Pintos Project2 Design Report

20210084 김지민, 20210216 양준영

1. Analysis of the current implementation.

A. Process execution procedure

i. 이번 프로젝트에서는 OS위에 user program을 돌리는 것이 목적이다. 따라서 어떻게 Command line을 받고 이것이 어떤 과정으로 process execution으로 이어지는지 살펴볼 필요가 있다. 현재 threads/init.c의 main에서부터 살펴본다.

ii. 현재 threads/init.c의 main에서 read_command_line 함수를 통해 argv를 받고 있는 모습을 볼 수 있다. 그리고 이것을 parse_options 함수의 인수로 넘겨주고 있는데, 이 함수에서는 option을 parsing 한 뒤 non-option argument 부분을 argv로 해서 반환한다.

iii. 그 뒤 run_actions(argv) 함수를 호출하여 argv을 인수로 넘겨주는데, 이 함수를 보면 구조체 action과 이를 요소로 갖는 actions가 있다. Actions의 첫 부분을 보면 {"run", 2, "run_task"}라 되어있다. 일단 이 부분만 고려한다. 그 뒤 코드를 보면 첫 번째 argv 즉, argv[0]이 actions 구조체의 name과 일치하는지 검사하는 부분이 있다. 이 구조체의 name은 'run'과 같은 명령어에 해당한다. 그 뒤 argv[1]부터 시작하여 arguments를 검사한 후 action 구조체의 function에 argv를 인수로 넘겨준다. 이 function은 run_task에 해당한다.

iv. run_task에서는 task에 argv[1]을 저장하고 process_wait (process_execute (task))을 호출함으로써 process_execute(task)가 끝날 때까지 process_wait이 기다린다. 이렇게 넘겨받은 task은 process_execute에서 file_name으로 인식된다. 즉, argv[1]에 program name과 arguments가 하나의 string으로 저장되어 있다.

v. 현재 userprog/process.c의 process_execute에 인수로 들어오는 file_name은 program name과 arguments들이 공백으로 연결된 하나의 string이다. 이것을 thread_create에 하나의 name으로 인수로 넘기면서

호출된다. thread_create 함수는 start_process 함수를 인수로 받는데, 이 함수는 넘겨받은 file name을 load하고 이를 running한다.

B. System call procedure

i. How to call syscall_handler() in userprog/syscall.c from user program

1. 유저 프로그램이 유저 모드에서 실행되다가 syscall을 호출하면(int $0x30을 호출하여), lib/user/syscall.c의 시스템콜을 호출한다. 이는 syscall-entry.S 파일을 호출하여 레지스터 값을 저장한다. 이후 저장한 intr_frame을 userprog/syscall.c에 전달하며, intr_frame의 $esp로 caller의 stack pointer를 전달한다. Caller의 스택 포인터의 32bit word에서 시스템 호출 번호를 받아 Syscall_handler에서 시스템 콜 번호에 해당하는 핸들러를 호출하여 처리한다. 또한, 첫 번째 인수는 스택 포인터의 다음 32bit word에 있다. 값을 반환하는 시스템 호출은 intr_frame의 $eax를 수정하면 된다.

ii. Source code 분석(syscall.c, intr-stubs.S, interrupt.c)

1. Syscall.c

A. Halt: pintos를 종료하는 함수이다.

B. Exit: 현재 프로세스를 종료시키는 syscall이다. 종료될 경우, 프로세스 이름을 출력시켜야 한다.

C. Exec: 현재 프로세스를 cmd_line에서 지정된 인수를 전달해 이름이 지정된 실행 파일로 바꾸는 것. 실패한다면 -1로 exit한다.

D. Fork: 자식 프로세스를 복제 및 실행시키는 syscall이다.

E. Wait: 자식 프로세스가 종료될 때까지 대기하는 함수로, 자식이 종료된 exit_status를 받아온다. 만약 자식이 예외 등으로 비정상 종료가 되었다면 -1을 반환한다.

F. Create: 파일의 이름과 경로 정보, 파일 크기를 받아 파일을 생성하는 syscall로 성공할 경우 참을 반환한다.

G. Remove: 파일을 삭제하는 명령으로 제거할 파일의 이름과 정

보를 받아 삭제한 경우 참을 반환한다.

