OS HW Report

9조 20175097 안준호 20175104 오정석

HW 2

Process Termination Messages

Problem Definition

프로세스가 exit 시스템 콜을 호출하면 정해진 포맷에 따라서 출력을 해주어야 한다.

Algorithm Design

system call 에서 현재 호출되는 시스템 콜을 분석하여 status 와 thread name 을 함께 출력한다. 이때 thread name 에서 주어진 인자를 적절히 프로그램명을 추출하여 thread name에 집어 넣어 주어야 한다. 또한 우리팀은 우선 argument passing 을 구현한후 이 부분을 구현하였다. 그전에는 process_exit 에서 예비로 모든 process 가 종료될때 출력을 시켰다.

Implementation

```
void exit(int status)
{
    struct thread *t = thread_current();
    t->exit = status;
    printf("%s: exit(%d)\n", thread_current()->name, status);
    thread_exit();
}
```

Argument Passing

Problem Definition

process_execute 에 프로그램을 주면, 스택에 적절히 argv arc를 쌓아 올려야 한다. 이 쌓아올린 stack 인자에 접근하여 system call 에서 인자를 추출하기 때문이다. 만약 process_execute("grep foo bar")으로 주어지면, thread->name 에 grep을 넣고, 스택에는 bar foo grep 순서로 집어 넣게 된다. 스택은 거꾸로 자라남으로, 먼저 arv를 쌓고 적절히 메모리의 접근성을 위해서 word-align을 넣어야 한다. 그후 argv를 가르키는 주소값과 argc 마지막으로 return address를 넣으면 된다. pintos docement 3.5.1을 참고 하였다.

Algorithm Design

1. process.c 에 load 가 file 임으로 주어진 file을 바탕으로 file name 과 argv들을 추출한다.
2. 새로 setup stack argv 함수를 만들어 load가 끝나고 stack 에 argv들을 정렬하여 넣어준다.
3. 넣어줄떄는 pintos document에 있는 대로, argv -> word align -> NULL -> argv의 주소값 -> argv 의 사작 주소값 -> return address (0으로 고정) 순서대로 넣어준다.
4. hex dump으로 주소값이 잘 들어 갔는지 확인한다.

Implementation

process_execute 에서 thread_create 를 하며, programe_name을 추출하여 thread_create에 넣는다. 이 방식으로 thread의 이름을 명령어의 이름과 일치시킨다. 만약 이작업을 하지않으면, thread 구조체의 name 필드에 명령어 + argv가 들어가 나중에 확인하기 어려워진다.

```
char programe_name[256];
int index = 0;
char* last = file_name[0];

while(last != '\0' && last != ' ') {
programe_name[index] = file_name[index];
last = file_name[index];
index++;
}
programe_name[index - 1] = '\0';
```

load는 파일에서 프로그램을 읽어오는 부분이다. 다음과 같이 충분한 길이의 char* 배열을 선언하고, strtok_r 을 " " 기준으로 분절하여 리턴 값이 포인터를 넣어준다. 여기서 구한 inputs들을 차례로 setup_stack_argv 함수를 이용해 구조체에 넣어주게 된다.

```
char* inputs[256];

char *token, *save_ptr;
int index = 0;

for (token = strtok_r (file_name, " ", &save_ptr); token != NULL; token =
strtok_r (NULL, " ", &save_ptr)) {
    //printf("t: %s\n",token);
    inputs[index++] = token;
}

/* Open executable file. */

file = filesys_open (file_name);
```

