Lab2 Final Report

Member #1
Name: Suwon Yoon
Student ID: 20210527

Member #2
Name: Jiwon Lee
Student ID: 20210706

1. Argument passing 구현

디자인 레포트에서 작성하였던 것처럼, start_process 함수 내부에서 load 함수를 호출하기 전에 argument parsing을 우선 하고, 그 결과 중 파일 이름만 load 함수로 전달하여 load 함수는 크게 변경할 부분 없이 만들었다. load 함수 내부에서 user stack을 구성하기 때문에 load 함수 호출 이후에 parsing한 argument를 사용하여 user stack을 구성하는 것이 필요하다. 이 동작은 put_args_into_user_stack 함수로 구현한다.

- put_args_into_user_stack 함수

  이 함수는 user stack을 구성하는 함수이다. esp를 가리키는 포인터의 주소, argument parsing의 결과로 나온 인자 개수, parsing된 후 인자들의 배열을 넘겨준다. 문서의 안내에 따라 real text를 stack에

  memcpy를 활용하여 스텍에 인자를 저장하는 것이 필요하다. 왜냐하면 parsing을 위해 할당한 메모리를 해제해줄 것이기 때문이다. 스텍을 구성할 때는 word-align과 argv, argc 값도 잘 넣어주저야한다. 마지막에 fake return address 또한 구성해야한다.

- file system lock

  제공된 file system code에 동시에 접근하여 실행하면 문제가 발생할 수 있다. 따라서 filesys_lock을 추가하여 제공된 file system code에 동시에 접근하는 상황을 방지한다. 락은 syscall_init에서 초기화하는 것이 합리적이다. 왜냐하면 결국 file system code는 syscall과 관련하여 사용될 것이기 때문이다. load 함수에서 filesys_open 함수를 사용하므

로 이 전에 lock을 얻어야한다. 이후에도 file system code를 자주 사용하므로 락은 마지막에 풀어주도록 한다.

2. Process termination message 구현

이를 위해서는 thread_create 함수에 전달하는 name 부분의 인자를 조절할 필요가 있다. 따라서 process_execute 함수에서 thread_create 함수를 호출하기 전에 argument를 omit 하기 위한 parsing을 진행한다. 위에서 한 parsing과 동일한 방식으로 하면 된다. 결론적으로는 첫번째 인자인 파일 이름만 thread_create 함수로 전달된다. parsing을 위해서 할당한 메모리를 해제하는 것도 중요하다.

여기서 중요한 점은 termination message를 출력하기 위해서 구현해야하는 기능이 많다는 점이다. 제공된 문서에서도 termination message를 출력하기 위한 편법을 알려주기도 하지만 정확한 구현을 한 후에 결과를 보는 것으로 할 것이다. 따라서 Process termination message 구현은 이후에 exit system call에 대한 설명을 할 때 이어서 작성하도록 하겠다.

3. System calls 구현

- system call을 위한 공통적인 논리
  1. verify user pointers before dereferencing.
  2. user stack에 system call을 위한 인자가 들어있다.
     a. 그 순서는 stack의 top에서부터, 32bit씩 차지하는데… system call number, first argument, second argument, … 순서이다.
     b. interrupt frame에 user stack의 esp 주소를 저장해두니까 interrupt frame의 esp 정보 참고해서 찾아가면 된다.
  3. syscall_init 함수 → intr_register_int 함수 → register_handler → syscall_handler 를 intr_handlers에 저장해뒀기 때문에, 나중에 intr_handler가 호출되면(interrupt frame이 구성된 이후에 호출됨)에서 0x30번 오면 syscall_handler가 호출된다. 이 syscall_handler 안에서 syscall number 를 보고 적절한 동작을 시키는 것이 핵심이다.
     a. switch문으로 번호를 판별하게 하고 각 번호에 따른 동작을 ‘함수’로 만들자.
     b. 함수에 인자가 있으면 syscall_handler에서 처리해서 ‘함수’로 보내주자.
     c. 함수에 리턴이 있으면 interrupt frame의 eax에 넣자.(38페이지)

위 기능을 위해 우선 src/userprog/syscall.c에 check_address 함수를 구현한다. 기본적으로 PHYS_BASE와 0x08048000 사이에 있는 것을 체크해주는 것으로 구현한다. 만약 허용되지 않는 주소가 온다면 exit(-1)을 호출한다.

