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

담당 교수 : 김영재

조 / 조원 :20190388 백상욱

개발 기간 :2023/09/18 ~ 2023/10/03

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

유저프로그램은 파일을 열거나 쓰고, 새로운 파일을 실행하고 메모리를 할당하는 등 많은 시스템적인 기능을 활용하고자 한다. 하지만 이는 유저 공간에서 직접 이루어지는 것이 아닌 커널 공간에서 이루어지게 된다. 이를 위해서 유저는 시스템 콜을 이용하고 이 시스템콜을 통해서 커널 혹은 운영체제는 유저가 필요로하는 기능을 수행한다. 하지만 현재 핀토스는 유저가 입력한 명령어나 프로그램 구체적으로는 그 안에 있는 시스템콜을 수행하지 못한다. 이는 현재 핀토스에 시스템 콜, 시트템 콜 핸들러, argument passing, 유저 스택이 온전히 구현되어있지 않고 메모리 이슈등의 문제들에 대한 예외처리가 되어있지 않기 때문인데, 이번 개발에서는 이를 수행가능하도록 직접 argument parsing, passing, 유저 메모리 스택 구현, 시스템 콜 구현 그리고 다양한 메모리 예외처리 등을 구현하고 개발하는 것이 목표다.

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

커널의 쓰레드는 유저 프로세스를 로드하고 실행하는 역할을 한다. 그리고 로드하는 과정에서 유저 프로세스에 대한 페이지 테이블을 생성하고 파일을 열고 각 세그먼트의 파일을 읽어온 다. 이후에는 유저의 스택을 생성 후 초기화하고 주요 인자들을 저장한다. 이때 저장하는 인자들을 이직접 파싱하고 패싱해서 저장해주는 것의 argument passing의 구현이다. 현재 핀토스는 입력받은 커맨드 라인 자체를 하나의 파일로 받아들이고 이를 구분하지 않은 채로 바로 파일을 오픈할 뿐 아니라 커맨드의 주요한 요소들을 유저 스택에 집어넣지 않는다. 이러헥 되면 결과적으로 유저프로그램이 실행되더라도 원하는 방향으로 작동하지 않게 되므로 결국 커맨드로 입력한 파일 또는 명령을 실행하려면 이를 유저 스택에 넣어줘야하는데 이를 구현해야한다.

1. User Memory Access

시스템 콜을 수행하는 것은 커널 공간에서 이루어진다. 즉 커널 공간에서 유저의 정보를 접근 한 후 이를 기반으로 시스템 콜을 수행해줘야한다. 이때 유저의 정보를 접근 하는 방법으로 커널이 유저의 가상 메모리 주소를 참조하고 접근하게 된다. 이를 구현하는 것은 쉽지만 중요한 지점은 주소를 참조했을 때 이 주소가 유요한지 검증하는 부분이다. 현재 접근한 유저의 가상 메모리 주소가 사실 가상 메모리 속에서 유저 영역이 아닌 kernel영역 (PHYS\_BASE이상의 주소)에 속하는지 첫 번째 확인해야하며 두 번째 user 영역에 존재하는 가상 주소이지만 이를 물리 주소로 변경했을 때 물리주소 중 Kernel영역에 속하지 않은지 또는 유요하지 않은지를 확인하여 이 경우 종료하도록 예외처리를 구현해야한다.

1. System Calls

현재 핀토스는 시스템콜 핸들러가 구현되어있지 않다. 시스템콜을 호출은 하지만 시스템콜 핸들러 내부를 보면 system call!이라는 출력만 나오고 그외에 작동은 하지 않는다. 내부의 핸들러를 직접 구현하여 프로세스가 정상적으로 작동하도록 만들어야한다. 구체적으로 이번과제에서 구현해야하는 핸들러는 halt, exit, exec, wait, read, write이며, read, write는 stdin, stdout에 대해서만 구현한다. 추가적으로 fibonacci, max\_of\_four\_int 시스템콜을 구현한다.

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

우선 명령어를 띄어쓰기를 기준으로 파싱하고 파싱한 결과를 문자열 배열에 따로 저장한다. 파싱에 대한 내용은 이후 파트에서 설명한다. 이 파싱한 결과를 저장한 배열을 가지고 문자열과 기타 내용을 스택에 넣어줘야하는데, 스택에 넣어주는 방법은 다음과 같다. 우선 유저의 가상 주소의 경우 핀토스는 총 4GB를 가지고 있다고 보며 이중 PHYS\_BASE이상의 영역은 커널 주소로 본다. 즉 우리는 PHYS\_BASE 미만의 영역부터 채워야한다. 스택은 아래로 차기 때문에 PHYS\_BASE-1 부터 아래로 채워나간다.   
stack을 채워나가는 방법은 "echo x"라는 명령을 줬을 때를 기준으로 설명한다.  
우선 parsing을 했다는 가정하에 echo, x 두개의 문자열로 분리가 되어있다.  
stack에 삽입할 때는 뒷 문자부터 앞으로 집어넣는다.  
즉 echo x지만 stack에 넣을 때는 x o h c e로 집어넣는다.  
결과적으로 echo x가 더 낮은 스택 주소부터 더 높은 스택 주소로 집어넣게 된다. 추가로 x와 echo 각각은 null 문자를 반영하여 2byte, 5byte씩 차지하게 된다.  
결과적으로 0xbffffffe :x ,0xbfffffff : \0  
0xbffffff9 : e , 0xbffffffa : c, 0xbffffffb : h, 0xbffffffc : o , 0xbffffffd : \0  
와 같이 stack에 저장된다.  
이후 문자열에 대해서 word align을 맞추기 위한 바이트가 들어가는데  
앞서 보면 2+5 = 7 byte를 차지하고 4byte align을 맞추기 위해서는 1byte가 필요하다. 이를 위해 1byte만큼 stack에 null 또는 0을 채워준다.  
0xbffffff8 : \0  
이후에는 각 문자열들의 시작 주소를 담아줘야한다. (argv)

