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

담당 교수 / 분반 : 박성용 교수님 / 3분반

이름 / 학번 : 문상호 / 20151149

개발 기간 : 2021.11.02 ~ 2021.11.14

1. **개발 목표**

**- file system 관련 system call 추가 구현**

**- pintos에 구현되지 않은 file system 관련 핸들링 구현**

**1. 실행 중인 프로그램의 executable file write 막기**

**2. critical section 보호 처리(syncronization 구현)**

1. **개발 범위 및 내용**
   1. **개발 범위**

|  |  |
| --- | --- |
| **/userprog** | **syscall.c syscall.h process.c** |
| **/threads** | **thread.c thread.h** |

**1. File Descriptor**

파일을 다룰 때 사용되는 개념으로, 프로세스에서 특정 파일에 접근할 때 사용해야하는 값이다. 해당 값을 통해 파일을 지칭할 수 있다.

**2. (이번 프로젝트에서 구현해야 하는) System Calls**

file system을 이용하여 파일 관련 작업을 하기 위해 필요한 system call들이다.

read: 파일로부터 byte를 읽어들인다.

write: 버퍼에서 file로 byte를 작성한다.

create: 새로운 파일을 만든다

remove: 파일을 삭제한다.

open: 파일을 오픈한다.

close: 파일을 닫는다.

filesize: 파일의 byte size를 리턴해준다.

seek: 열린 파일 fd에서 읽거나 쓸 다음 BYTE를 주어준 위치로 변경한다.

tell: 파일의 읽거나 쓸 다음 BYTE의 포지션을 리턴한다.

**3. Synchronization in Filesystem**

파일 시스템에서 synchronization이 구현되지 않는다면, 읽기와 쓰기가 동시에 발생할 수 있어 유저의 의도와는 다른 부정확한 결과가 발생하게 된다. 이를 구현할 시 해당 문제점이 해결된다.

* 1. **개발 내용**

**1. File Descriptor:**

file 구조체 포인터 배열의 자료구조를 사용하기로 하였다. 구현해야 하는 system call들이 보통 fd값을 인자로 받는데, 이 때 해당 fd값을 index로 갖는 배열에 접근한다면 O(1)의 시간 복잡도로 해당 파일에 접근할 수 있게 된다.

**2. System Calls: 구현할 각 system call에 대해 간략히 서술**

- read : fild descriptor가 2보다 크거나 같은 경우의 읽는 작업을 추가로 구현(**file\_read** 호출)한다

- write : fild descriptor가 2보다 크거나 같은 경우의 쓰는 작업을 추가로 구현(**file\_write** 호출)한다

- create : **filesys\_create** 함수를 호출하여 파일을 생성한다.

- remove : **filesys\_remove** 함수를 호출하여 file을 name으로 갖는 파일을 제거해준다.

- open : **filesys\_open**을 호출하여 file을 name으로 갖는 파일을 찾아 열어준다. 이 때 열려는 파일이 현재 켜져 있는 파일과 다를 경우 file\_deny\_write 함수를 호출하여 write deny처리해준다.

- close : **file\_close** 함수를 호출하여 파일을 닫는다.

- filesize : **file\_length**를 호출하여 file descriptor가 가리키는 file의 size를 리턴한다.

- seek : **file\_seek**를 호출하여 file descriptior가 가리키는 file을 찾아 그 위치를 position 으로 옮긴다.

- tell : **file\_tell**을 호출하여 file descriptior가 가리키는 file을 찾아 그 위치를 리턴한다

**3. Synchronization in Filesystem: Lock, Semaphore를 어떻게 이용할 수 있는지 각각에 대해 설명**

- semaphore: sema\_up은 sema value를 1만큼 증가시키고, sema\_down은 sema value를 1만큼 감소시키는데, sema\_down 시 value가 0이라면 해당 실행을 멈추고, sema가 1이 될 때까지 기다리게 된다. parent thread의 execute가 child thread의 start process가 끝나기전에 끝나는 것을 막고 기다려주기 위해서 사용할 것이다.

