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

**담당 교수 / 분반** : 박성용 / 01

**이름 / 학번** : 최준수 / 20151619

**개발 기간** : 2020/10/31 ~ 2020/11/15

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술
* Project1이 완료된 상태의 pintos에는 user program의 argument들이 빠짐없이 전달되도록 하는 argument passing 기능과 몇가지 기본적인 system call들이 구현되어 있다. Project2에서는 여기에 더불어 파일 시스템과 관련된 system call을 추가하고 여러 프로세스들이 공유된 데이터에 대해 동시에 접근하지 못하도록 동기화(synchronization) 기법을 적용해주어야 한다. 보다 효율적인 동기화 기법을 구현하기 위해 이번 프로젝트에서는 semaphore 방법을 사용한다. 또한 실행중인 프로그램의 executable file이 지워지지 않도록 Denying Writes to Executables 기능을 구현해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목을 구현해야 하는 이유, 혹은 구현 시 기대되는 결과를 간략히 서술
  + 1. File Descriptor

File system과 관련된 system call을 구현하기 위해서는 각 프로세스마다 파일에 대한 정보를 리스트로 만들어 추가하여야 한다. 따라서 다음과 같이 파일 정보를 저장하는 구조체를 만들었다.  
struct fd\_info {...}

Thread 별로 최대 128개의 파일을 열수 있다고 FAQ에 명시되어 있으며, Thread가 종료될 때 close되지 않은 file descriptor들을 모두 닫아주어야한다.

* + 1. (이번 프로젝트에서 구현해야 하는) System Calls

이번 프로젝트에서 File system과 관련된 system call을 구현할 때 추가해줘야 할 함수는 다음과 같다.  
(userprog/syscall.c) bool syscall\_create() // filename을 입력 받아 file 생성  
(userprog/syscall.c) bool syscall\_remove() // filename을 입력 받아 해당 파일을 삭제  
(userprog/syscall.c) int syscall\_open() // filename을 입력 받아 해당 파일을 엶  
(userprog/syscall.c) void syscall\_close() // file descriptor를 입력 받아 해당 파일을 닫음  
(userprog/syscall.c) int syscall\_read() // file descriptor를 입력 받아 size만큼 파일을 읽은 뒤 buffer에 저장  
(userprog/syscall.c) int syscall\_write() // file descriptor를 입력 받아 size만큼 buffer의 내용을 파일에 씀  
(useeprog/syscall.c) int syscall\_filesize() // file descriptor를 입력 받아 해당 파일 크기를 return함  
(userprog/syscall.c) void syscall\_seek() // file descriptor와 position을 입력 받아 해당 파일의 위치를 찾음  
(userprog/syscall.c) unsigned syscall\_tell() // file descriptor를 입력 받아 현재 위치의 다음 byte를 return함

3. Synchronization in Filesystem

Semaphore로 구현된 synchronization이 적용되는 함수들은 다음과 같다.

(userprog/process.c) int process\_wait () //자식이 끝날 때까지 기다린다.

(threads/thread.c) tid\_t thread\_create() //동기화에 필요한sema\_init 수행

(threads/thread.c) void thread\_exit() //동기화에 필요한 sema\_up 수행

(userprog/process.c) void start\_process() //exec에 필요한 sema\_up 수행

(userprog/syscall.c) tid\_t syscall\_exec() //load 될 때까지 기다린다.

(threads/thread.h) struct process\_info{...} //sema, load\_sema 변수 추가

추가로 구현되는 file system call사이에 동기화가 필요하며, 수정되어야 할 부분은 다음과 같다.

(userprog/syscall.c) void syscall\_handler() //file system call 사이에 동기화 수행

(userprog/syscall.c) void syscall\_init() //동기화에 필요한 sema\_init 수행

* **Denying writes to Executables**

실행중인 파일, 즉 아직 끝나지 않은 프로세스의 executable 한 파일에 쓰기를 막아야 하며 이때 수정되어야 할 부분은 다음과 같다.

(userprog/process.c) bool load() //load 된 파일에 대해 deny\_write

(userprog/process.c) void process\_exit() //종료시 allow\_write

(threads/thread.h) struct thread{ ... } //실행가능한 파일 포인터 저장

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

struct fd\_info {

int fd;

struct file\* fp;

struct list\_elem elem;

}

이라는 구조체를 (threads/thread.h)에 선언하고 기존에 있던 struct process\_info에 fd\_info를 저장하기 위한 struct list file\_list를 선언하였다. FAQ에 file을 최대 128개까지 열 수 있다고 되어있지만 많은 수의 파일을 열고 관리하기 위해 pintos의 list 자료구조를 사용하였다.

2. System Calls: 구현할 각 system call에 대해 간략히 서술 (하나의 system call 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

**create** : 문자열 형식의 file 이름을 입력 받아 filesys\_create 함수를 호출해 파일을 생성한다.  
**remove** : 문자열 형식의 file 이름을 입력 받아 filesys\_remove 함수를 호출해 파일을 제거한다.  
**open** : 문자열 형식의 file 이름을 입력 받아 filesys\_open 함수를 호출하여 return 값인 파일 포인터를 구한다. 그리고 그 포인터를 fd list에 삽입한다..  
**close** : 입력 받은 file descriptor에 대하여 해당 fd가 저장되어 있는 노드 포인터를 구한다. 그 다음 파일 리스트에서 해당 노드를 제거 한 후 file\_close 함수를 호출하여 파일을 닫는다.   
**read** : 입력 받은 fd가 0인 경우, 즉 standard input인 경우 input\_getc() 함수를 통해 읽어 들인 글자를 buffer에 저장한다. 파일을 읽는 경우에는 주어진 fd로 파일 포인터를 찾고 file\_read 함수를 호출하여 파일을 읽는다.  
**write** : 입력 받은 fd가 1인 경우, 즉 standard output인 경우에 대하여 putbuf 함수를 통해 buffer에 저장된 내용을 출력한다. 파일에 쓰는 경우 주어진 fd로 파일 포인터를 찾고 file\_write 함수를 호출해 파일을 쓴다.  
**filesize** : 입력 받은 fd에 대하여 파일 포인터를 찾고 file\_length 함수를 호출하여 size를 return한다.  
**seek** : 입력 받은 fd에 대하여 파일 포인터를 찾고 file\_seek 함수를 호출하여 position이라는 위치를 찾는다.  
**tell** : 입력 받은 fd에 대하여 파일 포인터를 찾고 file\_tell함수를 호출하여 현재 위치의 다음 byte를 return한다.

