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

1. **개발 목표**

프로젝트 1에서는 핀토스의 시스템 콜 중 일부분을 구현하였고 User Program이 핀토스에서 실행되도록 하는 기본적인 툴을 구현했다. 프로젝트 1에서 구현하지 않았던 file system로 인해 핀토스는 파일 시스템 관련 작동을(create, remove, open, close, filesize, read, write, seek, tell) 하지 못하는 상태이다. 본 프로젝트에서는 file system과 관련된 시스템 콜을 구현하여서 핀토스에서 파일을 다룰 수 있도록 하는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

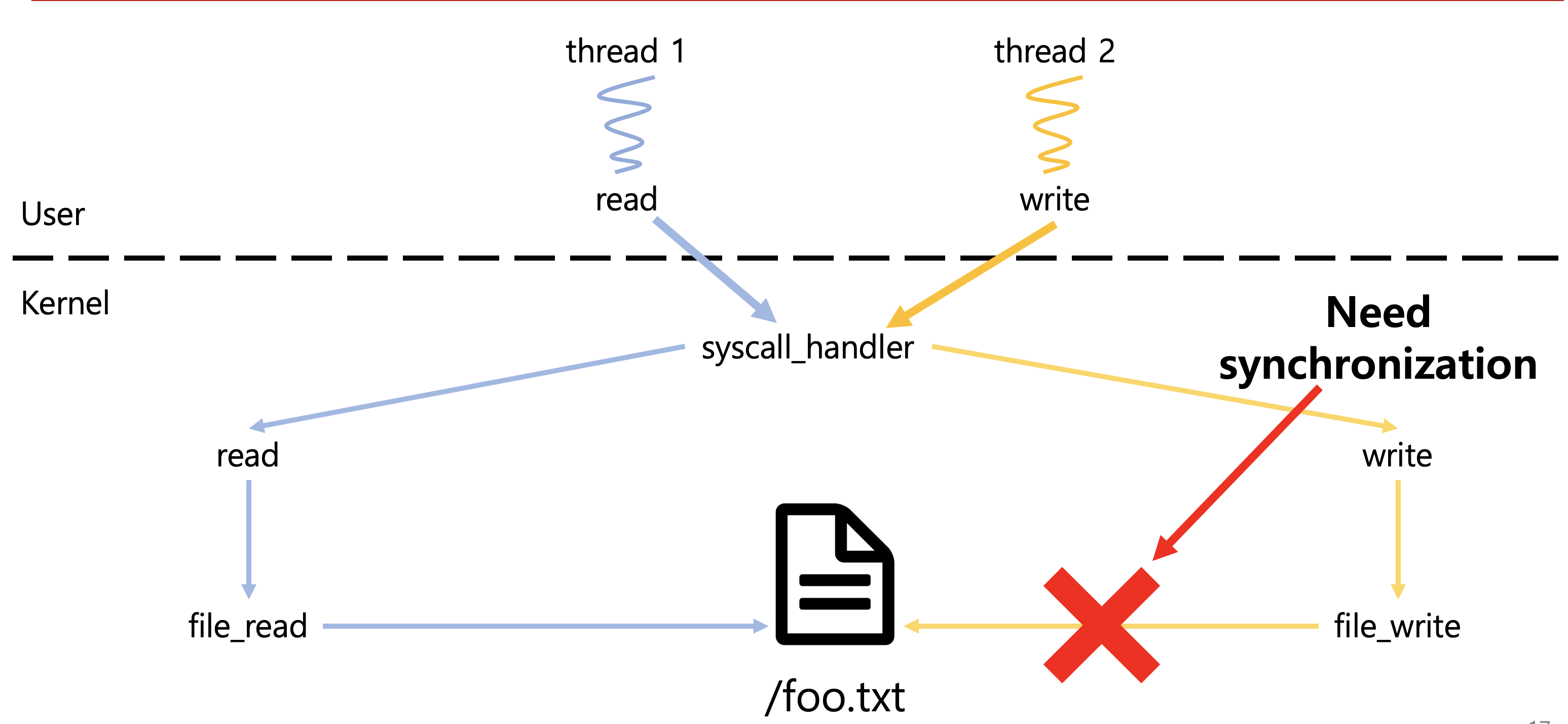
핀토스에서 모든 프로세스는 자신만의 file descriptor를 가진다. 이는 프로세스에서 파일을 구분해주는 양의 정수 값으로 프로세스에서 하나의 파일을 열 때 배정받게 된다. 0은 표준입력, 1은 표준 출력, 2는 표준 에러를 나타내기 때문에 파일입출력에는 3이상의 정수를 사용한다. 같은 파일이어도 여러 번 파일을 연다면 각각 다른 file descriptor를 가지게 되고 파일을 닫을 때도 이를 기준으로 실행된다. 이렇게 file descriptor는 프로세스에서 파일을 열고 닫을 때 필요한 번호로 각 thread마다 일정한 개수의 file descriptor을 가지고 있어야 한다. 본 실습에서는 이러한 file descriptor를 thread 구조체에 구현하여 핀토스에서 file system 시스템 콜을 성공적으로 수행할 수 있도록 한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