H. Open: 파일을 여는 함수로, 파일을 열 수 없다면 -1을, 아니면 음아 아닌 정수를 반환한다.

I. Filesize: 열린 파일의 크기를 바이트 단위로 반환한다.

J. Read: 열린 파일의 데이터를 읽어오는 함수이다. 열리지 않았다면 -1을 반환한다.

K. Write: 열린 파일에 데이터를 작성하는 함수이다.

L. Seek: 열린 파일에서 다음 파일 내 데이터의 위치를 받아 position으로 옮긴다.

M. Tell: 열린 파일에서 읽거나 쓸 바이트의 위치를 반환한다.

N. Close: 파일을 닫는 함수이다. 프로세스를 exit하거나 terminate하면 열린 파일을 implicit하게 닫는다.

2. Intr-stubs.S

A. Intr_entry

i. Internel/External 인터럽트가 intrNN_stub 루틴 중 하나로 실행된다. Intr_frame 구조체, frame_pointer, vec_no를 스택에 push하고 여기로 점프한다. 스택에 저장하고 커널에서 필요한 레지스터를 설정하고 핸들러를 호출한다. 이후 intr_exit을 통해 나온다.

B. Intr_exit

i. 호출자의 레지스터를 다시 불러오고 스택의 추가 데이터를 폐기한 후 호출자에게 반환한다.

C. Intr_stub

i. 256개의 stub로 각각은 해당 인터럽트 벡터의 진입점으로 사용된다.

ii. 함수 포인터들의 배열인 intr_stubs의 정확한 지점에 함수 각각의 주소를 넣는다.

iii. Interrupt.c에서 intr_handler에서 vec_no에 따라 배열의 값에 접근해 핸들러가 실행된다.

3. Interrupt.c

A. Intr_get_level: 현재 인터럽트 상태를 반환한다.

B. Intr_set_level: level로 지정한 인터럽트를 활성화(비활성화)하고 이전 인터럽트 상태를 반환한다.

C. Intr_enable: 인터럽트를 활성화하고 이전 인터럽트 상태를 반환한다.

D. Intr_disable: 인터럽트를 비활성화하고 이전 인터럽트 상태를 반환한다.

E. Intr_init: 인터럽트 시스템을 초기화한다.

F. Register_handler: privilege 단계에서 핸들러를 호출하기 위해 레지스터가 VEC_NO를 인터럽트한다. 인터럽트 상태가 LEVEL로 설정된 상태에서 핸들러가 호출된다.

G. Intr_register_ext: external interrupt VEC_NO를 호출한다. 인터럽트를 비활성화한 상태에서 실행될 것

H. Intr_register_int: internal interrupt VEC_NO를 호출한다. 인터럽트를 비활성화한 상태에서 실행될 것

I. Intr_context: external 인터럽트를 처리하는 동안 참을 반환한다.

J. Intr_yield_in_return: external 인터럽트를 처리하는 동안 핸들러가 인터럽트가 반환되기 전까지 새로운 프로세스를 실행하도록 양보하는 것.

K. Pic_init; PIC(Programmable Interrupt Controller)를 초기화한다.

L. Pic_end_of_interrupt: 주어진 IRQ에 대해 PIC에 end-of-interrupt singal을 보낸다. 만약 IRQ를 승인하지 않으면 다시는 전달되지 않을 것

M. Make_gate: 함수를 호출하는 gate를 생성한다.

N. Make_intr_gate: 주어진 DPL에서 함수를 호출하는 인터럽트 게이트를 호출한다.

O. Make_trap_gate: 주어진 DPL에서 함수를 호출하는 trap gate를 생성한다.

P. Make_idtr_operand: LIDT 명령을 위한 피연산자로 사용될 때, 주어진 LIMIT과 BASE를 양보하는 descriptor를 반환한다.

Q. Intr_handler: 모든 인터럽트, fault, 예외를 처리하는 핸들러이다. 이 함수는 intr-stubs.S에서의 assembly 언어를 통해 호출된다.

R. Unexpected_interrupt: interrupt frame F와 예기치 않은 인터럽트를 처리한다. 예기치 않은 인터럽트는 등록한 핸들러가 없는 인터럽트이다.

S. Intr_dump_frame: 디버깅을 위해 interrupt frame F를 console에 덤프한다.

T. Intr_name: 인터럽트 VEC의 이름을 반환한다.

C. File system

i. 현재 실행 중인 file에 write하는 것을 막는 것이 마지막 구현이다. 문서를 보면 filesys/file.c의 file_deny_write 함수와 file_allow_write 함수를 사용함으로써 구현할 수 있다고 되어있다.

