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

담당 교수 : 박성용

조 / 조원 : 오세정

개발 기간 : 2022.09.20~2022.09.30

1. **개발 목표**

* 이번 Pintos project에서는 Argument Passing을 통한 명령어와 각 명령어에 필요한 인자를 분리하는 작업을 하고, Kernel과 User 영역의 메모리를 분리하여 System Call이 문제없이 작동되도록 한다. 이때 구현하는 System Call은 exec(), exit(), write(), read(), halt(), wait()가 된다.

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

Pintos 명령을 통해서 들어온 명령어와 인자가 분리된 뒤에, 각 명령어와 인자, 그리고 주소값들이 esp를 통해서 Stack에 저장된다. 이는 나중에 system call handler에 의해 참조된다.

1. User Memory Access

is\_user\_vaddr() 함수를 통해서 참조하고자 하는 주소가 참조 가능한 주소인지 (Kernel 영역의 주소를 참조하고자 하는 것은 아닌지) 확인하여 Kernel 영역을 보호한다.

1. System Calls

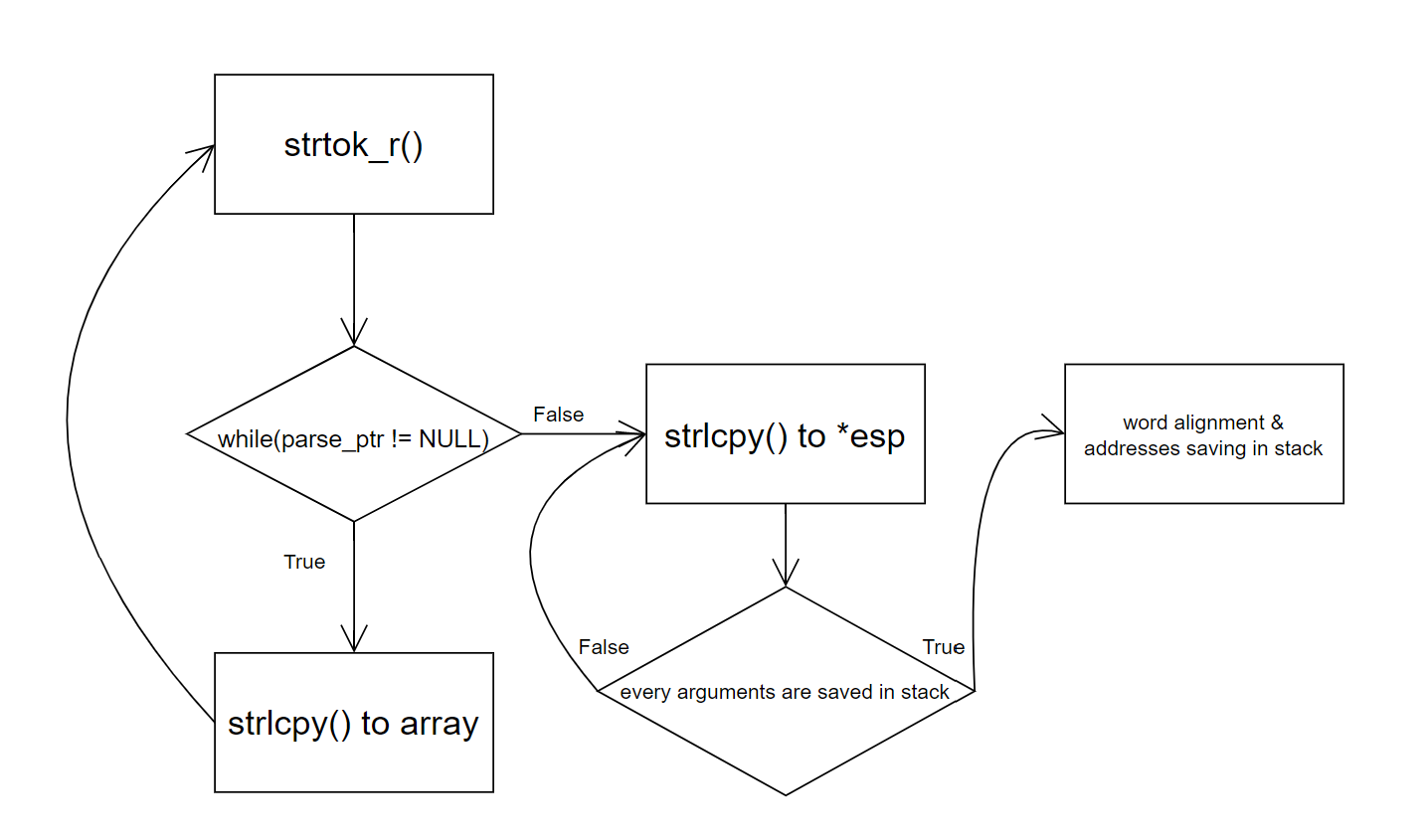
System call 을 구현함으로써 User 영역과 Kernel 영역을 연결해 줄 수 있다. 이때 중간 매개체는 esp로 쌓은 스택이 되며, 각 명령어에 필요한 system call들을 실행시킬 수 있게 된다.