많은 User progam들은 파일 시스템을 필요로 한다. 이에 핀토스에서는 기본적인 파일 시스템 API를 제공하고 있다. filesys/filesys.h, filesys/filesys.c, filesys/file.h, filesys/file.c에는 핀토스에서 파일 시스템 관련하여 주어지는 함수로 기본적인 기능을 제공해주지만 많은 한계점이 있다. 내부적으로 동기화가 구현되지 않았으며, 파일 크기가 정적으로 정해진다. 또한 파일 데이터가 연속적인 메모리 공간에 할당되고 서브 디렉토리를 사용할 수 없고 파일 이름의 길이에 제한이 있다. 본 실습에서는 create, remove, open, close, filesize, read, write, seek, tell 파일 시스템 콜을 구현하여 이러한 파일 시스템의 한계점을 개선한다.

3. Synchronization in Filesystem

여러 프로세스가 함께 메모리에 올라가 수행되고 있는 멀티 프로세서 환경에서 파일을 읽거나(read) 쓰는(write)과정은 동기화가 필요하다. 동시에 두 프로세스가 하나의 파일에 쓰는 경우, 하나의 프로세스가 파일에 쓰고 하나의 프로세스가 파일을 읽는 과정에서 동기화를 해주지 않으면 잘못된 데이터가 생성된다. 따라서 파일을 쓰거나 읽는 과정을 Critical Section으로 지정하여 동시에 여러 프로세스가 CS을 실행할 수 없도록 한다. 본 프로젝트에서는 read, write, open시스템 콜에서 lock을 통한 동기화를 실행해 CS를 보호하고 정상적인 파일 시스템 콜을 구현한다.



* 1. **개발 내용**

1. File Descriptor

FD는 여러개의 정수를 저장할 수 있는 다향한 자료구조로 구현된다. 본 프로젝트에서는 각 thread에 길이가 128인 file\* 배열을 사용해서 FD를 관리하도록 한다. 배열의 인덱스가 fd값이 되고 원소의 포인터는 file를 가르키고 있도록 한다. 배열 이외에도 연결 리스트를 사용할 수 있지만 이는 동적할당을 하는 과정에서 오류와 메모리 누수가 발생할 수 있으며 원소에 접근하기 위해서 O(N) 시간이 필요하다는 점에서 배열이 낫다고 판단했다. 열 수 있는 파일의 개수가 128개로 정해져 있다는 점에서 배열을 사용하여도 문제가 없을 것이다. 또한 연결 리스트와 다르게 FD를 O(1)시간에 접근할 수 있고 단순한 알고리즘으로 구현할 수 있다.

