

# **Pintos 프로젝트 2\_2. User Program**

## **(설계 프로젝트 수행 결과)**

**과목명 : [CSE4070-01] 운영체제**  
**담당교수 : 서강대학교 컴퓨터공학과 박성용**

**조원 : 28조 김태훈**  
**개발기간 : 2016. 11. 04. -2016. 11. 06.**

# 최 종 보 고 서

**프로젝트 제목: Pintos 프로젝트 2\_2. User Program**

**제출일: 2016. 11. 12.**

**참여조원: 28조 김태훈**

## I. 개발 목표

Pintos OS에서 user program의 실행이 가능하도록 하는 다양한 기능을 구현한다. 특히 user program과 OS의 interaction을 가능하게 하는 여러 system call을 구현하는데 중점을 둔다.

이전 Project에서 기본적으로 구현한 내용 중에서 synchronization과 관련하여 발생한 다양한 문제들을 보완하고 file system에 접근하는 다양한 system call을 추가적으로 구현한다.

## II. 개발 범위 및 내용

### 가. 개발 범위

현재의 Pintos OS에는 가장 기본적인 user program 실행 환경만 구현되어 있다. command-line을 통해서 들어온 argument passing과 user program의 실행을 위해 필요한 system call infrastructure를 구현한다.

이번 프로젝트에서는 multi-child 환경에서의 synchronization, memory garbage collection 등의 문제를 보완하고 file 관련 create, remove, open, close, read, write, seek, tell, file\_size system call을 system call handler에 추가한다.

### 나. 개발 내용

1. 이전 프로젝트에서 발생한 가장 큰 문제점은 parent thread가 2개 이상의 child를 생성했을 때 wait system call이 제대로 동작하지 않는 점이었다. semaphore를 사용한 synchronization에 문제가 있는 것으로 보인다. 이를 해결하여 2개 이상의 child를 생성한 parent thread가 이를 잘 관리하도록 한다.

2. 이전 프로젝트 결과를 토대로 multi-oom test-case를 돌리면 memory loss가 발생한다. 이 역시 synchronization 문제에 의해 생성된 zombie process 때문인 것으로 보인다. 이를 해결하기 위해 zombie buster code를 parent의 thread\_exit에 추가한다.

3. file system 관련 system call을 구현한다. 이 과정에서 다양한 동적 할당이 이루어지는데 이를 적절하게 모두 처리하여 memory loss가 발생하지 않도록 한다. 또한 한 thread가 file을 read 하는 동안 다른 thread가 read, write를 하지 못하도록 mutex lock을 이용한 synchronization을 구현한다. 또한 Read-only-executable(rox test cases)을 예외 처리 하여 동작 중인 process의 executable file에 write를 하지 못하도록 file\_deny\_write를 적절하게 걸어줘야 한다.

### III. 추진 일정 및 개발 방법

#### 가. 추진 일정

11.04 : 이전 프로젝트에서 발생한 문제의 원인 분석 및 해결

11.05 : file system 관련 system call 구현 및 synchronization 문제 해결

11.06 : Project 마무리 및 보고서 작성

#### 나. 개발 방법

Pintos manual과 강의 자료, Operating Systems 교재를 참고하여 관련 사항을 학습한다. 특히 synchronization의 구현을 위한 semaphore에 관한 내용은 수업 시간에 배우지 않았으므로 주의 깊게 살펴본다. semaphore를 0으로 initialize한 상태에서 down을 하면 해당 thread는 semaphore가 다시 up될 때까지 wait 상태에 머물게 된다. sema\_up을 해당 thread와 synchronize 할 thread에서 semaphore를 up하면 down 되어 있던 thread는 다시 execution을 시작한다. 이를 바탕으로 up, down을 실행할 함수를 결정하고 이를 잘 적용하여 synchronization을 잘 구현한다. 프로젝트 목표가 제대로 완수 되었는지 확인하기 위해서 Pintos OS 내부 test tool을 활용한다.