* 1. **개발 내용**
* Argument Passing
  + **커널 내 스택에 argument를 쌓는 과정 설명**
  + 먼저 각 string은 널 문자를 포함해야 하므로 strlen() 함수로 구한 문자의 길이에 널 문자까지 포함하여 저장하기 위해 1을 더하여 esp 포인터에 저장한다.
  + 이때 argv[2] argv[1] 의 형태와 같이 가장 마지막 인자부터 스택에 쌓으며, 모든 인자의 값을 스택에 쌓은 뒤 각 인자의 주소와 인자의 개수 등도 스택에 쌓아준다.
  + 모든 인자의 값을 쌓아준 다음, word\_align을 위해서 4의 배수로 맞춰주며, 빈 공간은 0의 값을 넣어준다. 그 후 인자의 주소와 argv의 주소, argc의 값을 넣어주기 위해 uint32\_t의 데이터 타입으로 저장해준다. 이는 크기를 고정시켜 word align을 용이하게 한다.
* User Memory Access
  + **Pintos 상에서의 invalid memory access 개념을 간략히 설명**
  + Pintos는 User 영역과 Kernel 영역을 구분하여 명령을 수행한다. 이때, 다른 두 Application이 서로의 영역을 침범할 수 없도록 Virtual Address를 사용하고, 각 Application이 Kernel의 메모리를 침범할 수 없도록 보호한다. 이때 보호를 해주지 않아 User 영역에서 실행되고 있는 Application이 Kernel의 영역을 참조하거나 수정하려고 하면 invalid memory access가 일어난다.
  + **Invalid memory access를 어떻게 막을 것인지 설명**
  + 사전에 구현되어 있는 is\_user\_vaddr() 함수를 사용하여 address\_protect()라는 함수를 구현한다. 이 address\_protect() 함수는 인자로 들어온 주소가 kernel 영역인지 아닌지를 검사하며, 만약 kernel 영역이라며 exit(-1)을 호출하여 application을 종료한다.
* System Calls
  + **시스템 콜의 필요성에 대한 간략한 설명**
  + User program 에서 바로 커널 영역의 메모리에 접근하게 되면 많은 문제가 발생한다. 의도치 않게 메모리를 침범하게 되면 kernel panic이 일어날 수 있고, kernel 자체가 망가져 버리는 일이 발생할 수도 있다. 따라서 User program에서 안전하게 kernel의 API를 활용하기 위하여 stack을 통해 값을 전달하고 받는 system call이 필요하다.
  + **이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)**
  + Halt() : shutdown\_power\_off() 함수를 호출하여 system을 종료시킨다.
  + Exit(): 현재 실행중인 file의 이름과 exit status를 출력한다.
  + Wait(): 현재 실행중인 child process가 종료되기를 wait 한다.
  + Exec(): 인자로 받은 file\_name에서 명령어를 파싱해 실행하고 실패시 -1을 리턴한다.
  + Read(): 이번 프로젝트에서는 STDIN을 통해 들어오는 값을 읽어들인다.
  + Write(): 이번 프로젝트에서는 STDOUT을 통해 전달된 buffer의 값을 write한다.
  + **유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명**
  + 예를 들어 User API에서 system call을 실행하면, 해당 system call에 넘겨지는 인자의 개수에 맞춰서 syscall1, syscall2, syscall3을 실행시킨다. 그 후 intr\_handler가 동작한 뒤, syscall\_handler가 동작하여 해당 system call의 number를 확인하여 stack에 쌓인 값들을 추출하여 Kernel API로 전달한다. 그 후 return 값은 eax 레지스터에 저장하여 user API로 전달한다.