3. Synchronization in Filesystem: Lock, Semaphore를 어떻게 이용할 수 있는지 각각에 대해 설명 (다른 방법을 서술해도 되지만 lock과 semaphore는 반드시 포함해야 함)

semaphore는 여러 프로세스가 공통된 메모리에 동시에 access 하는 상황인 race condition을 막기위한 동기화 방법이다. 이에 대한 메커니즘은 다음과 같다.

여러 프로세스가 동시에 공유하는 sema라는 변수가 있고 이의 초기화 값이 n이라고 하자. 한 프로세스가 공유 메모리에 접근할 때, sema\_down 함수를 통해 접근하며 sema라는 변수가 1 내려간다. 공유 메모리에서 빠져나올 때는 sema\_up을 호출하며 변수가 1 올라간다. 만약 sema변수가 0이 되었을 때 sema\_down을 호출한다면 해당 프로세스를 block 시키고 waiting queue에 넣어둔다. 어떤 프로세스가 sema\_up을 통하여 sema 변수를 1 올렸을 때, waiting queue에 들어있는 프로세스를 unblock 시켜 sema\_down을 진행한다.

이러한 메커니즘을 통해 공유메모리상에서 동기화를 진행할 수 있다. 해당 프로젝트에서는 sema 변수는 1로 초기화 하여 binary semaphore를 이용하도록 한다. 이렇게 구현하면 critical section에 한번에 하나의 프로세스만 접근 가능하다.

pintos의 자료구조인 struct lock은

struct lock {

struct thread \*holder;

struct semaphore semaphore;

}

로 선언되어있으며 위에서 설명한 semaphore에 holder라는 포인터가 추가된 것을 알 수 있다. lock은 semaphore와 달리 lock을 acquire한 thread만이 release할 수 있다. semaphore는 thread 1에서 sema\_down을 하더라도 thread 2에서 sema\_up을 할 수 있지만 lock은 이러한 형태가 불가능하다. 따라서 여러 thread에서 공유하여 synchronization을 수행해야되는 경우는 lock이 아닌 semaphore를 사용하여야한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10/31~11/5: file descriptor & System call 개발

11/6~11/12: Synchronization 추가

11/13~11/15: 보고서 작성 및 제출

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
* **System Call**

**(userprog/syscall.c) bool syscall\_create()** : 문자열 형식의 file 이름을 입력 받고 우선적으로 null인지 체크한 뒤 null인 경우 오류처리를 한다. 그 다음 filesys\_create 함수를 호출해 파일을 생성한다.  
**(userprog/syscall.c) bool syscall\_remove()** : 문자열 형식의 file 이름을 입력 받고 우선적으로 null인지 체크한 뒤 null인 경우 오류처리를 한다. 그 다음 filesys\_remove 함수를 호출해 파일을 제거한다.  
**(userprog/syscall.c) int syscall\_open()** : 문자열 형식의 file 이름을 입력 받고 filesys\_open 함수를 호출하여 return 값인 파일 포인터를 구한다. 그 다음 해당 파일 포인터에 대해 노드를 할당해 file descriptor를 부여하고 프로세스의 파일 리스트에 삽입한다.  
**(userprog/syscall.c) void syscall\_close()** : 입력 받은 file descriptor에 대하여 해당 fd가 저장되어 있는 노드 포인터를 구한다. 그 다음 파일 리스트에서 해당 노드를 제거 한 후 file\_close 함수를 호출하여 파일을 닫는다. 마지막으로 리스트에서 제거한 노드를 free로 해제한다.   
**(userprog/syscall.c) int syscall\_read()** : 입력 받은 fd가 0인 경우, 즉 standard input인 경우 input\_getc() 함수를 통해 읽어 들인 글자를 buffer에 저장한다. 파일을 읽는 경우에는 주어진 fd로 파일 포인터를 찾고 file\_read 함수를 호출하여 파일을 읽는다.  
**(userprog/syscall.c) int syscall\_write()** : 입력 받은 fd가 1인 경우, 즉 standard output인 경우에 대하여 putbuf 함수를 통해 buffer에 저장된 내용을 출력한다. 파일에 쓰는 경우 주어진 fd로 파일 포인터를 찾고 file\_write 함수를 호출해 파일을 쓴다.  
**(useeprog/syscall.c) int syscall\_filesize()** : 입력 받은 fd에 대하여 파일 포인터를 찾고 file\_length 함수를 호출하여 size를 return한다.  
**(userprog/syscall.c) void syscall\_seek()** : 입력 받은 fd에 대하여 파일 포인터를 찾고 file\_seek 함수를 호출하여 position이라는 위치를 찾는다.  
**(userprog/syscall.c) unsigned syscall\_tell()** : 입력 받은 fd에 대하여 파일 포인터를 찾고 file\_tell함수를 호출하여 현재 위치의 다음 byte를 return한다.

* **Synchronization**wait와 exec에 필요한 동기화 기법으로 semaphore를 이용하려면 기본적으로 sema 변수가 필요하다. 따라서 (threads/thread.h) struct process\_info{...} 구조체에 sema, load\_sema 변수 추가하여야 한다. 그 후 해당 sema 변수를 초기화 하여야 하는데 이를 위하여 (threads/thread.c) tid\_t thread\_create() 함수에 초기화 기능을 추가시킨다.

wait는 자식 프로세스가 끝날 때까지 기다리는 역할을 해야 하므로 (userprog/process.c) int process\_wait ()함수는 기존 busy wait으로 구현되어 있던 부분을 sema\_down() 함수로 대체한다. 그러면 자식 프로세스가 sema\_up()을 하기 전까지 block 된다. 자식 프로세스는 자신이 죽었다는 사실을 sema\_up()을 통해 부모 프로세스에게 알려야 하므로 (threads/thread.c) void thread\_exit() 함수에 sema\_up() 을 추가시킨다.

exec은 실행시키는 파일이 load가 끝날 때까지 동기화 되어야 한다. 따라서 (userprog/syscall.c) tid\_t syscall\_exec() 함수는 기존 busy wait으로 구현되어 있었으나 이를 wait과 마찬가지로 sema\_down()으로 대체한다. 그러면 load 하는 프로세스가 sema\_up()을 하기 전까지 block 된다. load가 끝나면 sema\_up()을 통해 기다리는 프로세스에게 알려야 하므로 load가 끝나는 부분인 (userprog/process.c) void start\_process() 부분에 sema\_up()을 추가시킨다.