이전에 문자열을 저장하는 것과 이들의 주소를 저장하는 부분을 구분하기 위해서 중간에 sentinel을 4byte넣어준다.  
0xbffffff4 ~ 0xbffffff7 : \0   
그리고 이후부터는 각 문자열이 시작된 주소를 4byte씩 입력해준다 즉  
0xbffffff0 ~ 0xbffffff3 : 0xbffffffd 를 1byte씩 집어넣음  
0xbfffffec ~ 0xbfffffef : bfffffff9 를 1byte씩 접어넣음  
이를 다 집어넣은 후 이 argv의 시작 주소를 4byte로 넣어줘야한다.  
결과적으로 0xbfffffe8 ~ 0xbfffffeb : 0xbfffffec를 1byte 씩 집어넣는다  
이후에는 argc, 즉 argument의 개수를 4byte로 집어넣어준다.  
위의 경우 echo x는 argc가 2이므로 0xbfffffe4 ~ 0xbfffffe7 : 2를 저장한다.  
마지막으로 return address를 0으로 하여 4byte를 추가한다.   
0xbfffffe0 ~ 0xbfffffe3 : 0을 넣는다. 결과적으로는 stack이 다음과 같이 구성된다.  


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

보통 invalid memory access는 두가지로 볼 수 있다. 가상메모리 상에서 그리고 실제 물리적 주소 상에서 발생하는 경우이다. 유저 프로세스는 가상 메모리를 사용하여 작동한다. 이때 4GB가 할당되어있다고 가정하면 0x00000000 ~ 0xcfffffff까지의 모든 영역을 사용할 수 있는게 아니다. kernel address영역이 특정되어있고 이 부분은 kernel이 사용하는 영역으로 user가 접근할 수 없다. 하지만 이 영역에 접근했다면 이는 invalid한 메모리 접근이다.  
두 번째는 물리적 주소, 정확히는 페이지 주소에 대한 잘못된 접근이다. 물리적 주소 또한 kernel 영역과 user 영역이 지정되어있다. 가상 주소를 물리적 주소를 접근했을 때, 이것이 kernel영역에 속한다는 등의 이유로 물리적 주소로 맵핑되지 않는 경우에 이는 잘못된 접근이다.

* + Invalid memory access를 어떻게 막을 것인지 설명

앞서 invalid memory access가 발생하는 경우를 살피면 이를 막는 방법은 쉽게 알 수 있다. 첫 번째 가상 메모리의 경우 해당 주소가 PHYS\_BASE(kernel영역과 user 영역의 경계)보다 크보다 같은지 확인한다. 이는 is\_kernel\_vaddr라는 함수로 구현되어있다. 이를 통해 만약 이보다 크다는 것이 확인되면 exit(-1)을 한다.  
두 번째 물리적 주소에 대해서는 해당 가상 주소에 대해서 mapping되는 user physical address가 존재하는지 확인해주면 되는데, 이는 pagedir\_get\_page라는 함수를 통해서 확인한다. return이 null이라면 맵핑되는 주소가 없다는 뜻이므로 invalid memory access이므로 exit(-1)을 한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명  
    유저 프로그램은 직접 커널에 존재하는 메모리나 코드에 접근할 수없다 하지만 원하는 작동을 하기 위해서는 커널이 제공하는 함수 및 작동을 사용해야할 때가 있다. 이때 직접적으로 이를 사용하는 것이 아닌 커널에게 요청하여 커널이 이를 작동하도록 해야하는데, 이 커널에 요청하는 것이 시스템콜이기 때문에 시스템 콜이 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 : 핀토스 운영 체제를 아예 꺼버리는 시스템 콜로 함수 인자나 반환값이 없다.  
    exit : 현재 돌아가고 있는 스레드를 종료(exit)하는 시스템콜로 인자로 스레드의 status를 받고 스레드의 이름과 이 status를 함께 출력해야한다.  
    exec : 인자로 실행 파일의 이름을 받는다. 그리고 인자로 받은 실행 파일을 새로운 프로세스로 만들어 실행시킨다. return으로는 만들어진 프로세스의 Pid를 반환하며 exec가 실패한다면 -1을 반환한다.  
    wait : 자식 프로세스의 pid를 인자로 받으며 해당 프로세스가 종료될때까지 기다린다. 종료되었다면 종료될때의 status를 알수 있어야한다.  
    read : 파일 디스크립터, 버퍼, size를 파라미터로 받는다. 파일디스크립터로 들어온 내용을 size만큼 읽어서 버퍼에 저장하며 읽은 만큼을 반환한다. 이번 과제에서는 stdin에 대해서만 구현한다.  
    write : 파일디스크립터, 버퍼, size를 파라미터로 받는다. 버퍼에 담긴 내용을 size만큼 파일디스크립터에 작성한다. 그리고 사이즈를 반환한다. 이번 과제에서는 stdout에 대해서만 구현한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명  
    우선 user는 프로그램을 실행하면서 필요한 시스템콜을 호출한다. 이렇게 되면 pintos/src/lib/user/syscall.c 경로 안에 있는 관련되어있는 시스템 콜이 불리게 된다. 이때 각 시스템 콜 내부에는 파라미터 개수에 맞는 syscall을 syscall에 해당하는 system number를 넣어서 호출된다. 가령 halt() 시스템 콜은 파라미터가 없으므로 syscall0(SYS\_HALT)을 호출하고 exit은 파라미터가 하나이므로 syscall1(SYS\_EXIT, status)로 호출을, write은 syscall3(SYS\_WRITE, fd, buffer, size)로 호출한다. 이때 각 syscall#은 인자로 넘어온 system call넘버, 와 파라미터들을 유저 스택에 집어넣는다. 이때 파라미터와 system call넘버를 역순으로 집어넣는다. 가령 syscall3(SYS\_NUMBER,arg0,arg1,arg2)라고 한다면 arg2 -> arg1 -> arg0 -> SYS\_NUMBER순으로 스택 주소를 내리면서 집어넣는다. 이렇게 집어넣음으로써 커널단에서 해당 정보를 접근할 때 순차적으로 SYS\_NUMBER, arg0, arg1, arg2 순으로 꺼내올 수 있게 된다.  
    이후에은 int $0x30을 호출한다. 이는 인터럽트를 거는 부분으로 커널로 하여금 시스템 콜 핸들러를 수행하도록 만들어준다. 이 지점부터는 커널이 수행한다. 커널은 인터럽트를 걸렸기 때문에 시스템콜 핸들러를 수행한다.  
    이는 pintos/src/userprog/syscall.c에 있는 시스템콜 핸들러 함수를 부르는 것으로 수행된다. 커널이 작동하는 이 함수는 intr\_frame 구조체 포인터를 통해 user stack pointer를 접근할 수 있다. 이를 통해 현재 유저스택의 정보를 확인할 수 있고 결과적으로 앞서 집어넣었던 SYS\_NUMBER과 필요한 argument를 꺼내올 수 있다. 이 SYS\_NUMBER에 따라 각각 시스템콜 핸들러를 구현해준다. 이때 시스템 콜 핸들러의 Return값이 있는 경우는 이 값을 intr\_frame 구조체의 eax에 집어넣어준다. 이를 통해 userprogram은 system call handler의 결과값을 알 수 있게 된다. 이런식으로 유저 프로그램은 시스템콜을 호출하고 커널은 이를 수행한 후 결과를 다시 유저에게 전달해주는식으로 구동한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