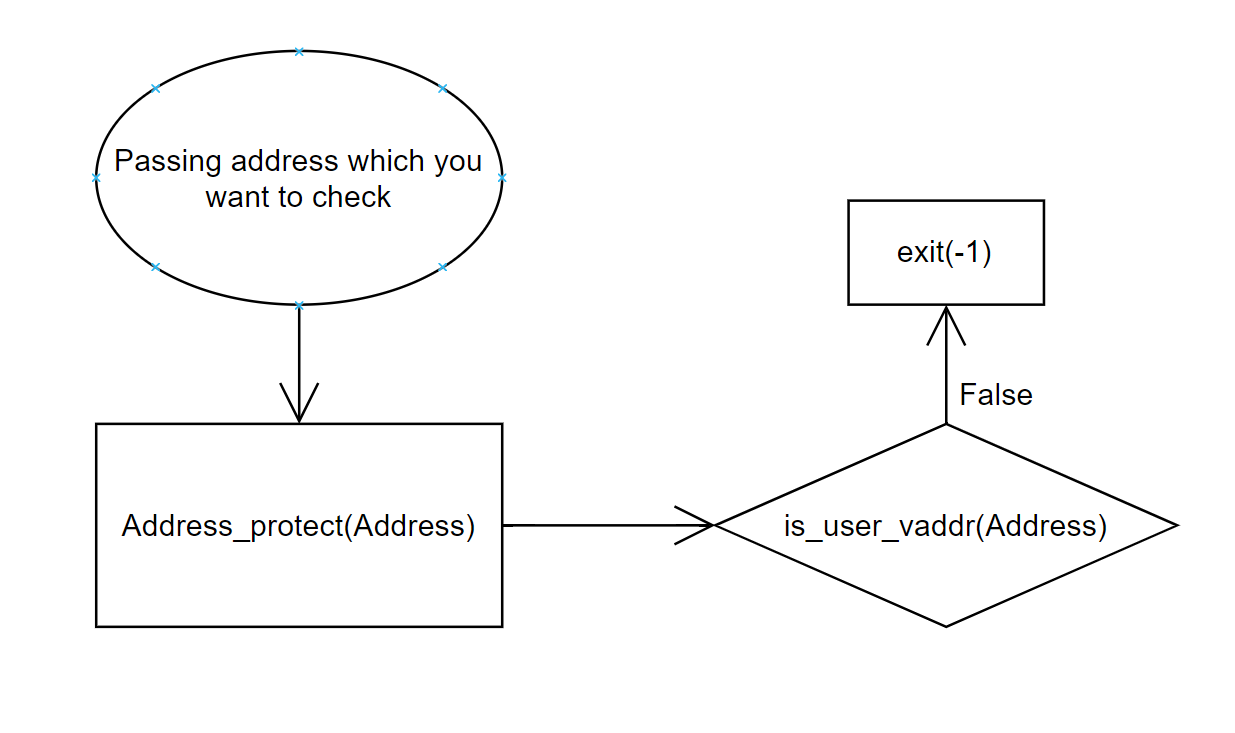
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* Argument Parsing : 2022.09.20~2022.09.23
* User Memory Access : 2022.09.24
* System Call Handler : 2022.09.25~2022.09.30
* 기본적인 추진 일정은 위와 같지만, 가장 마지막 구현 단계였던 System call handler를 진행하며 Argument Parsing 후 Stack에 데이터를 저장할 때 부분이나, User Memory Access 보호를 어느 지점에서 확인할지 등등 동시다발적으로 구현하며 진행하였다.
  1. **개발 방법**
* Argument Parsing
  + process.c에 load() 부분에 Parsing 하는 부분과 이후 esp를 통해 stack에 값을 저장하는 부분을 구현한다.
  + strtok\_r 함수와 strlcpy, strlen 함수등을 이용해서 구현한다. 이때, 기존 인자로 넘겨준 file\_name의 원본은 보호하기 위해 복사하여 parsing 하도록 한다.
  + 이후 stack에 쌓을 때 esp의 값을 줄여가며 값을 쌓는다. 이는 pintos에서 stack은 주소값이 작아지며 stack이 쌓이기 때문이다. 이때 strlcpy를 통해서 널 문자까지 잘 복사되도록 하고, 해당 값이 저장되는 주소를 다른 배열에 저장하여 이후 스택에 쌓아주도록 한다.
  + Word\_align을 위해서 스택에 쌓은 총 길이를 저장하고, 길이에 따라 4의 배수로 맞춰준다. 그 이후 uint32\_t의 사이즈로 크기를 고정해주고 esp에 주소와 argv, argc의 주소를 저장한다.
* User Memory Access
  + Syscall.c의 syscall\_handler()에서 참조하는 주소의 값이 kernel 영역을 참조하고 있는지 확인한다.
  + address\_protect(const void \* vaddr)에서 is\_user\_vaddr(vaddr)를 체크하여 kernel 영역을 참조하고 있을 경우 exit(-1)을 호출한다.
  + 해당 함수를 통해 syscall\_handler에서 먼저 switch 문으로 들어가기 전, address\_protect(f->esp)로 esp가 커널 영역을 참조하고 있는지 확인한다. 그리고 각 system call을 호출할 때에도 각 참조하는 주소를 전부 address\_protect()로 invalid memory access가 일어나지 않도록 체크한다.
* System Call Handler
  + Halt(): shutdown\_power\_off()를 호출한다.
  + Exit(): 먼저 thread\_current()를 통해서 현재 thread에 대한 정보를 받고, status를 현재 thread에 exit\_status에 저장한다. 그 후 thread의 name에서 process 이름만 파싱하여 exit에 대한 정보를 print 해준다. 그 후 thread\_exit을 통해서 exit()을 수행한다.
  + Wait(): 인자로 받은 pid를 process\_wait으로 넘겨 wait을 수행한다. process\_wait() 함수의 경우 전달받은 tid를 통해 현재 thread의 child 들을 전부 탐색하며 해당 child의 tid와 인자로 전달받은 tid가 동일한 child를 찾는다. 이때 해당 child의 status를 받아오고, list에서 지우며, semaphore를 통해서 synchronization이 꼬이지 않도록 구현한다.
  + Exec(): 전달받은 file\_name을 파싱하여 실행할 파일의 이름을 구하고, filesys\_open을 통해서 해당 파일을 실행한다. 만약 없는 파일을 실행하려 하거나, 파일 오픈 시 문제가 생기면 -1을 return 한다. 그 후, process\_execute() 함수를 호출한다.
  + Write(): 인자로 파일 디스크립터와 쓸 string이 담겨있는 buffer와 해당 buffer의 길이인 length가 넘겨진다. 이때, 해당 프로젝트1에서는 STDIN, STDOUT을 통한 입출력만을 구현하므로 STDOUT의 fd number인 1임을 check 하고 putbuf 함수를 통해서 write를 구현한다.
  + Read(): 인자로 파일 디스크립터와 읽을 string이 담겨있는 buffer와 해당 buffer의 길이인 length가 들어온다. Read() 역시 이번 프로젝트1에서는 STDIN, STDOUT만 고려하므로 fd number가 0임을 check하고 NULL 문자를 읽어들일 때까지 buffer를 읽는다.

