

# Operating System Project 3

Team #2

20165167 정현준. 20185125 이상범

프로젝트의 진행 순서는 Problem 2->Problem 3->Problem 1->Problem 4 입니다. 그러나 Problem 1, 3, 4는 상호 유기적으로 연관되어 있기 때문에 명확한 순서를 두고 구현하지 않았습니다. 그럼에도 보고서에는 순서대로 작성하였기 때문에 보고서에 이곳 저곳의 내용이 섞여 서술되었습니다.

## Problem 1. Process Termination Messages

### 1. Problem definition

현재 pintos에는 시스템 콜이 구현되어 있지 않다. 어떤 시스템 콜이 호출되어 System call handler가 실행되면, system call handler는 “system call!”을 출력할 뿐 아무것도 하지 않는다. Problem 1에서 우리는 어떤 user program이 종료될 때 아래와 같이 지정된 함수를 호출하여 지정된 형식의 문장을 출력하도록 만들어야 한다.

```
printf ("%s: exit(%d)\n", ...);
```

이 때 %s 에서는 process가 실행될 때 파일 이름의 인자로 전달된 명령어를 출력하고, %d 에서는 exit code를 출력해야 한다. 하지만, 유저 프로세스가 아닌 프로세스가 종료될 때나 halt 시스템 콜이 호출될 때에는 위 문장을 출력해선 안된다.

### 2. Algorithm design

Termination message의 구현은 세 부분으로 나뉜다. %s의 출력 부분과 %d의 출력부분, printf를 넣을 함수의 위치(즉, 시스템 콜 EXIT의 구현)이다. EXIT 시스템 콜의 구현의 경우 /pintos/src/lib/user/syscall.c를 참고하면 이번 프로젝트에서 구현해야 할 시스템 콜들이 사용할 함수를 알 수 있다. EXIT 시스템 콜의 경우 void exit (int status) 함수를 호출해야 한다. /pintos/src/lib/user/syscall.c에는 동시에 시스템 콜을 이행하는 함수들이 호출 시 입력 받는 argument 개수에 따라 그 argument들을 어떻게 사용하게 되는지를 정의한 매크로 함수가 있는데, 이를 통해 시스템 콜 함수들의 실행 시 넘겨주어야 할 argument를 esp에서 일정 수만큼 가감해서 사용하면 된다는 것을 알 수 있다. 또한 intr\_frame에서 읽은 esp로부터 syscall number를 알 수 있는데, 이는 시스템 콜의 구현에서 사용된다.

Problem definition의 printf문은 시스템 콜을 handle하는 과정에서 thread\_exit ()하기 전의 위치에 위치하여야 한다. 현재 pintos의 시스템 콜 handler로 사용되는 syscall\_handler ()함수에는 syscall\_handler ()함수가 호출되었음을 알리는 printf문과 syscall\_handler ()가 호출되자마자 스레드를 종료시키는 thread\_exit ()만 존재한다. 실제로 program/process execution 과정

에서 여러 번의 시스템 콜이 발생하므로 이를 handle할 수 있어야 하므로 이 시스템 콜을 다룰 방법에 대해 생각해 보아야 하고 이에 따라 `thread_exit()`의 위치 또한 변경되어야 한다. 시스템 콜을 다룰 방법은 Problem 3에서 다루고, `thread_exit()`의 위치를 바꾸는 부분에 대해서만 Problem 1에서 다루었다.

`%s`는 termination되는 프로세스의 이름으로, 이번 프로젝트에서 pintos는 한 프로세스당 한 스레드만을 사용하므로, 이 `%s`는 종료되는 프로세스의 이름임과 동시에 그 프로세스를 이루는 유일한 스레드의 이름이다. 따라서 `%s`로 출력할 인자로 `EXIT` 시스템 콜(`SYS_EXIT`)을 호출한 프로세스, 스레드의 이름을 사용하면 된다.

`%d`를 알기 위하여는 `EXIT` 시스템 콜을 어떻게 구현할지에 대해 먼저 알아야 한다. 이는 원래 implementation에 서술되어야 하나 이곳에 서술하기로 한다. `EXIT` 시스템 콜은 `userprog/syscall.c`의 `syscall_handler()`함수에서 정의되는데, `void exit(int status)`를 `syscall_handler()`함수 바깥에 정의하고 `syscall_handler()`함수에서 `exit()`함수를 호출함으로서 기능하게 한다. 따라서 Problem definition의 `printf`문은 이 `exit()`함수 내부에 위치하여야 한다. 이는 매우 직설적으로 `%d`로 넘겨주어야 할 인자가 무엇인지에 대한 힌트가 되는데, `void exit(int status)`의 인자인 `status`를 `%d`의 인자로 넘겨주어야 한다. 이 `status`는 시스템 콜 handler인 `syscall_handler()`함수에서 `exit()`함수를 호출하면서 넘겨주는 인자인데, 이 인자에 대한 설명은 앞서 첫 번째 문단에서 말한 바 있다.

