

# **Pintos Project 2: User Program (2)**

담당 교수 / 분반: 박성용/제1전공

이름 / 학번: 박준혁/20171643

개발 기간: 10/04~10/12

## I. 개발 목표

본 User Program (2) Phase는 지난 'User Program (1) Phase'에서 구축한 pintOS System을 확장하는 단계로, '**File-Related System Call**'을 구현하는 것이 핵심이다. 'File-Related System Call'이란, 본인이 임의로 만든 비공식 용어로, OS File System의 기능을 Application Program에게 제공하는 각 종 System Call을 의미한다. 그러한 System Call에는 우리가 흔히 System Programming 시 접해본 open, read, write 등의 API 함수들이 포함되며, 우리는 본 Phase에서 이들을 pintOS Manual과 프로젝트 안내 PDF를 토대로 구현한다. 또한, File System을 Access하는 Kernel Code 부분에 대한 User Program의 접근을 Mutual Exclusion 등의 Synchronization 기법으로 보호하여 상기한 System Call들의 올바른 동작을 도모한다.

이 밖에도 'File Descriptor Table', 'Denying WRITES to Executable File'도 본 Phase의 중요 포인트이며, 지난 Phase에서 구현했던 exit, exec, wait과 같은 'Process & Thread 관리 System Call'들의 기능을 보완하고 개선하는 작업도 필요하다. 이러한 일련의 과정을 통해 단순히 File-Related System Call들의 동작 방식을 이해하는 것뿐만 아니라, 궁극적으로 운영체제가 어떻게 User Application Program에게 'Process & Memory Virtualization', 'Execution Flow', 'File System' 등을 제공하는지, 그리고 나아가 그들을 제공하는 이유가 무엇인지를 다각도로 이해할 수 있다.

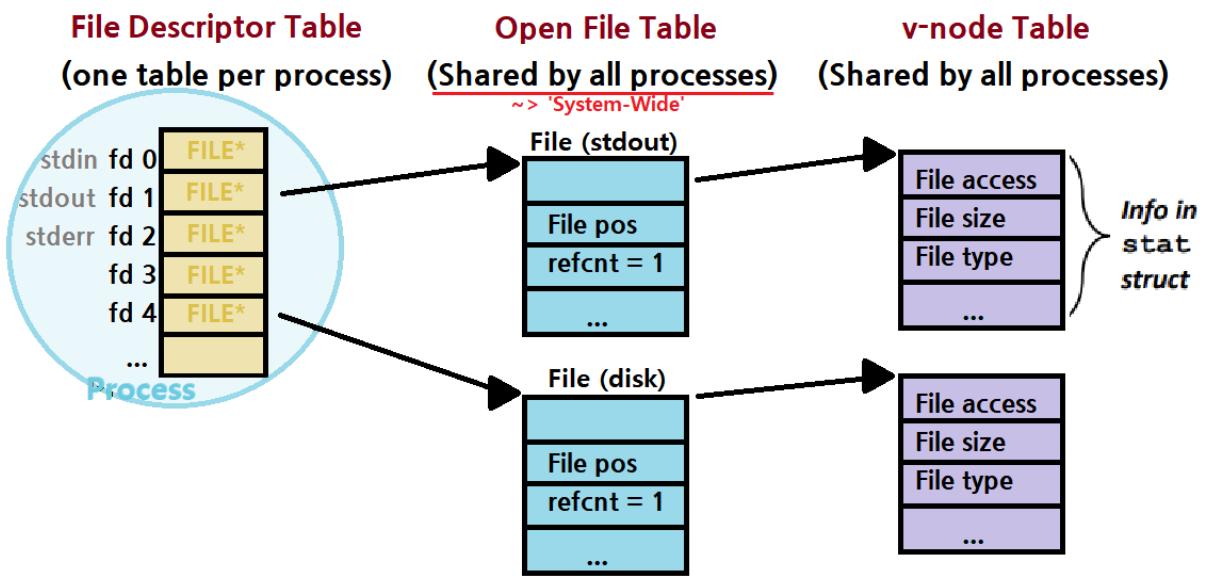
이러한 목표를 토대로 본인은 실제로 10월 4일부터 금일(10월 12일)까지 약 1주간 성실히 개발에 집중하였으며, 그 과정에서 상기 원리들에 대해 본인은 어떻게 분석하고 이해하였는지, 그리고 그러한 이해를 어떻게 Code-Level에서 표현하였는지, 그 표현 과정에서 있었던 Troubleshooting은 어땠는지 등을 이하 항목에서 차례대로 설명하겠다.

## II. 개발 범위 및 내용

### A. 개발 범위

#### i. File Descriptor

본 프로젝트 Phase에선 open, read, write를 말미암은 'File-Related System Call'의 구현이 핵심이라 언급했다. 이를 위해선 무엇이 가장 선행되어야 할까. 가장 먼저 해야 할 일은 뭘까? 그것은 바로 'File Descriptor Table'의 구축이다. System Programming 및 Operating Systems 강의에서 다 차례 다루었듯이, **모든 Process는 각자 자신의 'File Descriptor Table (약칭 FDT)'을 가진다.** 자신이 Open한 File들이 무엇 무엇이 있는지를 이 FDT에 기록해야 그와 관련한 각 종 File Operation을 적재적소에 할 수 있기 때문이다. 아래 그림을 보자.



위와 같이 Process마다 FDT가 있으며, 해당 FDT의 각 Slot은 대응되는 각 File에 대해, File마다 존재하는 'Open File Table'을 가리키는 그러한 구조이다. 본 Phase에선 Open File Table을 위시한 File System의 기본적 구조는 이미 제공되기 때문에, 우리는 Process마다 FDT를 마련하기만 하면 된다.

## ii. (File-Related) System Calls

이어서, (i)에서 언급한 File Descriptor Table이 구축된다면, 이들을 토대로 지난 Phase에서 구현하지 않았던 'File-Related System Calls'를 완성한다. 그러한 System Call들에는 **create, remove, open, close, filesize, read, write, seek, tell**이 있으며, 이들은 모두 pintOS 'Base File System (Code 관점에서는 src/filesys/filesys.h or file.h 등)'의 기본 제공 함수를 적절히 변형하여 구현할 수 있다. 그 중 일부의 경우는 Synchronization이 필요하며, 이에 대한 설명은 바로 아래에서 이어간다.

## iii. Synchronization in File System

'File-Related System Call'의 동작에 있어서 Synchronization은 어떤 것이 필요할까. pintOS 상에서 복수의 Process가 돌아간다고 가정하자. 이들이 모두 동시에 여러 차례 read나 write System Call을 사용한다고 하자. write System Call의 경우, src/filesys/file.c 파일의 file\_write 함수를 사용해야 하는데, file\_write 함수 내부를 보면 해당 File의 inode Structure에 접근하고 있음을 어렵지 않게 확인할 수 있다. 아직 inode Structure에 대한 명확한 이해는 필요가 없으나, 과거 System Programming에서 학습했듯, 이 Structure는 OS 내에서 각 File에 대해 고유하게 존재한다. 그 말은 즉, 'Shared Data Structure'인 것이

다. 그리고, 상기한 ‘복수 Process’ 상황은 ‘**Shared Data Structure**’에 대한 **Competition**인 것이다. 따라서 우리는 Synchronization을 통해 이러한 Competition 상황에서 Shared Data가 적절히 분배되고 변형될 수 있도록 보장해야 한다.

한편, pintOS에선 Running Program의 Executable File에 대한 Write Operation을 금지해야 하는 조건이 있으며, 이에 대한 처리도 본 Step에서 다뤄야 한다.

이 밖에도 앞서 ‘개발 목표’에서 언급한 바 있듯, 본 Phase Test Case (특히나 ‘multitoom’)에서 중점적으로 확인하는 ‘**Memory Leakage**’ 문제를 해결하기 위해 `exit`, `exec`, `wait` System Call들의 개선 작업도 개발 범위에 포함된다. 아래에서부터는 이 부분을 (ii) 항목에 포함해 설명도록 하겠다.

