OS Project2 Design Report

20180038 박형규, 20180480 성창환

1. Analysis on current Pintos system

1.1 Process execution procedure

현재 pintOS에 구현되어있는 프로세스가 실행되었을 때 어떤 방식으로 실행이 되는지 알아보자. 먼저 main 함수에서 run_action(argv) 함수가 실행된다. 이 때, run option이 활성화되어 있는 경우 run_task() 함수를 호출한다. run_task() 함수에서는 user process가 실행될 수 있도록 process 생성을 시작하고 process 종료를 대기하는 역할을 한다. 이는 process_wait(process_execute(argv))를 통하여 이루어진다. 이 때 process_wait()에서 하는 역할은 자식 프로세스가 종료될 때 까지 대기하는 역할인데, 이는 naive pintOS는 구현이 되어 있지 않다. 인자로 넘어온 process_execute() 함수에서는 process 생성 함수인 start process 함수를 thread_create() 함수의 인자를 통하여 호출하고 그로부터 반환받은 tid를 return해준다. thread_create() 함수에서는 thread를 생성하여 run queue에 추가해준다. thread_create() 함수의 인자로 호출된 start_process() 함수에서는 success라는 boolean type의 variable에 load() 함수의 결과를 저장하는데, 이 load()함수는 시작한 process를 메모리에 적재시키는 역할을 한다. 이는 setup_stack()의 함수 등을 통하여 이루어진다. 만약 load() 함수의 반환값이 옳지 못하다면(에러가 발생했다면) thread_exit()을 통하여 스레드를 종료시킨다. 이러한 Process execution procedure를 flow chart로 표현하면 아래 그림과 같다.

1.2 System call procedure

naive pintOS에는 system call handler가 구현되어 있지 않아서 system call이 호출될 수 없다. 따라서 이번 프로젝트에서는 system call handler를 구현하여 pintOS에서 응용 프로그램이 정상적으로 동작할 수 있도록 할 것이다. 먼저 naive pintOS에서 system call procedure을 살펴보자. User mode에서 system call이 발생하면 Kerner mode로 이동하여 Interrupt Vector table에 존재하는 syscall_handler()를 호출한다. 하지만 현재 syscall_handler()가 구현되어 있지 않으므로 아무런 동작을 하지 못하고 thread_exit() 하게 된다. 그래도 system call procedure의 가장 핵심적인 부분인 user mode에서 syscall이 발생하였을 때 kernel mode로 넘어가서 syscall에 대해 해결한 뒤 다시 user mode로 돌아오는 식으로 구현되어있다.

1.3 File system

이번 project에서 file system을 focusing 하여 다루는 것은 아니지만 user program이 file system으로부터 loaded되고, 우리가 이번 project에서 implement할 많은 syscall들이 file

system과 관련하여 다루어지므로 file system에 대해서도 살펴보아야 한다. naive pintOS에 구현되어 있는 file system은 많은 limitation을 가지고 있다. 먼저 internal synchronization이 없어서 여러 process가 file system에 접근하려고 하면 오류가 발생할 수 있다. 두 번째로 file size가 처음 create할 때의 size로 fixed되어 있다. 그리고 file의 data가 disk sector의 contiguous range를 차지한다. 따라서 External fragmentation이 disk memory 관련 문제가 될 수 있다. 이외에도 자잘한 limitation으로 subdirectories가 존재하지 않고, file name이 14자로 제한되어 있으며 file system을 repair하는 tool이 존재하지 않아서 한번 corrupt되면 복구할 수 없다는 문제점이 존재한다.

2. Solutions for each requirements

2.1 Process Termination Messages

프로세스가 종료될 때 process termination message를 출력하는 것은 syscall에서의 exit()에서 구현된다. process termination message를 kernel thread가 terminate될 때나, halt syscall이 발생할 때는 print하면 안된다.

2.2 Argument Passing

먼저 process_execute() 함수에서는 file_name 문자열을 파싱하고 파싱한 첫 번째 토큰을 thread_create() 함수에 thread name으로 전달해야 한다. 따라서 file_name 문자열을 strtok_r() 함수를 이용하여 파싱하는 과정이 필요하다. 다음으로 start_process() 함수에서는 file_name 문자열을 파싱하고, user stack에 파싱된 token을 저장하는 함수를 이용하여 stack에 파싱한 token들을 저장해야 한다. 따라서 먼저 strtok_r() 함수를 이용하여 띄어쓰기를 기준으로 file_name 문자열을 파싱하고, 새롭게 구현한 user stack에 파싱된 token을 저장하는 함수를 호출하여 stack에 파싱한 token들을 저장한다. 그렇다면 stack에 프로그램 이름과 인자들을 저장하는 함수는 어떻게 구현해야 할까? 먼저 이 함수의 인자로는 프로그램의 이름과 인자가 저장되어 있는 메모리 공간, 인자의 개수, 스택 포인터를 가리키는 주소가 필요하다. 이 함수에서는 프로그램의 이름 및 인자를 stack에 push하고, 프로그램 이름 및 인자의 주소들 또한 stack에 push 한다. 그 뒤에 argv와 argc를 push 하고 fake address를 저장한다.