ii. 우선 구현하기에 앞서 파일이 어떤 방식으로 다뤄지는지 알아볼 필요가 있다. 프로세스가 파일을 열고 읽고 작성하는 모든 과정은 시스템 콜로 다뤄진다(open, read, write 시스템 콜). 프로세스가 파일을 열 때, OS에게 open 시스템 콜을 통해 권한을 요청하게 되는데, 이 open함수는 file을 연 후 file descriptor라는, 음이 아닌 정수 값을 반환한다.

iii. 파일 디스크립터는 파일을 지칭하는 정수로, 파일이 열리면 open함수는 남은 정수 중 가장 작은 정수를 반환한다. 이 숫자는 프로세스마다 할당된 파일 디스크립터 테이블의 인덱스로서 이용된다. 즉, 파일 디스크립터 테이블은 각 프로세스가 현재 open한 파일들을 가리키는 포인터들을 모아둔 구조라고 생각하면 된다.

iv. 나중에 close함수에서는 입력 받은 파일 디스크립터에 해당하는 파일을 찾아 종료시킨다. 해당 파일 디스크립터 또한 파일 디스크립터 테이블에서 삭제시킨다.

v. 문서에서 활용하라고 한 file_deny_write 함수와 file_allow_write 함수를 살펴보자. file_deny_write 함수는 file 구조체 속 file->deny_write 값이 false면 deny_write를 true로 바꾼 후 filesys/inode.c의 inode_deny_write를 호출한다.

1. 여기서 아이노드란, 해당 파일에 대한 정보를 갖고 있으며 각 파일마다 하나의 아이노드를 가진다.

vi. inode_deny_write 함수에서는 inode->deny_write_cnt의 값을 1 증가시키는데, 이 값이 0이어야 write가 가능하고 0보다 크면 write를 할 수 없다. 또한 inode->deny_write_cnt가 inode->open_cnt 보다 작아야 한다. 이는 이 파일을 연 opener 수보다 write를 막은 수가 더 많아서는 안 된다는 의미이다. 결론은 file_deny_write 함수에 file을 인수로 넣어 호출하면 해당 file에 대한 write를 방지할 수 있다.

vii. file_allow_write함수는 file->deny_write가 true라면 false로 바꾸고 inode_allow_write를 호출한다. 거기선 Inode->deny_wirte_cnt를 하나 감소시킨다. 즉, 이 함수는 인수로 넣은 file에 대한 write 거절을 하나 줄여준다.

viii. 정리하자면, file_deny_write (file)을 호출하면 inode->deny_write_cnt를 증가시켜 해당 file에 대한 write를 방지하고, file_allow_write (file)을 호출하면 해당 파일의 inode -> deny_write_cnt를 하나 감소시킨다. 이 값이 0이 된다면 write를 진행할 수 있다.

2. Design plan

A. Process termination messages

i. 프로세스가 종료될 때 프로세스의 이름과 exit code을 출력해야 한다. 이때 user process가 아니거나 halt 시스템 콜일 때는 출력되면 안 된다. 따라서 이는 뒤에 나올 "System Call" 부분의 exit 함수에서 구현할 수 있다. exit함수는 현재 user program을 terminate시키고 status를 커널에 반환해야 한다. 이때 인수로 받는 status가 곧 exit code이므로 exit 함수

중간에 printf("%s: exit(%d)₩n", thread_current() -> name, status)를 삽입하면 될 것 같다.