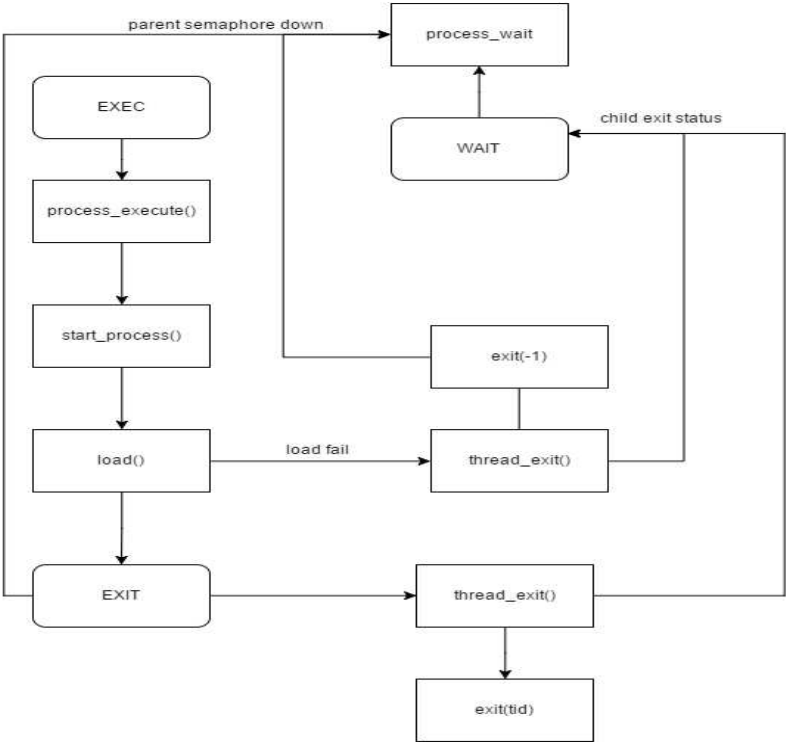
이번 프로젝트에서는 synchronization을 위해 semaphore 말고도 lock을 사용하였다. lock은 1로 초기화 된 semaphore로 critical section에 한 번에 한 thread만 들어가도록 하기 위해 활용한다. 이번 프로젝트에서 발생할 수 있는 주요 critical section은 file read, write 관련 함수 접근에서 발생할 것으로 예상된다.

#### 다. 연구원 역할 분담

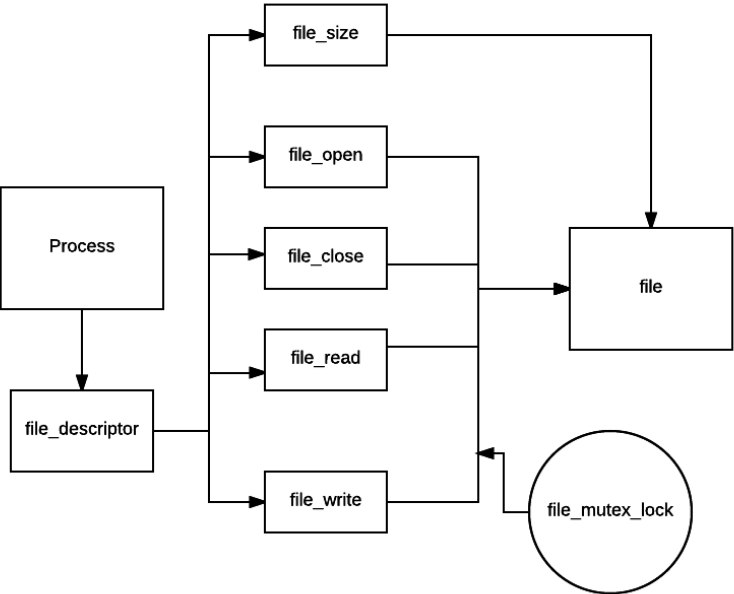
혼자서 진행하는 프로젝트이므로 충분한 시간을 두고 철저한 계획 하에 프로젝트를 진행한다. 일정 부분 이상 진행되지 않으면 중간 결과를 확인 할 수 없으므로 구현 사항에 우선 순위를 정하여 그 순서대로 프로젝트를 진행 한다.

IV. 연구 결과

1. 합성 내용: 전체적인 흐름도는 다음과 같다.



fileysys system call의 흐름도는 다음과 같다.