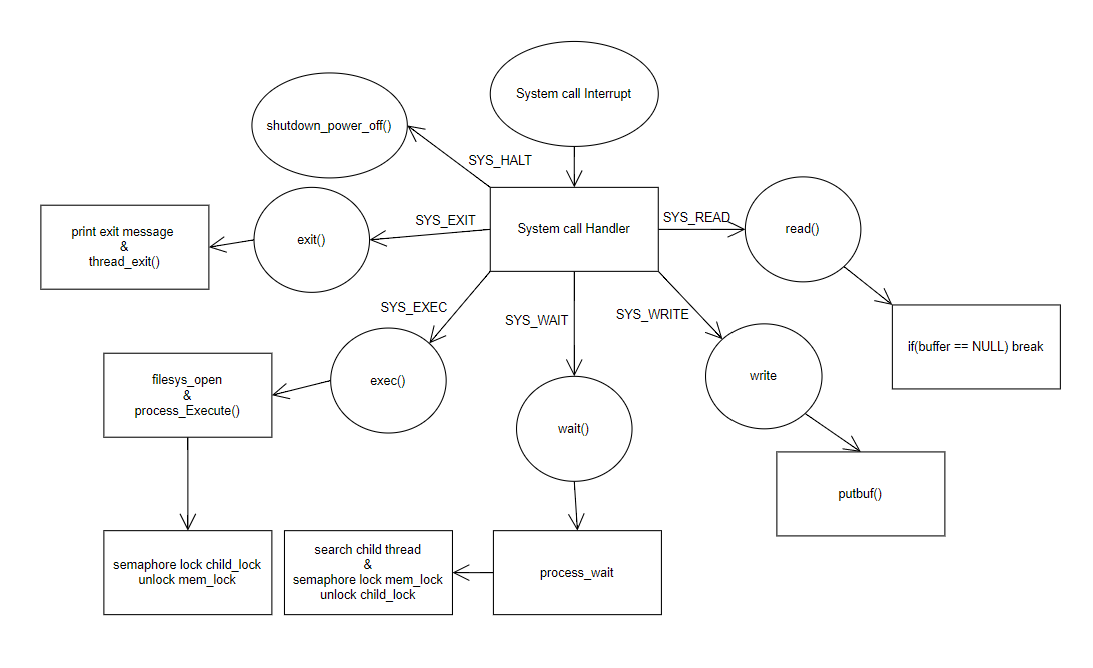
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



