Project2 Final report

20210084 김지민, 20210216 양준영

1. Process Termination Messages
    A. Solution
        i. syscall.c 변경사항
            1. “exit” 함수에 “printf("%s: exit(%d)₩n", thread_name(), status);” 코드를 추가함으로써 프로세스의 이름과 exit code를 출력하도록 한다.
    B. Discussion
        i. 기존 디자인과의 비교: 코드 한 줄밖에 나오지 않았기에 계획한대로 코드를 수정할 수 있었다.
        ii. 어려웠던 점: 딱히 없었다.
2. Argument Passings
    A. Solution
        i. 기존 process_execute 함수에서는 program 이름과 argument들이 공백으로만 구분된 채 하나의 문자열 ‘file_name’에 들어있었다. 이 함수에서 start_process 함수의 매개변수로 이 file_name을 그대로 넘겨주는데, start_process에서는 이 file_name을 그대로 load 함수에 넣음으로써 프로그램명으로서 쓴다. 또한 start_process에서 interrupt frame을 초기화하는데, 우리는 argument들을 stack에 push하는 것까지 완료해야 한다. 따라서 이 frame을 초기화하고 load 함수를 호출하는 “start_process” 함수에서 parsing 작업과 스택 push 작업을 구현하였다.
        ii. process.c 변경사항
            1. process_execute함수에서 thread_create 함수를 호출할 때 file_name을 그대로 thread의 name으로 쓰는 것을 볼 수 있다. 따라서 strtok_r 함수를 이용하여 program_name 변수에 프로그램명만 따로 저장한 후, thread_create 함수에 file_name 대신

program_name을 넣어준다.

2. thread_create 함수를 호출하기 전 program_name의 이름으로 실행 가능한 파일이 있는지 검사한 후 없다면 -1을 return 한다.

3. 이후 호출된 start_process 함수에서 load 함수를 호출하기 전 프로그램명과 argument들을 분리해야 한다. 따라서 load 함수를 호출하기 전, for문과 strtok_r 함수를 이용하여 file_name을 공백 기준으로 분리한 후 각 단어는 argu 문자열 배열에 저장한다. 이때 단어가 저장될 때마다 argument의 개수를 의미하는 argu_num도 1씩 증가시킨다.

4. load를 호출한 이후 argument들을 stack에 저장하는 작업을 거쳐야 한다. 나눠준 자료를 보면, stack의 top에서부터 argument들을 마지막 것부터 차례대로 push하고, word-alignment를 거친 후 각 문자열이 저장된 stack 주소를 차례대로 push 등의 순서로 진행된다.

5. 우선 각 argument들을 마지막 것부터 push하기 위해 for문을 argu_num-1부터 시작하여 1씩 감소하여 0번까지 진행한다. If_.esp 값을 (argu[i] 길이 + 1)만큼 미리 감소시킨 후(마지막 null값까지 들어가야 하므로 1을 더 빼 줘야 함), 해당 주소에 argu[i]값이 들어갈 수 있도록 memcpy 함수를 호출한다. 이때, null값까지 들어가야 하니 size는 문자열 길이 + 1로 넣어야 한다. 그리고 현재 esp값을 argu_address[i]에 저장한다. Word-alignment를 거친 후 각 문자열을 저장한 스택 주소도 push해야 하기 때문에 argu_address배열에 저장하는 것이다. 참고로 argu_address[argu_num]에는 0을 저장하여 argument배열의 끝을 표시하도록 한다.

6. 각 argument들을 다 push했다면, word-alignment과정을 통해 esp가 4의 배수 값을 가지도록 한다. 4의 배수가 될 때까지 "1 감소 + 0 삽입"과정을 반복한다.

7. Word-alignment를 거친 후 argu_address[argu_num]부터 argu_address[0]까지 push한다. 이때, memcpy에 argu_address[i]가 아닌, & argu_address[i]를 인수로 넣어야 argu_address[i]에 저장된 스택 주소가 제대로 스택에 push될 수 있다. (만약 argu_address[i]

를 삽입하면, argu_address[i]가 가진 스택 주소에 있는 문자열을 스택에 push하게 된다.)