setup stack은 다음과 같이 구성된다. 여기서 주의할 점은, 메모리에서 한 주소가 1바이트를 저장하기 때문에, 4바이트를 저장하기 위해서는 *esp 를 4만큼 감소 (스택을 아래로 감소) 시켜야 된다는 것이었다.

```
void setup_stack_argv (void **esp, int argc, char** argv) {
    int alignment_length = 0;
    int len = 0;
    int i=0;
    // pushing argv to esp
    for(i=argc - 1; i >= 0; i--) {
        len = strlen(argv[i]);
```

```
            // 1 for null pointer sentinel
            *esp -= len + 1;
            memcpy(*esp, argv[i], len + 1);
            argv[i] = *esp;
            alignment_length += len + 1;
        }
        // memory alignment
        int rest = alignment_length % 4;
        if(rest == 0) {
            //do nothing
        } else {
            *esp -= 4 - rest;
        }

        //push null
        *esp -= 4;
            *(uint32_t **)*esp = 0;

        //push argv[i] address
        for(i=argc - 1; i >= 0; i--) {
            // 1 for null pointer sentinel
            *esp -= 4;
            *(uint32_t **)*esp = argv[i];
        }

        //push argv address
        *esp -=4;
        *(uint32_t **)*esp = *esp + 4;

        //push argc
        *esp -=4;
        *(uint32_t **)*esp = argc;

        //push return address (namely 0)
        *esp -=4;
        *(uint32_t **)*esp = 0;
    }
```

System Calls

Problem Definition

이제 대망의 시스템 콜을 만들어야 한다. user programe은 다양한 kernel의 작업을 요구하는 작업에 대해 system call을 통하여, 작업이 적절한지 검사 받고 실행되게 된다. 따라서 우리는 아까 스택에 넣어논 argv들을 kernel에서 끄집어 내어, system call을 호출해야 한다.

Algorithm Design

1. user programe 이 kernel에 인터럽트 0x30을 걸어 system call을 호출한다.
2. userprograme이 0x30 인터럽트를 걸어서 system call 을 호출한다. 관련 내용은 syscall-nr.h에 기록되어 있다.

3. system call handler 에서 스택에서 인자를 꺼내어 어떤 시스템 콜이었는지 분류하여 인자를 넣는다.
4. 주소가 user programe 영역인지, 영역이라면 할당된 영역인지 검사하여야 한다. 또한 null 인지도 검사하여야 한다.

Implementation

struct lock file_lock; 을 추가하여, file 작업을 할때 다른 프로세서의 간섭이 없도록 하였다. 또한 syscall handler에서 switch 문을 이용하여 적절하게 작업을 분배하였다.

다음 함수를 이용하여, 인자를 원하는 만큼 가져와, arg에 넣어주었다.

```
void get_argument(void *esp, int *arg, int count)
{
    int i = 0;
    uint32_t *base_esp = (uint32_t *)esp + 1;
    for (i = 0; i < count; i++)
    {
        check_address(base_esp);
        arg[i] = *base_esp;
        base_esp += 1;
    }
}
```

또한 다음 함수들을 이용하여 주소가 user 영역에 할당되어 있는지, file이 valid 한 주소인지, 그리고 file pointer가 null인지 검사한다. 이때 pintos document에 나와 있듯이 file 이 valid 한지 확인하는 방법은 두가지 방법이 있다. 첫번째로는 pagedir_get_page가 적절한 file을 호출하는지 확인하는 방법이며, 두번째 방법은 핀토스에서 file포인터가 잘못되어 인터럽트를 호출했을때, 스레드를 종료하는 것 보다 레지스터에 관련내용을 넣고 검사하는 방법이다. 두번째 방법이 좀더 빨라서 구현하고자 하였으나, 중간에 오류가 많이 떠서 결국 첫번째 방법을 이용하였다.

```
void check_address(void *addr)
    {
    if ((uint32_t)0x8048000 >= (uint32_t *)addr || (uint32_t)0xc0000000 <=
(uint32_t *)addr)
    {
        //printf("Out of user memory area [0x%x]!\n", (uint32_t *)addr);
        exit(-1);
    }
}

void check_file_valid(void *addr)
{
    if (!is_user_vaddr(addr))
    {
        exit(-1);
    }
    if (pagedir_get_page(thread_current()->pagedir, addr) == NULL)
    {
        exit(-1);
    }
```

```
}

void check_file_null(void *addr)
{
    if (addr == NULL)
    {
        exit(-1);
    }
}
```