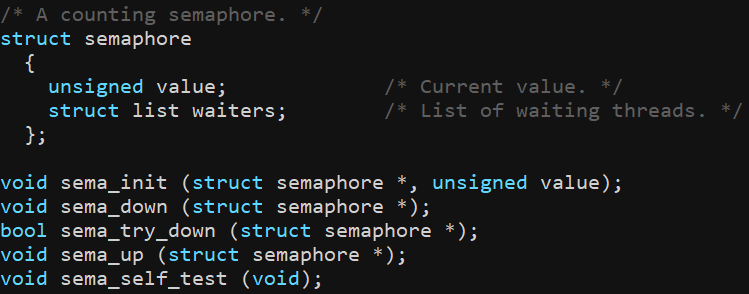
2. System Calls

|  |  |
| --- | --- |
| System Call | 설명 |
| create | 주어진 이름을 가지는 새로운 파일을 생성한다. 파일 생성 성공 여부를 반환한다. |
| remove | 주어진 이름의 파일을 삭제한다. 파일 삭제 여부를 반환한다. |
| open | 주어진 이름의 파일을 열어준다. 파일에 해당하는 file descriptor를 반환한다. |
| close | 주어진 fd의 파일을 닫아준다. |
| filesize | 주어진 fd의 파일 크기를 반환한다. |
| read | 주어진 fd의 파일을 주어진 size만큼 buffer에서 읽는다. 읽은 파일의 크기를 반환한다. 표준 입력이라면 콘솔 창에서 읽는다. |
| write | 주어진 fd의 파일을 주어진 size만큼 buffer에 쓴다. 쓴 파일의 크기를 반환한다. 표준 출력이라면 콘솔 창에 출력한다. |
| seek | 주어진 fd의 파일에서 읽거나 쓸 위치를 변경한다. |
| tell | 주어진 fd의 파일에서 현재 읽거나 쓸 위치를 반환한다. |

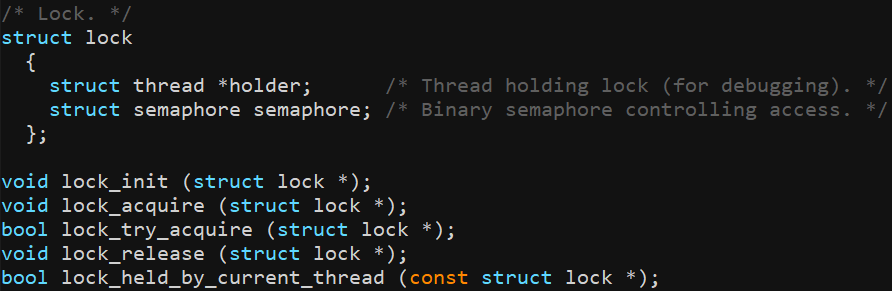
3. Synchronization in Filesystem

파일 시스템에서 동기화가 필요한 경우는 파일을 쓰고, 읽을 경우이다. 특히 여러 프로세스가 같은 파일에 동시에 쓰거나, 한 프로세스가 파일을 읽고 한 프로세스가 파일에 쓰는 경우를 동기화로 방지해야 한다. 동기화 기법에는 interrupt 비활성화, Monitor등이 있지만 본 프로젝트에서는 Lock과 Semaphore를 사용한다.

Semaphore는 0이상의 정수 값으로 Down, Up의 두 작동으로 변경된다. Down는 semaphore값이 양수가 될 때까지 기다렸다가 semaphore값을 하나 내리고 종료된다. Up은 semaphore값을 올린다. 즉 기다리고 있는 Down을 깨우는 역할을 한다. semaphore가 0으로 초기화되면 한 번 일어나는 작동에 대해 동기화를 해주게 되고 1로 초기화되면 한 resource에 대한 접근을 통제하는 데 사용된다. 아래는 핀토스에서 제공하는 semaphore API이다.



Lock은 초기값이 1인 semaphore과 비슷한 기능을 한다. lock은 release, acquire 두 작동으로 변경되는 정수 값이다. release는 semaphore의 up과 acquire는 semaphore의 down과 같은 동작을 한다. semaphore과는 다르게 lock은 해당 lock을 acquire한 thread만이 이 lock을 released할 수 있다. 이러한 특성으로 resource에 대한 접근을 통제하는 경우에는 lock을 사용하는 것이 semaphore보다 적절하다. 아래는 핀토스에서 제공하는 lock API이다.