## 2. 제작 내용:

(1) **filesys system call** : filesys system call은 filesys library 내부의 다양한 함수를 활용하여 구현하였다. create는 filesys\_create, remove는 filesys\_remove, open은 file\_open, close는 file\_close, read는 file\_read, write는 file\_write, seek는 file\_seek를 이용하였다. 이 과정에서 mutex lock을 활용하여 critical section 문제를 해결하였다. file descriptor 발급, 관리 등과 관련된 structure에 대한 설명은 다음 내용에서 다루기로 한다.

(2) **file descriptor**: file descriptor 관리에 사용한 structure는 다음과 같다.

```
struct sync_tool
{
    struct semaphore wait;
    struct semaphore exec;
    struct semaphore exit;
    struct lock fd_lock;
    int exit_status;
    tid_t parent;
    struct list child_list;
    struct list file_list;
    int fd_gen;
    struct file* exec_file;
    struct list_elem elem;
};

struct file_data
{
    int fd;
    struct file* file;
    struct list_elem elem;
};

struct lock file_rw;
```

기존의 sync\_tool에 mutex lock fd\_lock, file\_list, fd\_gen을 추가하였다. fd\_lock은 fd 발급시 발생하는 race condition을 해결하기 위해 추가하였다. file\_list는 struct file\_data로 이루어진 linked list이며 fd\_gen은 thread 별 file descriptor generation을 위한 변수이다. 새 fd가 생성될 때마다 1씩 증가하도록 하였으며, 0, 1은 stdin, stdout을 위한 fd이므로 fd\_gen은 thread가 생성될 때 2로 초기화하였다. file에 대한 정보는 file\_data에 저장된다.

(3) **system call handler infrastructure:** 여러 system call을 처리하기 위한 switch 문 구조와 각종 함수들을 이용한 read, write, execute, wait, exit, halt 총 6개의 system call을 구현하였다. read, write, halt는 이미 구현된 관련 함수를 사용하였으며, parent와 child 간의 synchronization이 필요한 execute, wait, exit에 해당하는 함수들 사이에는 semaphore를 활용하였다.

synchronization을 위해 struct thread에 새로 추가할 내용은 struct sync\_tool이라는 구조체로 묶었다. 여기에 list\_elem elem을 추가하여 list의 다양한 기능을 활용할 수 있도록 하였다.

(3) **synchronization:** 이전 프로젝트에서 multi child waiting 문제가 해결되지 않았던 이유는 parent의 wait semaphore를 활용하여 process wait synchronization을 구현하였기 때문이다. 이 경우 여러 child 중 하나만 종료되더라도 parent의 waiting도 끝난다. 이를 child의 wait semaphore를 활용하는 것으로 바꾸어 multi child waiting 관련 다양한 test case를 모두 해결하였다. file system call 구현 중 read, write와 관련한 critical section access 문제는 mutex lock을 활용하여 어렵지 않게 해결하였다.

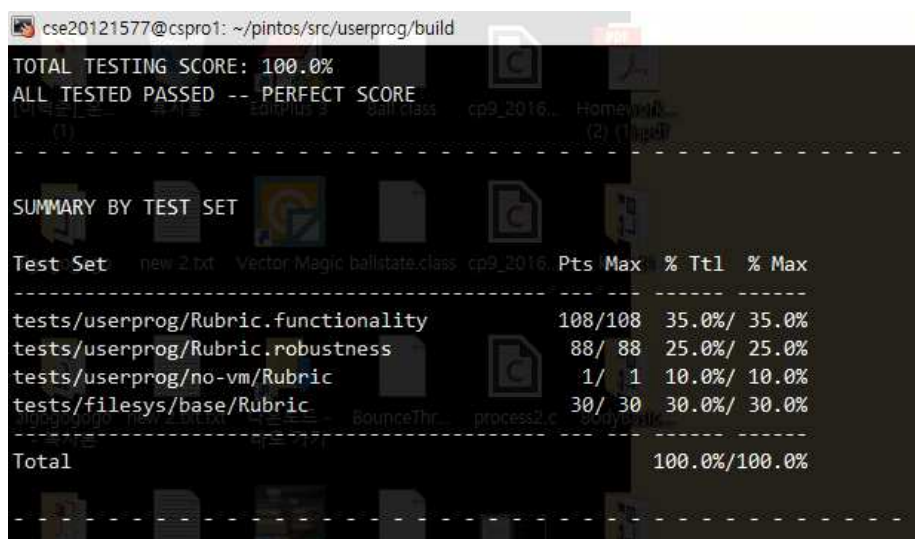
(4) **trial and error:** 이전 프로젝트에서 발생한 문제를 보완하고 file system 관련 test case를 모두 통과했음에도 불구하고 multi-oom test case가 통과되지 않자 원인을 파악하던 도중 zombie process가 계속 남아 memory loss를 발생시키고 있는 것을 발견했다. 이를 해결하기 위해 각 parent가 exit 전에 자신의 child\_list에서 zombie process를 탐색하고 이를 정상적으로 모두 종료 시킨 후 자신도 종료되는 code를 추가하였다. 이를 통해 multi-oom test case를 pass 할 수 있었다.

