list에서 그 child를 제거한 뒤 해당 child가 exit되도록 한다.

- exit 함수에서는 status를 argument로 받아 현재 process 이름과 status를 콘솔에 출력하고 thread\_exit() 함수를 호출한다. thread\_exit() 함수에서는 process\_exit() 함수를 호출하게 되므로, process\_exit() 함수에서 child가 종료되었음을 알려주는 semaphore를 설정한다. process\_exit() 함수에서 semaphore의 값을 증가시켜 해당 process가 exit될 것임을 알리면 이를 위의 process\_wait()함수에서 받게 되며, process\_wait()함수에서 해당 process를 parent의 child list에서 제거하였다면 exit을 수행한다.

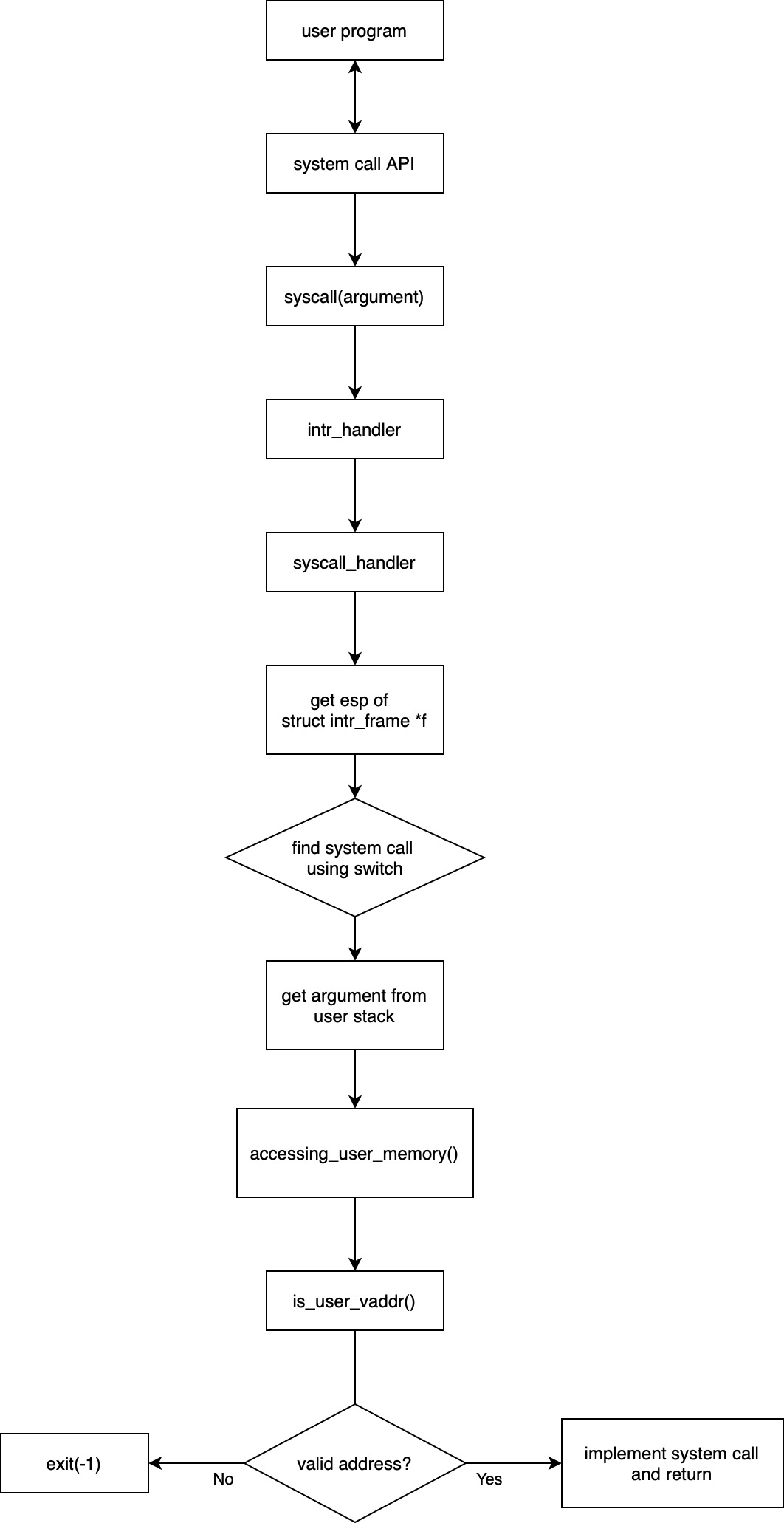
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