1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 날짜 | 수행 내용 |
| 11.02~11.05 | 이론 이해 및 프로젝트 계획 |
| 11.06~11.11 | 시스템 콜 구현 및 디버깅 |
| 11.12~11.15 | Synchronization 구현, Documentation |

* 1. **개발 방법**
  + 수정해야하는 소스코드

- threads/threads.h 의 thread 구조체에 file descriptor 배열을 선언한다. thread 구조체에 l\_lock semaphore, parent thread를 선언한다.

- threads/threads.c 의 init\_thread()에서 file descriptor와 l\_lock를 초기화시켜주고 parent thread를 설정해준다.

- userprog/syscall.h 에 file system call의 prototype을 작성한다.

- userprog/syscall.c 에 file system call을 구현하고 system call handler가 file system call을 호출할 수 있도록 한다. file system call에서 모든 메모리 접근에 대해 invalid memory access를 하고 Null 또는 잘못된 메모리에 접근하는 경우 프로세스를 종료한다. 각 system call은 filesys.h에서 제공하는 함수를 적절히 호출한다.

- userprog/exception.c 에서 not present page fault가 발생하는 경우 프로세스를 종료한다.

- userprog/process.c 의 process\_execute()에서 child thread를 생성한 후 child thread가 load될 때까지 기다린다. 또한 강제로 종료된 child thread들은 wait을 호출해서 받아준다. start\_process()에서 thread가 load된다면 parent의 l\_lock을 깨워서 이를 알린다.

* + 수정하거나 추가해야 하는 자료구조

-threads/thread.h 의 thread 구조체에 struct file\* fd[128], struct semaphore l\_lock 추가

* + 수정하거나 추가해야 하는 함수

- threads/threads.c 의 init\_thread()

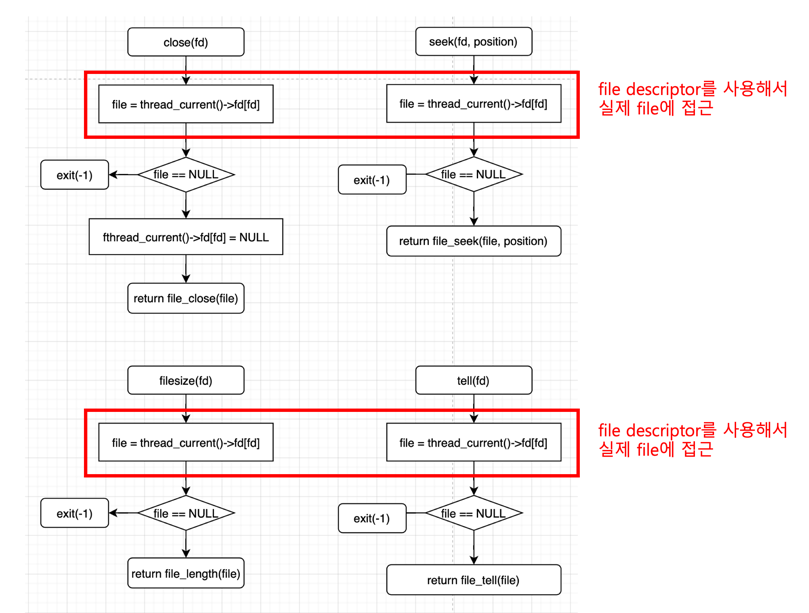
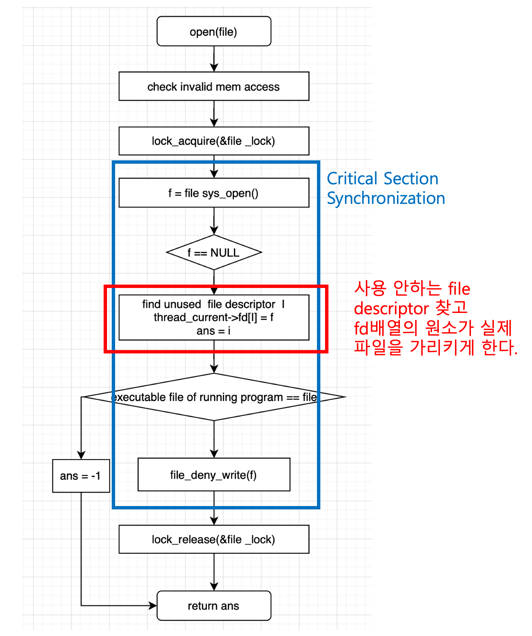
- userprog/syscall.c 의 syscall\_handler(), create(), remove(), open(), close(), filesize(), read(), write(), seek(), tell(), exit()