1. System Calls



* 1. **제작 내용**

1. Argument Passing
   * 먼저 passing 된 file\_name을 cp\_fn이라는 변수에 저장해서 parsing 시 원래 문자열에 생길 수 있는 문제를 방지한다. strlcpy() 함수를 통해서 복사하고 strtok\_r() 함수를 통해 tokenize 한다. 이때 기준은 “ “로 매 iteration마다 parse\_filename[] 이라는 배열에 인자를 저장한다.
   * 이후 해당 값을 stack에 저장해야 하는데, 이때 esp 포인터를 통해서 저장한다. Strlen(parse\_filename[idx])+1 과 같이 NULL 문자 역시 stack 영역에 저장되도록 1을 더해서 저장한다. 그리고 해당 값이 저장된 stack의 주소를 arg\_addr[] 배열에 저장한다. 이는 추후 esp 스택에 또한 역순으로 주소값을 저장하기 위함이다.
   * Word\_align은 4의 배수가 되도록 각 인자가 저장될 때 길이를 더하고, 총 길이가 4의 배수가 되도록 word\_align을 맞춘다. 그 후, argv, argc 주소를 전부 esp 스택에 저장한다.
2. User Memory Access
   * Is\_user\_vaddr() 함수를 사용해서 참조하고 있는 해당 영역이 kernel의 메모리인지 아니면 user 영역의 메모리인지를 판별한다.
   * 해당 is\_user\_vaddr 함수는 threads/vaddr.h 에 정의되어 있으며, PHYS\_BASE라는 User 영역의 시작 주소와 현재 인자로 들어온 주소값을 비교하여 더 클 경우 kernel 영역의 주소를 참조하는 것이 되므로 false를 return 한다.
   * 그 후, 해당 주소가 kernel 영역을 참조하는 것이 되면, 비정상적인 접근이므로, exit(-1) 함수를 호출하여 프로그램을 강제 종료시킨다.
   * Exit() 함수는 아래 system call에서 자세히 설명하겠다.
3. System Calls