## B. 개발 내용

### i. File Descriptor

File Descriptor Table의 구현은 간단히 Array로 할 수 있다. pintOS Manual에 따르면 pintOS 상에선 FDT의 최대 크기로 128을 상정하고 있고, 그 정도 Size의 Array이면 File Descriptor에 대한 Search를 할 때 단순 Iterative Method를 적용하더라도 Overhead가 크지 않기 때문에, 구현이 간편하고 효율적이기도 한 Array가 가장 적합하다고 판단했다. 따라서 src/threads 디렉토리의 thread.h 파일에 **128** 크기의 정적 배열을 선언하였다. 이 때, Array의 Type은 pintOS에서 File Structure를 나타낼 때 사용하는 ‘`struct file *`’ Type으로 지정한다. 즉, **Array의 Index가 곧 File Descriptor**인 것이고, 해당 Index의 Slot에 위치한 Data가 File Structure (Open File Table)라 볼 수 있겠다.

### ii. System Calls

‘Base File System’을 이용할 ‘File-Related System Call’은 앞서 언급하였듯 `create`, `remove`, `open`, `read`, `write`, `filesize`, `seek`, `tell`, `close`가 있으며, 이들의 기능은 다음과 같다.

- 1) **create:** 넘겨 받은 File Name을 이름으로 가지는 File을 지정된 Size로 생성한다. 생성이 성공적이면 `True`를, 실패했다면 `False`를 반환하며, 이 작업은 src/filesys/filesys.h에서 미리 제공하는 `filesys_create` 함수로 간단히 처리할 수 있다.
- 2) **remove:** 넘겨 받은 File Name을 이름으로 가지는 File을 삭제하는 System Call로, Open 또는 Close 여부와 상관 없이 삭제할 수 있으며, 1)과 마찬가지로 filesys.h에 정의된 `filesys_remove` 함수를 간단히 호출하여 처리할 수 있다.

- 3) **open:** File Name에 해당하는 File을 File System에서 찾아 Calling Process에게 Open시켜주는 System Call로, `fsys_open` 함수를 통해 해당 File의 Open File Table 구조체를 반환 받은 후, 해당 File Structure에 File Descriptor를 지정하는 기능이 핵심이다.
- 4) **filesize:** System Call 이름 그대로 File의 Size를 반환하는 함수로, `src/filesys/file.h`에 정의된 `file_length` 함수에게 기능을 위탁한다. 참고로, 본 System Call은 User Program으로부터 File Descriptor를 넘겨 받기 때문에 fd에 대한 예외 처리가 필요할 것이다.
- 5) **read:** File Descriptor가 가리키는 Open File에서 Size만큼의 Byte를 읽어 들여 Buffer에 저장하는 System Call로, 지난 Phase에서 Console 입력만을 처리했다면, 본 Phase에선 Process가 Open한 모든 File에 대해 작동할 수 있어야 한다.
- 6) **write:** Size만큼의 Byte를 Buffer로부터 읽은 다음 File Descriptor가 가리키는 Open File에 출력하는 System Call로, `read`와 마찬가지로 모든 Open File에 대해 동작할 수 있어야 한다. 참고로, `open`, `read`와 함께 Synchronization이 필요한 System Call이다.
- 7) **seek:** File Descriptor가 가리키는 Open File의 '다음에 읽거나 쓸 위치'를 Position이란 인자로 변경하는 System Call로, C Library Function의 `fseek` 기능을 생각하면 된다. `file.c`의 `file_seek` 함수를 통해 간단히 구현할 수 있다.
- 8) **tell:** 7)에서 설명한 '다음에 읽거나 쓸 위치'를 알려주는 System Call이다. 역시나 마찬가지로 간단히 `file.c`의 `file_tell` 함수를 통해 구현하며, 이런 류의 간단한 System Call들은 예외처리(fd 오류 또는 File Structure 오류)가 핵심이겠다.
- 9) **close:** File을 Close하는 System Call로, 간단히 `file_close`라는 함수를 불러서 처리할 수 있다. 다만, `file_close`에선 (close System Call에 인자로 넘어온) File Descriptor가 가리키는 File Structure를 NULL화하는 기능이 없기에 이를 추가하는 작업이 필요하다.

한편, 지난 Phase에서 구현했던 `exit`, `exec`, `wait`을 보완 및 개선하여 Operating System의 Memory Leakage를 방지하는 기능 역시 필요하다. 지난 Phase User Program (1)을 돌이켜 보면, `exit`, `wait`, `exec`의 기본적인 기능은 모두 구현되어 있지만 다음과 같은 Critical한 문제점이 존재했다.

**"만약, User Program이 비정상적으로 행동하면 어떡할 것인가?"**

이때, '비정상적인 행동'이라 함은, Operating System에 대한 악의적 공격(일부러 Child Reaping을 진행하지 않는 행위 등)부터, Page Fault상황과 같이 프로그래밍 실수로 인한 비정상 종료 등을 의미하는데, 우리가 앞선 Phase에서 구축한 pintOS는 이러한 상황을 올바르게 처리하지 못한다. 왜냐? 그러한 처리를 애초에 개발하지 않았기 때문이다.

따라서, 본 User Program (2) Phase에선 Memory Leakage 방지를 위한 여러 가지 작업을 해주어야 하며, 이들을 System Call 관점에서 간단히 소개하면 다음과 같다.

- (1) **exec(process\_execute)**: Parent Process가 exec를 통해 process\_execute 함수를 호출하는데, 이러한 Child Process 생성 및 실행 과정에서 'Current(Parent) Process가 Newly-created Child Process의 Load보다 먼저 종료되지 않도록(Load 실패를 처리할 수 있도록)' 고정한다. (Counting Semaphore가 사용될 수 있음)
- (2) **exit**: Process가 종료될 때, 굳이 User Program이 wait System Call을 호출하지 않더라도 예하 Child Process가 모두 Reaping되도록 처리한다. 마치 실제 Linux에서 init Process가 Orphaned Children을 Reaping하듯이 말이다.
- (3) **wait**: wait 자체는 수정할 것이 없다. 다만, exec System Call 수행 과정에서 'Child Process Load 시의 비정상 종료 가능성'을 처리하기 위해 이 wait System Call을 호출하도록 만든다. (wait 수정 x)

이러한 작업을 수행한다. 참고로, exit System Call에선 해당 Process의 File Descriptor Table을 Clearing하는 작업도 있어야 할 것이다.

### iii. Synchronization in File System

File System을 다루는, 정확히는 **File System 관련 OS Kernel Code**에 접근하는 **System Call들**에 **Synchronization**이 필요하다고 언급하였다. 하지만, 모든 File-Related System Call에 필요한 것은 아니다. 각 'File-Related System Call'은 위에서도 알 수 있듯, filesystem 디렉토리의 file.h 또는 filesystem.h의 Procedure를 호출해서 Main Job을 맡기고 있는데, 이때, 그 '호출 함수'들 중 '복수 Process가 Data Structure를 Share함'이란 특성을 가지고 있는 함수는 앞서 예시로 들었던 file\_read와 같은 함수만이 존재한다. 주로 inode Structure에 대한 접근 등이 특징인데, 이러한 함수들을 종합하면 open, read, write System Call에서 호출하는 함수들로 추릴 수 있다. 따라서 우린 open, read, write를 보호해야 한다.

이때, 우리는 **Lock(Binary Semaphore)**이나 **Semaphore(Counting Semaphore)**를 도입하여 **Synchronization**을 구현할 수 있다. 두 방식은 결국 'Critical Section을 보호하기 위함'이라는 목적 아래 동작하기에 유사하지만, 그럼에도 약간의 차이가 있다.

