Project 2 design-report

CSED312-01 운영체제 PintOS - Threads

October 27, 2019

27조
A: 최우석(단일계열, 49003157, f2sound)
B: 주봉조(컴퓨터공학과, 20100374, jbj0708)

1. HOW TO ACHIEVE EACH REQUIREMENT?

1. Process Termination Messages

pintos docs 분석:

어떠한 이유로든간에 user program이 종료되면, 해당 프로세스의 이름과 exit_code를 출력하기.

프로세스의 이름은 process_execute() 함수를 거쳐가야 한다(커맨드라인 argument는 생략)

커널 스레드가 종료되거나(이는 유저프로세스 종료가 아님), halt 라는 시스템콜이 호출되면 프린트하지 말기.

구현계획:

프로세스의 파일 이름은 `process_execute (const char *file_name)` 에서 알 수 있다.

프로세스의 exit code는 스레드의 `thread_status`이며, 프로세스 이름은 `char name[16]`이다. `process_exit()`에서 이 두 정보를 print해준다.

2. Argument Passing

pintos docs 분석:

현재 `process_execute()`는 새로운 프로세스에 대해 passing arguments를 지원하지 않고 있다. 이 함수를 확장하여, 프로그램 파일 이름을 argument로 사용하는 대신 공백으로 단어로 나누는 기능을 구현해라. 첫 번째 단어는 프로그램 이름이고 두 번째 단어는 첫 번째 argument이다.
즉, `process_execute ("grep foo bar")` 는 두 개의 arguments `foo` 및 `bar`를 전달하여 `grep`을 실행해야 한다.

command line 에서 여러 공백은 단일 공백과 동일하므로 `process_execute ("grep foo bar")` 는 원래 예제와 동일하다. command line argument의 길이에 제한을 둘 수 있다. 가령 argument를 단일 페이지(4kB)에 맞는 argument로. (핀토스 유틸리티가 커널에 전달할 수 있는 명령행 인수에는 128바이트 관련 제한이 없긴 하다).

내가 원하는 방식으로 argument strings를 parse 할 수 있다. 감이 안 잡힌다면, "lib/string.h" 에 프로토타입이 있는 `strtok_r()` 을 살펴보십시오. 매뉴얼을 보면 자세한 내용을 알 수 있습니다(프롬프트에서 man strtok_r 실행).

```
char* strtok_r (char *s, const char *delimiters, char **save_ptr)
 /* s는 분리하고자 하는 문자열, delimiters는 구분자(무엇을 기준으로 분리할것인가). 여기에서 분리자를 공백으로 줘야 한다.
    save_ptr은 함수 내에서 토큰이 추출된 뒤 남은 녀석을 가리키기 위한 것이다.
    즉 strtok_r의 리턴은 s의 가장 앞에 있는 녀석이고, 이후 두번째 녀석에 접근하고 싶다면 두 번째 strtok 호출 전 s = save_ptr 해줘야 한다.
    사용에 대한 자세한 예제는 https://codeday.me/ko/qa/20190508/495336.html 참조. */
```

1.1. Program Startup Details (docs p.36)

`'/bin/ls -l foo bar'`라는 command는 어떻게 다뤄지는가?

1. command 가 단어로 쪼개진다. `'/bin/ls'`, `'-l'`, `'foo'`, `'bar'`.
2. 위 단어들을 스택의 가장 위에 넣는다. (포인터로 참조되므로 각 단어의 순서는 중요하지 않다.)
3. 각 단어(string)의 주소와 null pointer sentinel 을 스택에 push한다. (right-to-left 순서로)

이 녀석들은 모두 `argv` 의 elements 다. null pointer sentinel은 `argv[argc]`가 널포인터일 경우를 대비한 것이다(C standard). 이 순서는 `argv[0]`이 가장 낮은 virtual address에 있도록 한다. 첫 번째 push 전에 스택포인터를 4의 배수로 내린다.

4. `argv`(argv[0]의 주소)와 `argc`를 순서대로 push한다.
5. 가짜 "return address"를 push한다.

entry function은 절대 return되지 않지만, 그것의 스택프레임은 다른 프레임 구조와 동일해야한다.

docs p.37의 스택테이블 참조하기

1.2. Stack(User virtual memory) (레이아웃은 docs p.26)

- virtual address 0 up to `PHYS_BASE` (which is defined in "threads/vaddr.h") and defaults to `0xc0000000`(3GB).
- Kernel virtual memory가 나머지 virtual address space를 점유한다.