- userprog/exception.c 의 page\_fault()

- userprog/process.c 의 process\_execute(), start\_process()

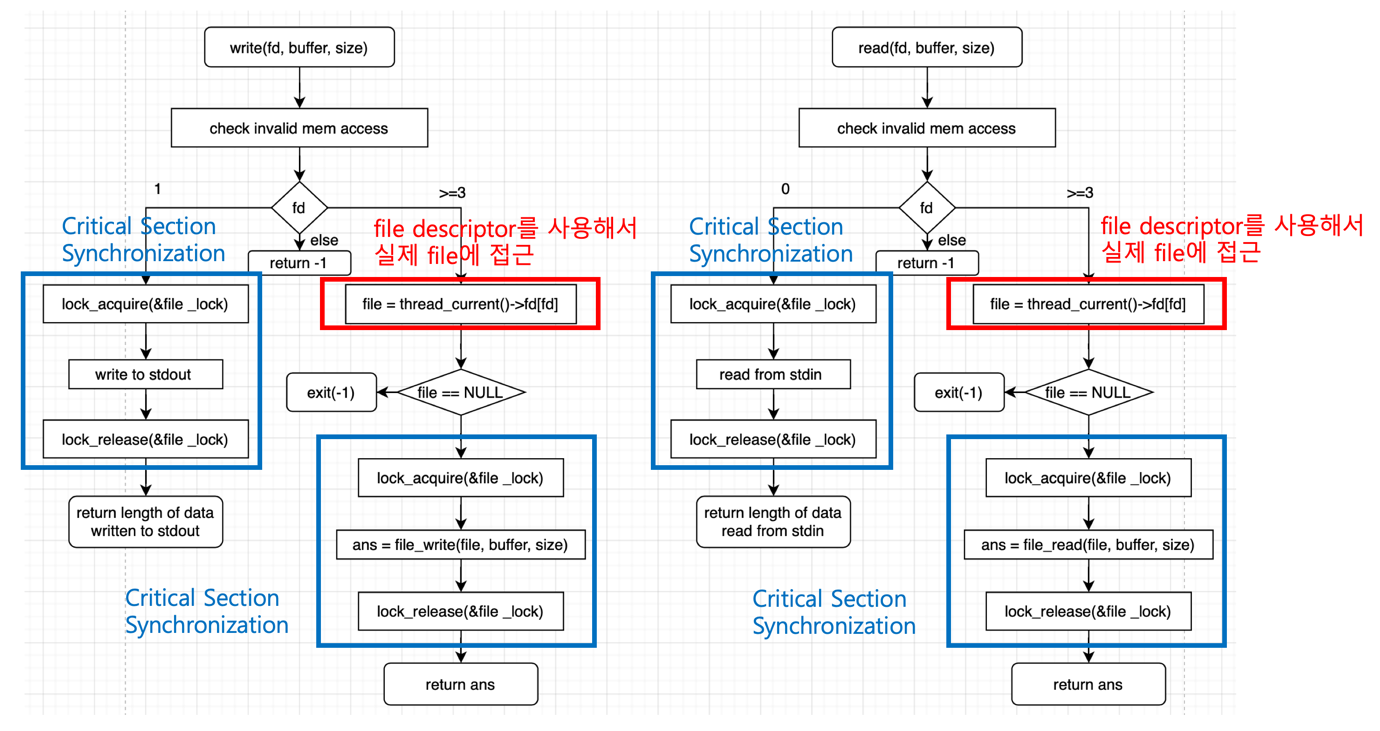
1. **연구 결과**
   1. **Flow Chart**

1. File Descriptor & System Call

****

File Descriptor는 파일이 open될 때 open함수에서 지정된다. open함수에서는 thread에서 사용되지 않는 fd를 찾고 이를 파일과 연결시켜준다. 이후 파일에 관련된 filesize, seek, tell, close등의 시스템 콜에서는 이 fd를 사용해서 파일에 접근한 후 파일에 대한 처리를 한다.

2. Synchronization

****

동기화는 write, read시스템 콜에서 요구된다. 데이터를 표준 출력 또는 파일로 써야하는 경우 코드 전, 후에 lock을 걸어 다른 프로세스가 Critical Section을 실행하지 못하도록 보호한다. 마찬가지로 데이터를 표준 입력 또는 파일에서 읽어올 때 코드 전, 후에 lock을 걸어 다른 프로세스가 Critical Section을 실행하지 못하도록 보호한다.

* 1. **제작 내용**

**1. File Descriptor**

|  |  |
| --- | --- |
| **threads/thread.h** | **설명** |