- **Lock Method**: Binary Semaphore로서, 진정한 Mutual Exclusion을 제공한다. 즉, '하나의 시간 단위'에 오직 하나의 Process만이 Critical Section에 접근할 수 있도록 한다. 이렇게 하면 말 그대로 해당 Code에 단 하나의 Process만이 접근할 수 있어 File System의 Shared Data Structure가 거의 완벽하게 보호될 수 있다. 그러나, '한 시간 단위에 한 Process'는 곧 (Data에 수정을 가하지 않는) 여러 Process의 접근을 '줄 세우기'하는 것이므로 '불필요한

상황에서의 Process 대기 Overhead'를 만들어낼 수 있는 문제점이 있다.

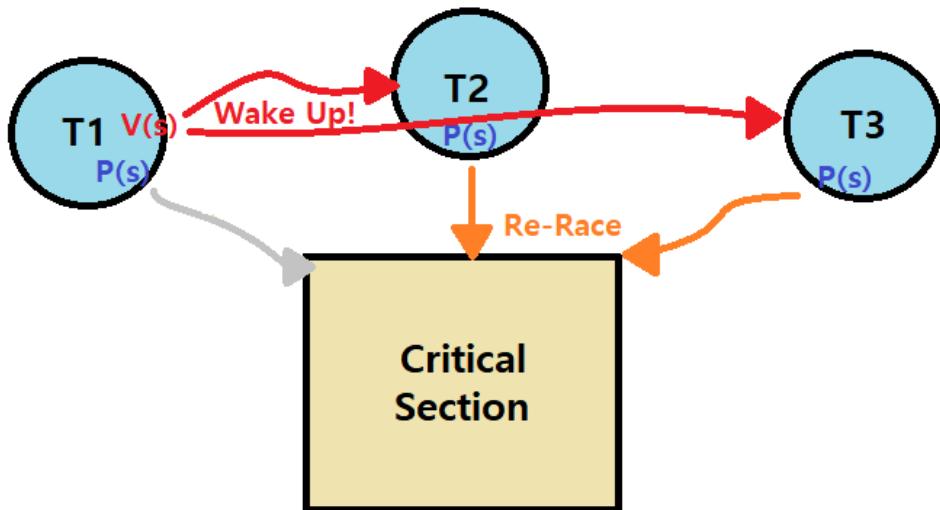


Figure 1 - Binary Semaphore (Lock, mutex)

- **Semaphore Method:** Counting Semaphore로, File System Kernel Code에 대한 'Readers-Writers Problem' 적용이 가능하다. File Structure가 가리키는 inode 등의 Shared Data Structure를 Shared Object로서 바라보고, Reader(read System Call을 호출한 Process) 또는 Writer(write System Call을 호출한 Process)에게 Priority를 부여해 상기한 '불필요한 상황'에서의 Process 대기 Overhead'를 낮출 수 있다. 그러나, 반대로 이 방식은 Lock과 다르게 미묘한 Execution Flow를 포착해야 하는 프로그래밍적 부담이 존재한다.

본인의 경우 이 부분에서 구현이 간편한 Lock Method를 채택하였다. 구현이 간편할 뿐만 아니라, pintOS에선 "Ensure that when one process is executing in its critical section, no other process can execute in its critical section."이란 조건도 존재하기 때문이다. 진정한 의미의 Mutual Exclusion을 권장하고 있는 것이다. 따라서, File System Code에 대한 Synchronization에 Lock Method를 도입한다.

한편, 이 밖에도 'Running Program의 Executable File에 대한 Write 문제'를 처리하기 위해 pintOS 제공 'file\_deny\_write/file\_allow\_write' 함수 사용이 필요하며, File System과는 별개의 이야기지만, 앞서 소개한 'exec/exit System Call의 개선' 과정(Load Routine의 수행 이전에 Parent Process가 먼저 죽어버리는 상황의 개선)에서도 Counting Semaphore의 적용이 필요하다. 요약하면, 본 Phase에선 크게 세 번의 Synchronization 작업이 필요하다고 할 수 있겠고, 그 중에서도 File System 관련해서는 Lock Method / Semaphore Method의 적용이 가능하고, 본인의 경우, 여기에선 전자의 방식을 채택한 것이다.

### III. 추진 일정 및 개발 방법

#### A. 추진 일정

지난 User Program (1) 때와 마찬가지로 개발 기간의 대부분을 '프로젝트 Phase'에서 요구하는 상황'의 이론적 분석에 집중하였으며, Code-Level Implementation은 사실상 이를 정도밖에 소요되지 않았다. 아무래도 본 Phase의 Solution이 지난 Phase의 그것과 유사하기 때문에 기초적인 Code-Level Troubleshooting이 많이 필요치 않아서였던 것 같다. 결과적으로 개발 일정을 요약하면 다음과 같다.

- 1) **Understanding 'Base File System' (10/04-10/08):** 프로젝트 안내 PDF와 pintOS Manual을 토대로 Base File System의 구조와 기능 제공 원리를 개괄적으로 분석하였다. 이때, filesystem 디렉토리의 file.c와 filesystem.c 파일을 중점적으로 분석하였으며, System Call 구현 시 어떤 Function이 쓰일 것인지를 각각 정리했다. ex) file\_write -> write System Call
- 2) **File Descriptor (10/08):** File Descriptor Table 구현을 상기 설명 및 분석을 토대로 가볍게 수행하였다.
- 3) **System Calls & Synchronization in Files System (10/08-10/10):** File-Related System Call을 포함하여 본 Phase에서 수정을 요하는 모든 System Call의 개발을 수행했다. 이론적 배경이 그다지 복잡하지 않았기에 Synchronization은 System Call 개발과 동시에 진행하였으며, 특히나 File-Related System Call의 구현은 10/08일 하루 만에 모두 완성하였다.

의외로 난해했던 지점은 pintOS 'multi-oom' Test Case를 말미암은 **Memory Leakage**의 처리 과정이었으며, 이 과정에 약 '하루 + 반나절' 정도 소요되었다. 그 중에서도 Counting Semaphore와 Busy Waiting을 구현하는 과정이 가장 주요했다. 본인의 경우, User Program (1)에서 간단한 wait Routine만을 구현했기에 User Program (2)에선 '필요한 부분' 모두에서 Wait(Reaping)을 설치해야 했고, 이 과정이 상당히 까다로웠다. 자세한 Code-Level Detail은 아래 항목에서 설명하겠다.

모든 Test Case의 통과를 확인한 시점은 10/10일이었으며, 이후 이를 정도 (10/11-10/12) 보고서 작성 및 코드 정리를 수행하였다.

## B. 개발 방법

### i. File Descriptor

File Descriptor Table의 구축은 계속 언급하는 것처럼, 상당히 간단히 수행하였다. 모든 Process가 128 사이즈의 Array를 갖추면 되는 일이기 때문에 threads 딕토리의 thread.h 파일에서 해당 Table을 정의하고, thread.c에서 Array의 각 Element를 NULL로 초기화하기만 하면 됐다. 특별한 함수의 추가는 필요 없었다.

### ii. System Calls

'File-Related System Call'의 구현 역시 마찬가지로 어렵지 않다. userprog 딕토리의 syscall.h에 각 System Call Handling Routine 함수들을 선언하고, syscall.c에서 각 함수들을 정의하면 된다. 기존의 System Call 추가와 다를 것이 없었으며, User Program (1)에서 정의했던 다음의 Macro를 요긴하게 사용했다. 이 중 '**POINTER\_CHECK**' 매크로 함수는 지난 Phase에선 'CHECK\_ROUTINE'이란 이름으로 정의되어 있었는데, 본 Phase에선 'File-Related System Call' 구현 간에 각 System Call 파라미터로 넘어오는 **Pointer Value들의 Validation**에 직접 사용했기에 이름을 아래와 같이 변경한 것이다. 한편, 'OPEN\_FILE\_ERROR'는 open 함수의 에러 처리 값이다.

```
/* Macros for readability of codes (whole) */
#define POINTER_CHECK(vaddr) if (vaddr == NULL || is_user_vaddr(vaddr) == false) exit(-1);
// POINTER_CHECK: checks if an argument is in the user address space
//           with pre-provided 'is_user_vaddr' function. And, check NULL also!
#define USER_ADDR_CHECK(param_num) for(int i=1;i<=param_num;i++){POINTER_CHECK(ARG_ADDR(i))}
// USER_ADDR_CHECK: checks all the parameters that a system call needs
//           with consequetively calling 'POINTER_CHECK' macro above!

#define OPEN_FILE_ERROR -1
// OPEN_FILE_ERROR: indicates that an error occurs in the 'open' syscall
```

한편, 앞서 계속 언급한 것처럼, 본 Phase에선 exec, exit System Call의 수정, 정확하는 Process Execution 과정에서의 'Load/start\_process 실패 Children'에 대한 Busy Waiting, 그리고 Process Exit 과정에서의 'Zombie Children' 및 'Remaining Children'에 대한 Busy Waiting이 필요한데, 이 작업은 exec System Call을 위탁 수행하고 있는 userprog/process.c의 process\_execute의 수정, 그리고 syscall.c의 exit System Call Routine 수정을 통해 수행하였다.