- lock: lock\_acquire와 lock\_release 함수를 파일 read/write cs섹션의 앞뒤 부분에 각각 호출하여 fd가 가리키는 파일이 다른 스레드간의 읽/쓰기가 돟시에 발생하지 않도록 처리해줄 것이다. 내부적으로 sema\_up과 sema\_down이 호출된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **11/2(화) ~ 11/7(일): 프로젝트 메뉴얼 확인 및 사용되는 코드 분석**
* **11/8(월)~ 11/11(목): file system 관련 system call 구현**
* **11/12(금) ~ 11/13(토): 통과되지 않는 테스트 해결을 위한 디버깅 작업**
* **11/14(일): 디버깅(테스트 2개 미해결) 및 보고서 작성** 
  1. **개발 방법**
  + 수정하거나 추가해야 하는 자료구조

**- synchronization: parent 가 child thread의 load가 끝날 때까지 기다리는 것을 처리해주기 위해 load\_semaphore를 추가한다.**

**- synchronization: load가 실패한 child thread의 핸들링(wait)이 필요해 load 작업의 상태값을 갖도록 load\_status를 추가한다.**

**- fd: files 라는 file 구조체 포인터 배열 타입을 추가하여 각 파일들의 포인터를 저장하여 이 멤버 변수를 통해 각 파일들에 접근한다. 또한 files\_length 값을 추가하여 파일을 생성할 때마다 fd가 고유한 값을 가지도록(1씩 증가) 해줄 것이다.**

* + 수정하거나 추가해야 하는 함수

- **void exit(int status): exit 전에 열려 있던 파일들을 모두 닫는 코드를 추가한다. 테스트 케이스 일부가 통과하지 않는 문제를 해결한다.**

- **tid\_t process\_execute (const char \*file\_name): start\_process가 실행 중인 child thread가 먼저 끝나는 것을 막기 위해 semaphore를 사용한다. load가 unsuccess 처리된 (load\_status = 2) 스레드가 있다면 process\_wait을 호출하여 프로세스를 회수한다.**

- **static void start\_process (void \*file\_name): load가 끝나면 semaphore를 해제시켜주어 부모 스레드가 작업을 마저 처리할 수 있도록 해준다. load를 성공하지 못했을 경우 load\_status=2로 변경한다.**

- **int read(int fd, void \*buffer, unsigned size): 기존의 read는 fd==0 (STDIN)일 경우에만 처리하였으나, 2 이상의 fd 값도 처리할 수 있도록 수정한다. lock\_acquire / lock\_release 를 사용하여 cs 섹션에서 synchronization 을 구현하고, 이외의 메모리 주소 검사, 유효한 fd값 검사, file의 NULL 여부 등을 체크하는 예외처리를 수행한다.**

- **int write(int fd, void \*buffer, unsigned size): write 역시 기존의 fd==1 (STDOUT)일 경우에만 처리하던 것을 2 이상의 fd 값도 처리할 수 있도록 수정한다. lock\_acquire / lock\_release 를 사용하여 cs 섹션에서 synchronization 을 구현하고, 이외의 메모리 주소 검사, 유효한 fd값 검사, file의 NULL 여부 등을 체크하는 예외처리를 수행한다.**

**- bool create(const char \* file, unsigned initial\_size): parameter로 받는 file 값이 유효한지 NULL 체킹 후, filesys\_create 함수를 호출하여 file을 생성한다.**

**- bool remove(const char\* file): parameter로 받는 file 값이 유효한지 NULL 체킹 후, filesys\_remove 함수를 호출하여 file을 삭제한다.**

**- int open (const char\* file): parameter로 받는 file 값이 유효한지 NULL 체킹 후, filesys\_open 함수를 호출하여 file을 연다. file\_deny\_write 함수를 호출하여 열려는 파일이 기존에 open된 파일과 같은 경우에는 write를 막는다. 또한 files\_length 값을 하나 증가시켜 다음 파일 오픈 시 fd가 고유한 값을 가질 수 있도록 처리한다.**