### 3. 시험 및 평가 내용:

Project 결과에 대한 평가는 make check의 pass/fail을 기준으로 진행하였다.

평가는 cspro2.sogang.ac.kr에서 진행하였으며 simulator로 qemu를 사용하였다.

그 결과는 다음과 같다.



```
cse20121577@csp01: ~/pintos/src/userprog/build
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

SUMMARY BY TEST SET
-----
Test Set      new 2.txt  Vector Magic ballstate.class  cp9_2016...  Home...
-----
tests/userprog/Rubric.functionality      108/108   35.0%/ 35.0%
tests/userprog/Rubric.robustness          88/ 88   25.0%/ 25.0%
tests/userprog/no-vm/Rubric               1/ 1    10.0%/ 10.0%
tests/filesys/base/Rubric                 30/ 30   30.0%/ 30.0%
-----
Total                                     100.0%/100.0%
```

총 76개의 test를 모두 pass 하였다.

(1) 다음 26개의 test를 통해 생산성을 test하였다.

```
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/wait-simple
pass tests/userprog/open-normal
pass tests/userprog/close-normal
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/create-normal
pass tests/userprog/read-normal
pass tests/userprog/read-stdout
pass tests/userprog/write-normal
pass tests/userprog/write-stdin
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
```

system call을 문제없이 수행하여 의도한 결과를 출력하여 생산성을 만족하였다.

(2) 다음 50개의 test를 통해 내구성을 test 하였다.

```
pass tests/userprog/open-twice
pass tests/userprog/close-twice
pass tests/userprog/close-bad-fd
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-bad-fd
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
```

```

pass tests/userprog/write-zero
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
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-seq-block
pass tests/filesys/base/lg-seq-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

```

각종 예외 상황을 적절하게 대처하여 Pintos OS의 비정상적인 종료를 방지하였다. 이로써 내구성을 만족하였다.

### (3) 보진 및 안정성 test

OS가 마주하는 예외 사항은 다양하다. 이에 대해 적절하게 대처하지 못하면 사용자의 불편을 초래하게 된다. 따라서 page fault를 방지하기 위해 모든 예외 상황을 파악하고 page fault가 발생하기 전에 해당 thread를 강제로 종료 시켰다. system call의 argument로 잘못된 값이 들어오는 경우는 다음과 같은 함수를 사용하여 안정성을 높였다.

```

void check_address(void* address){
    struct thread *cur=thread_current();
    if (address==NULL){
        thread_current()->sync.exit_status=-1;
        thread_exit();//exit(-1);
    }
}

```



```

    }
    else if(is_kernel_vaddr(address)){
        thread_current()->sync.exit_status=-1;
        thread_exit();//exit(-1);
    }
    else if(pagedir_get_page(cur->pagedir,address)==NULL){
        thread_current()->sync.exit_status=-1;
        thread_exit();//exit(-1);
    }
}
}

```

## V. 기타

### 1. 연구 조원 기여도: 김태훈 (100%)

2. 해당 프로젝트를 진행한 cspro1.sogang.ac.kr은 서버의 낙후화로 다양한 문제점들이 발생하였다. bochs simulator를 사용할 수 없어 qemu simulator만을 사용하여야 했으며, git이 설치되어 있지 않아 source code version control에 어려움이 있었다. 하지만 느린 서버 환경이 다양한 synchronization 문제를 발생시켜 이를 해결하며 Pintos OS 자체의 성능을 향상하는데 도움이 되었다.