### 3. Implementation

`EXIT` 시스템 콜을 포함하여, 시스템 콜을 handle 하기 위해 우선 `syscall_handler()`함수 내부에서 `switch - case` 문으로 handle 해야 할 시스템 콜의 경우를 나누어 주어야 한다. 어떤 시스템 콜을 handling 해야 할지는 `syscall_handler()`함수가 인자로 받는 `intr_frame`에서 `esp`를 참조하면 알 수 있다. `EXIT` 시스템 콜의 handling은 `intr_frame->esp == SYS_EXIT`인 case에 대해 시행된다. 이 경우 case문 내에서 `void exit(int status)` 함수가 호출되며, 그 인자로 `esp+4` 값을 넘겨주게 된다. `esp`에서 더해주는 값은 `hex_dump()`함수를 이용한 디버깅을 통해 정확히 유추하여야 한다. 이 때 `exit` 함수의 argument인 `status`의 자료형이 `int`이므로 `esp+4` 값을 넘겨줄 때, `*(uint32_t *)`로 typecasting 해야 한다. 이는 뒤의 Problem 3에서도 마찬가지로, 시스템 콜 handler에서 시스템 콜을 시행할 함수에 인자로 넘겨주는 `esp`에 얼마의 base를 더해야 하는지, 어떤 type으로 typecasting 해야 하는지에 대한 설명은 앞으로 생략하기로 한다.

이번 Problem 1에서 `exit()` 함수는 아래와 같이 구현하였다. Problem definition의 `printf`문과 `thread_exit()`, 스레드의 이름을 리턴하는 함수로 구성되어있다.

```
void exit(int status) {  
    printf("%s: exit(%d)\n", thread_name(), status);
```

```

thread_current()->exit_status = status;
}

```

이 때의 `thread_name()` 함수가 리턴하는 값은 프로그램 실행을 위해 입력된 `command`의 첫 번째 `argument` 즉, 후술할 `argv[0]`이어야 한다. 이 `argument`가 스레드의 이름이기 때문이다. 스레드의 이름은 `process_execute()` 함수 내부에서 선언되는 `thread_create()` 함수의 첫 번째 인자로 결정된다. 그런데 `pintos` 구현상 입력된 `command` 전체를 스레드의 이름으로 사용할 인자로 전달하고 있으므로 `thread_create()` 함수를 호출 하기 전에 `command`를 파싱하여, 그 첫 번째 인자를 `thread_create()`에 스레드 이름으로 넘겨주는 과정이 필요하다. 여기서는 `command` 전체를 파싱하지 않고 첫 번째 인자만 얻으면 된다. 이를 위해 `strncpy` 함수와 `strlen` 함수를 사용한다.

```

strncpy(cmd_name, file_name, strlen(file_name)+1);
for(i=0; cmd_name[i]!='\0' && cmd_name[i] != ' '; i++);
cmd_name[i] = '\0';

```

`file_name`은 입력된 `command`이고 `cmd_name`은 스레드 네임으로 `thread_create()`에 넘겨줄 첫 번째 인자이다. Command 전체를 파싱하여 `argv` 배열과 `argc` 값을 얻는 과정은 Problem 2에 서술되었다.

## Problem 2. Argument Passing

### 1. Problem definition

프로그램이 실행될 때 `process_execution()`은 실행할 프로그램 이름을 받고 적절한 `argument`를 프로그램에 넘겨주어야 한다. Command line 상에서 프로그램의 실행은 한 줄로 이루어지는데, 예를 들자면 다음과 같다. `grep foo bar` 라는 명령은 `grep`이라는 프로그램을 실행하되, `foo`와 `bar`를 `argument`로 프로그램에 넘겨주라는 의미이다. `program_execution()`은 `grep`을 프로그램 이름으로, `foo`와 `bar`를 `argument`로 적절히 사용하여야 한다. 현재 `pintos`에는 이러한 `argument passing`이 구현 되어있지 않아 `process_execution()`이 입력된 모든 `command`를 프로그램 이름으로 받아들이고 있다.

### 2. Algorithm design

`Argument passing`의 구현을 위하여 필요한 작업은 두 가지이다. 첫 번째로 입력된 `command`의 `delimiter`를 띄어쓰기 “ “로 하여 `parsing` 하여야 한다. 두 번째로 `parsing`된 `arguments`들을 프로세스의 `stack`에 쌓아 프로세스가 입력된 `argument`들을 참조할 수 있도록 만들어 주어야 한다.

`stack`의 순서는 아래 그림과 같다(Pintos material 3.5.1 Program Startup Details).