이후 할 일은 syscall_handler를 구현하는 것이다. 이 함수는 interrupt frame을 인자로 받는데 여기는 user stack의 top을 가리키고 있는 esp 변수가 있다. esp의 top에는 system call number가 push되어 있고 그 위에는 해당 system call의 argument들이 들어있다. 이는 lib/user/syscall.c 함수의 syscall# 함수에서 어셈블리어를 해석하면 알 수 있다. 다시 돌아가서 syscall_handler는 switch case 문을 활용하여 어떤 system call 함수를 호출할지를 판단한다. 여기서 고려해야하는 부분은 check_address 함수를 활용하여 user program이 제공하는 인자에 대한 검사를 해야한다는 것이다. 그 예시로는 user stack의 상단에 있는 syscall number를 가리키는 주소에 대한 검사가 있다.

지금까지 살펴본 부분은 system call을 구현하기 위해 기본적으로 필요한 부분에 대한 것이었다. 이제 살펴볼 부분은 exit, exec, wait 함수에 대한 부분이다. halt 구현은 문서에 나와있는대로 shutdown_power_off 함수의 호출로 간단히 구현할 수 있으니 별다른 언급은 하지 않겠다. 시스템콜 기능을 하는 함수의 이름은 유저가 호출하는 시스템콜의 이름과 동일하게 하였다.

- exit / exec / wait / struct thread

  주어진 문서를 보면, exit에서 wait으로 신호를 줘야하고, load 이후에 exec으로 신호를 줘야한다는 것을 알 수 있다. 이를 위해 struct thread에 추가할 정보들이 다주 존재한다는 공통점을 가지고 있기때문에 독립적으로 다루기 보다는 한번에 정리하는 것이 바람직하다.

  exit 함수를 위해서는 exit_status를 위한 정수, exit 관련 동기화를 위한 세마포어, 어떤 자식을 기다리고 있는지에 대한 정보를 위한 정수를 sturct thread에 추가한다. exec 함수를 위해서는 exec관련 동기화를 위한 세마포어, 그리고 child의 load가 성공했는지를 기록하는 boolean을 sturct thread에 추가한다. 마지막으로 부모, 자식 관계를 위한 struct list와 struct thread *가 필요하다.

  기본적으로 exit, exec, wait 함수는 모두 각각에 대응되는 process.c 내에 있는 함수를 활용하여 구현한다. exit은 process_exit, exec은 process_execute, wait은 process_wait으로 구현한다.

- exit 함수

  exit_status를 설정하고 thread_exit함수→process_exit함수로 process에 있는 자원을 할당해제하고 종료메시지 출력한다. 정상적으로 종료되는 프로세스는 모두 exit 함수를 거치기 때문에 이 함수에서 종료 메세지를 출력하는 것이 바람직하다.

- wait 함수

곧바로 -1을 리턴해야하는 경우는 총 3가지로 생각할 수 있다. 1. child_tid가 커널에 의해 종료된 경우, 2. child_tid가 invalid한 경우(calling process의 child가 아닌 경우), 3. 특정 child_tid에 대해서 process_wait이 이미 호출된 경우이다. 이 중 두번째와 세번째 경우는 이 함수에서 처리할 수 있다. 하지만 첫번째 경우는 process_exit에서 처리한다. 커널에 의해 종료된 경우는 exit함수를 통해 종료된 경우가 아니기 때문에 exit_status가 기존의 값과 동일할 것이다. 이를 활용하여 exit 함수에서 -1을 리턴하여 기다리는 것 없이 종료시키도록 한다. 그 밖에는 자식들의 list를 순회하며 child_tid와 같은 값을 같은 값을 가지는지를 판단하여 기다리는 자식을 기록해둔다. 그리고 기다릴 자식에 대해서 struct thread에 있는 세마포어를 활용하여 기다린다. 부모는 기다림이 끝난 이후 즉, 자식이 종료된 이후에는 자식을 child_list에서 제거하는 작업도 한다.