이 과정에서 모든 Process가 비정상적으로 종료될 때 '-1'을 Exit Status로 지정할 수 있도록 userprog/exception.c의 Page Fault 함수를 좀 더 보완하는 작업도 필요했다. (Test Case 중 일부 Failing Tests의 원인 분석을 통해 이 필요성을 포착하였음)

참고로, Busy Waiting 과정에서, Child Process를 담고 있는 List에 대한 Iteration Code가 가독성이 떨어지는 측면이 있어 아래와 같은 새로운 Macro를 process.h 파일에 추가하였다.

```
/* Macros for readability of codes (child list iteration codes) */
#define ITERATE_C_LIST iter=list_begin(c_list);iter!=list_end(c_list);iter=list_next(iter)
// ITERATE_C_LIST: repeat the list data structure iteratively
#define EACH_CHILD (list_entry(iter, struct thread, child_elem))
// EACH_CHILD: get the individual entry from the list data structure
```

### iii. Synchronization in File System

상기 설명처럼, File System 부분 Kernel Code에 대한 Synchronization은 Binary Semaphore인 Lock을 통해 수행했다. 해당 Kernel Code는 filesystem 디렉토리의 file.c, filesystem.c에 정의되어 있고, 이 Routine들을 우리는 System Call Handler에서 직접 호출해 User Program에게 기능을 제공한다. 따라서, userprog/syscall.c의 open, read, write System Call 구현 부분에 이러한 Lock을 적용했다. pintOS에서 미리 제공하는 lock Structure를 사용했으며, 이를 위해 syscall.h에 Global Variable로 해당 Mutex를 선언하는 것이 필요하다. 그리고 System Call Handler가 처음 초기화될 때 이 Mutex도 함께 초기화해야 한다.

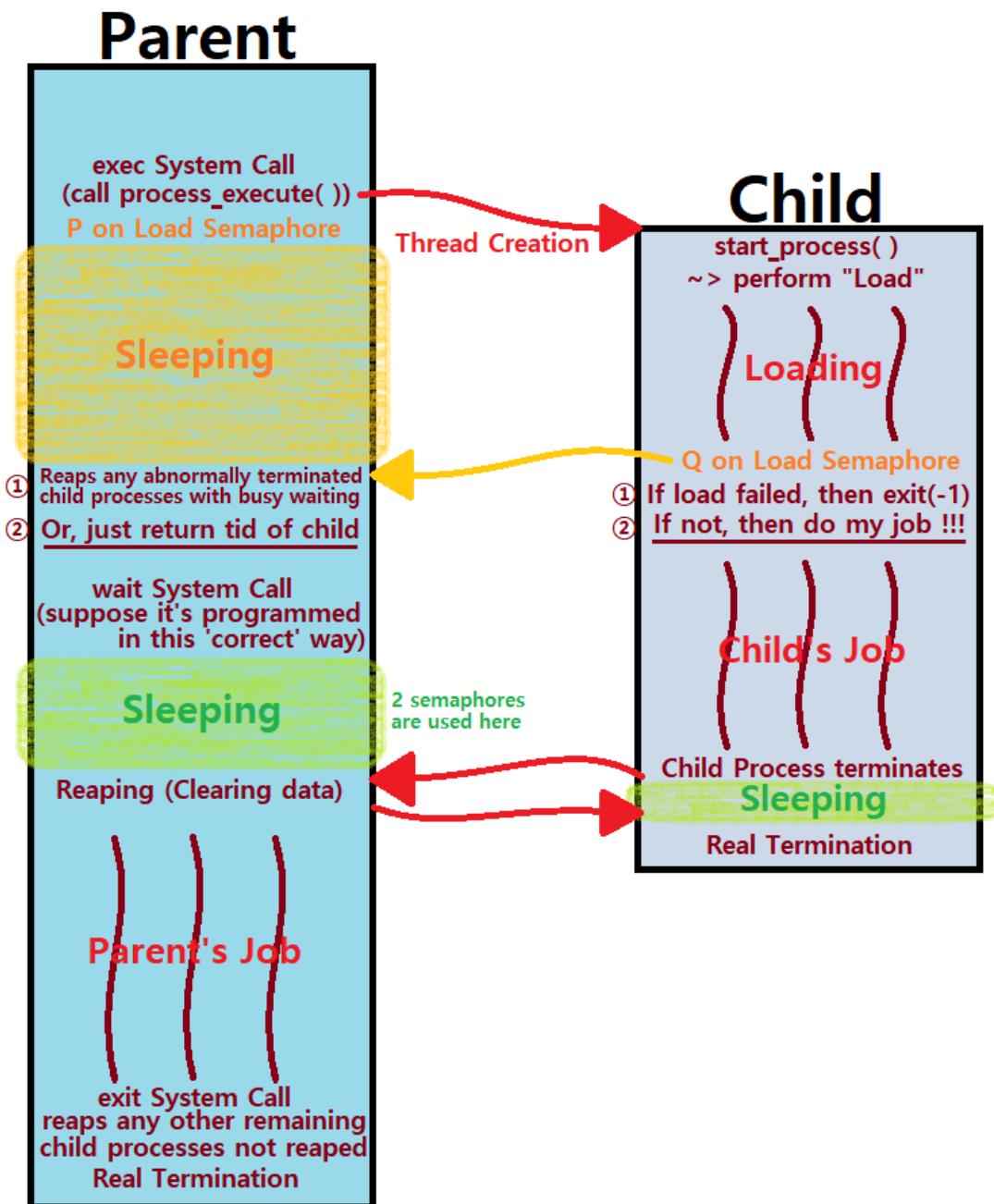
'Denying Write Operations to Executable File'의 구현은 각 Process가 open System Call로 File Open 시 해당 File이 Running Program의 ELF와 동일하면 file\_deny\_write 함수를 호출하도록 지정하여 수행했다. 이 함수에서는 해당 File의 Open File Table에 있는 특정 Flag를 Set하고, 이를 통해 다른 Process에게도 이 'Denying' 사실을 알릴 수 있다. 타 Process에선 write System Call에서 이를 확인해 ELF에 대한 Write를 미연에 방지할 수 있다.

exec System Call, process\_execute 함수 수행 과정에서의 Synchronization은 지난 Phase에서 Parent Semaphore, Child Semaphore를 도입했던 방식과 동일하게 수행하였다. process\_execute 함수에서는 Thread Creation & Start Process & Load Routine의 연쇄적 작업을 수행하는데, 그 작업을 호출한 후 새로 도입된 'Counting Semaphore (Load Semaphore라 하자)'에 P Operation을 수행해 Sleep한다.

같은 시간에, Thread Creation & Start Process를 통해 새롭게 생성된 Child Process는 Load Routine을 수행한다. Load가 끝나면, 그 직후 바로 Child Process가 수행되는 것은 아니다. process.c 파일의 start\_process Routine을 확인해보면, 해당 Routine의 종료 직전에 Child Process의 Job을 실행시킨다. 따라서, 그 사이에서 무언가 일을 할 수 있는 것이다.