Address	Name	Data	Type
0xbfffffc	argv[3][...]	"bar\0"	char[4]
0xbfffff8	argv[2][...]	"foo\0"	char[4]
0xbfffff5	argv[1][...]	"-l\0"	char[3]
0xbffffed	argv[0][...]	"/bin/ls\0"	char[8]
0xbffffec	word-align	0	uint8_t
0xbffffe8	argv[4]	0	char *
0xbffffe4	argv[3]	0xbfffffc	char *
0xbffffe0	argv[2]	0xbfffff8	char *
0xbffffdc	argv[1]	0xbfffff5	char *
0xbffffd8	argv[0]	0xbffffed	char *
0xbffffd4	argv	0xbffffd8	char **
0xbffffd0	argc	4	int
0xbffffcc	return address	0	void (*) ()

그림 1. Argument Passing Process Statck State

위는 “/bin/ls -l foo bar” 명령어가 입력되었을 때 결과로 보여야할 스택이다. PHYS\_BASE 는 0xc0000000라고 가정하였다. Pintos는 pintos material 3.5 80x86 Calling Convention의 순서에 따라 프로세스를 실행하는데, caller가 argument를 pop 하고 시스템콜을 호출하면서 프로세스를 실행한다. 스택의 word-align 은 parsing한 command를 모두 푸시한 뒤 esp를 4배수로 맞추어 주는 과정이다. argv는 입력된 command를 parsing하여 얻은 argument를 가지고 있는 배열이고 입력된 순서에 따라 저장된다. argc는 parsing된 argument의 개수이다.

그림에 묘사된 순서에 따라 먼저 parsing된 argument들을 역순으로 stack에 push한다. 다음 word-align를 하고 스택에 입력된 인자들의 주소 값을 역순으로 push하는데, 이 때에는 인자의 마지막 공백으로 NULL (0)값이 들어간다. 이후 argv의 시작 주소, argc를 push하고 return address를 0으로 세팅하여 push 한다.

Argument의 parsing을 확인하는 방법은 다음과 같다. 입력된 인자가 argv에 잘 저장되었는지 배열을 순서대로 출력해 본다. 또한 가장 마지막 argument가 저장된 argv의 index가 argc-1에 해당하는지를 확인해 본다. Process stack은 hex\_dump() 함수를 사용하여 확인해 볼 수 있다.

### 3. Implementation

입력된 command는 process\_execute ()함수와 thread\_current ()함수를 거쳐 start\_process ()함수로 전달된다. start\_process ()함수는 load ()함수를 호출하는데, load ()함수가 호출하는 setup\_stack ()함수는 stack pointer인 esp를 세팅한다. start\_process ()함수는 내부에서, load ()함

수가 stack setup 성공 여부를 반환하고(bool success) 앞으로 세팅할 esp에 대한 정보도 가지고 있으므로 start\_process ()에서 argument를 parsing하고 stack을 쌓기로 한다.

첫 번째, 입력된 command의 parsing 구현.

start\_process ()함수 내부에서 load ()함수가 호출되기 전에 command를 parsing한다. Integer 형의 argc와 character pointer array 인 argv를 초기화 하고, while문과 strtok\_r ()함수를 사용하여 file\_name 인자를 통해 받은 command를 parsing하여 argv에 저장하고, 저장한 횟수를 argc에 저장하여 parsing을 완료한다. 또한 호출되는 load ()의 첫 번째 인자로 argv[0], 즉 실행되는 process 이름을 넘겨줄 수 있도록 변수명을 수정한다.

두 번째, parsing한 argument의 stack 구현.

void stack\_esp (char \*\*argv, int argc, void \*\*esp) 함수를 새로 만든다. 새로 구현한 stack\_esp () 함수는 앞서 parsing한 command가 저장되어있는 argv와 argc, 그리고 esp, stack 시작주소를 입력받아 stack을 쌓는다. start\_process ()함수 내부에서 load ()함수가 호출되고 success = 1을 반환하였을 때 stack\_esp ()함수가 호출되어 stack을 쌓는다. start\_process ()함수 내부에서 stack\_esp()함수는 앞서 parsing한 argv와 argc와 load ()함수에 의해 세팅된 esp인 &if\_.esp를 인자로 입력받는다.

argv의 배열 요소별 주소를 push해야 하므로 argv의 배열 요소가 push될 때 마다 그 주소를 미리 선언한 addr 배열에 저장하여 나중에 사용한다는 점과, word-align을 위해 argument 들의 total length를 input\_length 변수에 저장하여 나중에 사용한다는 점이 특징이다. 자세한 구현은 코드를 참고.

### Problem 3. System Calls

#### 1. Problem definition

시스템 콜이란 운영 체제의 kernel이 제공하는 서비스를 이용하기 위해, 유저 프로그램이 커널에 접근하기 위해 이용하는 인터페이스이다. 앞서 Problem 1에서 설명했듯이 pintos에는 시스템 콜이 구현되어 있지 않지만 ~/pintos/src/userprog/syscall.c 파일에 skeleton이 구현되어 있다. 따라서 Problem 3에서는 13종의 시스템 콜을 Problem 3에서 구현하였다.