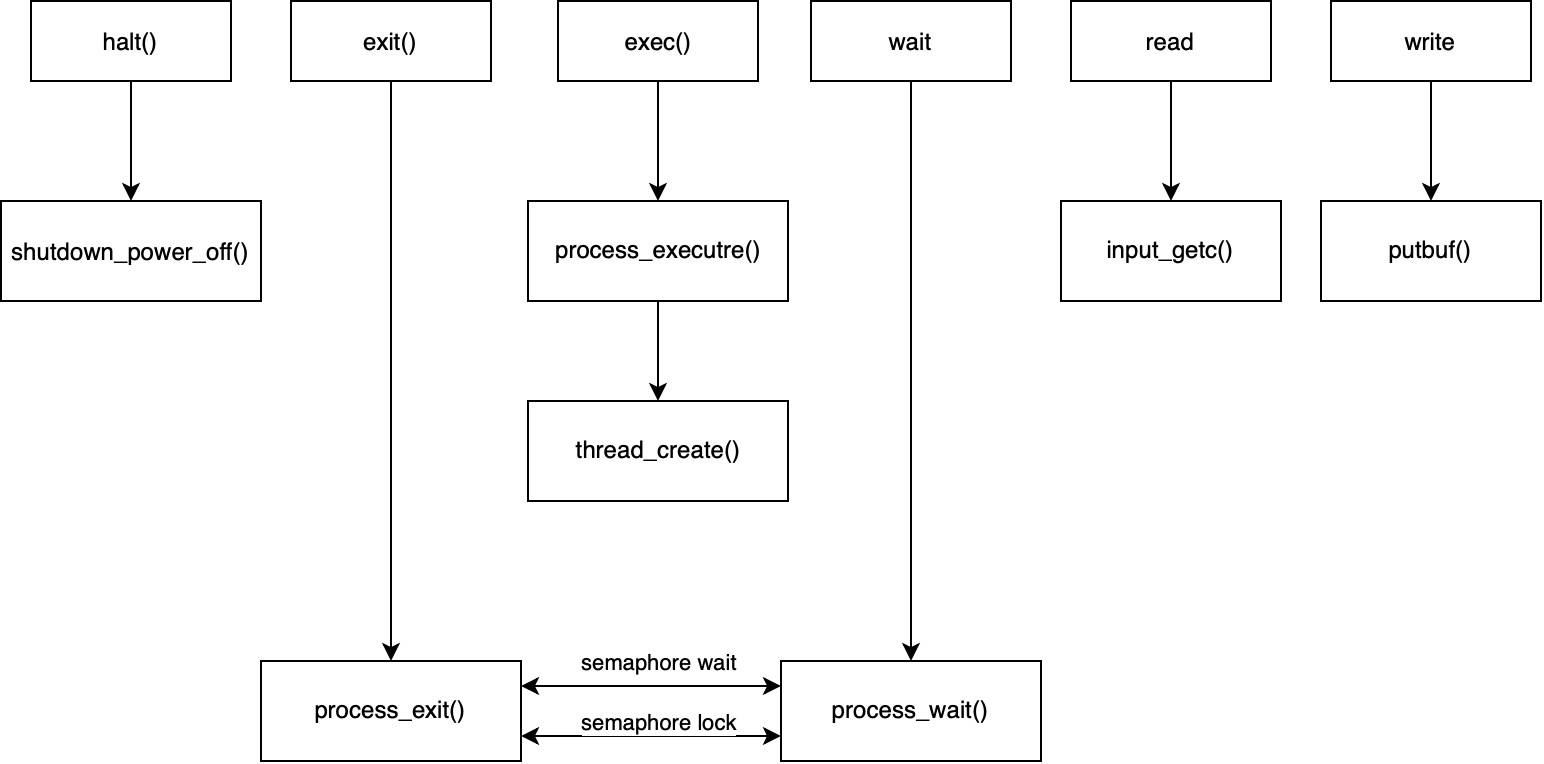
1. User Memory Access



1. System Calls
   * 전체적인 흐름도



* + 각 system call 흐름도



* 1. **제작 내용**

1. Argument Passing

텍스트이(가) 표시된 사진

자동 생성된 설명

pintos/src/userprog/process.c 파일의 load() 함수에서 argument parsing 작업을 하는 코드를 작성하였다. strtok\_r() 함수를 사용하여 공백을 기준으로 argument가 분리되도록 하고자 하였으며, 첫 while문에서 argument의 개수를 세어 argc에 저장하고 두번째 while문을 통해 argc만큼의 길이를 가진 char\*\* 형 배열 argv에 각 argument를 저장하였다. 원래의 file\_name이 손상되지 않도록 fname\_cpy, tmp 등에 file\_name을 복사하여 argument parsing 작업을 수행하였다. argument parsing 작업 후 공백을 기준으로 하여 각 argument는 argv[]배열에, argument 개수는 argc에 저장된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

load() 함수에서는 이후 setup\_stack() 함수를 통해 만들어진 stack에 argument를 passing하기 위한 함수인 construct\_stack() 함수를 호출한다. construct\_stack()에서는 위의 2-B에서 보였던 것처럼 스택에 argument를 거꾸로 쌓는 작업을 수행하게 되며, 초기에 PHYS\_BASE에 위치해 있는 esp를 감소시키며 해당 부분에 argument를 채워 나가게 된다. construct\_stack()의 코드는 다음과 같으며 현재 esp는 PHYS\_BASE 값을 가지고 있는 상태로 시작한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