8. 위 과정을 다 거치고 나면 esp는 argu_address[0]이 push되어 있는 곳을 가리키게 되는데, 이 값을 argu0 변수에 저장하고 esp를 4 감소시킨 후 argu0 값을 저장한다. 이는 argu_address[0]이 저장된 스택 주소를 push하는 것이다.

9. 또한 argument의 개수도 push해야 하므로 argu_num 값도 push한다.

10. 이후 fake return address로 0을 push하면 스택 push작업은 완료된다.

11. 마지막으로 if_.edi에는 argument의 개수를, if_.esi에는 문자열의 시작주소를 넣어줌으로써 완료한다.

B. Discussion

i. 기존 디자인과의 비교: 기존 디자인에서는 start_process 함수에서만 parsing 작업이 필요하다 생각했는데, 실제로 구현 단계에서 생각해보니 process_execute함수에서 thread의 name으로 argument까지 다 들어간다는 것을 깨달았다. 따라서 process_execute 함수에서도 parsing 작업을 통해 프로그램명만 name에 들어갈 수 있도록 했다. 또한 esp의 값만 바꾸는 것을 생각했는데, 인수로 스택에 들어간 argu_num값과 argument들의 배열 시작주소를 가리키는 레지스터 값도 바꿔야 한다고 생각하였다. 사실 이건 안 해도 될 것 같았지만, 혹시 몰라 edi값과 esi 값을 각각 argu_num과 argument 시작주소로 저장하였다.

ii. 어려웠던 점: 아직 system call을 구현하기 전에 test를 돌렸는데 계속 kernel panic이 나왔었다. 스택 삽입이 잘못된 줄 알고 새로 짠 코드 부분만 고려했었는데, 알고 보니 project1에서 짰던 부분이 여기서 panic을 일으키고 있었다. 이전에 thread.c에 추가한 check_prior 함수의 조건문 부분에 "!intr_context"부분을 추가하니 무사히 통과했었다. 이를 통해 기존 코드가 예상치 못한 결과를 가져올 수 있다는 것을 깨달아 완전히 새로운 코드 위에 다시 코드를 작성하였다.

.

3. System call

A. Syscall.c Solution

i. 먼저 syscall.c의 handler에서 f->esp에 핸들러 번호, 인자들이 순차적으로 담겨오기 때문에, f->esp의 값이 정상적인지를 check()를 통해 확인하고 정상적이라면, 이로부터 핸들러 번호를 받는다. 또한, 인자들은 f-esp+4, f->esp+8, f->esp+12로 전달되기 때문에 해당 값을 형변환하여 전달하였다. 형변환을 위해 #define으로 매크로를 선언하여 변환을 편하게 만들었다.

ii. 핸들러 번호를 받고 case switch문을 이용하여 번호에 맞는 함수를 실행시킨다. 핸들러 번호는 syscall-nr.h에 enum 형태로 존재하여 SYS_HALT와 같이 선언하여 사용하였다.

iii. Halt: halt의 경우, 어려운 점 없이, 핀토스 자료에 적힌 shutdown_power_off()를 실행하면 된다,

iv. Exit: 이 함수의 경우, file_descripter에서 열린 파일들을 전부 닫고 null로 설정한 후, thread_exit()을 실행하였다. 또한, wait에서 기다리는 스레드를 위해 해당 스레드의 exit을 status로 선언하여 종료 상태를 저장해 놓는다.

v. Exec: 이 경우에도 process_execute를 실행하도록 하였다.

vi. Wait: 이 경우에 process_wait(pid)를 실행하도록 하였다.

vii. Create: file이 null인지만 확인하고 filesys_create를 실행한다

viii. Remove: file이 null인지만 확인하고 filesys_remove를 실행한다.