- 프로세스당 하나를 할당받는다. 커널이 프로세스를 switch하면, 프로세서의 page directory base register를 바꾸면서 user virtual address spaces도 함께 switch한다. (`pagedir_activate()` in "userprog/pagedir.c")
- 유저프로그램은 본인의 user virtual memory에만 접근할 수 있다. 나머지는 예외처리된다. 커널은 현재 실행중인 유저프로세스의 virtual memory에도 접근할 수 있다.
- 유저스택의 사이즈는 고정되어있다. 이는 Project3 에서 확장할 것이다.

Argument Passing 구현을 위한 질문

- 유저프로그램의 실행으로 `process_execute (const char *file_name)` 함수가 실행될 때, 스택의 할당과 스택으로 push(fn_copy) 하는 코드는 어디에 있는 건가?

유저프로그램의 실행을 위해 `file_name`이 전달되는 큰 그림은 아래와 같다.

a. 프로그램 실행 --> `process_execute()` 안에서 `thread_create()` 및 `start_process()`.

b. `start_process (void *file_name_)` 안에서 유저프로그램 `load(file_name, &if_.eip, &if_.esp)`.

`eip`는 실행할 명령의 주소, `esp`는 현재 진행하는 함수의 제일 아래부분의 스택포인터이다.

여기에서 파일이 실행가능하다면, `intr_exit`(in "threads/intr-stubs.S") 인터럽트로부터의 리턴을 시뮬레이션(?)하여 유저프로세스를 실행함. 이 `intr_exit`은 스택의 모든 arguments를 `struct intr_frame` 형식으로 가져오기 때문에, 우리는 스택포인터(`&esp`)를 우리의 스택프레임으로 가리키게 한 다음에 그것을 점프한다(?).

c. `load (const char *file_name, void (**eip) (void), void **esp)` 안에서 `setup_stack (esp)`를 통해 스택포인터, 즉 스택을 초기화(이를 통해 esp는 `PHYS_BASE`로 초기화됨).

구현계획:

a. `process_execute()`에서 `thread_create()`를 호출하기 전 파일 이름을 `strtok_r`을 통해 토큰화한 뒤, 이를 `thread_create()`의 첫 번째 인자로 넣어준다.

`process_execute()`에서 `thread_create()`를 호출할 때 `start_process (void *file_name_)`이 호출된다(이 때 `file_name`은 `thread_create()`에 argument로 전달된 토큰이 아닌 `process_execute()`의 argument로 전달된 `file_name`이다). 이후 `start_process()` 내에서 `load (file_name, &if_.eip, &if_.esp)` 를 호출하는데, 이 때 argument로 전달되는 `file_name`은 역시 토큰이 아닌 그냥 생짜 `file_name`이다.

b. `load (const char *file_name, void (**eip) (void), void **esp)` 에서 실질적으로 파일을 여는데,이 때 전달된 `file_name`을 `file_name`의 첫번째 토큰으로 변경해야 한

다. (`file = filesys_open (file_name)`에서의 `file_name`을 토큰의 제일 첫번째 토큰으로 변경)

`load ()` 내에서 호출되는 `set_up (esp)` 를 변경해야 한다. `static bool setup_stack (void **esp)`를 보면, 스택을 위한 페이지를 할당받는 게 성공하면 argument로 받은 `esp`가 `PHYS_BASE`로 초기화되는 것을 확인할 수 있다.

1.1. Program Startup Details 에 언급했듯, 우리는 command의 각 단어의 주소를 `argv`에 넣어 이 `argv`와 null pointer sentinel를 스택에 넣어야 한다. 이렇게 하기 위해서는, `set_up ()`를 호출하는 `load ()`에서 `file_name`을 토대로 `char **argv` 및 `int argc` 를 초기화 한 뒤, `esp`와 함께 이 둘도 `set_up ()`의 argument로 보내주어야 한다.

`load ()` 에서 `strtok_r()`을 이용하여 각 단어를 `argv[argc++]`에 넣는다.

`file = filesys_open (file_name)`을 `file = filesys_open (argv[0])` 으로 변경한다.

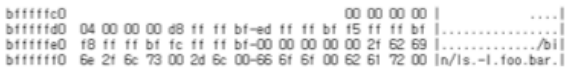
`setup_stack ()` 에 세 개의 argument (`void **esp, char** argv, int argc`) 가 전달되도록 변경한다.

`load ()` 에서 `setup_stack ()` 호출문의 argument를 위 세 개로 변경한다.