추가로 구현되는 file system call 동기화 역시 semaphore을 이용한다. 따라서(userprog/syscall.c)의 전역변수로 filesys\_mutex를 추가하며 이는 (userprog/syscall.c) void syscall\_init() 함수에서 초기화를 수행한다. file system call 간의 동기화를 위해 (userprog/syscall.c) void syscall\_handler()에서 file system call 마다 앞 뒤로 sema\_down() 과 sema\_up()을 추가하여 동기화를 구현한다.

* **Denying writes to Executables**

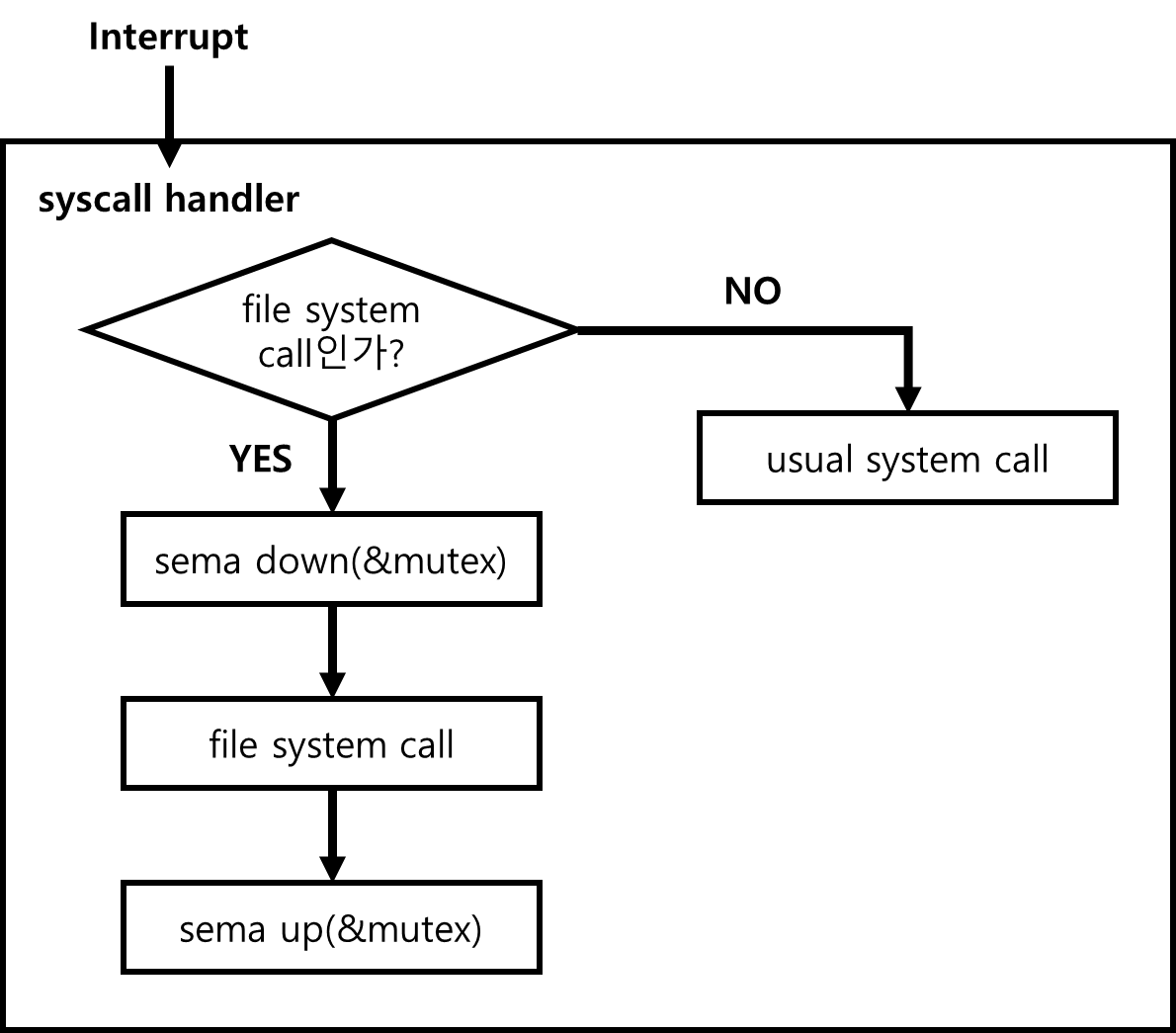
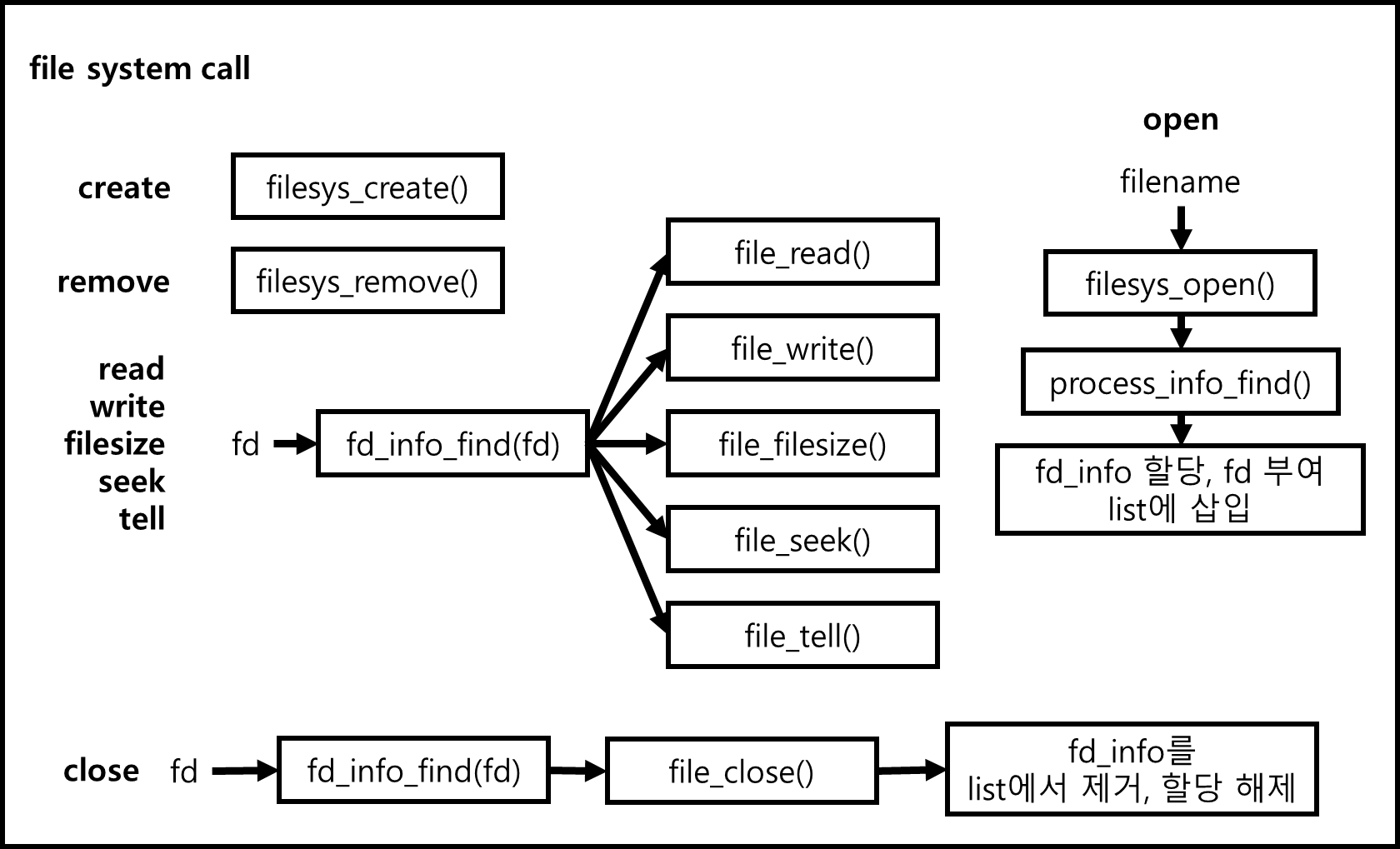
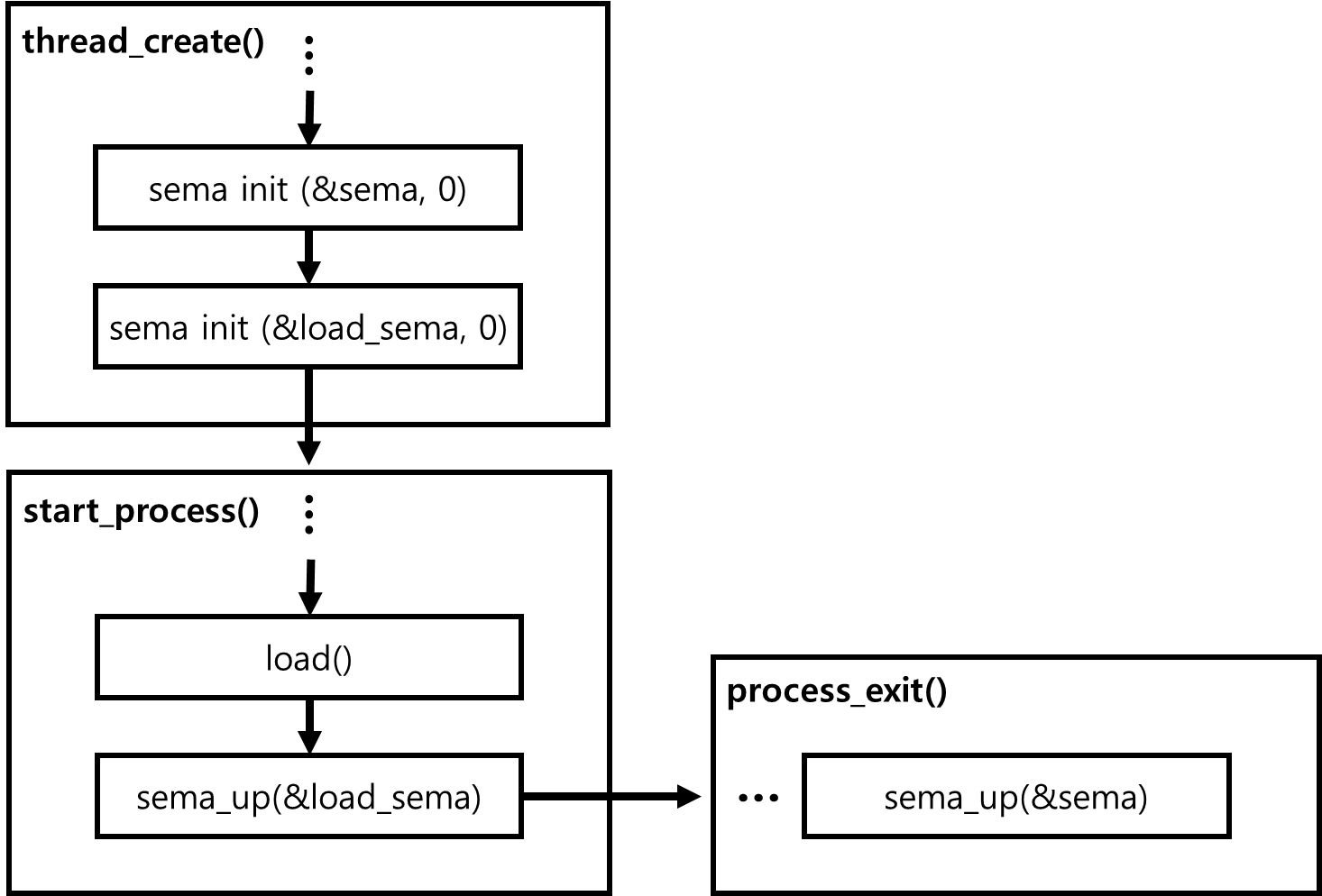
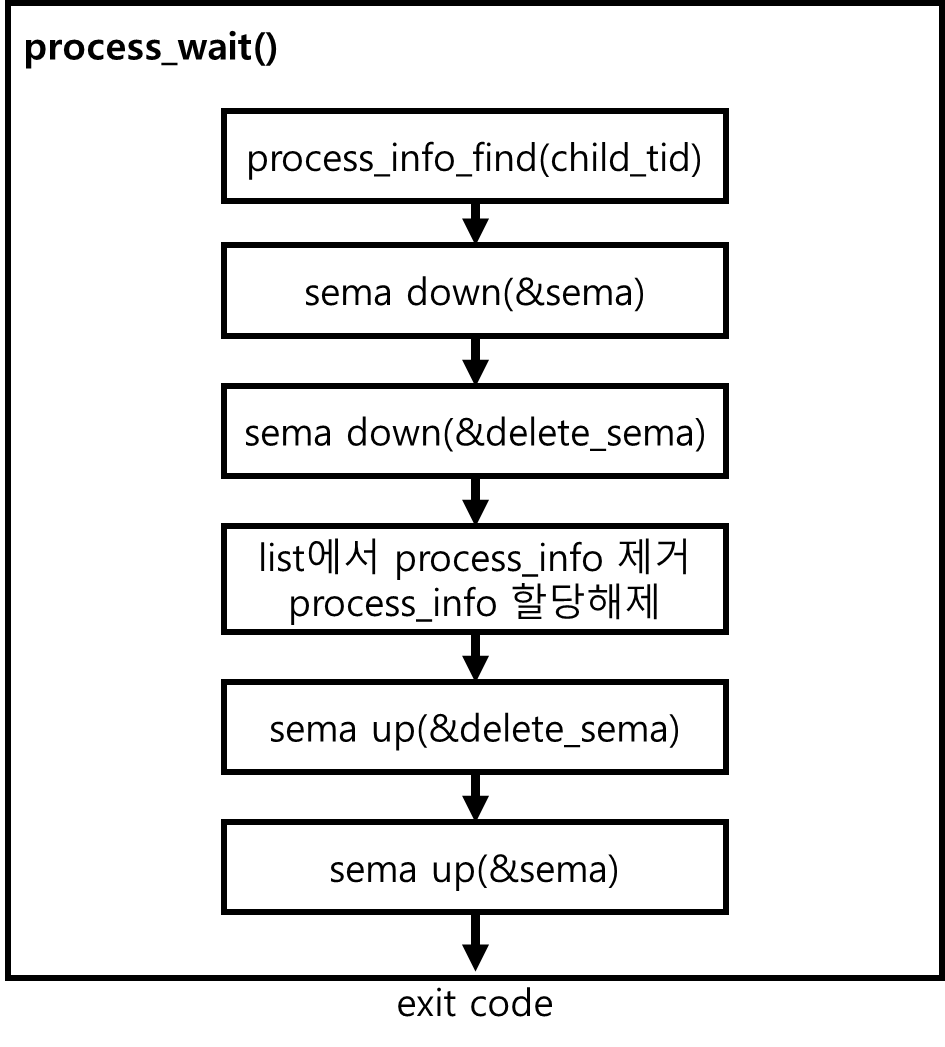
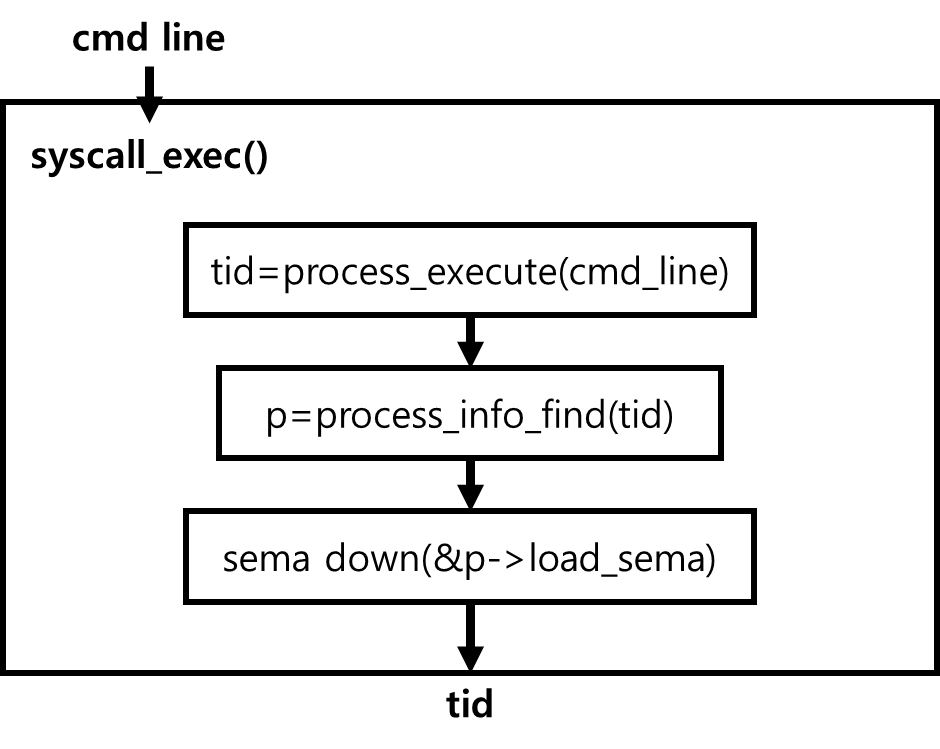
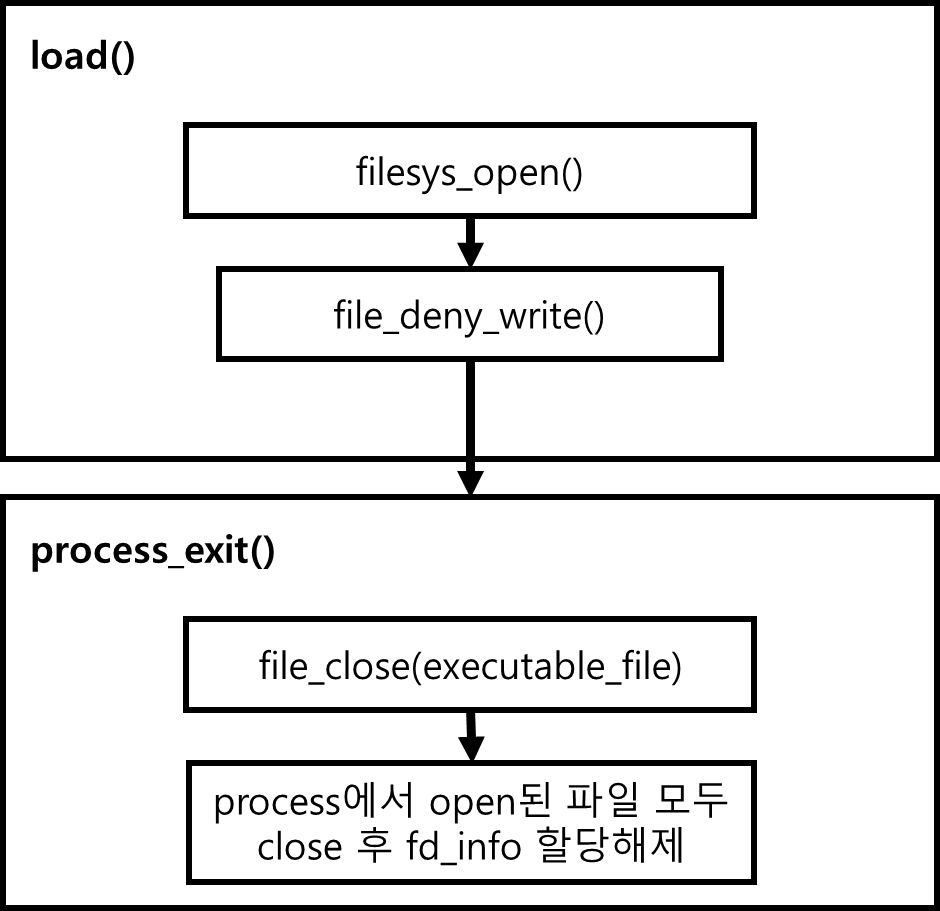
Executable file open은 load 함수 수행 중에 호출된다. 따라서 (userprog/process.c) bool load() 에서 file\_open() 수행후에 해당 파일에 대해 file\_deny\_write()를 수행한다. 이를 통해 해당 파일의 삭제를 막을 수 있다. 이는 해당 파일을 실행하는 프로세스가 종료될 때까지 유지되어야 하며 해당 프로세스가 종료되면 file\_allow\_write() 수행해야한다. 그러므로 (userprog/process.c) void process\_exit() 함수에서 file\_close() 함수를 통하여 file allow write를 수행한다. 이 때 해당 파일의 포인터가 필요한데 이는 (threads/thread.h) struct thread{ ... } 구조체에 관련변수를 추가하고 load 수행시점에 저장하도록 구현한다.