09/18 ~ 09/27까지는 기존의 내용과 메뉴얼, project 1 ppt 내용을 숙지  
09/28 : argument parsing/passing 구현

09/29 : write, read system call 구현  
09/30 : halt, exit, exec system call 구현  
10/1 : wait system call 구현  
10/2 : wait system call 구현, exec 수정  
10/3 : invalid memory access 및 여러가지 예외 처리에 대해서 추가 구현

* 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)**

argument passing을 할 때는 그에 앞서 argument를 parsing해야한다. parsing에 대해서 얘기하자면 띄어쓰기를 기준으로 나눠주고 이를 각각 저장해주는 것과 띄어쓰기를 기준으로 첫 번째 단어가 열고자하는 파일의 이름이기 때문에 이를 filesys\_open에 넣어주기 위해서 다른 문자열에 저장하는 역할을 한다. 이를 구현하기 위해서 char real\_file[128], char \*token, char \*token\_ptr, char\* token\_arr[32]를 선언했다. real\_file은 filesys\_open의 파라미터로 줄 첫 번째 토큰, 가령 echo x라면 echo, additional 1 2 3 4라면 additional에 해당하는 부분을 저장한다. token\_arr는 띄어쓰기를 기준으로 잘라놓은 각 토큰을 저장하는 문자열 배열이다. 그리고 token 과 token\_ptr은 각각 띄어쓰기를 기준으로 파싱하기 위해서 추가로 사용할 포인터이다.

그리고 이 포인터들을 직접 1byte씩 순회하면서 앞서 말한 방법으로 Stack에 삽입해준다.   
User Memory Access를 하는 개념 자체는 간단하다 추가의 구현 없이 intr\_frame의 변수인 esp를 참조하면 이것이 user의 스택포인터이기 때문에 바로 접근할 수 있다. 다만 이때 esp로 참조한 주소가 invalid한지 확인하기 위한 함수가 필요하다. 이때 invalid한 2가지 경우를 모두 검사하는 check\_addr\_valid라는 함수를 만든다. 이 함수내부는 가상주소에서의 invalid access를 검사하기 위해 PHYS\_BASE 이상의 주소를 갖는지 확인하고 두 번째 물리주소에서 invalid access를 검사하기 위해서 pagedir\_get\_page함수를 사용하는데, 이의 구체적인 내용은 4.B에서 설명한다.  
System Call은 기본적으로 syscall.c 내부의 syscall\_handler 함수의 코드를 직접 구현해야한다. 이를 구현하는 방법은 다음과 같다. 우선 파라미터로 들어온 intr\_frame 구조체의 esp를 확인하여 유저의 가상 주소로 접근한다. stack\_pointer가 당장 가리키고 있는 주소 4byte에는 syscall\_number에 대한 정보를 가지고 있으므로 그 값을 syscall\_number로 저장하고 이를 switch문에 사용한다. 그리고 syscall\_number에 따라 case를 halt ,exit, exec, wait, read, write인 경우로 나눠준다.

추가로 위의 함수들을 구현할 때 halt는 shutdown\_power\_off함수를 사용한다.   
exit은 thread\_exit함수를 사용하는데 이때 thread의 이름과 Status를 출력해줘야하는데 thread의 이름을 출력하기 위해서 thread\_name()함수를 사용한다.  
  