`setup_stack ()` 내에서, argument로 받은 `argv, argc`를 스택에 push한다.

스택은 다음과 같다.

Address	Name	Data	Type
0xbffffffc	argv[3][...]	bar\0	char[4]
0xbffffff8	argv[2][...]	foo\0	char[4]
0xbffffff5	argv[1][...]	-l\0	char[3]
0xbfffffed	argv[0][...]	/bin/ls\0	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()



(`word-align` 은 접근속도를 빠르게 하기 위해 4의 배수로 맞추기 위해 추가하는 녀석이라고 한다. 위 사진처럼 하자.)

스택의 push 방법은, `*esp`를 스택의 원하는 위치로 조정한 뒤 `*esp`에 넣고자 하는 녀석을 대입해주면 된다.

가령, 초기 스택포인터를 초기화하는 것은 `*esp = PHYS_BASE;` 이다. 이후 차례로 넣고자하는 녀석의 크기만큼 스택을 확장해준뒤(가령 `argv[i]`에 있는 한 단어를 넣고자 한다면, 하나의 문자는 1바이트이므로 확장해야 하는 스택의 사이즈는 `argv[i]`의 사이즈가 되며, 이는 `strlen[i]+1` 이다 (`'\0'`도 포함). 즉 `*esp -= (strlen(argv[i]))+1` 을 이용하여 스택 확장), 주소를 넣고자 한다면 `*(int*)*esp = argc;`, 내용을 복사하여 넣고 싶다면 `memcpy(*esp, argv[i], strlen(argv[i])+1);` 처럼 한다.

return address의 크기는 4이며, `*(int*)*esp = 0;` 이다.

3. System Call

pintos docs 분석:

3-1. User Memory Access

모든 시스템콜은 유저메모리를 read하는 게 필요하다. 그 중 몇몇 시스템콜은 유저메모리를 write하는 게 필요하다.

커널은 유저프로그램으로부터 제공받는 포인터을 통해 유저메모리에 접근해야 한다. 하지만 그 포인터가 이상한 녀석일 수도 있다. (null이라던가, unmapped virtual memory라던가, 커널 virtual address space를 가리키고 있다거나(above `PHYS_BASE`)...) 이러한 포인터들은, 해당 프로세스를 종료하고 자원을 회수함으로써 거절되어야 한다.

위 문제를 해결하기 위한 방법은 두 가지가 있는데, 첫 번째는 유저로부터 제공된 포인터가 타당한지 검증하는 방법이고("userprog/pagedir.c"와 "threads/vaddr.h"를 봐야 한다), 두 번째 방법은 유저포인터가(스택포인터) `PHYS_BASE` 아래에 있는지 확인하는 방법이다("userprog/exception.c"의 `page_fault()`를 수정해야한다). 일반적으로 두 번째 방법이 빠르다.

두 경우 모두 메모리누수가 일어나지 않도록 철저히 확인해야 한다. 어떤 경우에서?

example) 시스템콜이 lock이나 힙에 할당된 메모리를 획득한 상황에서, 잘못된 유저포인터와 조우하게 된다면 lock을 release하거나 메모리페이지를 free해야만 한다. 첫 번째 방법으로 포인터를 판단한다면 이 상황은 비교적 간단하게 해결될 수 있다. 하지만 만약 두 번째 방법(`PHYS_BASE`를 확인하는 방법)으로 포인터를 판단한다면 좀 어렵다. 왜냐하면 메모리 접근으로부터 error code를 리턴할 방법이 없기 때문이다. 이를 위해 핀토스는 두 번째 판단방법을 사용하는 사람을 위해 추가적인 code를 제공한다(docs p.27).

"threads/vaddr.h" 에 있는 `bool is_user_vaddr(const void *vaddr)` 함수를 이용하면 두 번째 방법을 사용할 수 있다. 이 함수는 인자로 넘겨받은 `vaddr`이 `PHYS_BASE`보다 아래에 있으면 true를 리턴한다.

유저프로그램 실행에 따른 page의 흐름

1. 유저프로그램이 실제로 실행되기 위해 `load()` 함수가 호출되면, `file_name`으로부터 파일을 열기 전에 스레드를 하나 생성하고, 그 스레드에 `pagedir_create()`를 통해 페이지를 하나 생성한다.
2. 스레드가 생성되고 페이지가 할당된 시점에서 `process_activate()`가 호출되고, 이 함수 안에서 `pagedir_activate(uint32_t *pd)` 함수가 호출된다. 이 함수에서는 어셈블리 코드를 통해 pd를 CPU's page directory base register(PDBR)에 넣는다. 이를 통해서 실질적으로 무언가가 되는 듯 하다.