B. Argument passing

i. 현재 process_execute 함수는 file_name이라는 인수로 program name과 arguments이 구분되어 있지 않은 하나의 string을 받고 있다. 이 file_name은 나중에 start_process에서 load함수를 호출할 때 그대로 인수로 넣는다. 따라서 이 file_name으로 load하기 전에 공백을 기준으로 program name과 arguments로 나누는 작업이 필요하다.

ii. start_process에서 'file_name' 변수를 load함수에 그대로 넣기 전, parsing하는 작업이 필요하다. threads/init.c의 parse_options 함수에서 parsing할 때 사용한 함수인 "strtok_r"를 이용하여 공백을 기준으로 file_name을 program name과 arguments로 나눈 후 load함수에는 program name만을 넣는다.

1. strtok_r (char *s, const char *delimiters, char **save_ptr): s를 delimiters를 기준으로 자르는 함수로, 한 글자씩 이동하다가 만약 delimiters를 만나면 해당자리에 NULL값을 삽입한 후 NULL 앞까지의 문자열을 반환한다.

iii. 커널에서 initial function에 대한 arguments를 스택에 넣어야 한다고 문서에 명시되어 있다. 따라서 strtok_r 함수로 file_name 변수를 parsing할 때 arguments는 새로운 문자열 배열에 저장한 후, 이 배열 속 arguments을 차례대로 스택에 넣는 작업도 구현해야 한다.

iv. 스택 포인트는 구조체 intr_frame의 esp 멤버를 통해 접근할 수 있다. Start_process에 intr_frame인 if_가 선언되어 있으므로 if_의 esp값을 수정하면서 arguments들을 스택에 저장한다. 이때 80x86 호출규약을 따른다(arguments는 스택에 오른쪽에서 왼쪽으로 push된다.)

v. 스택 삽입 과정에서 중간에 0을 삽입함으로써 word-align을 맞춰야 할 것이다.

C. System call

i. 수정할 방식

1. System call handler: Syscall handler에서 frame의 32bit word를 읽어 핸들러 번호와 인자를 받는다. 이후 해당 핸들러에 맞는 핸들러를 호출하는 방식으로(아래의 구현한 halt와 같은 함수를 호출) syscall handler를 구현할 것이다.

2. User process manipulation(halt, exit, exec, wait): halt의 경우는 Pintos를 종료하는 함수를 호출하고, exit은 실행 중이던 스레드를 종료하므로 thread_exit을 통해 구현할 예정이다. Exec의 경우는 파일 이름을 받아 해당 파일을 실행 파일로 바꿀 것이다. Wait의 경우 대기를 해야하므로 semaphore를 이용하여 lock을 거는 방식으로 구현할 계획이다.

3. File Manipulation(file system): Process처리는 process.c에 있는 함수들을 이용하여 process_fork와 같은 함수를 사용하고, file 관리는 file system에 있는 함수들을 활용하여, filesys_create, file_seek, file_tell과 같은 함수들을 사용할 수 있다면 최대한 활용하여 사용할 것이다. 파일을 동시에 작업할 수 없도록 만들어야 하므로 filesys.c, file.c에 파일을 관리하는 함수를 만들어 해당 함수를 호출하는 방식으로 구현할 예정이다.

ii. 데이터 구조와 구체적 알고리즘

1. Syscall handler: handler 번호를 받은 후 switch 문을 이용하여 알맞은 핸들러를 호출할 예정이다.

2. 다른 함수들의 경우 특별한 데이터 구조를 사용하진 않겠지만, lock, aquire가 필요할 수 있어 semaphore를 사용할 수 있을 것으로 생각된다.

D. Denying writes to executables

i. 시스템 콜 open에서 파일을 연 후, file_deny_write 함수를 호출함으로써 이 파일에 대한 inode -> deny_write_cnt를 하나 늘린다. 이렇게 하면 file_allow_write에 의해 이 값이 감소되어 0이 될 때까지 write를 할 수 없게 된다.

ii. file_allow_write 함수 같은 경우, filesys/file.c의 file_close에서 호출하고 있으므로 파일이 close되면 자동으로 이 함수가 호출될 것이다. 따라서

이 함수를 따로 추가로 넣어주지 않아도 될 것이다.