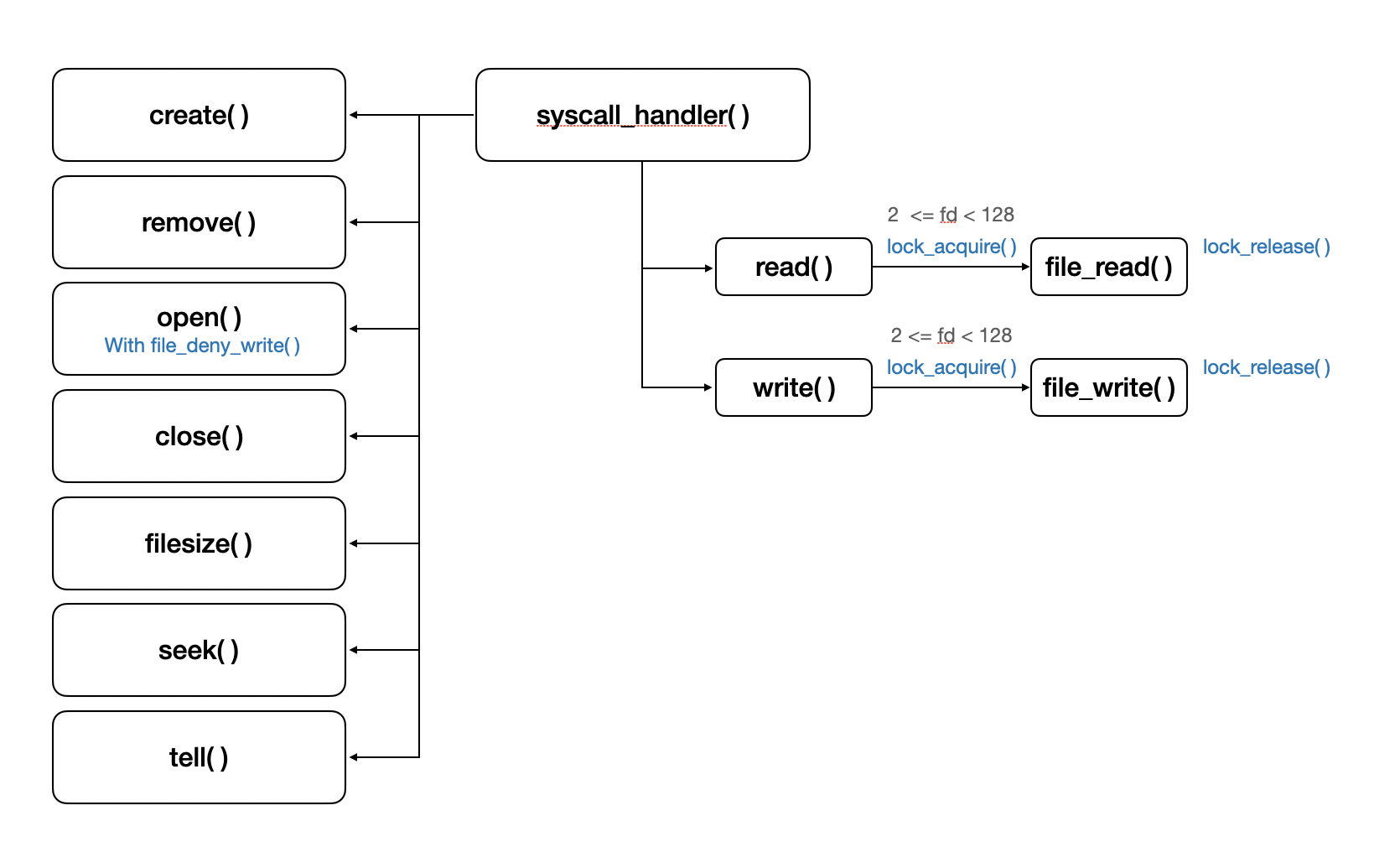
**- void close (int fd): parameter로 받는 fd가 가리키는 file 값이 유효한지 NULL 체킹 후, file\_close 함수를 호출하여 file을 닫는다.**

**- int filesize (int fd): parameter로 받는 fd가 가리키는 file 값이 유효한지 NULL 체킹 후, file\_length 함수를 호출하여 file size를 리턴한다**

**- void seek(int fd, unsigned position): parameter로 받는 fd가 가리키는 file 값이 유효한지 NULL 체킹 후, file\_seek 함수를 호출하여 file 위치를 두 번째 인자로 받은 position으로 변경한다.**