즉, 여기서 우리는, Load의 성공/실패 여부와 상관없이 Parent Process에 대한 Load Semaphore를 풀어주어야 할 것이다. Q Operation으로 깨우는 것이다. 그리고, 만약, Load가 성공적이었다면(success 변수가 True) Child Process를 수행하고, 그렇지 않았다면 해당 Child를 "exit(-1);"로 죽이는 것(이 비정상 종료에 대한 처리가 바로 상기한 Busy Waiting으로 이뤄짐)이다. 우리는 이러한 흐름의 동기화 작업을 입혀주어 exec, exit System Call이 제대로 돌아갈 수 있게 만들어야 한다.

아래는 전체적인 Process 생성 및 해제 흐름에 대한 도식이다. 어떤 부분에서 어떤 목적으로 Synchronization이 도입되었고, Busy Waiting은 어떻게 사용되는지 등을 전반적으로 이해할 수 있다. (Parent는 exec 이후 wait을 하도록 설계되었다 가정)

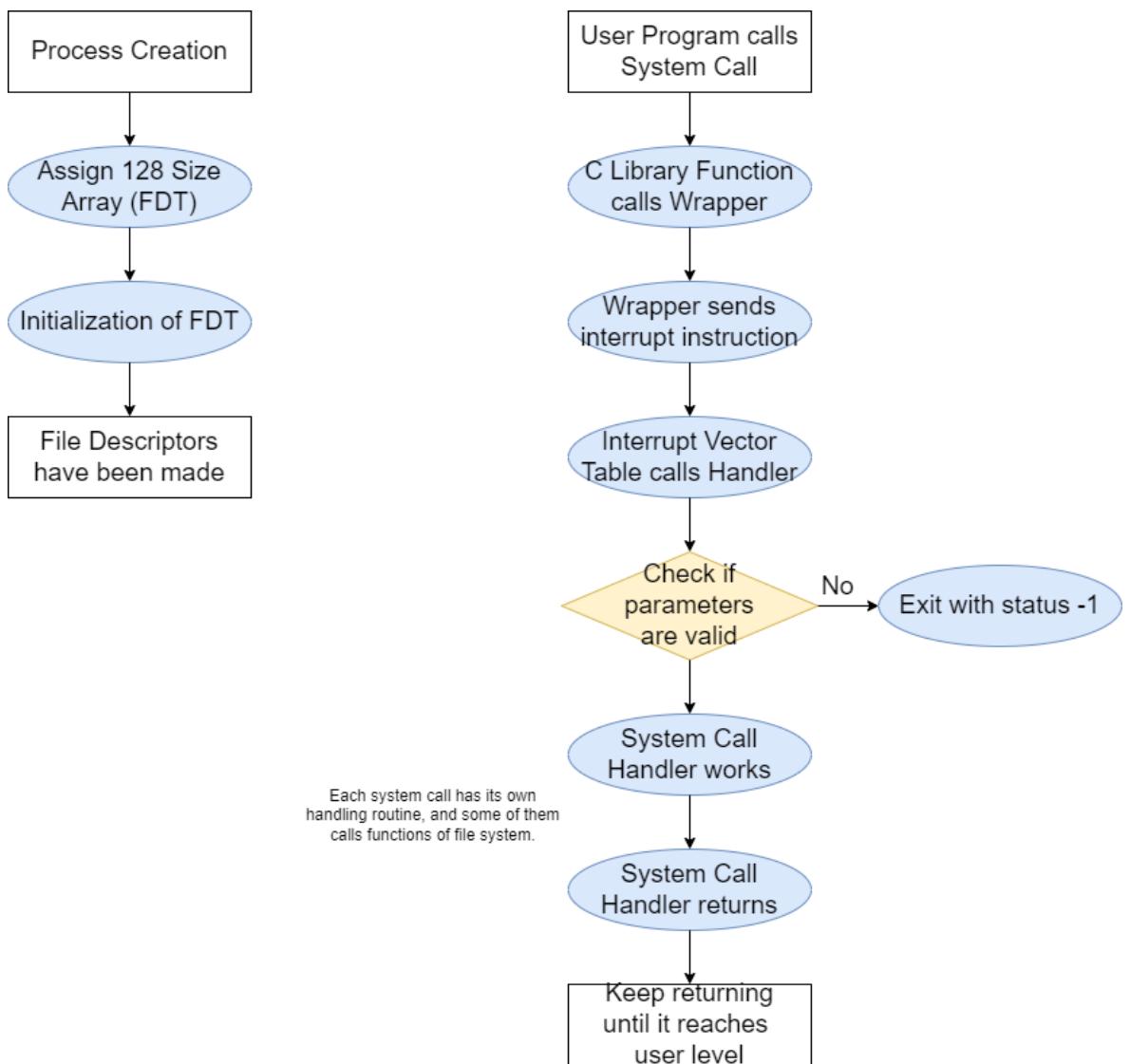


## IV. 연구 결과

### A. Flow Chart

#### i. File Descriptor

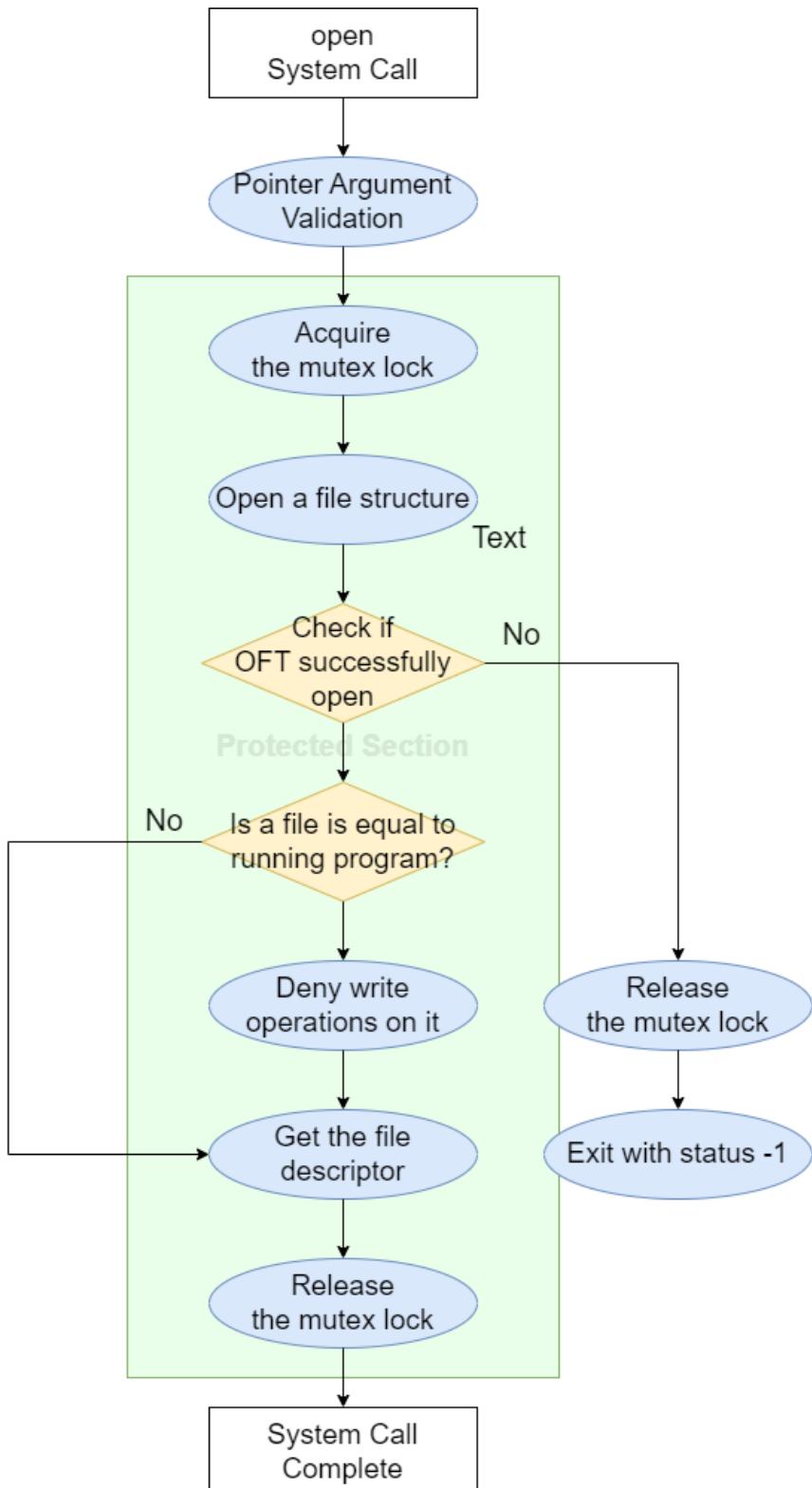
아래 시각 자료에서 좌측 Flow Chart는 File Descriptor 생성 과정을, 우측 Flow Chart는 System Call 구현 과정을 나타낸다. System Call의 경우 지난 Phase에서의 Flow Chart와 차이가 없다. 기본적인 System Call의 구조는 변함이 없기 때문이다.