exec는 process\_execute함수를 사용하여 구현한다.  
wait은 process\_wait함수를 이용해서 구현한다. 이때 wait의 경우 child proccess의 tid를 알고 있어야하고 그들의 status를 가져올수도있어야하는데, 이를 구현하기 위해서 thread struct에 변수를 추가해야하는데, exit\_status를 추가한다. 그리고 부모 process가 child process의 정보를 접근할 수 있도록 하기 위해 pintos에 구현되어있는 linkedlist 구조체 child\_list, child\_elem을 선언한다. 그리고 wait은 synchronization이슈가 존재하기 때문에 semaphore 구조체 child\_sem을 추가한다. 그리고 child thread의 메모리가 먼저 죽어버리는 이슈가 존재할 수 있기 때문에 하나의 세마포를 추가한다. 이는 이후에 더 구체적으로 설명한다.  
read는 input\_getc함수를 사용하여 buffer에 있는 바이트를 모두 입력하면 된다.  
write함수는 putbuf 함수를 사용하여 buffer에 있는 내용을 fd에 집어넣으면 된다.  
기존에 있는 함수를 활용한 것들의 구체적인 내용과 반환값은 뒤에 기술한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명 텍스트, 도표, 평행, 번호이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

텍스트, 라인, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명

 텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

1. System Calls

도표, 평면도, 라인, 영수증이(가) 표시된 사진

자동 생성된 설명

* 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

argument passing이 실질적으로 이루어지는 파트는 pintos/src/userprog/process.c 내부의 load함수다. 이때기존의 pintos 코드를 그대로 사용하면 filesys\_open의 결과가 NULL이 나온다. 이는 커맨드라인을 저장하고 있는 file\_name이 그대로 파라미터로 들어가기 때문이다. 가령 echo x라면 echo가 들어가야하는데 echo x가 들어가기 때문에 오류가 생긴다. 이를 해결하고 argument를 stack에 넣어주기 위해서 공백을 기준으로 file\_name을 자르고 저장해준다. 이때 strtok\_r함수를 사용한다. 기존에 사용하는 C library strtok함수는 reentrance 이슈로 사용하면 안된다고 오류가 나와서 이 함수를 사용한다. 우선 file\_name을 복사해서 tmp\_file이라는 새로운 문자열에 저장한다. 이는 file\_name이 const char이기 때문에 편하게 사용하고자 새롭게 추가한 변수이다. 이후 token = strtok\_r(tmp\_file," ",&token\_ptr);로 파싱할 준비를 한다. 토큰화는 공백을 기준으로 하고 세 번재 파라미터로 포인터가 필요하기 때문에 token\_ptr라는 변수를 추가로 선언해 사용한다. 이후 while (token != NULL)로 해당 문자열이 끝날 때 까지 공백을 기준으로 토큰을 꺼내온다. 그리고 token의 개수가 몇개인지를 파악하기 위해서 token\_count라는 변수를 선언한다. 이 Token\_count는 while문안에서 token을 저장할때마다 1씩 올려주며 이후 스택에 집어넣을 때 이 값을 기준으로 for loop을 진행한다. 가령 token\_arr[token\_count ++] = token; token = strtok\_r(NULL," ",&token\_ptr);로 진행한다. 그리고 token\_count가 0일 때, 즉 첫 번째 토큰일 때, 이것이 실제 파일 명이기 때문에 이를 real\_file이라는 새로운 문자열 변수에 저장한다.   
while문을 빠져나온 후 real\_file을 filesys\_open의 파라미터로 넣어준다.  
이로써 우선 token\_arr라는 문자열 배열에 token들을 잘 저장했다.  
이후 이를 스택에 넣어줘야하는데, 이 과정은 setup\_stack이 이루어진 후에 넣어준다.  
스택에 넣어주기 위해서는 token count만큼을 for loop을 돌고 각 for loop안에서는 token\_arr[i]의 strlen만큼을 다시 for loop을 돌아준다. 이때 중요한 것은 순서를 역순으로 돌아야한다는 것이다. 이는 앞서 설명했듯 스택에 넣을 때는 더 높은 주소부터 낮은 주소로 내려오면서 넣어줘야하기 때문에 주소 기준으로 오름차순으로 정렬하기 위해서는 역순으로 돌아오면서 내려어야한다. 결과적으로   
for (int i = token\_count -1 ; i>=0;i--){  
 for (int j = strlen(token\_arr[i) ;j>=0;j--)   
 \*esp = \*esp -1  
}  
와 같이 반복문을 돌아준다. 이때 위처럼 반복문의 시작에서 바로 주소를 1 낮춰준 뒤, token\_arr[i][j]로 참고하여 특정 문자를 입력으로 넣어준다. 그리고 이때 for\_align이라는 변수를 1씩 증가시켜준다. 이는 문자열을 스택에 넣어줄 때, word align을 맞추기 위한 변수로 주소에 스택에 집어넣을 때 마다 1씩 증가시켜준다. 그리고 문자열을 모두 넣어준 이후에 이 각 문자열들의 첫 시작 주소를 저장해야하기 때문에, 안 쪽의 for loop이 끝났을 때의 esp 주소를 저장해야한다. 이때 token\_arr를 재활용하여 token\_arr[i]에 \*esp를 저장한다. 결과적으로 코드는 다음과 같다.  
for (int i = token\_count -1 ; i>=0;i--){  
 for (int j = strlen(token\_arr[i) ;j>=0;j--) {  
 \*esp = \*esp -1  
 \*\*(char \*\*)esp = token\_arr[i][j]  
 for\_align ++