시스템 콜을 구현할 때 주의해야 할 점으로 유저 프로그램은 kernel 영역 메모리 주소를 참조해서는 안된다는 것이 있다. 따라서 이는 ~/pintos/src/threads/vaddr.h에 구현되어 있는 is\_user\_vaddr 함수를 이용하여 해결하도록 하였다.

다음으로 고려해야 할 점은 read 시스템 콜이나 write 시스템 콜 같은 시스템 콜은 critical section에 접근하여 동작하는 file system code에도 관여하기 때문에 synchronization이 구현되지 않을 경우 오류가 발생할 수 있다. 따라서 시스템 콜이 synchronization될 수 있도록 구현해야 할 것이다.

## 2. Algorithm design

시스템 콜은 `syscall_handler`를 통해 호출된다. 이들은 `system call number`를 통해 호출되는데 이는 `~/pintos/src/lib/syscall-nr.h`에 정의되어 있다. 이를 `~/pintos/src/userprog/syscall.c`의 `syscall_handler`에 순서대로 `syscall number`에 맞게 `switch case`문으로 구현하였다. `Switch case`문에서는 먼저 해당 시스템 콜에서 사용하는 `virtual address`가 `user virtual address`인지 확인하고 해당 시스템 콜 함수를 호출하였다. `User virtual address`인지 확인할 때는 앞서 설명한 `is_user_vaddr`함수를 사용하는 `bad_address`라는 함수를 만들어 `user virtual address`가 아니라면 `exit(-1)`을 호출하도록 하였다.

`pintos`에서는 하나의 `process`에서는 하나의 `thread`만을 다루기 때문에 `thread`를 이용해 구현해도 `process`를 다루는 것과 같은 효과를 가진다. 따라서 `thread`를 이용해 알고리즘을 디자인하였고 `thread`를 다룰 때 사용하기 위해 `~/pintos/src/threads/thread.h`의 `struct thread`의 `#ifdef USERPROG` 밑 부분에 여러 변수를 추가하였다. `~/pintos/src/userprog/process.c`의 `process_wait()` 위의 주석을 참고하면 해당 `thread`의 `child`가 종료되어 해당 `thread`의 `exit status`를 반환해야 한다고 한다. 따라서 `struct thread`에 `int exit_status`를 추가하여 각 `thread`가 `exit_status`를 가질 수 있도록 하였다. 또한 `child`에 대해 다뤄야 하기 때문에 `child`의 `list`와 `list_elem`를 추가하고 `synchronization`을 위해 `semaphore child_lock`을 추가하였다. 또한 `child process`가 종료되고 `sema_up`을 할 때, `parent process`에서 `list_remove`를 실행해야 하는데 이때 먼저 `memory`에서 사라지는 것을 막기 위해 `semaphore past_lock`을 추가하였다. 마지막으로 `parent thread`가 `child thread`가 `load`되기 전에 종료되는 것을 방지하기 위해서 `parent`와 `semaphore load_lock`을 추가하였다.

그 밖에도 `open` 시스템 콜 함수 구현 중, 각각의 프로세스가 `file descriptor`를 필요로 하므로 이를 구현하였다. `Pintos`에서는 `process`와 `thread`가 1대1 매칭되므로 `struct thread`에 `file descriptor fd`를 추가하였다.

각 시스템 콜의 함수는 아래 표에서 `syscall_handler`에 들어간 순서대로 표를 만들어 각각의 `Algorithm design`을 정리하였다.

표 1. Algorithm Design of each system call

System call	Algorithm design
halt	시스템 콜 <code>halt</code> 는 <code>threads/init.h</code> 에 구현되어 있는 <code>shutdown_power_off()</code> 를 불러와 <code>Pintos</code> 를 끝내는 기능을 가지고 있다. 따라서 시스템 콜 함수에 <code>shutdown_power_off()</code> 함수를 불러오도록 디자인하였다.
exit	시스템 콜 <code>exit</code> 는 <code>current user program</code> 을 끝내고 커널로 <code>status</code> 를 반환하는 함수이다. <code>Problem 1</code> 에서 요구한 <code>termination message</code> 를 출력하도록 디자인하였다. 이후 현재 <code>thread</code> 의 <code>exit status</code> 에 <code>status</code> 를 저장하고 <code>thread</code> 의 <code>file descriptor</code> 를 확인하여 <code>close</code> 한다. 그 후 <code>thread_exit()</code> 함수를 호출하여 <code>exit</code> 를 실행하