ix. Open: file이 null인지 확인하고 아니라면, filesys_open으로 file을 연 후, 해당값이 없다면 -1을 반환하고, 있다면, thread_current()의 file descriptor에 저장한다. 이 때 이름을 확인해 열린 파일이라면 file deny를 한 후, 해당 위치를 반환한다. Syn-read을 위해 open, read, write에 lock을 걸어 synchrounous를 유지하였다.

x. Filesize: file이 null인지만 확인하고 file_length를 실행한다.

xi. Read: buffer가 null이거나 user address가 아닌지 확인한 후, fd 번호에

따라 명령어를 수행한다. Fd==0이면 stdin이기 때문에, input_getc를 통해 값을 받아 buffer에 저장한다. 이 때, 받은 값이 ₩0라면 종료하고 길이를 반환한다. 만약 fd>2이라면 stdin, stdout, stderr가 아니므로, file descriptor에 저장된 값을 file_read로 가져온다(이 전에 먼저 NULL인지 확인한다. read에서도 synchronous를 유지하기 위해 lock을 사용하였다.

xii. Write: buffer가 null이거나 user address가 아닌지 확인한 후, fd 번호에 따라 명령어를 수행한다. Fd==1이면 stdout이기 때문에, putbuf를 통해 값을 받아 buffer에 저장한다. Fd>2이라면 stdin, stdout, stderr가 아니므로, file descriptor에 저장된 파일을 확인하여 deny_write라면 deny_write를 실행하고 아니라면 file_write로 적는다. (이 전에 먼저 NULL인지 확인한다. write에서도 synchronous를 유지하기 위해 lock을 사용하였다.

xiii. Seek: file이 null인지만 확인하고 file_seek를 실행한다.

xiv. Tell: file이 null인지만 확인하고 file_tell를 실행한다

xv. Close: file이 null인지만 확인하고 file_close를 실행한다

xvi. Check: 주어진 address가 올바른 주소인지를 확인하는 함수로, 주소가 NULL이거나, user_vaddr가 아니라면 exit(-1)을 하도록 구현하였다.

xvii. check_file: 주어진 fd에 파일이 정상적으로 존재하는지를 확인하는 함수로, 비어있다면 exit(-1)을 하도록 구현하였다.

B. Process.c 변경점

i. Process_execute에 현재 실행중인 thread가 종료되었음을 부모에게 알릴 수 있도록, sema_down(&thread_current()->load)를 추가하였다. 또한, 비정상적이게 종료된 자식들을 기다리도록 자식 리스트인 children을 탐방하며 exit==-1인 스레드들을 wait하도록 구현하였다.

ii. Start_process: 자신의 부모 스레드에 자식이 실행중임을 알리기 위해 sema_up(&thread_current()->parent->load)를 해준다.

iii. Process_wait: 자식이 종료되는 것을 기다리기 위해 자신의 자식들 중 tid가 같은 자식들의 종료상태를 받을 수 있도록 sema_down(&now->wait)을 하고, 부모의 종료상태를 자식의 것을 받아온다. 또한, 자식 리스트에서 해당 자식을 지우고, sema_up(&now->kill))을 통해 자식들이

전부 종료된 후에 exit을 반환할 수 있도록 하였다.

iv. Process_exit: 자식들이 종료된 것을 알리고 죽을 수 있도록 sema_up(&cur->wait), sema_down(&cur->kill)을 추가하였다.

C. Exception.c 변경점

i. OS에서 page_fault가 발생해선 안되므로, page fault가 발생하는 원인 중 존재하지 않는 경우인 not_present, user가 false거나, 커널 스레드인 경우 exit(-1)을 통해 종료될 수 있도록 변경하였다.