"pagedir.h"의 함수들 중에서 `void *upage`, `void *kpage`를 인자로 받는 함수들이 있다. 첫 번째 방법에서 유저로부터 제공된 포인터가 타당한지를 검증하는 코드를 이 함수들 안에 짜야 하는 걸까?

시스템콜은 어떤식으로 호출되는거죠?

- "userprog/syscall.c"의 `static void syscall_handler(struct intr_frame *f UNUSER)`함수에서 인자로 `intr_frame *f`가 들어온다.
- `intr_frame`구조체는 "thread/interrupt.h" 에 선언되어있다.
- "lib/user/syscall.c" 를 보면 syscall 번호를 어셈블리어로 호출하하고 있음을 각 함수의 주석을 읽어보면 알 수 있다. 이 때, `syscall0`, `syscall1`, `syscall2`, `syscall3` 함수의 `asm volatile` 호출 부분을 보면 각각 `$4`, `$8`, `$12`, `$16`에 `esp`를 `add`하는 것 같은 수상쩍은 낌새를 눈치챌 수 있다.
- "lib/user/syscall.c" 를 자세히 보면, `$4`, `$8` 의 숫자가 각 함수의 인자의 수와 4의 배수로 매칭되는 것을 알 수 있다. 즉 인자가 없는 `syscall0(NUMBER)` 은 syscall의 번호에 해당하는 4바이트(?)만 할당하면 되기에 스택포인터를 4만큼 늘려주는 건가보다.
- "lib/user/syscall-nr.h" 에 각 syscall이 enum으로 명시되어있다. "lib/user/syscall.c"의 인자인 `NUMBER`를 이용하여 요 녀석들을 번호로 호출하는 것 같다.
- 정리해보자면, "userprog/syscall.c" 에서 인자로 받은 `intr_frame *f`의 데이터 중 시스템콜을 나타내는 숫자가 포함되는 것 같다. 즉 `esp`를 4만큼 더해준 뒤(스택을 늘려준 뒤) 그 자리로 syscall 어셈블리 함수의 인자가 차례대로 들어오는 듯하다.
- 즉 "lib/user/syscall-nr.h"에 선언된 시스템콜들의 순서에 맞게 번호에 따라 호출해주는 함수를 모두 구현해야 한다.

시스템콜을 호출하는 함수는 어디에 어떤 식으로 선언해야 하는 거죠?

- "lib/user/syscall.c" 에서 각 시스템콜 함수들을 위에서 언급한 `syscall0()`, `syscall1()` 등으로 호출하는 것을 볼 수 있다.
- 우리는 실제로 각 시스템콜 함수를 구현해야 하는데, 그 구현은 "userprog/syscall.c"에 하면 된다. 그리고 각 시스템콜 함수의 호출은 동일한 파일의 `static void syscall_handler (struct intr_frame *)` 이 담당한다.
- 앞서 확인해봤듯, `intr_frame` 구조체는 `esp`를 가지고 있다. 그러므로 이 핸들러의 인자로 들어오는 녀석의 `esp`의 위치를 확인해줌으로써 유저포인터가 제대로 된 녀석인지 확인하는 것이 아닐까 조심스레 예측해본다.
- 모든 시스템콜의 내용 및 파라미터 등의 정보는 pintos docs p.29에 나와있다.

구현계획:

User Process Manipulation

- `void halt (void)`

핀토스 종료. `shutdown_power_off()` 를 호출하면 된다.

- `pid_t exec (const char *cmd_line)`

`process_execute()`를 호출한 뒤, 이 리턴값이 `TID_ERROR`라면 -1을 리턴하고, 아니면 리턴값을 그대로 리턴한다.

- `int wait (pid_t pid)`

자식프로세스(`pid`로 식별, 즉 인자가 자식프로세스이다)를 기다리고, 자식의 exit status를 검사한다. 이를 구현하기 위해, 현재 프로세스(스레드)는 "thread/synch.h"에 정의된 `void cond_wait (struct condition *, struct lock *)`를 통해 자식프로세스의 종료를 기다리고, 반대로 자식프로세스는 `void cond_signal (struct condition *, struct lock *)`를 통해 종료를 알리게 한다. pintos docs를 보면 자식프로세스가 정상적으로 종료되지 않고 커널에 의해 종료되는 상황을 구분해두었는데, 이 때 이 함수는 -1을 리턴해야 한다. 또한 `pid`가 현재 이 함수를 호출한 스레드의 자식이 아닐 경우에도 바로 -1을 리턴해야한다. 참고로 스레드는 상속관계로 `wait` 할 수 없다. 즉, 자식의 자식을 인자로 `wait`을 호출할 수 없다. 또한, 이미 동일한 자식을 인자로 `wait`을 호출했을 경우에도 즉시 -1을 리턴한다.