	도록 하였다.
exec	시스템 콜 exec는 child process를 생성하고 해당 process에 입력 받은 cmd_line의 프로그램을 실행시키는 시스템 콜이다. Process 생성에 성공 시 생성된 child process의 pid를 반환하고, 실패할 경우 -1을 반환한다. Parent process는 생성된 child process의 프로그램이 메모리에 적재될 때까지 semaphore를 이용해 대기한다(struct thread의 load_lock이용).
wait	시스템 콜 wait은 입력 받은 tid의 child process가 종료할 때까지 대기하고 자식 프로세스가 제대로 종료됐는지 확인하는 시스템 콜이다. 이를 위해 시스템 콜 wait에서는 ~/pintos/src/userprog/process.c의 process_wait()을 호출한다. 이를 통해 child process가 종료되지 않았을 때 parent process가 대기하도록 하고 정상적으로 종료 시 exit status를 반환하고 그러지 못했다면 -1을 반환하도록 디자인하였다.
create	시스템 콜 create은 입력 받은 file 변수를 이름으로 가지고 입력 받은 initial_size bytes의 크기를 가진 새로운 파일을 만드는 시스템 콜이다. 성공적으로 만들었다면 true를 반환하고, 그러지 못했다면 false를 반환하도록 디자인되었다.
remove	시스템 콜 remove는 입력 받은 file 변수를 이름으로 가지는 파일을 삭제하고 성공했으면 true를 실패했으면 false를 반환하는 시스템 콜이다.
open	시스템 콜 open은 입력 받은 file 변수를 이름으로 가지는 파일을 열고 성공했을 경우 file descriptor를 반환하고 실패했을 때 -1을 반환하도록 디자인하였다.
filesize	시스템 콜 filesize는 입력 받은 file descriptor로 open된 file의 크기를 byte 단위로 반환하도록 하였다.
read	시스템 콜 read는 입력 받은 fd의 open된 파일의 데이터를 읽는 시스템 콜이다. 성공 시 실제로 읽은 바이트 수를 반환하고 실패 시 -1을 반환한다. 입력 받은 buffer는 읽은 데이터를 저장할 주소 값이고 size는 읽을 데이터의 크기이다. fd가 0일 경우 키보드로부터 입력 받은 데이터를 buffer에 저장하도록 하였다.
write	시스템 콜 write는 입력 받은 fd의 open된 파일의 데이터를 기록하는 시스템 콜이다. Read와 마찬가지로 성공 시 기록한 바이트 수를 반환하고 실패 시 -1을 반환한다. buffer는 기록할 데이터를 저장한 주소 값이고 size는 기록할 데이터의 크기이다. fd가 1일 경우 buffer에 저장된 데이터를 화면에 출력하도록 하였다.
seek	시스템 콜 seek는 입력 받은 fd의 open된 파일의 위치를 이동하도록 하는 시스템 콜이다. 입력 받은 position을 통해 이동할 거리를 결정한다.
tell	시스템 콜 tell은 입력 받은 fd의 open된 파일의 위치를 반환하는 시스템 콜이다. 성공했을 경우 파일의 위치를 반환하고 실패 시 -1을 반환하도록 하였다.
close	시스템 콜 close는 입력 받은 fd값의 파일을 닫는 시스템 콜이다. Process가

	종료될 때 이 시스템 콜이 호출되어 해당 process가 가진 모든 file descriptor의 open된 파일을 닫도록 하였다.
--	---

### 3. Implementation

algorithm design에서 언급했듯이 올바른 virtual address를 참조해야 하기 때문에 void bad\_vaddr(const void \*vaddr) 함수를 만들어 is\_user\_vaddr 함수를 이용한 if문으로 user virtual address가 아닐 경우 exit(-1)이 실행되도록 구현하여 사용하였다.

각 시스템 콜 함수의 구현은 위의 Algorithm Design에서와 같이 시스템 콜의 종류가 많기 때문에 표로 정리하였다. 보다 자세한 내용은 코드 참조.

**표 2. Implementation of each System Call**

System call	Implementation
halt	pintos를 종료하는 기능을 가진 시스템 콜이므로 pintos에서 제공하는 ~pintos/src/device/shutdown.c의 shutdown_power_off() 함수를 호출하도록 구현하였다.
exit	먼저 Problem 1에서 요구한 termination message를 출력하도록 구현하였다. 이후 현재 thread를 가리키는 thread_current()의 exit status에 입력 받은 status 값을 넣어주었다. 이후 for loop를 통해 해당 process가 가지고 있는 file descriptor를 close 시스템 콜을 이용해 닫아 주었다. 이후 pintos가 제공하는 ~pintos/src/threads/thread.c의 thread_exit()함수를 이용하여 thread를 종료하도록 구현하였다.
exec	기본적으로 ~pintos/src/userprog/process.c의 process_execute(file)을 반환하도록 구현하였다. 또한 본래 pintos에는 child process와 parent process가 구현되어 있지 않으므로 위의 algorithm design에서 설명한 변수, semaphore, list들을 ~pintos/src/threads/thread.h의 struct thread 안에 추가하여 사용하였다. 또한 ~pintos/src/threads/thread.c의 init_thread로 초기화 해주었다. 이후 pass되지 않는 exec-missing을 해결하기 위해 parse한 cmd_name이 유효한지 확인하기 위해 if문을 사용하여 확인한 후 유효하지 않을 경우 -1을 반환하도록 구현하였다.
wait	기본적으로 ~pintos/src/userprog/process.c의 process_wait(pid)를 반환하도록 구현하였다. 수정 전의 pintos는 parent process인 init process가 child process인 user process가 끝날 때까지 기다리지 않고 -1을 반환하여 pintos를 종료한다. 하지만 wait_process를 구현하여 parent process가 child process를 기다리도록 한다. for loop를 사용하여 이전에 구현했던 child_list를 통해 모든 child process를 검색한다. 이후 입력 받은 child_tid가 검색한 child thread의 tid와 같으면 ~pintos/src/threads/synch.c의 sema_down을 통해 block상태로 parent process가 대기하도록 구현하였다. 이후 child process를 list_remove를 통해 제거하고 sema_up을 통해 block을 해제한다. 이후 exit_status를 반환