2.3 System Call

먼저 syscall handler를 구현하기 전에 syscall handler에서 하는 역할을 살펴보면 syscall handler에서 system call number에 해당하는 system call을 호출한다. 그리고 esp 주소와 인자가 가리키는 주소가 user range인지 확인해야 한다.(pintOS는 user range를 벗어난 주소를 참조할 경우에 page fault가 발생한다.) 그리고 user stack에 있는 인자들을 kernel에 저장하고 system call function의 return 값은 interrupt framce의 eax에 저장해야 한다. 이를 위해 먼저 우리는 주소의 유효성을 검사하는 함수를 구현해야 한다. 이 함수에서는 포인터가 가리키는 주소가 user range(0x8048000 ~ 0xc0000000)인지 확인한다. 만약 user

range를 벗어난 영역을 참조하였을 경우에는 exit(-1)을 통해 process를 종료한다. 다음으로 user stack에 있는 인자들을 kernel에 저장하는 함수를 구현해야 한다. 이 함수에서는 stack에서 인자들을 4byte 단위로 꺼내어서 arg 배열에 순차적으로 저장한다. 이 때, stack에서 가져오는 인자의 개수는 count 개수 만큼이다. 다음으로 syscall_handler 함수는 user stack에 저장되어 있는 system call number를 이용하여 구현한다. 먼저 stack pointer가 user range인지 확인하고, 저장된 인자 값이 pointer일 경우 user range의 address인지 확인한다. 그 뒤에 알맞은 system call을 찾아 system call 함수를 호출한다. 우리가 이번 project에서 구현하게 될 system call 함수는 총 13개이다. 먼저 halt()는 shotdown_power_off()를 사용하여 pintos를 종료시킨다. exit()은 실행중인 스레드 구조체를 가져오고 process 종료 메세지를 출력하며 thread를 종료시킨다. exec()은 cmd_line으로 주어진 program을 실행하고 새로운 process의 pid를 return한다. 만약 실패할 경우 -1을 return한다. wait()은 pid로 주어진 child process가 terminate되기를 기다린다. create()은 파일 이름과 크기에 해당하는 file을 생성하며 file 생성 성공 시 true를 반환하고, 실패시 false를 반환한다. remove()는 파일 이름에 해당하는 file을 제거하며 파일 제거 성공 시 true를 반환하고, 실패 시 false를 반환한다. open()은 파일 이름에 해당하는 file을 open하며 성공하였을 시 file descriptor의 number에 해당하는 nonnegative integer을 return하고 실패하면 -1을 return한다. filesize()는 파일의 size를 return한다. read()는 열려져 있는 file을 size만큼 read하고 이를 buffer에 저장한다. write()은 열려져 있는 file에 size byte만큼 buffer에서 쓴다. seek()는 open file fd에서 다음에 읽거나 쓸 위치를 position으로 바꾼다. tell()은 open file fd에서 다음에 읽거나 쓸 위치를 return해준다. close는 file fd를 닫아준다.

2.4 Denying Writes to Executables

실행중인 파일에 데이터를 기록하게 되면 프로그램이 원래 예상했던 데이터와 다르게 변경된 데이터를 읽어서 오류가 발생할 수 있다. 따라서 이를 해결하기 위해 현재 실행 중인 process에 대한 write access를 denying 해야 한다. 이를 위해서는 파일이 실행되는 위치를 정확히 파악하고, 실행 중인 프로그램 파일의 데이터가 변경되는 것을 예방할 수 있도록 해야 한다. 그리고 프로그램이 종료 되었을 경우에만 실행중인 파일의 데이터가 변경될 수 있도록 해야한다. 이를 구현하기 위해 먼저 thread 구조체에 현재 실행중인 file을 나타내는 list type의 variable을 선언해야 한다. 그 뒤 load() 함수에서 실행 할 파일을 얼 때 데이터 변경을 예방하도록 해야한다. 따라서 실행할 file을 open할 때 lock 을 획득하도록 하고, file이 없는 경우나 file 실행이 끝났을 경우 lock을 release한다. 그리고 thread 구조체에 새롭게 추가한 현재 실행중인 file을 현재 실행할 파일로 할당한다. 이후에 file을 open 할 때 file_deny_write() 함수를 호출하여 process가 execute되는 도중에 file에 write을 예방한다. process_exit() 함수에서는 process가 종료되었을때 실행 중인 file의 data가 변경 될 수 있도록 해야한다. 따라서 현재 process가 실행 중인 파일을 file_close() 함수를 이용하여 닫는다. 이 함수가 호출되면 file_allow_write() 함수가 internal 하게 호출되어 file의 data가 변경되는 것을 허락하게 된다.