- struct process

  부모는 자식이 죽었을 때 자식의 상태를 유추할 수 있어야만한다. 지금까지의 구현상으로는 부모가 자식에 대한 정보를 저장할 때 자식의 tcb를 가리키게 저장한다. 하지만 이는 자식에 대한 정보가 모두 자식의 tcb에 있기 때문에 자식이 종료될 경우 자식의 정보를 더이상 받아올 수 없는 문제가 존재한다. 따라서 자식이 없어져도 자식의 상태를 담아둘 수 있는 구현이 필요하다. 이를 위해 struct process를 추가한다. 이 구조체에는 exit_status와 자식의 struct thread를 가리키는 포인터가 존재한다. 이후에 디버깅 과정에서 다른 정보가 추가되기는 할 것이다. 자식의 정보를 저장하기 위해 구조체를 추가로 만들었으니 추가로 메모리를 할당하여 사용한다. 따라서 이에 대한 관리를 할 필요가 있다. 이는 process_exit 함수와 process_wait 함수에서 각각 한다. 기다린 자식이 종료됬다고 알려주면 부모는 자식 정보가 담긴 struct process를 할당해제한다. process_exit에서는 해당 프로세스가 종료될 때 자식의 정보를 담는 리스트를 순회하며 struct process를 모두 할당해제 한다. 부모가 종료되는 관점에서 자식의 정보는 더이상 필요없기 때문이다. 사실 여기서 할당해제를 안하면 할 곳도 없기도 하다.

  자식의 정보를 추가적으로 다루기 때문에 실제 자식의 tcb에 존재하는 정보와 부모가 다루는 자식 리스트 안의 정보가 다를 수 있다. 이를 동일하게 유지하는 것이 중요하다. 따라서 exit 함수에서 exit할 때 자신이 상태를 부모의 자식 리스트에게도 알리는 작업이 필요하다.

  한 가지 더 짚고 넘어가자면 부모와 자식의 관계는 thread가 생성될 떄 형성된다. 따라서 thread_create 함수에서 이 작업을 해준다. 또한 init_thread에서 위에서 struct thread에 추가한 변수들을 추가하는 작업도 필요하다.

- exec 함수

  문서에서 제시한 조건에 따르면 exec 함수가 실행하려는 코드가 제대로 load 되었는지를 확인한 후에야 다시 실행될 수 있다. exec 함수를 호출한 thead가 부모로, exec 함수로 실

행되는 thread가 자식의 관계를 가지게된다. 따라서 부모가 자식을 기다릴 수 있게 하는 기능을 세마포어를 활용하여 구현한다. 만약 부모의 입장에서, 기다린 자식이 제대로 load 되지 않은 경우 -1을 리턴하며 부모는 종료된다.

- struct file_desc

  이제 파일과 관련된 system call 구현을 할 차례이다. 이를 위해서 파일을 다루는 구조체를 하나 만들어야한다. 파일은 기본적으로 핀토스에서 제공해주는 코드가 있기 떄문에 그것을 잘 사용하기만 하면 된다. struct file_desc는 파일 디스크립터를 위한 정수와 실제 열린 파일을 가리키는 file *가 필요하다. struct thread에 추가할 변수는 총 두개이다. 하나의 프로세스가 여러 파일을 열 수 있으므로 이를 다루는 struct list하나, 그리고 파일 디스크립터 숫자를 부여할 정수 하나이다. 이에 대한 초기화는 init_thread에서 해준다. struct thread에 추가한 리스트는 새로 할당된 struct file_desc 를 원소로 하기 때문에 이에 대한 할당해제를 해주는 기능을 process_exit 함수에 추가해준다. exit하는 process가 열어둔 모든 파일을 닫는 것으로 구현된다.

- get_file_desc 함수

  이 함수는 특정 process와 (pintos는 1thread 1process 구조이므로 구현할 때는 실제 정보를 담고 있는 struct thread를 사용함) 파일 디스크립터 번호를 주면 해당 파일에 대한 정보를 담고 있는 struct fild_desc의 포인터를 반환하는 함수이다. 만약 프로세스에 파일 디스크립터 번호가 없으면 NULL을 리턴한다. 이 함수는 이후 파일 관련 함수를 구현할 때 자주 사용한다.