**- void seek(int fd, unsigned position): parameter로 받는 fd가 가리키는 file 값이 유효한지 NULL 체킹 후, file\_tell 함수를 호출하여 file 위치를 리턴한다.**

1. **연구 결과**
   1. **Flow Chart**

****

* 1. **제작 내용**

**1-1. /threads/thread.h**

- semaphore:

**- parent 가 child thread의 load가 끝날 때까지 기다리는 것을 처리해주기 위해 load\_semaphore를 추가한다.**

**- load가 실패한 child thread의 핸들링(wait)이 필요해 해당 load 작업의 상태값을 갖도록 load\_status를 추가한다.**

**- fd: files 라는 file 구조체 포인터 배열 타입 추가, 각 파일들의 포인터를 저장하여 이 멤버 변수를 통해 각 파일들에 접근할 것이다**

|  |
| --- |
| **struct thread**  **{**  **...**  **// init: 0, load success 1, load unsuccess 2**  **int load\_status;**  **// child thread load를 기다려주기 위한 semaphore**  **struct semaphore load\_semaphore;**  **...**  **int files\_length;**  **struct file \*files[256];**  **...**  **};**  **struct thread \* get\_thread\_by\_tid(tid\_t tid);** |

**1-2. /threads/thread.c**

**- semaphore 초기화를 처리하고, fd 값들을 초기화시켜준다. files\_length는 STDIN[fd=0], STDOUT[fd=1]을 고려하여 2로 시작할 수 있도록 정해준다.**

|  |
| --- |
| **static void**  **init\_thread (struct thread \*t, const char \*name, int priority)**  **{**  **...**  **// load status init**  **t->load\_status = 0;**  **// except STDIN / STDOUT fd**  **t->files\_length = 2;**  **sema\_init(&t->load\_semaphore, 0);**  **...**  **for(i = 0; i < 128; i++) {**  **t->files[i] = NULL;**  **}**    **intr\_set\_level (old\_level);**  **}** |

**- get\_loaded\_unsuccess\_thread 함수를 추가하였다. load\_stuats 값으로 load가 제대로 처리되지 못한 스레드를 리턴하는데, 이를 통해 실패한 프로세스를 오류가 나지 않도록 처리할 수 있다.**

|  |
| --- |
| **struct thread \***  **get\_loaded\_unsuccess\_thread () {**  **struct thread \* child\_thd;**  **struct list\_elem \* elem;**  **int load\_status;**  **for (elem = list\_begin(&all\_list); elem != list\_end(&all\_list); elem = list\_next(elem)) {**  **child\_thd = list\_entry(elem, struct thread, allelem);**  **load\_status = child\_thd->load\_status;**  **if (load\_status == 2) {**  **return child\_thd;**  **}**  **}**  **return NULL;**  **}** |

**1-3. /userprog/syscall.c**

- **void syscall\_init (void) & lock 선언**

**- synchronization을 위해 global로 선언한 lock을 초기화한다.**

|  |
| --- |
| // global lock  struct lock rw\_lock;  void  syscall\_init (void)  {  // lock init  lock\_init(&rw\_lock);  intr\_register\_int (0x30, 3, INTR\_ON, syscall\_handler, "syscall");  } |

- **void exit(int status)**

**- exit 전에 열려 있던 파일들을 모두 닫는다. 테스트 케이스 일부가 통과하지 않는 문제를 해결한다.**

|  |
| --- |
| void exit(int status) {  struct thread\* curr = thread\_current();  curr->exit\_status = status;  // file close (except STDIN/STDOUT)  for (int i = 2; i < 128; i++) {  if (curr->files[i] != NULL)  close(i);  }  // curr->load\_status = 0;  printf("%s: exit(%d)\n", curr->name, status);  thread\_exit();  } |