D. Thread.c 변경점

i. 종료된 상태를 저장하는 exit, 자식들이 exit을 전달하고 죽고, 부모는 자식들이 모두 죽은 후 wait이 끝날 수 있도록 하는 wait, kill, 부모의 process_execute가 자식의 start_process보다 먼저 끝나는 것을 막기 위한 load 3가지의 세마포어를 추가하였다. 또한, 자신의 부모와 자식을 저장하기 위한 parent, children을 추가하고, children list를 관리하기 위해 list_elem child_elem을 추가하였다. 또한, file descriptor를 관리하기 위해 최대치인 128개만큼의 file 배열을 선언하였다.

E. Discussion

i. 기존 디자인과의 비교: 자식이 부모에게 전달해야 할 정보나, 자식이 부모보다 먼저 죽는 것을 생각하지 못해 이에 필요한 semaphore를 추가하였다. 또한, read, write의 동기화 문제를 해결하기 위한 lock 또한, 추가하였다. File descriptor를 저장하는 문제를 해결하기 위해 file descriptor 배열을 thread에 추가하였다.

ii. 어려웠던 점

1. 가장 어려웠던 문제는 process들 사이의 wait에 관한 문제였다. 부모가 자식보다 먼저 끝나는 것을 생각하지 못하여 semaphore를 추가하는 것에 어려움을 겪었다. 특히, 비정상적으로 종료된 자식들을 확인하는 과정을 추가하지 않아 test를 통과하지 못해 이 부분을 추가하는데 애를 먹었다.

2. 중간중간 address들이 정상적인지 확인하지 않은 부분들이 있어, page fault되는 문제가 있었다. 어디에서 검사를 하지 않았는지를

몰라서 이를 찾는데 어려움이 있었다.

3. Start_process에서 load를 성공했을 때에만 스택에 값을 저장했어야 했는데, 실패했을 때도 값을 저장해 multi-oom에서 낭비되는 자원이 있었던 것으로 파악되었다. 이 문제로 인해, multi-oom을 통과하지 못해 상당한 시간을 소요하였다.

4. Denying Writes to Executables

A. Solution

i. 현재 실행 중인 프로그램 파일에 대한 write를 막는 작업을 구현해야 한다. syscall.c의 open 함수와 write 함수에서 파일을 열고 쓰려고 할 때 조건 검사를 하여 file_deny_write 함수를 호출한다.

ii. syscall.c 변경사항

1. open함수에서 file을 열 때, 열고자 하는 file과 현재 실행 중인 file이 같다면 file_deny_write 함수를 호출해야 한다. filesys_open 함수로 file을 연 후 파일 디스크립터 배열에 할당하기 전에 strcmp로 현재 파일명과 인수로 받은 파일명이 같은 지 조사한다. 만약 같다면 file_deny_write를 호출하여 해당 파일에 대한 write 거부를 하나 증가시킨다.

2. write 함수에서 file_write 함수를 호출하기 전에 if문으로 현재 쓰려고 하는 파일의 deny_write 값이 true인지 검사한다. true하면 이 파일에 대한 쓰기가 거부된 상태이므로 file_deny_write함수를 호출함으로써 write 거부를 하나 증가시킨다.

B. Discussion

i. 기존 디자인과의 비교: 기존에는 open함수에서만 증가시키면 될 것이라 생각했는데, write 함수에서도 file_deny_write 함수를 호출해야 현재 실행 중인 파일에 대한 모든 write들을 막을 수 있었다.

ii. 어려웠던 점: write함수에서 file_deny_write 함수를 호출할 때, 조건문으로 deny_write값을 조사하는 과정에서 계속 오류가 났었다. 이는 file 구조체가 syscall.c 안에 정의되어 있지 않아 생기는 문제였다. 구조체를 추가시켜도 계속 오류가 뜨기에 에러 메시지를 보니 이번엔 구조체 속

"off_t"에 대한 정의가 포함되어 있지 않다고 되어 있었다. 그래서 #include "filesys/off_t.h"를 추가시킨 끝에 드디어 deny_write 값에 접근할 수 있었다.