우선 argv 배열에 저장된 argument들을 stack에 거꾸로 쌓는 작업을 수행하기 위해argc-1 인덱스부터 0번 인덱스까지 반복문을 수행하며 argv 배열의 크기만큼 esp를 감소시키고, 그 공간에 argument의 값을 memcpy를 통해 stack에 저장한다. stack에 argument가 저장된 주소를 저장하기 위해 argv[i]의 값은 현재 esp의 값으로 업데이트 해준다. 모든 argument를 저장하였다면, word alignment를 위해 PHYS\_BASE와 현재 stack pointer의 차가 4로 나누어질 때까지 0을 stack에 넣어 준다. 이후 Null pointer sentinel을 stack에 넣고, 아까 반복문에서 저장했던 argument address들을 순서대로 stack에 넣어 준다. 그 후 argv argument address의 시작 주소를 다시 한 번 stack에 넣어주고, 마지막으로 argc와 return address를 넣어 준다. 이 때 실제로 반환하지 않으므로 return address로는 fake address인 0을 넣어 준다. esp를 감소시킬 때 (char\*) 등을 사용했더니 오류가 발생하여 (uint32\_t\*)를 사용하여서 수행하였다.

위와 같은 과정을 통해 argument passing 작업이 수행된다.

1. User Memory Access

텍스트이(가) 표시된 사진

자동 생성된 설명

Invalid memory access를 막기 위하여, pintos/srs/userprog/syscall.c 파일에 void accessing\_user\_memory(const void\* vaddr) 함수를 선언하였다. 이 함수는 pintos/src/thread.vaddr.h 파일에 정의된 is\_user\_vaddr() 함수를 사용해서 해당 주소가 유효한 주소인지 판단하게 되며, 만약 유효하지 않은 주소라면 -1을 exit system call에 보내 해당 프로그램이 종료되도록 하는 기능을 수행한다.