* **Memory leak prevention**

process가 생성 될 때 마다 process\_info 구조체가 동적 할당되기 때문에 부모가 wait을 종료시에 더이상 필요성이 없으므로 할당 해제해줘야 한다. 이를 위해 (userprog/process.c) int process\_wait (tid\_t child\_tid) 함수를 process\_info\_list에서 해당 process\_info를 제거하고, 관련 메모리 할당 해제하도록 수정해야한다.

file\_open이 수행될 때마다 fd\_info 구조체가 동적 할당되기 때문에 close system call을 호출 시 해당 fd\_info를 할당해제 해야 한다. 따라서(userprog/syscall.c) void syscall\_close() 함수에서 해당 fd\_info를 할당 해제하는 부분을 추가하고 file\_close를 통해 file 구조체를 할당 해제하도록 한다. 프로세스가 open 되어 있는 file 을 닫지 않고 종료될 수가 있는데 이 때 열려 있는 파일을 모두 닫고, 연결된 fd\_info 구조체를 모두 할당 해제해야 한다. 따라서(userprog/process.c) void process\_exit() 함수에서 프로세스에 open 되어있는 file 들을 모두 확인하여 각각을 close한 후 fd\_info를 할당해제 해주는 부분을 추가한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
* **file system call의 동기화 구현**
* **file system call 구현**
* ****
* **process의 semaphore 관리 방식**
* ****
* **process\_wait semaphore 구현 및 메모리 누수 방지**
* ****
* **syscall\_exec semaphore 구현**
* ****
* **denying writes to executables 구현 및 메모리 누수 방지**
* ****
  1. **제작 내용**
* II. B. 개발 내용의 각 3가지 항목에 대하여 실직적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
* 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
* **File System Call**

각 프로세스마다 파일에 대한 정보를 리스트로 만들어 추가하여야 한다. 따라서 다음과 같이 파일 정보를 저장하는 구조체를 만들었다.  
struct fd\_info  
{  
 int fd;  
 struct file\* fp;  
 struct list\_elem elem;  
};  
fd는 file descriptor를 의미하며 fp는 파일 포인터이다. elem은 리스트의 노드들을 연결해주는 포인터다.

sytem call관련 함수들을 구현하기에 앞서 이번 프로젝트에서는 프로세스들 간의 동기화가 이루어져야 한다. 따라서 각각의 system call 함수들이 호출되기 전후로 sema\_down, sema\_up 함수를 호출하여 동기화가 이루어지도록 하였다.