}  
 token\_arr[i] = \*esp  
}  
  
이후에는 우선 word align을 맞춰준다. pintos는 4byte align을 맞춰야하므로 앞서 선언한 변수 for\_align을 4로 나눈 나머지를 4에서 빼준 만큼을 null을 넣어준다. 이때 굳이 별도로 값을 채우지 않고 주소를 내리면 알아서 안에 Null이 차기 때문에 이는 다음과 같다.   
for\_align = 4 - (for\_align % 4) \*esp = \*esp - (for\_align + 4)  
이때 4를 추가해서 주소에서 빼는 이유는 sentinel, 즉 문자를 넣은 지점과 이들의 주소를 저장한 지점의 구분을 null로 4byte만큼 해줘야하기 때문에 이부분에서 추가해서 집어넣어주는 것이다. 이후에 다시 token\_count만큼을 for loop을 돌아줘야한다. 여기서 token\_count의 개수만큼 그들의 시작 주소를 넣어준다. 즉 echo x라면 echo의 시작 주소와 x의 시작 주소를 각각 저장해준다. 이때도 역시 for loop을 역순으로 돌아준다. 즉 x의 시작 주소를 먼저, echo의 시작 주소를 넣어준다.   
결과적으로 코드는 다음과 같다.  
for (int i = token\_count -1 ; i>=0; i--){  
 \*esp = \*esp - 4  
 \*\*(uint32\_t \*\*)esp = (uint32\_t)token\_arr[i]  
}  
주소는 4byte씩 넣어주기 때문에 Esp는 4씩 감소한다. 그리고 마지막으로 이러한 argv의 시작 주소들의 시작 주소를 그 다음에 넣어줘야하기 때문에 i==0일 때,   
\*esp = \*esp - 4  
\*\*(uint32\_t \*\*)esp = (uint32\_t)(\*esp + 4)와 같이 넣어줌으로써 argv의 시작 주소를 넣어준다. 그리고 argc를 넣어줘야한다. argc는 앞서 선언한 token\_count와 동일한 값이기에 \*esp = \*esp - 4, \*\*(uint32\_t \*\*)esp = token\_count와 같이 argc를 넣어준다.  
마지막으로 return address를 0으로 넣어준다 :\*esp =\*esp -4, \*\*(uint32\_t \*\*)esp = 0;  
이로써 스택에는 필요한 요소를 모두 넣어줬다.

1. User Memory Access

user memory의 invalid access를 관리하기 위해서 check\_addr\_valid라는 함수를 pintos/src/userprog/syscall.c 소스 코드 안에 구현한다. 이 함수는 pagedir 와 가상 주소를 입력으로 받는다. 그리고 가상메모리 상에서의 invalid memory access는   
단순하게 해당 파라미터로 온 주소가 PHYS\_BASE보다 크거나 같은지 확인하고 그렇다면 exit(-1)을 한다. 다음으로 현재 가상 메모리 주소가 물리적 주소로 맵핑이 잘되는지 아닌지를 확인하기 위해서는 pagedir\_get\_page라는 함수를 사용한다. 이는 핀토스 내부에 구현되어있는 함수로 pintos/src/userprog/pagedir.c 함수안에 작성되어있다. 기능과 return값을 설명하자면 page directory와 가상 주소를 입력으로 받고 pagedir를 참고하면 가상주소가 맵핑되는 physical address를 return으로 준다. 반면 mapping이 되어있지 않다면 NULL을 반환해준다. 그래서 이 함수에 pagedir, 가상주소를 입력으로 넣어서 NULL이 나온다면 exit(-1)을 수행한다. 결과적으로 소스코드는 다음과 같다.  
int check\_addr\_valid(uint32\_t \*pagedir, void \*addr){  
 if (addr >= PHYS\_BASE){  
 exit(-1)  
 }  
 if (!pagedir\_get\_page(pagedir,addr)){  
 exit(-1)  
 }  
 return 1 }  
다만 이때 결국 파라미터로 pagedir를 넣어줘야하는데 이를 system call handler안에서 구하려면 cur\_thread = thread\_current();로 현재 돌아가고 있는 스레드를 가져온다. 그리고 이 thread구조체의 pagedir멤버를 참고한다. pagedir = cur\_thread->pagedir;  
그리고 이 pagedir을 check\_addr\_valid의 첫 번째 파라미터로 넣어준다.   
이를 실제로 사용해서 메모리 주소를 확인하는 방법은 다음과 같다.  
우선 syscall\_number와 무관하게 stack pointer의 주소를 바로 확인한다. 이 주소가 invalid하다면 syscall\_number자체가 invalid하고 접근을 할 수 없기 때문에 이를 바로 확인한다. : check\_addr\_valid(pagedir, (f->esp));   
이후에 메모리 확인이 필요한 지점은 syscall에 필요한 파라미터를 저장하고 있는 주소를 모두 확인해야한다. 가령 syscall1이라면, 즉 스택에서 인자를 하나만 꺼내오는 syscall 이라면 check\_addr\_valid(pagedir, (f->esp + 4)); 와 같이 체크한다.  
2개를 가져온 check\_addr\_valid(pagedir, (f->esp + 4)); check\_addr\_valid(pagedir, (f->esp + 8));와 같이 체크한다. 이 처럼 각 인자들은 4를 간격으로 저장되어있으므로 사용해야하는 파라미터들에 대해서 4간격으로 모두 체크해준다.   
이때 추가로 SYS\_EXEC, 즉 execute를 할 때는 주소를 한번 더 체크해줘야한다. 이 부분은 처음에 예외처리를 하지 못해서 오류가 발생했는데, 이를 체크해야하는 이유는 Exec는 입력으로 들어온 새로운 파일을 실행할 때, 그 파일의 invalid memory여부를 체크해줘야하기 때문이다. 오류가 발생한 소스코드를 보면 다음과 같다.  
exec ((char \*) 0x20101234)  
이는 결국 invalid한 주소를 가지고 실행하려다보니 page fault가 발생하는 것이다. 위의 주소를 저장하고 있는 스택 자체는 valid 하지만 저 참조하는 주소가 invalid하기 때문인 것이므로, exec는 다음과 같이 stack에 저장되어있는 주소를 한번 참조하고 그 참조된 주소가 valid한지를 체크해줘야한다. 결과적으로 exec에서 valid memory를 참조하는지 확인하는 코드는 다음과 같다.  
check\_addr\_valid(pagedir, (f->esp + 4))  
file\_pointer = (char \*)(f->esp +4) exec의 입력으로 들어온 주소가 참조하는 주소 저장  
check\_addr\_valid(pagedir, \*(void \*\*)file\_pointer) file\_pointer가 참조하는 주소가 valid한지 확인  
이로써 invalid memory access에 대한 처리를 완료했다.