#### ii. Synchronization in File System

File System에서의 Synchronization은 아래와 같이 묘사할 수 있다. 참고로, Process Execution 과정에서의 Synchronization은 본 항목의 물음과 매치되지 않으므로 생략

한다. 다만, 바로 이전 장의 시각 자료로 그 Flow를 충분히 이해할 수 있다.



read, write System Call에서의 Synchronization Flow는 위의 Flow Chart에서 사실상 Critical Section 내부의 동작 내용만 바뀔 뿐, 흐름은 변함이 없으므로 생략한다. 나머지 File-Related System Call에서는 Synchronization이 필요치 않다.

## B. 제작 내용

### i. File Descriptor

앞서 계속 강조해온 것처럼, File Descriptor의 구현은 아래와 같이 Array 자료구조를 도입해 간단히 처리할 수 있다. threads/thread.h 파일 내부이다.

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                      /* Page directory. */
    struct semaphore load_lock;             /* Semaphore for implementing exec */
    struct thread *parent;                  /* This is a helper for 'load_lock' */

    struct semaphore parent_lock;           /* Semaphore for implementing wait */
    struct semaphore child_lock;            /* Sem for suspending memory clear */

    struct list child_list;                 /* List storing children of process */
    struct list_elem child_elem;            /* List element declaration */

    struct file *fd[FD_MAX];                /* File Descriptor Table */

    int exit_status;                       /* Exit status of a process */
#endif
```

#ifdef USERPROG 정의를 통해, 상기 Data Structure는 모두 User Program, 즉, Process에게 할당될 것임을 알 수 있다. FD\_MAX라는 매크로 변수는 정수 128로, pintOS Manual에서 지정(권장)한 File Descriptor 최대 크기이다. 참고로, 본 항목의 내용은 아니지만, 아래에서 설명할 load\_lock Semaphore와 해당 Lock Routine의 구현에 사용되는 Parent Pointer도 확인할 수 있다.

이어, thread.c 파일에서 File Descriptor Table을 초기화한다. 간단한 반복문을 통해 모두 NULL로 초기화하고 있다.

```
#ifdef USERPROG
    /* Create a counting semaphore */
    /* for protecting load routine */
    sema_init(&(t->load_lock), 0);
    t->parent = running_thread();
    /* Create a counting semaphore */
    /* for child & memory locking */
    sema_init(&(t->parent_lock), 0);
    sema_init(&(t->child_lock), 0);
    /* Initialize the children list */
    list_init(&(t->child_list));
    list_push_back(&(running_thread()->child_list), &(t->child_elem));
    /* Initialize the File Descriptor Table */
    for (i = 0; i < FD_MAX; i++) (t->fd[i]) = NULL;
#endif
```

Load Semaphore와 Parent Pointer의 초기화도 확인할 수 있다. 이때, Parent Pointer의 경우 '현재 Process Creation 시점'에서의 Running Thread, 즉, Newly Created Child Process의 Parent Process를 그대로 대입하여 초기화하고 있음을 주목하자. Thread Creation이 완료되면 Running Thread가 가리키는 대상이 변화하게 될 것이다.

이렇게 간단히 File Descriptor를 구현한다.

## ii. System Calls

## iii. Synchronization in File System

System Call은 userprog/syscall.c 파일이 Main이다. 우선, System Call Handler 내부의 (논리적) System Call Table인 switch문을 확인해보자.

```
case SYS_CREATE:      /* Create a file. */
USER_ADDR_CHECK(2);
f->eax = create(ARG(1, char*), ARG(2, unsigned));
break;

case SYS_REMOVE:      /* Delete a file. */
USER_ADDR_CHECK(1);
f->eax = remove(ARG(1, char*));
break;

case SYS_OPEN:        /* Open a file. */
USER_ADDR_CHECK(1);
f->eax = open(ARG(1, char*));
break;

case SYS_FILESIZE:    /* Obtain a file's size. */
USER_ADDR_CHECK(1);
f->eax = filesize(ARG(1, int));
break;

case SYS_READ:        /* Read from a file. */
USER_ADDR_CHECK(3);
f->eax = read(ARG(1, int), ARG(2, void*), ARG(3, unsigned));
break;

case SYS_WRITE:        /* Write to a file. */
USER_ADDR_CHECK(3);
f->eax = write(ARG(1, int), ARG(2, void*), ARG(3, unsigned));
break;
```

공간 상의 이유로 그 일부만 첨부한다. 지난 Phase에서 가독성 제고를 위해 도입한 ARG, USER\_ADDR\_CHECK와 같은 매크로 함수를 적절히 사용하여 어렵지 않게 구현할 수 있다. switch문에서 입력 정수(esp Pointer가 가리키는 정수)의 값을 토대로 미리 정의된 Enumeration Type Number와 대조하여 그 정수에 대응하는 알맞은 Handling Routine을 호출하고 있다. 본 Phase에선 이들을 포함해 총 9개의 File-Related System Call을 구현하였고, exec, exit System Call

을 수정/개선해 총합 11개의 System Call을 구현하였다.

```
/* Create routine: checks the value of pointer variable
   and simply calls 'filesystem_create' func of filesystem.h */
bool
create(const char* file, unsigned initial_size)
{
    POINTER_CHECK(file);
    return filesystem_create(file, initial_size);
}

/* Remove routine: implemented just like 'create' above */
bool
remove(const char* file)
{
    POINTER_CHECK(file);
    return filesystem_remove(file);
}
```

자, 이어서 'File-Related System Call'을 개별적으로 확인해보자. 좌측에는 create, remove System Call의 구현 내용이 나타나 있다. 두 System Call 모두 User로부터 Pointer를 넘겨받기 때문에 앞서 소개한 POINTER\_CHECK 매크로 함수로 해당 포인터 유효성을 확인하고 있다. 둘 모두 간단히 File System 함수를 부르고 있다.

이어서 대망의 open 함수를 보자. 코드와 함께 설명하기 때문에 따로 System Call 구현 내용과 Synchronization을 구분하지 않고 두 개념을 함께 설명하겠다.