- **int read(int fd, void \*buffer, unsigned size)**

**- 기존의 read는 fd==0 (STDIN)일 경우에만 처리하였으나, 2 이상의 fd 값도 처리할 수 있도록 수정하였다. lock\_acquire / lock\_release 를 사용하여 cs 섹션에서 synchronization 을 구현하였다. 이외의 메모리 주소 검사, 유효한 fd값 검사, file의 NULL 여부 등을 체크하는 예외처리를 수행한다.**

|  |
| --- |
| int read (int fd, void \*buffer, unsigned size) {  if(!(fd >= 0 && fd < 128) || !is\_user\_vaddr(buffer + size)) {  exit(-1);  }  // STDIN 읽어오는 부분  if (fd == 0) {    int i;  for (i = 0; i < (int)size; i++) {  ((uint8\_t \*)buffer)[i] = input\_getc();  if (((uint8\_t \*)buffer)[i] == 0) {  break;  }  // \*(uint8\_t \*)buff += 1;  }  return i;  } else {  // 이외  struct file \*file = thread\_current()->files[fd];  if (file == NULL) {  // except handlng  exit(-1);  }    // locking 처리  lock\_acquire(&rw\_lock);  int byte = file\_read(file, buffer, size);  // unlocking 처리  lock\_release(&rw\_lock);  return byte;  }  } |

- **int write(int fd, void \*buffer, unsigned size)**

**- write 역시 기존의 fd==1 (STDOUT)일 경우에만 처리하던 것을 2 이상의 fd 값도 처리할 수 있도록 수정하였다. lock\_acquire / lock\_release 를 사용하여 cs 섹션에서 synchronization 을 구현하였다. 이외의 메모리 주소 검사, 유효한 fd값 검사, file의 NULL 여부 등을 체크하는 예외처리를 수행한다.**

|  |
| --- |
| int write (int fd, const void \*buffer, unsigned size) {  if(!(fd >= 0 && fd < 128) || !is\_user\_vaddr(buffer + size)) {  exit(-1);  }  // STDOUT  if (fd == 1) {  putbuf(buffer, size);  return size;  } else if (fd >= 2 && fd < 128) {  struct file \*file = thread\_current()->files[fd];  if (file == NULL) {  exit(-1);  }    // locking 처리  lock\_acquire(&rw\_lock);  int byte = file\_write(file, buffer, size);  // unlocking 처리  lock\_release(&rw\_lock);  return byte;  }    return -1;  } |

**- bool create(const char \* file, unsigned initial\_size)**

**- parameter로 받는 file 값이 유효한지 NULL 체킹 후, filesys\_create 함수를 호출하여 file을 생성한다. 해당 함수는 성공 여부를 리턴하는데, 그 값을 그대로 받아 리턴한다.**

**- read와 write 외의 부분에는 lock이 필요없다고 판단되어 주석처리하였다.**

|  |
| --- |
| bool create (const char \*file, unsigned initial\_size) {  if (file == NULL) {  exit(-1);  }  // locking 처리  // lock\_acquire(&lock);  bool success = filesys\_create(file, initial\_size);  // unlocking 처리  // lock\_release(&lock);  return success;  } |

**- bool remove(const char\* file)**

**- parameter로 받는 file 값이 유효한지 NULL 체킹 후, filesys\_remove 함수를 호출하여 file을 삭제한다. 해당 함수는 성공 여부를 리턴하는데, 그 값을 그대로 받아 리턴한다.**

**- read와 write 외의 부분에는 lock이 필요없다고 판단되어 주석처리하였다.**

|  |
| --- |
| bool remove (const char \*file) {  if (file == NULL) {  exit(-1);  }  // locking 처리  // lock\_acquire(&lock);  bool success = filesys\_remove(file);  // unlocking 처리  // lock\_release(&lock);  return success;  } |

**- int open (const char\* file)**

**- parameter로 받는 file 값이 유효한지 NULL 체킹 후, filesys\_open 함수를 호출하여 file을 연다.**

**- file\_deny\_write 함수를 호출하여 열려는 파일이 기존에 open된 파일과 같은 경우에는 write를 막는다.**

**- files\_length 값을 하나 증가시켜 다음 파일 오픈 시 fd가 고유한 값을 가질 수 있도록 처리한다.**

|  |
| --- |
| int open (const char \*file) {  if (file == NULL) {  exit(-1);  }  // locking 처리  // lock\_acquire(&lock);  struct file \*opened\_file = filesys\_open(file);  if (opened\_file == NULL) {  // unlocking 처리  // lock\_release(&lock);  return -1;  }  // WRITE 거부 처리  if (!strcmp(thread\_name(), file)) {  file\_deny\_write(opened\_file);  }  struct thread \*thd = thread\_current();  int fd = thd->files\_length;  thd->files[fd] = opened\_file;  thd->files\_length += 1;  // unlocking 처리  // lock\_release(&lock);  return fd;  } |