* 먼저 이번 프로젝트에서 구현한 system call은 halt(), exit(), wait(), exec(), read(), write()과 같이 총 6개이다.
* Syscall\_handler의 인자로 넘어오는 intr\_frame 구조체 f를 통하여서 esp주소를 참조할 수 있고, 이를 통해 해당 system call의 NUMBER를 알아서 적절한 함수를 호출할 수 있다. 또한, 각 시스템 콜마다 필요로 하는 인자가 다르기 때문에 esp 포인터에 적절한 숫자를 더해줌으로써 해당 값에 접근할 수 있다. 이때 참조하는 모든 영역을 address\_protect() 함수로 감싸줘서 커널 영역으로의 비정상적인 접근은 일어나지 않는지 검사한다.
* Halt, read, write의 경우 위에서 언급한대로 구현하면 되기 때문에 생략하도록 하겠다.
* 먼저 exit의 경우, 인자로 status 값을 넘겨받는데, 이는 현재 thread, 곧 프로세스의 상태가 저장되어 있는 값이다. 따라서 해당 status 값을 thread의 exit\_status에 업데이트 해준다. 현재 thread에 대한 정보는 thread\_current() 함수를 통해서 얻어올 수 있다.
* 또한 thread->name 을 참조하여서 현재 실행되고 있는 process의 이름을 가져올 수 있는데, 이때 해당 이름은 parsing이 되어있지 않은 상태이므로, 가장 앞에 있는 실행 파일의 이름(곧 명령어)을 추출하기 위해 while문을 사용ㅎ지 않고 한번만 파싱해준 다음에 해당 내용과 status를 정리하여 stdout으로 출력해준다. 그리고 난 뒤, thread\_exit() 함수를 호출하여 thread를 제거한다.
* 그 다음 exec()의 경우, 마찬가지로 인자로 넘어온 file\_name이 파싱이 되어있지 않은 상태이므로, 한번만 파싱하여 실행하고 있는 명령어, 곧 파일 이름을 추출해준다. 그리고 난 뒤, 해당 이름으로 filesys\_open을 호출하여 해당 파일을 오픈할 수 있는지 확인한다. 만약, 오픈할 수 없다면 -1을 리턴하여 실행이 불가함을 알린다.
* 만약 제대로 실행이 되었다면 process.c에 정의되어 있는 process\_execute() 함수를 실행하여 해당 파일이름으로 프로세스를 실행시킨다.
* Process\_execute() 함수는 사전에 pintos에 내장으로 구현되어 있는 함수로 먼저 palloc\_get\_page() 함수를 통하여서 필요한 메모리를 할당한다. 그리고 난 뒤, strlcpy() 함수를 호출하여 현재 인자로 넘어온 file\_name의 값을 복사해주고, 해당 이름으로 thread\_create()를 호출하여 쓰레드를 생성한다. 이때 인자로 start\_process() 함수를 같이 넘겨주어, 최종적으로 load 함수가 실행되도록 한다. 모든 과정이 마무리 된 다음에는 palloc\_free\_page() 함수를 통하여서 사전에 복사를 위해 할당해놓은 메모리를 free 해준다.
* 마지막으로 wait 시스템 콜의 경우에는, process\_wait() 함수를 바로 호출하게 된다. 그리고 해당 함수가 호출된 뒤에는 현재 thread의 child 리스트를 확인하며 인자로 넘어온 child의 thread의 id와 동일한 id를 가진 child를 찾는다.
* 이때 해당 child의 tid와 인자로 넘어온 child\_tid가 동일한 thread를 찾았을 경우, 해당 thread의 child 상태를 변경하기 위해 sema\_down을 실행해준다. 이 과정으로 인해 다른 쓰레드들과의 공유자원을 안전하게 사용할 수 있게 된다. 그리고 난 뒤에, exit\_status를 업데이트 해준 뒤, list에서 child를 제거해준다. 그 후, 해당 child가 가지고 있던 memory에 대해 공유자원 사용이 끝났으므로 sema\_up을 메모리 자원에 대한 semaphore에 해준다.
* 그리고 난 뒤 exit\_status를 리턴해준다. 사전에 exit\_status의 값을 -1로 설정함으로 만약 for문을 돌며 해당 child를 찾지 못했을 때는 -1을 리턴하도록 한다.

1. Additional System calls

* 구현하지 못함
  1. **시험 및 평가 내용**
* **구현하지 못함.**