system call을 수행할 때 각 argument의 주소에 대해 accessing\_user\_memory 함수로 유효성을 체크하여 invalid memory access가 일어나지 않도록 하였다. 예를 들자면 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그런데 test 중 pintos/srs/userprog/exception.c에서 계속 kernel panic 문제가 발생하는 것을 확인하였고, 이를 해결하기 위해 exception.c 파일의 150번 줄에 user가 아니라면, 즉 user program이 kernel 영역을 참조하게 되는 문제가 발생한다면 -1을 exit system call에 보내서 해당 프로그램이 종료되도록 하는 코드를 추가하여 문제를 해결하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. System Calls

system call을 구현하기 위해 pintos/src/userprog/syscall.c의 syscall\_handler() 함수를 완성한다. syscall\_hadler()에서는 인자로 넘어온 intr\_frame 구조체의 esp를 이용하여 해당 system call의 정보가 담긴 stack에 접근한다. 그 후 stack에 저장된 system call number를 switch문의 case로 사용하여 각 system call을 구별하였다. 각 system call의 인자 수에 맞는 argument 개수만큼을 stack에서 가져오고, accessing\_user\_memory() 함수를 통해 해당 포인터들의 유효성을 체크하여 유효한 포인터라면 해당 system call을 호출하도록 하였다. system call이 수행되고 나서 반환값이 있다면 intr\_frame 구조체 f의 eax에 저장하였다.

아래는 syscall\_handler의 일부분으로, 이와 같은 방식으로 코드를 구현하였다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

각 system call은 별도의 함수를 작성하여 구현하였다.

- halt 함수는 별도의 인자를 받지 않는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

system call 내부에서는 shutdown\_power\_off() 함수를 호출하도록 구현하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

- exec 함수에서는 file\_name(cmd\_line) 하나를 인자로 받는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

system call 내부에서는 process\_execute() 함수를 호출하도록 구현하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

process\_execute() 함수에서는 thread\_create() 함수를 통해 새로운 thread를 만들게 되고, 이때 새로 만들어진 process는 현재 process의 child가 된다. 이와 같은 parent-child 관계를 나타내기 위해 pintos/src/threads/thread.h의 thread 구조체에 child\_list와 child\_elem을 추가한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

pintos/src/threads/thread.c의 init\_thread() 함수에서 child list를 초기화하는 작업이 수행되며, thread\_create()에서 list\_push\_back() 함수를 사용(185번 줄)하여 child가 parent의 thread\_list에 들어가도록 설정하였다. 이와 같은 작업을 통해서 parent-child 관계가 제대로 연결되도록 하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

thread\_create() 함수로 process가 제대로 만들어졌다면 해당 process의 id가 반환된다.

- read 함수에서는 file discriptor와 buffer, size의 세 개를 인자로 받는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

system call 내부에서는 file descriptor 값이 0인 경우 STDIN이므로 데이터를 size만큼 읽어 buffer에 저장하는 기능을 수행한다. input\_getc()함수를 통해 standard input을 받는 방식으로 구현한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

- write 함수에서는 file discriptor와 buffer, size의 세 개를 인자로 받는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

system call 내부에서는 file descriptor 값이 1인 경우 STDOUT이므로 buffer의 데이터를 size만큼 출력하는 기능을 수행한다. putbuf() 함수를 통해 standard output을 하는 방식으로 구현한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이 프로젝트에서는 wait과 exit 기능을 구현하기 위해서 semaphore 방식을 선택하였다. 따라서 pintos/src/threads/thread.h의 thread 구조체에 semaphore 구조체 변수 wait과 lock을 추가한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

semaphore 구조체 변수 wait과 lock은 pintos/src/threads/thread.c의 init\_thread() 함수에서 둘 다 0으로 초기화된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

wait은 child가 죽었음을 알리기 위한 semaphore이고, lock은 child가 exit되기 전에 parent의 child\_list에서 제거하기 위한 semaphore이다. semaphore 구조체는 pintos/src/threads/synch.h에 선언되어 있고, 프로젝트에서는 semaphore 구조체의 value element를 이용하여 system call wait과 exit을 구현하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이 프로젝트에서는 pintos/src/threads/synch.c에 저장되어 있는 sema\_down(), sema\_up() 함수를 통해 semaphore 구조체의 값을 변경하였다. 간략히 말하자면, sema\_down()은 semaphore 구조체의 value 값을 감소시키는 역할의 함수이며 sema\_up()은 semaphore 구조체의 value 값을 증가시키는 역할의 함수이다.