File system 관련 system call 함수들 중 일부는 fd를 입력 받아 해당 fd를 가지는 파일 포인터를 찾는 과정이 있다. 중복된 코드의 작성을 방지하기 위해 이 과정을 함수로 구현하였는데 내용은 다음과 같다.  
struct fd\_info \*  
fd\_info\_find(int fd) {  
 struct process\_info \*pinfo = process\_info\_find(thread\_current()->tid);  
  
 if(list\_empty(&pinfo->file\_list))  
 return NULL;  
  
 struct list\_elem \*p = list\_begin(&pinfo->file\_list);  
  
 for (; p!=list\_end(&pinfo->file\_list); p=list\_next(p)) {  
 struct fd\_info \*finfo= list\_entry(p,struct fd\_info,elem);  
 if (fd==finfo->fd)  
 return finfo;  
 }  
 return NULL;  
}

다음은 system call 함수들의 구체적인 제작 내용이다.

(userprog/syscall.c) bool syscall\_create(const char \*file, unsigned initial\_size) : 문자열 형식의 file 이름을 입력 받고 우선적으로 null인지 체크한 뒤 null인 경우 syscall\_eixt(-1)을 호출하여 user program을 종료 시킨다. 그 다음 filesys\_create 함수를 호출하여 크기가 initial\_size byte인 파일을 생성한다.  
(userprog/syscall.c) bool syscall\_remove(const char \*file) : 문자열 형식의 file 이름을 입력 받고 우선적으로 null인지 체크한 뒤 null인 경우 syscall\_eixt(-1)을 호출하여 user program을 종료 시킨다. 그 다음 filesys\_remove 함수를 호출해 파일을 제거한다. filesys\_remove 함수는 root directory에서 파일을 찾아 제거하는 함수이다.  
(userprog/syscall.c) int syscall\_open (const char \*filename) : 문자열 형식의 file 이름을 입력 받고 우선적으로 null인지 체크한 뒤 null인 경우 오류처리를 한다. 그 다음 filesys\_open 함수를 호출하여 return 값인 파일 포인터를 저장하고 만약 이 포인터가 null일 경우 한번 더 오류처리를 한다. 파일 포인터가 유효한 경우 파일 정보를 저장하는 노드를 할당 받아 file descriptor와 함께 저장한다. fd는 static int로 선언되어서 각기 다른 스레드들이 파일을 열때마다 1씩 증가하여 서로 다른 fd를 부여 받는다. 또한 fd는 2부터 초기화 되는데 이는 0, 1이 각각 standard input, standard output이기 때문이다. 파일 포인터와 fd가 저장된 노드는 해당 스레드의 파일 리스트에 삽입된다.  
(userprog/syscall.c) void syscall\_close(int fd) : 입력 받은 file descriptor에 대하여 해당 fd가 저장되어 있는 노드 포인터를 fd\_info\_find 함수를 호출하여 구한다. 만약 해당 fd를 갖는 노드가 없을 경우 바로 종료한다. 노드 포인터를 구한 경우에는 일단 파일 리스트에서 해당 노드를 제거 한 후 file\_close 함수를 호출하여 파일을 닫는다. 마지막으로 리스트에서 제거한 노드를 free로 해제한다.   
(userprog/syscall.c) int syscall\_read(int fd, void \*buffer, unsigned size) : 입력으로 fd, buffer, size를 받게 되면 우선적으로 buffer의 주소가 유효한지 확인한다. 그 다음 fd가 0인 경우, 즉 standard input인 경우 size만큼 input\_getc() 함수를 통해 한 글자 씩 buffer에 저장한다. 파일을 읽는 경우에는 fd\_info\_find 함수를 호출하여 주어진 fd로 파일 포인터를 찾고 file\_read 함수를 호출하여 size 크기만큼 파일을 읽는다.  
(userprog/syscall.c) int syscall\_write(int fd, const void \*buffer, unsigned size) : syscall\_read와 마찬가지로 먼저 buffer의 주소가 유효한지 확인하고 standard output인 경우에 대하여 putbuf 함수를 통해 buffer에 저장된 내용을 출력한다. 파일에 쓰는 경우 fd\_info\_find 함수를 호출하여 주어진 fd로 파일 포인터를 찾고 file\_write 함수를 호출해 size 크기만큼 파일을 쓴다.  
(useeprog/syscall.c) int syscall\_filesize(int fd) : 입력 받은 fd에 대하여 fd\_info\_find 함수를 호출하여 파일 포인터를 찾고 file\_length 함수를 호출하여 size를 return한다.  
(userprog/syscall.c) void syscall\_seek(int fd, unsigned position) : 입력 받은 fd에 대하여 fd\_info\_find 함수를 호출하여 파일 포인터를 찾고 찾고자 하는 위치인 position을 파일 포인터와 함께 file\_seek 함수의 파라미터로 보내준다.  
(userprog/syscall.c) unsigned syscall\_tell(int fd) : 입력 받은 fd에 대하여 fd\_info\_find 함수를 호출하여 파일 포인터를 찾고 file\_tell함수를 호출하여 현재 위치의 다음 byte를 return한다.

* **Synchronization**

기존의 busy wait 방법으로 구현되어 있던 부분은 다음과 같다.

(userprog/process.c) int process\_wait ()

부모가 자식을 기다리기 위해 자식이 종료되었는지 busy wait으로 확인한다.

(userprog/syscall.c) tid\_t syscall\_exec()

실행하는 파일이 로드 될 때까지 기다리기 위해 로드 정보를 busy wait으로 확인한다

이들을 semaphore로 구현하기 위하여 process info 구조체에 다음과 같은 변수를 추가하였다.

struct process\_info{  
...  
struct semaphore sema;  
struct semaphore load\_sema;  
}.

**process\_wait ()에서의 동기화 구현**

process\_info구조체의 변수인 sema는 자기 자신과 부모만 접근하는 sema이다. 자식 프로세스의 sema는 자식이 생성될 때 바로 down되도록 하고, 만약 부모가 자식을 wait하게 된다면 sema\_down()을 통하여 sema\_up이 될 때까지 block 되어야 한다.