1. System Calls

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

우선 pintos/src/userprog/syscall.c 내부의 syscall\_handler 내부에서 작성을 시작한다.  
가장 먼저 해야하는 것은 syscall number를 확인하여 이게 어떤 syscall인지를 학인해야한다. 이는 현재 stack pointer의 주소에 저장되어있다. 그래서  
syscall\_number = \*(uint32\_t \*)(f->esp)로 저장한다. 그리고 이 숫자를 switch문으로 넣어 각 케이스에 대해서 구현한다.  
우선 SYS\_HALT인 경우는halt()라고 정의한 함수를 호출한다. 이때 파라미터는 없다. 그리고 halt()내부는 shutdonwn\_power\_off()를 호출한다. 이는 핀토스 운영 체제 자체를 종료하는 함수로 반환 값도 따로 존재하지 않는다.  
SYS\_EXIT의 경우는 exit(int status)함수를 호출하는데, 내부적으로는 thread\_exit()을 호출한다. 이는 간단하게 보면 해당 thread를 종료시키는 함수이다. 이것만으로도 해당 프로세스를 종료할수 있지만, 이번과제에서는 구체적으로 어떤 쓰레드가 어떤 status로 종료했는지 출력해야한다. 가령 echo x를 정상 작동하고 종료했다면  
echo: exit(0)과 같이 출력해야한다. 이를 위해서 직접 printf로 출력을 하는데, status의 경우 함수의 파라미터로 들어오기 때문에 출력할 수 있지만 thread의 이름을 출력해야하는데, 이는 thread\_name()함수를 사용하면된다. thread\_name()은 current thread 구조체의 멤버인 name을 반환하는 함수이다. 그래서 결과적으로 다음과 같이  
출력해주고 종료하면. exit은 수행이 완료된다.  
printf("%s: exit(%d)\n",thread\_name(),status);  
thread\_exit();  
이후 wait을 구현하기 위해서 exit에서 새롭게 코드가 작성되는데 이는 해당 파트에서 설명을 하면서 추가하겠다.   
SYS\_EXEC의 경우 exec(const char \*cmd\_line)를 호출한다. 그리고 이는 내부에서 process\_execute(cmd\_line)을 호출한다. process\_execute함수는 내부적으로 구현되어있는 함수인데, 이는 pintos/src/userprog/process.c에 정의되어 있다.  
기존 process\_execute의 대략적인 내용은 입력으로 들어온 file\_name에 대해서 새로운 Thread를 하나 만들어주고 그 thread에 대한 tid를 반환하는 것이다.  
결과적으로 exec함수는 입력으로 들어온 command line을 하나의 filename으로 보고 실행시키는 함수라고 볼 수 있다. 허나 이때 예외처리를 해주지 않으면 몇가지 오류를 경험했다. 첫 번째 예외처리를 해주는 부분은 thread\_create라는 함수에 첫 번재 파라미터로 들어가는 file\_name을 수정해줘야한다는 것이다. 기존의 코드처럼 FIle\_name이 첫 인자로 들어간다면 echo x라고 했을 때 Thread\_create의 첫 인자로 Echo가 아닌 echo x가 통째로 들어가는 것이다. 근데 이것이 아닌 Echo가 들어가는 것이 pintos의 내부 구현 및 검증 통과를 위해서 필요한 내용이다. 왜냐하면 echo x를 넣게 된다면 이것이 thread의 name이 되는데 핀토스에서는 thread의 name이 echo x가 아닌 Echo가 되길 원하기 때문에 이 부분을 수정해준다.  
수정하는 방법은 앞서 argument passing을 위해서 parsing을 할 때 띄어쓰기를 기준으로 첫 번째 토큰을 저장해주는 방식을 그대로 사용했다. 방식은 그대로이기 때문에 구체적인 설명은 넘어간다. 동일한 방식으로 real\_file이라는 문자열에 파싱한 파일명을 저장하고 이를 thread\_create의 첫 인자로 넣어준다.   
추가로 예외처리를 위해 수정해야하는 부분은 바로 위 real\_file에 저장한 파일명이 valid한지, 파일이 잘 열리는지 체크하는 부분이다. 사실 이것을 발견한 것은 테스트 중 exec\_missing에 대해서 실패를 하는 부분에서 발견했다. 해당 파일의 소스코드는   
msg ("exec(\"no-such-file\"): %d", exec ("no-such-file")); 이다.  
이를 보면 잘못된 파일명이 넘어가지만 이에 대해서 예외처리를 하지 않아서 오류가 발생하는 것을 확인할 수 있다. 이를 정상 종료하기 위해서 load함수에서 사용하는 filesys\_open함수를 사용한다. filesys\_open(real\_file)의 결과가 NULL이라면 해당 파일로 열리지 않는 것 이기 때문에 이 경우에 -1을 return함으로써 새로운 프로세스가 만들어지지 않았다는 사실을 알린다. 결과적으로 예외처리에 대해서 추가된 코드는 다음과 같다.   
strlcpy(tmp\_file, file\_name, sizeof(tmp\_file)+1);   
token = strtok\_r(tmp\_file, " ",&token\_ptr);   
while (token != NULL) {  
 if (token\_count==0){   
 strlcpy(real\_file, token, sizeof(real\_file)+1);   
 }  
 break;   
}   
if (!filesys\_open (real\_file)) {  
 return -1;  
}   
tid = thread\_create (real\_file, PRI\_DEFAULT, start\_process, fn\_copy);  
  