	하도록 구현하였다. For loop가 끝났음에도 해당 tid를 가진 child process를 찾지 못하는 것과 같은 예외 상황 발생 시에는 -1이 반환되도록 하였다.
create	file이 NULL인지 아닌지 확인하고 NULL이 아니라면 pintos가 제공하는 ~/pintos/src/filesys/filesys.c의 filesys_create()를 반환하도록 구현하였다.
remove	file이 NULL인지 아닌지 확인하고 NULL이 아니라면 pintos가 제공하는 ~/pintos/src/filesys/filesys.c의 filesys_remove()를 반환하도록 구현하였다.
open	Open부터는 file descriptor를 사용하기 때문에 위의 algorithm design에서 설명했던 struct thread에 구현한 fd[]를 이용하였다. file이 NULL일 경우에는 exit(-1)을 실행시키도록 하고 bad_vaddr 함수를 이용해 virtual address를 체크하였다. pintos에서 제공하는 ~/pintos/src/filesys/file.c의 filesys_open(file) 함수를 이용하여 file을 열어 fp에 저장하도록 하고 for loop를 사용해 NULL인 fd값을 찾아 fp를 저장하고 fd값을 반환하도록 구현하였다. fp가 NULL인 경우는 -1을 반환하도록 하였다. 추가적으로 synchronization을 위해 파일을 열기 전 ~/pintos/src/threads/synch.c의 lock_acquire()를 사용하고 마지막에 lock_release를 이용하여 lock을 해제하도록 하였다.
filesize	현재 thread의 fd(입력값)이 NULL인지 확인하여 NULL이면 exit(-1)을 사용하여 끝낸다. 그렇지 않으면 ~/pintos/src/filesys/file.c에 구현된 file_length() 함수를 반환하여 파일의 크기를 반환하도록 하였다.
read	bad_vaddr()실행 후 두 가지 파일이 동시에 접근할 수 있으므로 synchronization 문제를 없애기 위해 lock_acquire()를 사용하여 lock을 걸었다. 이후 fd가 0이라면 키보드로부터 받은 입력을 buffer에 저장하고 buffer의 크기를 반환하도록 한다. fd가 2보다 클 경우 Pintos가 제공하는 file_read()를 사용하여 파일을 buffer에 저장하도록 하고 buffer의 size를 반환하였다. 이후 lock_release()를 통해 lock을 해제하였다. 현재 thread의 fd가 NULL일 경우에는 즉시 lock_release()와 exit(-1)을 실행시키도록 구현하였다.
write	bad_vaddr()실행 후 두 가지 파일이 동시에 접근할 수 있으므로 synchronization 문제를 없애기 위해 lock_acquire()를 사용하여 lock을 걸었다. 이후 fd가 1이라면 buffer에 저장된 값을 화면에 출력하고 buffer의 크기를 반환하도록 하였다. fd가 2보다 클 경우 Pintos가 제공하는 file_write()를 사용하여 buffer에 저장된 값을 size만큼 파일에 기록하도록 하고 buffer의 크기를 반환하였다. 이후 lock_release()를 통해 lock을 해제하였다. 현재 thread의 fd가 NULL일 경우에는 즉시 lock_release()와 exit(-1)을 실행시키도록 구현하였다.
seek	현재 thread의 fd(입력값)이 NULL인지 확인하여 NULL이면 exit(-1)을 사용하여 끝낸다. 그렇지 않으면 pintos가 제공하는 ~/pintos/src/filesys/file.c에 구현된 file_seek() 함수를 실행하여 파일의 위치를 이동하도록 하였다.
tell	pintos에서 제공하는 ~/pintos/src/filesys/file.c에 구현된 file_tell() 함수를 실행하여 파일의 위치를 반환하도록 하였다.

close	현재 thread의 fd(입력값)이 NULL인지 확인하여 NULL이면 exit(-1)을 사용하여 끝낸다. 그렇지 않으면 해당 fd를 NULL로 초기화 하고 pintos에서 제공하는 ~/pintos/src/filesys/file.c에 구현된 file_close ()함수를 실행하여 입력 받은 fd에 해당하는 파일을 닫도록 하였다.
-------	---

#### Problem 4. Denying Writes to Executables

##### 1. Problem definition

Process가 실행 중에 write를 할 수 있을 경우, 문제가 발생하여 예기치 못한 문제가 발생할 수 있다. (파일 데이터가 변경되어 예상했던 데이터와 다른 데이터가 read될 수 있다.) 따라서 Problem 4에서는 process가 실행 중일 때, write되는 것을 방지하기 위한 코드를 구현하였다. 열려 있는 파일에 write하는 것을 막기 위해서는 ~/pintos/src/filesys/file.c에 구현되어 있는 file\_deny\_write()를 사용하였다.