- wait 함수에서는 pid 하나를 인자로 받는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

system call 내부에서는 process\_wait() 함수를 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

process\_wait() 함수에서는 현재 thread의 child\_list를 순회하면서 해당 pid를 갖는 child process가 있는지 검색하게 되고, 만약 해당하는 child가 있다면 child가 종료되기까지 기다리도록 한다. child가 존재하지 않는다면 -1을 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

해당하는 child가 죽으면 process\_exit() 함수의 sema\_up(&cur->wait)으로 인한 wait의 값 변화를 통해 알 수 있는데, 그렇게 되면 sema\_down(&child->wait)을 통과하게 되어 child에 대한 제거 작업을 수행하게 된다. 해당 child의 status를 얻고 parent의 child\_list에서 그 child를 제거한 뒤에 sema\_up(&child->lock)으로 lock의 값을 변화시켜 child가 exit될 수 있도록 signal을 준다.

원래는 semaphore 구조체 변수 wait만 구현했었는데, 그렇게 되면 child의 exit status를 얻고 parent의 child\_list에서 제거하기 전에 child process가 exit되어 버려서 test의 수행이 제대로 되지 않는 문제가 발생하였다. 따라서 child에 대한 작업이 완전히 끝난 뒤 child process가 exit을 수행하도록 막아주는 semaphore 구조체 변수 lock을 추가하였다.

- exit 함수에서는 status 하나를 argument로 받는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

system call 내부에서는 현재 process 이름과 status를 콘솔에 출력하고 thread\_exit() 함수를 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

thread\_exit() 함수에서는 process\_exit() 함수를 호출하게 되므로, process\_exit() 함수에서 child가 종료되었음을 알려주는 semaphore를 설정한다.

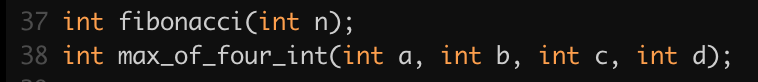
텍스트이(가) 표시된 사진

자동 생성된 설명

process\_exit() 함수에서는 sema\_up(&cur->wait)을 통해 해당 process가 exit될 것이라는 signal을 주고 sema\_down(&cur->lock)으로 기다린다. 이 signal은 위의 process\_wait() 함수에서 받게 되며, process\_wait()함수에서 해당 child가 parent의 child\_list로부터 완전하게 제거되었다면 sema\_up(&child->lock)으로 lock의 값이 바뀌게 되므로 sema\_down(&cur->lock)을 통과하여 exit이 완전히 수행되게 된다.

1. Additional System calls

새로운 system call을 구현하기 위해 우선 pintos/src/lib/user/syscall.h에 fibonacci()와 max\_of\_four\_int()의 system call API prototype을 작성하였다.



기존의 system call은 argument를 3개까지 받을 수 있었기 때문에 max\_of\_four\_int()에서 4개의 argument를 받을 수 있도록 해 주기 위해서 pintos/src/lib/user/syscall.c에서 syscall4로 argument를 4개 받는 형태를 작성해 주었다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

또한 같은 pintos/src/lib/user/syscall.c 파일에 fibonacci()와 max\_of\_four\_int()의 system call API를 작성하였다.

**텍스트이(가) 표시된 사진

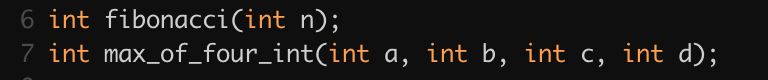
자동 생성된 설명**

이후 pintos/src/lib/syscall-nr.h 파일에서 system call의 번호를 저장하는 enum 객체에 fibonacci()와 max\_of\_four\_int()를 추가하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

pintos/src/userprog/syscall.h 파일에는 fibonacci()와 max\_of\_four\_int()의 prototype을 작성하였다.



pintos/src/userprog/syscall.c 파일에서 fibonacci()와 max\_of\_four\_int()가 수행하는 기능을 작성하였고, 같은 파일의 system\_handler()에서 switch문을 통해 두 system call을 호출할 수 있도록 하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이때 system\_handler()에서 system call 호출 시 fibonacci()와 max\_of\_four\_int()에서 n과 a, b, c, d가 저장된 주소가 유효한 주소인지 판단하는 작업을 수행하여 system call을 수행하는 동안 invalid memory access가 없도록 하였으며, 두 system call의 반환값은 eax에 저장하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

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

**텍스트이(가) 표시된 사진

자동 생성된 설명**