이를 구현하기 위하여 (threads/thread.c) tid\_t thread\_create()에서 자식 프로세스(쓰레드)를 생성하는 동시에 sema\_init(&sema,0) 을 호출하여 자식이 살아있는 동안은 계속 sema가 0이 되도록 한다. 자식이 종료될 때인 (threads/thread.c) void thread\_exit()에서 sema\_up(&sema)를 호출한다.

위와 같이 semaphore를 구성하였기에, 자식을 기다리는 함수 (userprog/process.c) int process\_wait ()을 다음과 같이 구현하였다.

자식이 살아있다는 것은 자식의 sema가 0이라는 뜻이므로 sema\_down()호출해 block되어 기다리게 하였고, 자식이 죽으면 자식의 sema가 1 증가하므로 block되어 있던 부모 프로세스가 unblock되어 이후 부분을 수행할 수 있도록 하였다.

**syscall\_exec() 에서의 동기화 구현**

process\_info의 변수 load\_sema는 syscall\_exec()과 execute되는 프로세스의 load 사이에 동기화를 담당한다. load\_sema는 위의 sema와 마찬가지로 (threads/thread.c) tid\_t thread\_create()에서 프로세스(쓰레드)를 생성하는 동시에 sema\_init(&load\_sema,0)을 수행하여 초기값을 0로 만들었고, load가 종료되는 부분인 (userprog/process.c) void start\_process() 부근해서 호출하도록 하였다. 따라서 (userprog/syscall.c) int syscall\_exec () 에서 process\_execute() 호출 뒤에 sema\_down(&load\_sema)를 호출하여 execute하는 process의 load가 종료될 때까지 block시키고 load가 종료되면 unblock되어 이후 부분을 실행할 수 있게 구현하였다.

**file system call 사이의 동기화 구현**

다른 프로세스(쓰레드)가 system call을 통하여 하나의 파일에 접근한다고 할 때, 이에 대해 동기화가 필요하다. file 관련 시스템 콜은 create, remove, open, close, filesize, read, write, seek, tell 이며 이에 대한 동기화를 semaphore로 구현하였다. 해당 sema의 초기화는 (userprog/syscall.c) void syscall\_init()에서 하도록 구현하였다. system call 호출을 관리하는 부분은 (userprog/syscall.c) static void syscall\_handler () 이므로 file system call 호출 전 후로 sema\_down() sema\_up()을 두어 한번에 하나의 file system call이 호출되게 하였다.

* **Denying writes to Executables**

file\_deny\_write()함수를 이용하면 다른 system call에 의해 해당 file의 inode에 쓰는 것을 막을 수 있다. 따라서 (userprog/syscall.c) bool load() 함수에서 file\_open() 함수로 file open이 성공하는 즉시, file\_deny\_write() 함수를 호출한다. 그러면 file\_allow\_write()함수가 호출되기 전까지 해당 파일에 쓰는 것이 금지된다. 그러나 file\_close() 함수에 file\_allow\_write()가 호출됨을 볼 수 있다. 따라서 기존의 load 함수에서는 맨 마지막에 file\_close()를 수행하였지만, 이는 해당 프로세스가 끝날 때인 process\_exit() 까지 유예되어야 한다. 따라서 thread 구조체에 다음과 같이 변수를 추가한다.

struct thread{  
...  
 struct file \*executable\_file;  
}

해당 변수는 load함수에서 file\_open된 파일 포인터를 저장하며, 이는 process\_exit() 함수에서 file\_close()를 수행하기 위해 참조되는 변수이다.

* **Memory leak prevention**

**process\_info 구조체 메모리 관리**

process\_info 구조체의 목적은 부모가 자식을 wait할때 자식의 종료코드를 얻기 위함이다. 따라서 wait을 마친다면 process\_info는 더이상 필요성이 없으므로 process\_info\_list에서 제거하고 이를 할당 해제해야 한다. 이를 위해 (userprog/process.c) int process\_wait (tid\_t child\_tid) 함수에서 list\_remove()를 통해 process\_info\_list에서 제거한 후 free()를 호출하여 할당 해제한다. 이 때 동기화 부분이 필요한데, 그 이유는 list 내를 순회하는, 즉 process\_info\_find() 함수를 수행하는 프로세스가 있을 경우에 다른 프로세스에서 process\_info\_list를 변경시킨다면 오류가 생길 수 있다. 따라서 process\_info\_find() 함수와 list\_remove() 함수사이에 동기화가 필요하다. 이를 위해 delete\_sema라는 전역 변수를 도입했으며 process\_info\_find()함수와 list\_remove() 함수 전후에 sema\_down()과 sema\_up()을 추가하여 한번에 하나의 함수만 수행되도록 하였다.

**fd\_info 구조체 메모리 관리**

fd\_info 구조체는 close system call을 호출시 process\_info에 있는 file\_list에서 해당 fd\_info를 제거한 후 free()를 통해 할당해제한다. 따라서(userprog/syscall.c) void syscall\_close() 함수에서 list\_remove()를 통해 file\_list에서 제거한 후 free()를 호출하여 할당 해제한다. 이 때 process\_info의 경우처럼 순회하는 함수 fd\_info\_find()가 존재하나, 동기화 부분이 필요하지 않다. 그 이유는 다음과 같다. 프로세스의 file list는 프로세스 본인만 접근 가능하며, file 관련 system call끼리 이미 동기화가 되어 있기 때문이다.

정상적인 종료 외에 프로세스가 open 되어 있는 file 을 닫지 않고 종료되는 상황이 있을 수 있다. 이 때 해당 process에 열려 있는 파일을 모두 닫고, 연결된 fd\_info 구조체를 모두 할당 해제해야 한다. 따라서 (userprog/process.c) void process\_exit() 함수에서 process\_info에 있는 file\_list를 순회하면서 하나하나 list\_remove() 후에 free()를 통해 할당 해제한다. 또한 fd\_info 뿐만 아니라, open 되어 있는 file 구조체 역시 file\_close()함수 통해 할당해제 해야 한다.

* 1. **시험 및 평가 내용**
* make check 수행 결과를 캡처하여 첨부