다음과 같이 시스템 콜을 구현하였다. halt는 단순히 power off 해주면 된다.

```
void halt(void)
{
    shutdown_power_off();
}
```

exit은 문제에서 요구하는 대로 출력을 해주면 된다. 적절하게 thread 구조체에 name을 넣어 주었기 때문에, name을 가져오면 된다. get argument에서 check address를 하였기 때문에 만약 인자가 잘못된 경우 exit(-1)이 호출되게 된다.

```
void exit(int status)
{
    struct thread *t = thread_current();
    t->exit = status;
    printf("%s: exit(%d)\n", thread_current()->name, status);
    thread_exit();
}
```

process exec를 하는 함수이다. process에 구현한 process_exec를 호출하게 된다.

```
//pid_t exec
int exec(const char *cmd_line)
{
    return process_execute(cmd_line);
}
```

process wait를 하는 함수이다. process에 구현한 process_wait를 호출하게 된다.

```
int wait(int pid)
{
    return process_wait(pid);
}
```

새로운 file을 만드는 시스템 호출이다. 주어진 file주소가 null인지 valid 한지 검사한후 filesys_create를 이용해 파일을 만들어 주게 된다.

```
//bool create
int create(const char *file, unsigned initial_size)
{
    check_file_null(file);
    check_file_valid(file);

    int returnVal;

    lock_acquire(&file_lock);
    returnVal = filesys_create(file, initial_size);
    lock_release(&file_lock);

    return returnVal;
}
```

위와 비슷한 원리로 file을 삭제하면 된다.

```
//bool remove
int remove(const char *file)
{
check_file_null(file);

int returnVal;

lock_acquire(&file_lock);
returnVal = filesys_remove(file);
lock_release(&file_lock);

return returnVal;
}
```

open부터 열린 파일을 스레드에 file descriptor을 이용하여 list나 배열에 저장해야 한다. 프로세스가 이미 존재하는 파일을 open 하는등의 작업을 처리하면, 커널은 필요한 동작을 수행하고 파일 디스크립터 값을 리턴해준다. 0은 STDIN 1은 STDOUT 그리고 2는 STDERR로 정의하여 사용하였다. file descritor은 프로세서 마다 하나 존재하여야 함으로 (pintos에서는 thread와 프로세서의 개념이 모호하다.) thread 구조체에 다음과 같이 정의하였다.

struct file *file[MAX_FILE_SIZE]; int file_struct_size;

또한 관련 처리는 다음과 같이 함수를 만들어 호출하여 사용하도록 하였다.

```
int insert_file(struct file *f);
void remove_file(struct file *f);
void init_file_struct();
```

```
struct file *file_from_fd(int fd);
void debug_file_struct();
```

open에서는 file이 null인지 검사하고, filesys_open을 통해서 file이 열리는지 검사한다. 또한 열러진 file_pointer 을 받아서 NULL과 비교하여 열렸는지 확인한다. 여기서 if(strcmp ...) 이후의 부분은 Denying write to Executable을 체크하기 위해서 검사하는 과정이다.

```
int open(const char *file)
{
    struct file *file_pointer;
    int returnVal;

    check_file_null(file);
    //check_file_valid(file);

    lock_acquire(&file_lock);
    file_pointer = filesys_open(file);
    //printf("%x\n",file_pointer);
    //printf("0x%x\n", file_pointer);
    if (file_pointer == NULL)
    {
        lock_release(&file_lock);
        return -1;
    }

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

    //printf("%x\n",file_pointer);
    //printf("%d\n",file_deny_state(file_pointer));

    returnVal = insert_file(file_pointer);
    //printf("[%d] 0x%x\n", 3, file_from_fd(3));

    lock_release(&file_lock);

    return returnVal;
}
```

주어진 fd를 바탕으로 file pointer를 가져오고 파일의 길이를 구하면 된다.

```
int filesize(int fd)
{
    int returnVal;
    struct file *file_pointer = file_from_fd(fd);
```

```
    //check_file_valid(file_pointer);
    check_file_null(file_pointer);
    lock_acquire(&file_lock);
    returnVal = file_length(file_pointer);
    lock_release(&file_lock);

    return returnVal;
}
```