- create, remove, open, filesize

  기본적으로 모두 file system과 관련된 코드를 작성할 때는 filesys_lock을 얻어야한다. 문제 출제자가 구현해 둔 file system 관련 코드를 활용하여 쉽게 구현할 수 있다. create 함수는 filesys_create 함수로, remove 함수는 filesys_remove 함수로, open은 filesys_open 함수로, filesize는 file_length 함수로 구현한다. open 함수는 파일을 여는 것이므로(create 함수는 파일을 생성하는 것) 이때 프로세스가 파일에 대한 관리를 할 수 있도록 파일 디스크립터 번호를 부여하고 파일을 다루는 리스트에 struct file_desc를 넣어준다.

- read, write, seek, tell, close

  이 함수들도 모두 file system 관련 코드를 사용하며 filesys_lock을 얻어야한다. read의 경우 파일 디스크립터 번호가 0인 경우 keyboard 입력을 받는 것으로 특수하게 처리를 해준다. 그 밖의 경우는 대응되는 파일 디스크립터로 파일에 접근하여 file_read 함수를 통해 작업을 한다. write의 경우 파일 디스크립터 번호가 1인 경우 console에 출력하는 것으로 특수하게 처리를 해준다. 그 밖의 경우는 대응되는 파일 디스크립터로 파일에 접근하여 file_write 함수를 통해 작업한다. seek 와 tell의 함수 역시 각각 file_seek, file_tell 함수를

통해 구현할 수 있다. 마지막으로 close 함수에서는 file_close 함수를 호출하고 중요한 점은 여기서 파일을 다룰 때 사용하는 구조체인 struct file_desc를 할당해제 한다는 것이다. process에서 파일을 다루는 리스트에서도 제거해줘야한다.

4. Denying writes to executables

struct thread에 현재 프로세스가 실행하고 있는 파일을 가리킬 수 있는 포인터를 하나 넣어둔다. 그리고 파일을 load 함수에서 파일을 filesys_open 함수를 호출한 직후 file_deny_write 함수를 통해 파일에 write 하는 것을 막는다. 그리고 프로세스가 종료될 때 file_allow_write 함수를 통해 다시 write를 가능하게 하고, file을 닫는식으로 구현한다. filesys_open 함수에서 파일을 잘 못 열 수도 있기 때문에 결과가 NULL인 경우에 대한 예외 처리 이후에 본 기능을 구현하여야 한다.

5. 간과하기 쉬운 예외들 - 디버깅

1. check_address 함수

   단지 하나의 포인터 값만 유저 가상 메모리에 있는지를 확인하는 것으로는 부족하다. 해당 포인터가 가리키는 이후의 주소에 대한 검증도 필요하다. 가령 4바이트의 값을 가지는 값인데 주소가 0x1234였다고 하면 0x1235, 0x1236, 0x1237에 대한 검사가 모두 필요한 식이다. 이에 대한 부분은 문서의 3.1.5 accessing user memory 부분을 참고하면 된다. 총 3가지의 예외 상황이 존재할 수 있으며, 첫쨰로 null pointer, 둘째로, user vm 밖의 공간, 셋쨰로 unmapped된 가상 주소가 있다. 이에 대한 처리를 모두 해줄 필요가 있다.

2. system call 인자에 대한 검증

   exec, create, remove, open, read, write 함수는 공통적으로 포인터를 인자로 받는다. 이에 대한 검사도 필요하다.

3. struct process

   struct process에 따로 기록해두지 않는 이상 자식이 먼저 thread_exit을 해버리면 부모는 자식에 대한 정보를 더 이상 볼 수 없게 된다. process_wait 함수에서 자식이 먼저 종료된 경우 자식의 tid를 알지 못하는 경우가 생길 수 있기 때문에 struct process에 자식의 tid 또한 기록해야한다. 안전하게 exit 함수에서도 process_wait 함수와 동일하게 자식의 tid를 보도록 구현한다.

4. process_exit 함수

process_exit에서 file descriptor 할당 해제할 때 close 함수에서 먼저 지우고 list_next하면 오류난다. list_next를 한 후에 free를 해줘야한다.