```
/* Open routine: opens the corresponding file structure
   of the target file, with assigning a proper descriptor.
   We should guarantee two things here.
   - only one process at a time can access this code.
   - if a file is as same as the running program, then
     deny any write operations on that file */

int
open(const char* file)
{
    struct file **fd_list;
    struct file *f;
    int i, idx;

    POINTER_CHECK(file);                                // check if a
    lock_acquire(&access_lock);                         // give mutual

    /* Open 'open file table' of the input file.
       If open fails, then return with -1 status */
    f = filesys_open(file);
    if (f == NULL)
    {
        lock_release(&access_lock);
        return OPEN_FILE_ERROR;
    }

    /* If a file is as same as the running program,
       then deny any write operations on this file */
    if (!strcmp(thread_name(), file))
        file_deny_write(f);

    /* Assign proper file descriptor to file struct */
    fd_list = &(thread_current()->fd);
    for (i = 3; i < FD_MAX; i++)
    {
        if (fd_list[i] == NULL)
        {
            fd_list[i] = f;
            idx = i;
            break;                                     // first-fit a
        }
    }
    if (i == FD_MAX)                                    // if FDT is f
    {                                                    // return with
        lock_release(&access_lock);
        return OPEN_FILE_ERROR;
    }
    lock_release(&access_lock);                         // retrieve th
    return idx;                                         // return the
}
```

앞선 두 System Call들과 마찬가지로 Pointer Value부터 Validate한다. 이어서, 'access\_lock' (Mutex / Binary Semaphore)에 P Operation을 수행해서 해당 OS Kernel Code에 대한 다른 Process의 접근을 방지한다.

Mutex로 보호가 이루어졌으니 Critical Section으로 들어간다. filesys.c의 파일 오픈 함수를 호출해 '현재 열고자 하는 파일'의 Open File Table (File Structure)을 얻는다. 이어, 현재 다루고 있는 파일의 이름이 '현재 수행 중인 프로그램'의 ELF 파일명과 동일하면 해당 파일에 대해 Write 연산 보호를 수행한다.

그 다음, 현재 Process의 FDT를 순차적으로 조회해, 처음으로 비어 있는 Slot에 OFT를 삽입한다. First-Fit Approach라고 볼 수 있는데, 이는 구현하기 나름인 부분이다. (Linux의 그것과 동일)

이렇게 File Descriptor Table 삽입이 완료되면, open System Call 초입에 걸어놓았던 Mutex Lock을 해제한다. 그 다음, 삽입한 Array Slot의 Index를 반환해 User Program에게 File Descriptor를 알리는 것이다.

이어서 read와 write System Call을 확인해보자. 기본적으로 두 System Call의 동작 구조는 매우 유사하다. 먼저 Pointer Value Validation을 한 후, open과 마찬가지로 Lock을 걸고, 이어서 User Program으로부터 넘겨 받은 File Descriptor Number를 토대로 대응하는 OFT를 Array에서 찾는다. 만약 File Descriptor가 Standard I/O이면 Console에서 입출력을 수행하고, 그렇지 않다면 찾은 OFT를 바탕으로 file.c에 정의된 함수를 호출해 파일 입출력을 수행한다. 수행을 마치면 걸었던 Lock을 풀고 사용

자에게 읽은/쓴 Byte 개수를 반환한다. 두 System Call을 이러한 구조를 공유한다. 따라서 write System Call의 코드만 첨부한다.

```
/* Write routine: writes 'size' bytes of buffer to the file
   pointed by fd, We should guarantee two things here.
   - only one process at a time can access this code.
   - if a file is as same as the running program, then
     deny any write operations on that file */

int
write (int fd, const void *buffer, unsigned size)
{
    struct file *f;
    off_t byte_cnt = 0;

    POINTER_CHECK(buffer);
    lock_acquire(&access_lock);

    if (fd == 1)
    {
        /* STDOUT_FILENO */
        putbuf(buffer, size); // writes as many bytes as possible
        byte_cnt += size;
    }
    else if (fd < 3 || fd >= FD_MAX)
    {
        /* File descriptor error */
        lock_release(&access_lock);
        exit(-1);
    }
    else
    {
        /* Any I/O-possible files */
        f = thread_current()->fd[fd];
        if (f == NULL)
        {
            lock_release(&access_lock);
            exit(-1);
        }
        if (f->deny_write == true) // if this file is running program,
            file_deny_write(f); // then deny write operations on it.

        byte_cnt = file_write(f, buffer, size); // write with 'file_write'
    }
    lock_release(&access_lock);

    return byte_cnt;
}
```

ELF에 대한 Write Operation 방지 코드도 포함되어 있음을 알 수 있다. OFT의 deny\_write Flag는 file\_deny\_write 함수 호출로 인해 Set되는데, 이는 즉, 자기 자신 Process에서 Running Program의 ELF를 Open할 때 Set이 된 Flag가 다른 Process에서 deny\_write Flag Set의 형태로 확인할 수 있다는 것이다. 즉, 복수의 Process가 소

```
/* Close routine: simply calls 'file_close' function. */
void
close(int fd)
{
    struct file *f;

    if (fd < 3 || fd >= FD_MAX) exit(-1);
    f = thread_current()->fd[fd];
    if (f == NULL) exit(-1);

    thread_current()->fd[fd] = NULL; // record as NULL
    file_close(f); // This func performs not only the closing-file routine,
                   // but also the 'file_allow_write' for running programs
```

통하는 것이다.

좌측은 close System Call이다.  
close, tell, seek, filesize System Call은 모두 동일한 구조를 가지  
 고 있는데, 이들은 처음에 User Program으로부터 넘겨 받은 File

Descriptor가 올바른지 체크하고, 해당 Descriptor가 가리키는 File Structure를 뽑아낸 다음, 해당 Open File Table이 NULL이 아닌지 체크한다. 예외 체크를 모두 정상적으로 통과하면 각 System Call에 대응하는 File System Function을 호출해 Main Job을 위임한다. 이때, close System Call에선, 함수 호출 직전에 File Structure가 들어 있던 Array Slot을 NULL로 바꾸어 비우는 작업이 필요하다.

이처럼 File-Related System Call은 모두 자신의 Main Job을 pintOS 제공 Base File System에게 맡기기 때문에 위에서 설명한 것처럼 Synchronization, Denying Write to ELF, Error Handling만 적절히 구성하면 구현이 완성된다.

이어서 바로 'Memory Leakage 방지 Routine'을 확인하자. 아래는 process.c 파일의 Process Execution 담당 함수 process\_execute (exec System Call)의 일부이다.

```
/* Create a new thread to execute FILE_NAME. */
tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);

/* Prevent the situation that the running process abnormally finishes 'exec' earlier than the load routine */
sema_down(&(thread_current()->load_lock));

/* If an error occurred in the loading phase, then free the allocated page of that newly created thread */
if (tid == TID_ERROR)
    palloc_free_page(fn_copy);

/* Reap every child that exited abnormally while loading or while start_process routine */
c_list = &(thread_current()->child_list);
for (ITERATE_C_LIST) // iterate the list of child processes
{
    entry = EACH_CHILD; // for each entry, if exited abnormally
    if (entry->exit_status == -1) // then, reap that child here!
        return process_wait(tid); // tid is a thread-ID of child
}

return tid;
```

thread\_create 함수로 Thread Creation을 수행한다. 이어, 새롭게 생성된 Thread는 start\_process Routine을 수행하고, 그곳에선 Load가 수행된다. start\_process 수행 시점부터는 Child Process의 Flow라고 생각하자.

같은 시간에 위의 Code는 당연 Parent Process가 이어서 수행하는데, Thread Creation 직후 Load Semaphore에 대한 P Operation을 통해 Parent가 스스로 잠든다. Child Process가 Load되고 수행하는 그 순간에 Parent가 먼저 수행을 마치는 Case를 방지하기 위함이다.

다시 Child Process로 돌아가자. Child Process는 Load를 마치고 아래의 코드를 수행한다. 일단 Load Routine의 성공/실패 여부는 고려하지 않는다.

```

success = load (file_name, &if_.eip, &if_.esp);

/* After the load routine ends successfully, wake
   up the parent process of current(child) process */
sema_up(&((thread_current()->parent)->load_lock));

/* If load failed, quit and exit with status -1.
   Reaping will be made in the process_execute routine */
palloc_free_page (file_name);
if (!success)
    exit(-1);

/* Start the user process by simulating a return from an
   interrupt, implemented by intr_exit (in
   threads/intr-stubs.S). Because intr_exit takes all of its
   arguments on the stack in the form of a `struct intr_frame',
   we just point the stack pointer (%esp) to our stack frame
   and jump to it. */
asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
NOT_REACHED ();

```

Load Routine을 마친 Child Process는 곧바로 Load Semaphore에 대해 Q 연산을 수행해 잠자고 있던, 다시 말해 Child의 Load를 기다리고 있던 Parent Process를 다시 깨운다. 그 다음, 앞선 Load 과정이 성공적으로 끝난 경우 자신의 업무를 수행하고, Load 과정이 임의의 원인으로 인해 실패하였다면 -1 Exit Status로 종료한다.

다시 Parent Process를 보자.