주어진 fd 가 STDIN 이면 input_getc 를 이용하여 버퍼에 원하는 만큼 입력을 받는다. 만약 STDOUT 이거나 STDERR이면, 그곳에서는 입력을 받을 수 없으므로, 에러를 호출하여야 한다.

```
int read(int fd, void *buffer, unsigned size)
{
    //check_file_valid(file_pointer)
    int returnVal;

    if (fd == STDIN)
    {
        *(uint32_t *)buffer = input_getc();
        size++;
        lock_release(&file_lock);
        return size;
    }
    else if (fd == STDOUT || fd == STDERR)
    {
        lock_release(&file_lock);
        return -1;
    }

    lock_acquire(&file_lock);
    struct file *file_pointer = file_from_fd(fd);
    check_file_null(file_pointer);
    returnVal = file_read(file_pointer, buffer, size);
    lock_release(&file_lock);

    return returnVal;
}
```

마찬가지로 STDOUT이면 STDOUT 버퍼에 원하는 만큼 입력을 출력시키게 된다. 나머지의 경우 file에 버퍼에 있는 내용을 써야 한다. if(file_deny_state ...) 이후의 부분은 Denying Writes to Executables 부분을 구현하기 위해서 처리하는 부분이다.

```
int write(int fd, const void *buffer, unsigned size)
{
    if (fd == STDOUT)
    {
        putbuf(buffer, size);
```

```
        return size;
    }

    int returnVal;
    struct file *file_pointer = file_from_fd(fd);

    lock_acquire(&file_lock);

    //check_file_valid(file_pointer);
    check_file_null(file_pointer);
    //printf("%x\n",file_pointer);
    //printf("%d\n",file_deny_state(file_pointer));
    /*
    if(file_deny_state(file_pointer) == (int)true) {
        lock_release(&file_lock);
        //exit(-1);
    }
    */

    returnVal = file_write(file_pointer, buffer, size);
    lock_release(&file_lock);

    return returnVal;
}
```

file을 원하는 위치에 seek 시킨다. 포지션이 현재 파일의 크기보다 크더라도 에러가 아니라 단지 0을 더해서 출력하기 때문에, position을 검사할 필요는 없다.

```
void seek(int fd, unsigned position)
{
    struct file *file_pointer = file_from_fd(fd);

    //check_file_valid(file_pointer);
    check_file_null(file_pointer);
    lock_acquire(&file_lock);
    file_seek(file_pointer, position);
    lock_release(&file_lock);
}
```

다음 읽어들일 부분을 return 한다. unsigned tell(int fd) { struct file *file_pointer = file_from_fd(fd);

```
    //check_file_valid(file_pointer);
    check_file_null(file_pointer);
    return file_tell(file_from_fd(fd)) + 1;
}
```

파일을 닫는다. 시스템에 예약된 STDIN, STDOUT, 그리고 STDERR는 닫을 수 없기 때문에 exit(-1)을 리턴해야 한다.

```
void close(int fd)
{
    struct file *file_pointer = file_from_fd(fd);
    //check_file_valid(file_pointer);
    check_file_null(file_pointer);

    if (fd == STDIN || fd == STDOUT || fd == STDERR)
    {
        exit(-1);
    }

    lock_acquire(&file_lock);
    remove_file(file_pointer);
    file_close(file_pointer);
    lock_release(&file_lock);
}
```

Denying Writes to Executables

Problem Definition

실행파일에 파일을 쓰려는 행위는 거부되어야만 한다. 왜냐하면, run code를 프로그램 실행 중간에 변화시키려는 시도는 예측할 수 없는 결과를 얻을 수 있기 때문이다.

Algorithm Design

1. open에서 file 이름과 현재 스레드의 이름이 같으면, 현재 스레드를 실행시키는 파일을 열려는 시도이다.
2. write 를 하려면 file을 열어야 함으로, write를 할 수 없도록 deny_write를 호출하여 deny_write를 true로 만들어야 한다.
3. write 에서 file_deny_write 를 검사하기 때문에, deny_write인지 검사할 필요는 없다.
4. open에서 deny_write 를 자동으로 close 해주기 때문에, 별도로 처리할 필요는 없다.

Implementation

syscall.c 의 open 에서 이름을 비교하여, deny_write를 체크해 주어야한다. pintos에서 기본적으로 제공하고 있는 file_deny_write 를 이용하였다.

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