##### 2. Algorithm design

Problem definition에서 언급했던 대로 file\_deny\_write()를 사용하여 알고리즘을 디자인하였다. 이를 위해 Pintos/src/userprog/syscall.c에서 pintos/src/filesys/file.c에 구현되어 있는 struct file (bool deny\_write;)을 사용하기 위해 해당 코드를 복사해왔다. 이후 ~/pintos/src/userprog/syscall.c의 open 시스템 콜, write 시스템 콜에 file\_deny\_write()를 추가하였다.

그리고 kernel panic을 방지하기 위해 ~/pintos/src/userprog/exception.c의 static void page\_fault 함수의 if문에 not\_present (원래 page\_fault함수 안에 bool not\_present;로 선언되어 있다.)를 추가하였다.

##### 3. Implementation

Open 시스템 콜에서는 filesys\_open()을 이용하여 파일을 열고 난 후, NULL인 fd를 찾기 전에 현재의 thread의 이름이 open하려는 파일의 이름과 같다면 file\_deny\_write(fp)을 사용하여 write를 막는다.

```
if(strcmp(thread_current()->name, file) == 0){
    file_deny_write(fp);
}
```

Write 시스템 콜에서는 현재 thread의 fd가 NULL인지 확인한 후 현재 thread의 fd에 해당하는 파일의 deny\_write(struct file에 구현)가 참이라면 file\_deny\_write()를 실행한다.

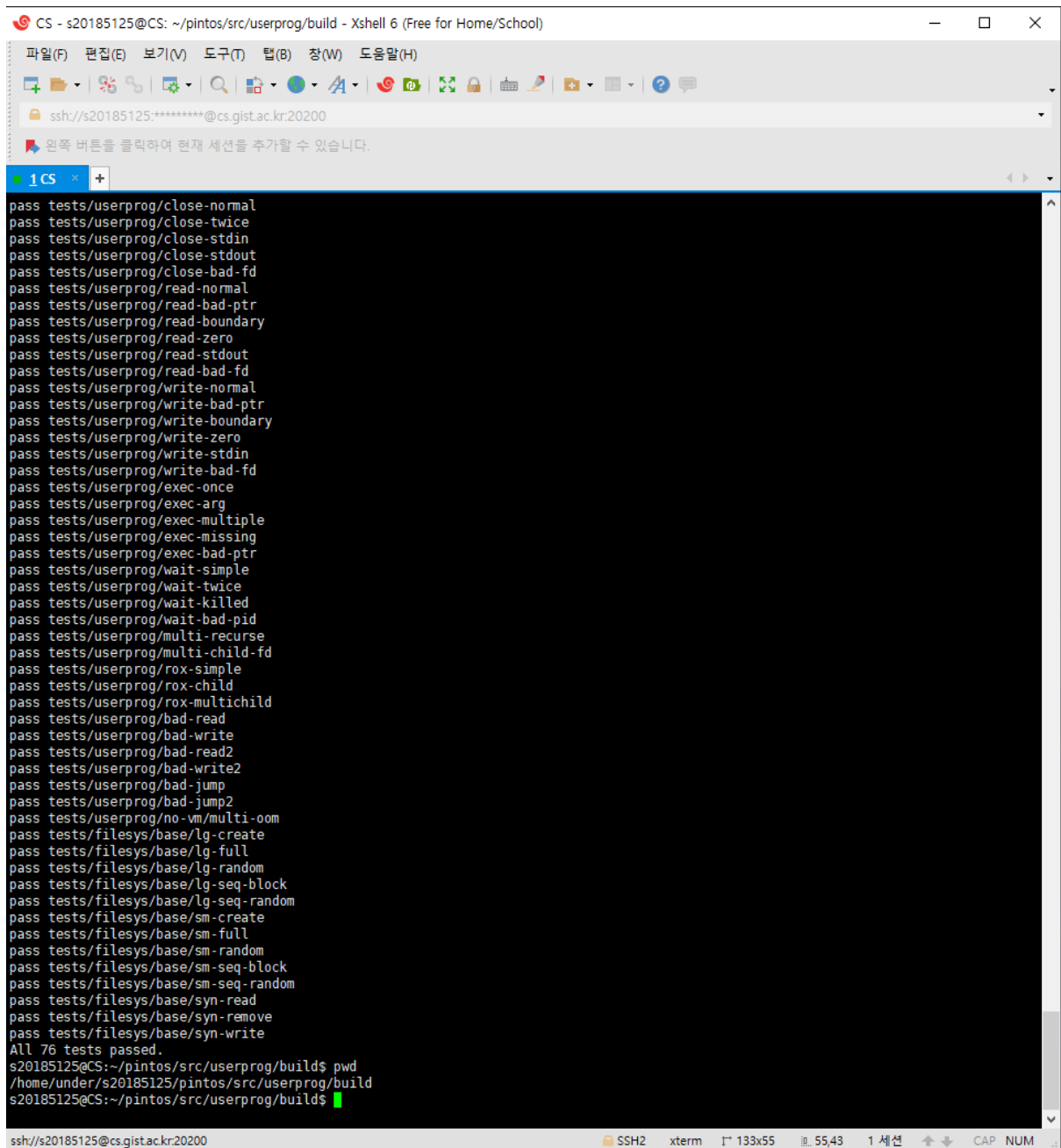
```
If(thread_current()->fd[fd]->deny_write){
    file_deny_write(thread_current()->fd[fd]);
}
```

```
}
```