그 다음은 SYS\_WAIT의 구현인데, 우선 wait(tid\_t pid)함수를 부른다. wait함수 안에서는 process\_wait(pid)을 호출한다. 참고로 인자로 들어오는 Pid는 child의 pid이다. 이때 process\_wait함수는 return -1이 되어있는데, 이를 직접 구현해서 chlid의 exit status를 반환할 수 있어야한다. 이를 구현하려면 부모 process가 child process의 정보를 알거나 접근할 수 있어야한다. 이를 위해서 pintos의 자료구조 중 하나인 Linkedlist를 사용한다. 기본 아이디어를 요약하자면 Linked list로 부모가 자식 process와 연결을 시키고, 연결된 것을 통해서 child의 exit status를 꺼내오도록 한다.  
이를 위해서 우선 thread 구조체에 몇가지 변수를 추가한다.  
pintos/src/threads/thread.h 파일 안에 struct thread 선언 부분에서   
#ifdef USERPROG의 밑에 struct list child\_list, struct list\_elem child\_elem을 추가한다.  
child list라는 것을 thread마다 선언함으로써 링크드 리스트에 접근할 수 있게하고 list elem이라는 구조체를 통해서 pintos는 별도의 메모리 할당없이 linkedlist로 연결된 thread 구조체를 참조할 수 있게 된다. 그리고 exit status라는 변수를 추가해야한다. 이를 통해서 직접 부모가 child의 thread 구조체에 접근하여 이 값을 뽑아온다.   
추가로 wait을 제대로 구현하기 위해서는 부모 process는 child process를 기다려야한다. 이를 효과적으로 구현하기 위해서 pintos에 구현되어있는 semaphore 구조체를 사용한다. 대략적인 그림은 thread가 초기화 될때 semphore를 0으로 초기화 한다. process\_wait에서는 sem\_down을 부르고 기다리고 있는다. 결과적으로 이게 1 이상이 될때 까지 기다린다. 그리고 child의 process가 exit할 때 이 값을 1로 증가시킨다. 이때 Process wait에서 sem\_down을 수행하면서 해당 프로세스를 종료한다. 이로써 process가 죽을 때 동기화되어서 wait을 수행한다. 이를 구체적으로 구현하기 위해서 필요한 코드를 살펴보자면 우선 pintos/src/threads/thread.c 소스코드에서 init\_thread함수를 수정해야한다. 마지막에 child\_sem을 0으로 init하고 child\_list를 list\_init한다 이대 0으로 init한다. 그리고 어떤 thread든 처음 만들어질 때, 앞서 초기화한 list에 자기를 넣어야한다. 이는 list\_push\_back함수를 사용해 집어넣는다.  
sema\_init(&(t->child\_sem),0);  
list\_init(&(t->child\_list));  
list\_push\_back(&(running\_thread()->child\_list), &(t->child\_elem));   
list\_push\_back함수를 사용하게 되면 구체적으로 현재 thread의 child\_elem이 child\_list로 들어간다. 이때 running\_thread()의 child\_list에 들어가는데, 이는 곧 부모 프로세스의 child\_list에 들어가는 것이다. thread를 새로 create할 때 이는 곧 부모이기 때문에 부모의 list에 child의 elem이 들어가고 이를 통해 부모는 child이 thread 구조체에 접근 가능하다. 이후 Process\_wait에서는 이 링크드 리스트를 순회하면서 인자로 들어온 child tid와 동일한 tid를 링크드 리스트내에서 찾아내고 지워줘야하는데 이는 우선 다음의 코드로 list를 순회한다.  
element = list\_begin(&(thread\_current()->child\_list))  
while (element != list\_end(&(thread\_current()->child\_list)))){  
내용  
element = list\_next(element)  
}  
내용을 보면 현재 부모 스레드의 child\_list의 시작 원소를 잡고 list의 끝 원소가 아니라면 안에 내용을 수행하고 다음 원소로 넘어간다.  
이때 안에서 수행하는 내용은 결국 해당 리스트에서 구조체 정보를 뽑아내고 이게 지금 찾는 Tid가 같은지를 확인하고 맞다면 list에서 지워줌으로써 정보를 지워주고 exit status를 return하는 것이다. 이는 list\_entry라는 내부 구현 함수로 우선 thread 구조체를 뽑아내는데서 시작한다. t = list\_entry(element, struct thread, child\_elem);  
위에서 t가 결국 링크드 리스트에 해당하는 Thread 구조체를 저장한다. 이제 이 멤버에 접근해서 파라미터에 있는 tid와 같은지 확인한다. 그리고 같다면 sema\_down을 시키고 List\_remove함수를 사용해 해당 thread의 child\_elem을 지운다.   
코드는 다음과 같다.   
t = list\_entry(element, struct thread, child\_elem);  
if (child\_tid == t->tid){  
sema\_down(&(t->child\_sem));   
exit\_status = t->exit\_status;   
list\_remove(&(t->child\_elem));   
return exit\_status; }