위와 같은 기능을 수행하기 위해서는, 모든 스레드가 자신의 부모에 대한 포인터와 자식 스레드에 대한 포인터를 가지고 있어야 한다. 또한, 현재 스레드의 상태를 나타내는 변수와 종료여부를 나타내는 변수

를 가지고 있어야 한다. 이를 위해 `thread` 구조체에 `Tid_t parent_pid`, `Tid_t child_pid`, `bool isRun` 변수를 추가한다.

후술할 exit 함수에서 `isRun`과 `status`를 기록한다.

`status = THREAD_DYING`일 경우, `isRun = true`라면 자식스레드가 커널에 의해 종료된 것이라고 판단할 수 있다. 만약 `isRun = false`라면 이미 부모스레드에 의해 종료된 것이라고 판단할 수 있다. 자식스레드가 직속(?) 자식인지를 구분하기 위해서는 `child_pid`를 이용한다. 이미 `wait`을 부른 경우는 `status = THREAD_BLOCKED`으로 판단할 수 있다(?).

- `void exit (int status)`

현재 프로그램을 종료하고, `status`를 커널로 리턴한다.

`status = THREAD_DYING` 및 `isRun = false` 로 변경해준 뒤 `process_exit()` 를 호출한다.

File Manipulation

- `bool create (const char *file, unsigned initial_size)`

`filesys_create()` 함수를 호출한다.

- `bool remove (const char *file)`

`filesys_remove()` 함수를 호출한다.

- `int open (const char *file)`

파일을 열고 파일의 id인 `fd`를 리턴한다. `filesys_open()`을 통해 파일을 연다. 파일이 열리지 않으면 -1을 리턴한다.

`fd`는 자식에게 상속되지 않으며, 같은 파일이고 심지어 같은 프로세스에의해 파일이 실행되더라도 매번 다른 값을 가져야 한다. 이를 위해 fd는, 전역으로 `int file_open_count` 변수를 선언한 뒤 `filesys_open()`이 성공하면 `file_open_count++`를 리턴하도록 구현할 것이다.

이하의 함수들은 `fd`와 매핑되는 `file*`를 어떻게 찾을 것인지를 고민해야 하는데, 잘 모르겠다..

- `int filesize (int fd)`

핀토스에서는 inode 라는 자료구조 안에 파일의 모든 정보를 저장한다. "filesys/file.c" 에서 `struct file` 구조체를 볼 수 있으며, 여기에 데이터로 `inode`를 가지고 있음을 확인할 수 있다. 이 `inode->data->length` 가 파일의 사이즈를 의미한다.

- `int read (int fd, void *buffer, unsigned size)`

`fd != 0`이라면 `file_read()`를 호출하고, `fd = 0`이라면 `input_getc()`를 이용해 키보드 입력을 버퍼에 넣는다.

- `int write (int fd, const void *buffer, unsigned size)`

`fd != 0`이라면 `putbuf()` 함수를 이용하여 버퍼의 내용을 콘솔에 입력한다. 이 때에는 필요한 사이즈만큼 반복문을 돌아야 한다. `fd = 0`이라면 `file_write()`를 호출한다.

- `void seek (int fd, unsigned position)`

`file_seek()` 함수를 호출한다.
0이 파일의 시작지점이다.

- `unsigned tell (int fd)`

`file_tell()` 함수를 호출한다.

- `void close (int fd)`

`filesys_close()` 함수를 호출한다. 이후 `process_exit()` 호출 시 열려있는 모든 파일에 대해 `close()`를 호출해 닫아주어야한다.

4. Denying Writes to Executables

구현계획:

`read()`와 `write()` 함수에서 `file -> deny_write`를 통해 파일을 쓸 수 있는지 확인할 수 있다. 만약 리턴값이 true라면 -1을 반환하며, false라면 `file_deny_write()`함수를 호출한 뒤 read 혹은 write를 진행한 뒤 `file_allow_write()` 함수를 호출한다.