**- void close (int fd)**

**- parameter로 받는 fd가 가리키는 file 값이 유효한지 NULL 체킹 후, file\_close 함수를 호출하여 file을 닫는다.**

**- 해당 fd값이 가리키는 file이 없도록 값을 비운다. 따로 커널 api에서 처리해주지 않으므로 이를 해주지 않으면 테스트 케이스에 실패하는 경우가 생긴다.**

|  |
| --- |
| void close (int fd) {  // file 확인  struct thread \*thd = thread\_current();  struct file \*file = thd->files[fd];  if (file == NULL) {  exit(-1);  }  // file close and initialization  file\_close(file);  // for solving closd twice test case  thd->files[fd] = NULL;  } |

**- int filesize (int fd)**

**- parameter로 받는 fd가 가리키는 file 값이 유효한지 NULL 체킹 후, file\_length 함수를 호출하여 file size를 리턴한다**

|  |
| --- |
| int filesize (int fd) {  // file 확인  struct file \*file = thread\_current()->files[fd];  if (file == NULL) {  exit(-1);  }    off\_t size = file\_length(thread\_current()->files[fd]);  return size;  } |

**- void seek(int fd, unsigned position)**

**- parameter로 받는 fd가 가리키는 file 값이 유효한지 NULL 체킹 후, file\_seek 함수를 호출하여 file 위치를 두 번째 인자로 받은 position으로 변경한다. void 타입이므로 리턴값은 없다.**

|  |
| --- |
| void seek (int fd, unsigned position) {  // file 확인  struct file \*file = thread\_current()->files[fd];  if (file == NULL) {  exit(-1);  }    file\_seek(thread\_current()->files[fd], position);  } |

**- void seek(int fd, unsigned position)**

**- parameter로 받는 fd가 가리키는 file 값이 유효한지 NULL 체킹 후, file\_tell 함수를 호출하여 file 위치를 리턴한다.**

|  |
| --- |
| unsigned tell (int fd) {  // file 확인  struct file \*file = thread\_current()->files[fd];  if (file == NULL) {  exit(-1);  }  off\_t pos = file\_tell(thread\_current()->files[fd]);  return pos;  } |

**1-4. /userprog/process.c**

- **tid\_t process\_execute (const char \*file\_name)**

**- start\_process가 실행 중인 child thread가 먼저 끝나는 것을 막기 위해 semaphore를 사용한다.**

**- load가 unsuccess 처리된 (load\_status = 2) 스레드가 있다면 process\_wait을 호출하여 프로세스를 회수한다.**

|  |
| --- |
| tid\_t  process\_execute (const char \*file\_name)  {  ...  struct thread\* thd = get\_thread\_by\_tid(tid);  if (thd != NULL) {  thd->parent\_tid = thread\_current()->tid;  // load semaphore waiting 처리  sema\_down(&thd->load\_semaphore);  }  // multi-oom 해결  struct thread\* temp\_thd = get\_loaded\_unsuccess\_thread();  if (temp\_thd != NULL) {  return process\_wait(temp\_thd->tid);  }  ...  return tid;  } |

- **static void start\_process (void \*file\_name)**

**- load가 끝나면 semaphore를 해제시켜주어 부모 스레드가 작업을 마저 처리할 수 있도록 해준다.**

**- load를 성공하지 못했을 경우 load\_status=2로 변경한다.**

|  |
| --- |
| static void  start\_process (void \*file\_name\_)  {  ...  struct thread \*cur\_thd = thread\_current();  // load semaphore wake up 처리(child process의 load가 끝남을 알려줌)  sema\_up(&cur\_thd->load\_semaphore);  if (!success) {  cur\_thd->load\_status = 2;  exit(-1);  }  ...  } |

* 1. **시험 및 평가 내용**

**- syn-read와 syn-write가 계속 fail이 떴다. child 프로세스가 제대로 생성되지 않아서 발생하는 문제인지, synchronization의 문제인지 아직 정확히 오류를 분석하지 못했다. 다음 프로젝트가 진행되기 전까지 해결하려 한다.**

* 