Exception.c 파일에 추가한 조건은 아래와 같다.

```
if(!user || is_kernel_vaddr(fault_addr) || not_present) {  
    exit(-1);  
}
```

## Appendix



The screenshot shows a terminal window titled "CS - s20185125@CS: ~/pintos/src/userprog/build - Xshell 6 (Free for Home/School)". The terminal displays a list of 76 tests, all of which passed. The tests are categorized into userprog and filesys. The userprog tests include close-normal, close-twice, close-stdin, close-stdout, close-bad-fd, read-normal, read-bad-ptr, read-boundary, read-zero, read-stdout, read-bad-fd, write-normal, write-bad-ptr, write-boundary, write-zero, write-stdin, write-bad-fd, exec-once, exec-arg, exec-multiple, exec-missing, exec-bad-ptr, wait-simple, wait-twice, wait-killed, wait-bad-pid, multi-recurse, multi-child-fd, rox-simple, rox-child, rox-multichild, bad-read, bad-write, bad-read2, bad-write2, bad-jump, bad-jump2, no-vm/multi-oom, and filesys tests include base/lg-create, base/lg-full, base/lg-random, base/lg-seq-block, base/lg-seq-random, base/sm-create, base/sm-full, base/sm-random, base/sm-seq-block, base/sm-seq-random, base/syn-read, base/syn-remove, and base/syn-write. The terminal output shows "All 76 tests passed." followed by the current directory path and a prompt.

```
pass tests/userprog/close-normal  
pass tests/userprog/close-twice  
pass tests/userprog/close-stdin  
pass tests/userprog/close-stdout  
pass tests/userprog/close-bad-fd  
pass tests/userprog/read-normal  
pass tests/userprog/read-bad-ptr  
pass tests/userprog/read-boundary  
pass tests/userprog/read-zero  
pass tests/userprog/read-stdout  
pass tests/userprog/read-bad-fd  
pass tests/userprog/write-normal  
pass tests/userprog/write-bad-ptr  
pass tests/userprog/write-boundary  
pass tests/userprog/write-zero  
pass tests/userprog/write-stdin  
pass tests/userprog/write-bad-fd  
pass tests/userprog/exec-once  
pass tests/userprog/exec-arg  
pass tests/userprog/exec-multiple  
pass tests/userprog/exec-missing  
pass tests/userprog/exec-bad-ptr  
pass tests/userprog/wait-simple  
pass tests/userprog/wait-twice  
pass tests/userprog/wait-killed  
pass tests/userprog/wait-bad-pid  
pass tests/userprog/multi-recurse  
pass tests/userprog/multi-child-fd  
pass tests/userprog/rox-simple  
pass tests/userprog/rox-child  
pass tests/userprog/rox-multichild  
pass tests/userprog/bad-read  
pass tests/userprog/bad-write  
pass tests/userprog/bad-read2  
pass tests/userprog/bad-write2  
pass tests/userprog/bad-jump  
pass tests/userprog/bad-jump2  
pass tests/userprog/no-vm/multi-oom  
pass tests/filesys/base/lg-create  
pass tests/filesys/base/lg-full  
pass tests/filesys/base/lg-random  
pass tests/filesys/base/lg-seq-block  
pass tests/filesys/base/lg-seq-random  
pass tests/filesys/base/sm-create  
pass tests/filesys/base/sm-full  
pass tests/filesys/base/sm-random  
pass tests/filesys/base/sm-seq-block  
pass tests/filesys/base/sm-seq-random  
pass tests/filesys/base/syn-read  
pass tests/filesys/base/syn-remove  
pass tests/filesys/base/syn-write  
All 76 tests passed.  
s20185125@CS:~/pintos/src/userprog/build$ pwd  
/home/under/s20185125/pintos/src/userprog/build  
s20185125@CS:~/pintos/src/userprog/build$
```