```

/* Reap every child that exited abnormally while
   loading or while start_process routine */
c_list = &(thread_current()->child_list);
for (ITERATE_C_LIST)           // iterate the list of child processes
{
    entry = EACH_CHILD;        // for each entry, if exited abnormally
    if (entry->exit_status == -1) // then, reap that child here!
        return process_wait(tid); // tid is a thread-ID of child
}

```

자신이 fork한 (복수의) Child Process가 비정상 종료한 경우를 대비해 Busy Waiting을 실시한다. 매 exec System Call마다 반복문을 수행해 Abnormally exited Children (load나 start\_process 수행 과정에서 비정상적으로 종료된 Process들)을 확인하고, 존재한다면 Reaping한다. 앞서 소개했던 C Macro인 ITERATE\_C\_LIST와 EACH\_CHILD로 가독성을 높였다. 이곳에서 (거의) 모든 비정상 종료 프로세스가 처리될 것이다.

이때, 한 가지 주요 'make check 고려 사항'이 있다. 상기한 설명만으로도 거의 모든 Case가 Cover될 것 같지만, 막상 make check를 수행하면 그렇지 않다. 그 이유는 Page Fault Handler를 수정하지 않아서이다. Load나 start\_process 과정, 또는 그 외의 수행 과정에서 특정 Signal이나 여타 다른 요소로 인해 비정상적으로 종료된 Process는 Termination 이전 잔존 코드 수행 도중 (메모리 맵핑도 마찬가지로 정상적으로 이

뤄지지 못한 상태에서 특정 메모리를 접근해 발생하는) Page Fault Error를 맞이할 수 있다. 이때, 이전 Phase에서 만들어놓은 Page Fault Handle는 모든 Case에 대해 "exit(-1);"을 수행하지 않기 때문에 '불특정 Exit Status로 종료하는 Process'를 만들어 낼 수 있다. 따라서 우리는 exception.c의 Page Fault 함수도 수정해야 한다.

```
/* If a page fault occurs, then the faulting process
   must exit with an exit status -1 */
if (is_kernel_vaddr(fault_addr) == true ||
    user == false || not_present == true) exit(-1);
```

이 코드를 추가하면 된다. **not\_present Flag**는 'Not-present Page'로 인해 발생하는 **Fault Exception**을 알리는데, 이를 이용해 Load 과정이 순탄치 않은 Process들이 -1 Exit Status와 함께 종료하도록 만들어주는 것이다. 이렇게 하면 앞서 설정한 Busy Waiting에 비정상 Children이 모두 걸리게 될 것이다.

한편, Process Exit 과정도 수정해야 한다. 바로 위까지 pintOS를 개발하면 모든 Test Case를 통과한다. 딱 하나, 'multi-oom'만 제외하고 말이다. tests 디렉토리에서 multi-oom.c 파일을 확인해보면, 이는 Memory Leakage를 야기하는 테스터인데, 이것을 통과하지 못한다는 것은 아직 하나 더 처리가 남았다는 것이다. 그것은 바로 exit System Call에 있다.

(지난 Phase에서 구현한) exit System Call은 얼핏 보면 문제가 없다. 그러나, 오랜 시간 고민하고 디버깅해보면 다음과 같은 생각이 문득 들게 된다.

**"만약, Current(Parent) Process가 Child Process를 생성하고 나서, wait System Call을 따로 부르지 않는다면 그 Child들의 Reaping은 어떻게 이뤄지는가?"**

이에 대한 해답은 우리는 이미 알고 있다. Linux System에선 init Process가 이를 입양(Adopt)해 Reaping한다고 배운 바 있다. 그리고 가만 생각해보면 pintOS에선 이 작업이 구현되지 않았다! 따라서 우리는 이 기능도 구현해야 한다. 하지만 굳이 init Process를 따로 둘 필요는 없다. init Process는 Linux System을 실행시키는 최초의 프로세스인데, 이는 실제로 Linux 상에서 각 Version마다 구현 방식이 다르다고 한다. pintOS에선 이를 어떻게 관리하는지가 Manual 상에는 언급되어 있지 않은데, 그냥 어렵지 않게 Kernel Code 그 자체가 init으로서의 역할을 수행하면 된다.

따라서 우리는 Kernel Code 그 자체, 그 중에서도 Process가 종료하는 시점에서 exit System Call을 통해 자신의 (미리 추적해놓은) 잔존 Child를 모조리 Reaping하는 식으로 구현할 수 있다. 이는 "Parent가 exit을 거치지 않고 종료되는 상황에선 어떡 할 것인가"라는 반문을 맞이할 수 있는 구현이지만, 본 Phase의 make check에선 이를 따로 확인하진 않으므로 무리가 없다.

따라서 아래와 같이 exit System Call을 수정하였다.

```
/* Exit routine: stores an exit status of running thread, with
   closing all the not-closed files in the FDT. After that, reaps
   every child process that this current process forked and calls
   'thread_exit' to do the main thread-clearing job */
void
exit (int status)
{
    struct file **f_list;
    struct list *c_list;
    struct list_elem *iter;
    int i;

    printf("%s: exit(%d)\n", thread_name(), status);
    thread_current()->exit_status = status; // record the exit status

    /* Close all the not-closed files in the FDT */
    f_list = &(thread_current()->fd);
    for (i = 0; i < FD_MAX; i++)
    {
        if (f_list[i] != NULL)
            close(i);
    }

    /* Reap every child process that this(parent) process forked */
    c_list = &(thread_current()->child_list);
    for (ITERATE_C_LIST) // iterate the list of child processes
        wait(EACH_CHILD->tid); // reap each entry(child) with 'wait'

    thread_exit();
}
```

Thread의 Exit Status를 지정한 후, 현재 종료하려는 Process의 File Descriptor Table  
을 비운다. 그 다음 잔존 Child Process에 대해 하나 하나 일일이 Busy Waiting하여  
모조리 Reaping한다.

이렇게 해서 User Program (2)의 모든 요구사항을 만족시킬 수 있다. 여기까지 설명한 내용을 토대로 make check를 돌리면 아래와 같다.

## C. 시험 및 평가 내용

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/sc-bad-sp pass tests/userprog/sr-bad-arg pass tests/userprog/sc-boundary pass tests/userprog/sc-boundary-2 pass tests/userprog/sc-boundary-3 pass tests/userprog/halt pass tests/userprog/exit pass tests/userprog/create-normal pass tests/userprog/create-empty pass tests/userprog/create-null pass tests/userprog/create-bad-ptr pass tests/userprog/create-long pass tests/userprog/create-exists pass tests/userprog/create-bound pass tests/userprog/open-normal pass tests/userprog/open-missing pass tests/userprog/open-boundary	pass tests/userprog/open-empty pass tests/userprog/open-null pass tests/userprog/open-bad-ptr pass tests/userprog/open-twice 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-bound pass tests/userprog/exec-bound-2 pass tests/userprog/exec-bound-3 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-recuse 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 80 tests passed.  
make[1]: Leaving directory '/sogang/under/cse20171643/pintos/src/userprog/build'
```

80개의 Test Case를 모두 통과하였다. 이렇게 해서 내 pintOS는 User Program을 모두 정상 수행할 수 있게 되었다. 이제는 이 pintOS가 임의의 User Program을 모두 돌릴 수 있을지 궁금하다. 분명 Test Case에서 체크하지 않는 맹점이 있을 것이라 생각하는데, 예를 들어, 현재 이 pintOS에 System Programming에서 만들었던 Shell Program이나 Concurrent Server Program을 돌리면 과연 예러 없이 잘 돌아갈 수 있을지 상당히 궁금하다. 이어지는 Project3, Project4 등을 열심히 수행해 추후 '그럴듯한 OS'가 완성된다면, 꼭 한 번 그러한 실험을 해보고 싶다.