이때 중요한게 구조체 thread에 멤버 중 새롭게 정의한 exit\_status라는 변수에 exit status를 할당하는 파트가 필요한데, 이는 system call handler 중 SYS\_EXIT에서 불리는 exit안에서 수행한다. exit에서는 파라미터로 exit status를 가지고 있기 때문에 여기서 직접 구조체의 멤버에 넣어준다 결과적으로 exit은 다음과 같이 수정된다.  
void exit (int status) {   
printf("%s: exit(%d)\n", thread\_name(), status);   
struct thread \* cur\_thread;   
cur\_thread = thread\_current();   
cur\_thread->exit\_status = status; //여기서 exit state를 저장을 해줘야함   
thread\_exit (); }  
그런데 위와 같이 수행하면 모든 테스트에서 fail하고 pagefault가 발생한다.  
이 부분에 대해서 검토한 결과 list\_remove로 child\_elem을 제거하거나 exit\_status를 뽑아오기 전에 race 컨디션으로 child가 먼저 exit되어버려서 메모리에서 해제된다면 없는 지점을 참조하기 때문에 fault가 발생한다. 결과적으로 child가 exit하는 것과 부모가 wait하는 것 사이에 다시 한번 동기화가 필요하다. 그래서 다시 thread 구조체에 seg\_sem이라는 semaphore 구조체를 하나 추가한다. 그리고 init\_thread에서도 sema init(seg\_sem)을 넣어서 seg 세마포를 초기화한다. 그리고 이 process\_exit에서 sem\_down(seg\_sem)을 선언하고 기다린다. 이 seg\_sem이 1이상이 되는 시점은 process\_wait에서 이루어지는데 exit\_status를 저장하고 list\_remove를 수행한 후에 sema\_up을 한다. 이로써 메모리는 안지워진다. 이후 필요한 정보를 다 가져온 후에는 seg\_sem이 up되므로 process\_exit에서는 seg\_down이 return되고 정상적으로 process를 종료시킨다. 결과적으로 process\_wait의 코드는 다음과 같고  
텍스트, 스크린샷, 소프트웨어, 운영 체제이(가) 표시된 사진

자동 생성된 설명  
텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

process\_exit은 아래 sema\_up,down이 두줄 추가되었다.  
다음으로 SYS\_READ에 대한 구현이다.  
SYS\_READ는 read(fd, buffer, size)라는 함수를 호출한다.  
read는 내부에서 input\_getc()를 사용해서 구현한다. input\_getc()함수는 키보드로 들어오는 값을 하나씩 return해준다. 이번 과제에서는 stdin, 즉 fd==0인 경우만 고려한다. 그래서 다음과 같이 구현한다. 일단 fd==0인 경우에 대해서 for loop을 도는데,   
for (read\_bytes =0 ;read\_bytes < size;read\_bytes++)다음과 같이 돈다.  
즉 사이즈만큼을 for loop을 돈다. 이 한번한번의 loop마다 input\_getc()의 결과를 문자로 저장하고 이걸 buffer에 집어넣는다. 이때 input\_getc의 결과가 '\0'이라면 이는 enter가 들어왔다는 말이므로 for loop을 반복한다. 그리고 read\_bytes를 반환한다. read함수는 결국 얼마나 읽었는지를 반환해야하기 때문에 위와 같이 구현한다. 만약 fd가 0이 아닌 경우는 그냥 -1을 반환한다.  
SYS\_WRITE은 write(fd,buffer,size)라는 함수를 호출한다. write은 stdout, fd가 1인 경우에 대해서만 구현하므로 fd==1인 경우에 대해서만 수행하고 putbuf라는 함수를 사용한다. putbuf는 buffer, size를 파라미터로 주는데 buffer의 내용을 size만큼 fd에 써주는 함수이다. 그리고 size를 return한다. write는 쓴만큼을 반환해야하기 때문이다. 그리고 fd==1이 아닌 경우는 -1을 return한다.  
텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

추가로 모든 return값이 있는 system call의 경우 그 return값을 eax레지스터에 저장해야지 유저가 이를 접근할 수 있다. 그래서 모든 return값은 f->eax에 저장한다.

1. Additional System calls

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

우선 pintos/lib/syscall-nr.h에 시스템콜 번호를 macro로 정의한다. SYS\_FIBO, SYS\_MAXF로 정의했다. 그리고 pintos/lib/user/syscall.h에   
int fibonacci(int), int max\_of\_four\_int(int, int, int int);라는 프로토타입을 지정하고  
pintos/lib/user/syscall.c에 내용을 정의한다.  
fibonacci는 하나의 파라미터만 들어가는 syscall1이기 때문에 syscall1(SYS\_FIBO, arg1)과 같이 함수를 구현한다. max\_of\_four\_int의 경우는 syscall4이다. 파라미터를 4개 사용하기 때문인데, 이는 추가로 구현해야한다.   
내용은 간단하다. pushl을 통해서 스택을 아래로 밀면서 arg3 arg2 arg1 arg0을 순차적으로 추가해준다. 이외에는 int $0x30을 걸어야하고, 파라미터가 4개, syscall number 1개가 stack에 들어갔기 때문에 esp를 20만큼 움직인다. 이렇게 함으로써 syscall4를 구현했다. 이를 바탕으로 max\_of\_four\_int는 syscall4(SYS\_MAXF,arg1,arg2,arg3,arg4)를 호출하도록 구현한다. 여기까지 했다면 system call handler를 호출하는 부분까지 작성한 것이다. 이제는 system call handler를 작성해야한다. 이 내용은 pintos/src/userprog/syscall.c에 작성한다. 앞서 사용한 switch 문에서 case SYS\_FIBO, case SYS\_MAXF를 추가한다. 나머지 시스템콜과 마찬가지로 주소에 대해서 check\_addr\_valid로 주소를 검증한 후, SYS\_FIBO는 fibonacci라는 함수를 호출한다. 이는 pintos/src/userprog/syscall.c 내에서 정의한 함수로 for loop을 돈다. 1부터 파라미터까지 순차적으로 sum에 값을 축적한 후 이 sum을 반환한다. max\_of\_four\_int도 마찬가지로 이 파일에서 새롭게 정의된 함수로 max=0으로 초기화 한 후 각가 if문을 통해 max보다 크다면 그 값을 max로 업데이트함으로써 네 숫자 중 가장 큰 값을 max에 저장하고 반환한다.  
텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

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

  자동 생성된 설명**

fibonacci(10), max\_of\_four\_int(10,20,62,40)의 값이 잘 출력되었고 exit(0)과 함께 정상종료 되었음을 볼 수 있다