| struct thread  {  (생략)  #ifdef USERPROG  (생략)  struct semaphore l\_lock;  struct thread\* parent;  struct file\* fd[128];  #endif  (생략)  }; | -thread구조체에 변수를 추가했다.  -자식 thread가 load될 때까지 기다려주는 semaphore  -부모 thread를 가르키고 있는 포인터  -크기 128의 file\* 배열, 3~127의 file descriptor에 대해 각 file구조체를 가르키는 포인터 |
| **threads/thread.c** | **설명** |
| static void  init\_thread (struct thread \*t, const char \*name, int priority)  {  (생략)  #ifdef USERPROG  for (int i = 0; i < 128; i++)  t->fd[i] = NULL;  (생략)  sema\_init(&(t->l\_lock), 0);  t->parent = running\_thread();  (생략)  #endif  } | -앞서 추가한 thread구조체의 변수들을 초기화한다.  -처음에는 open한 파일이 없기 때문에 모든 file descriptor에 해당하는 file이 없다. 모든 fd의 원소를 NULL로 초기화해준다.  -semaphore l\_lock을 초기화해준다.  -현재 실행되고 있는 thread는 t thread를 생성한 부모 thread이다. t->parent포인터가 부모 thread를 가리킬 수 있도록 한다. |

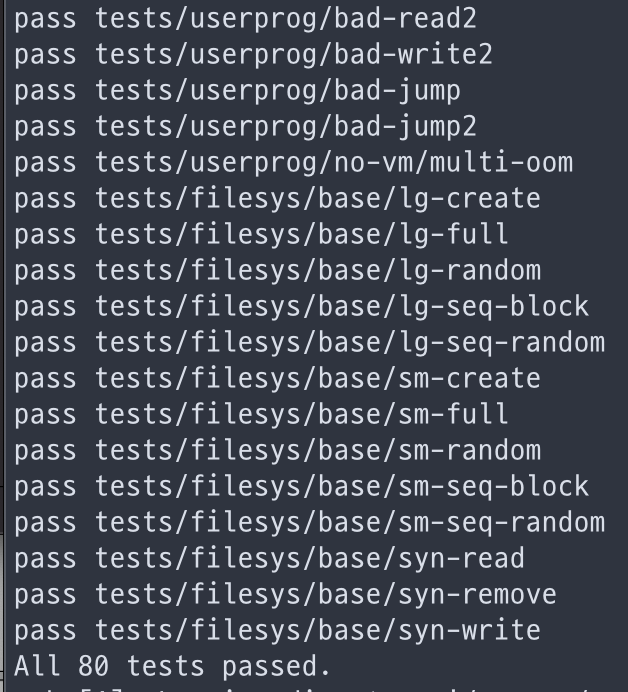
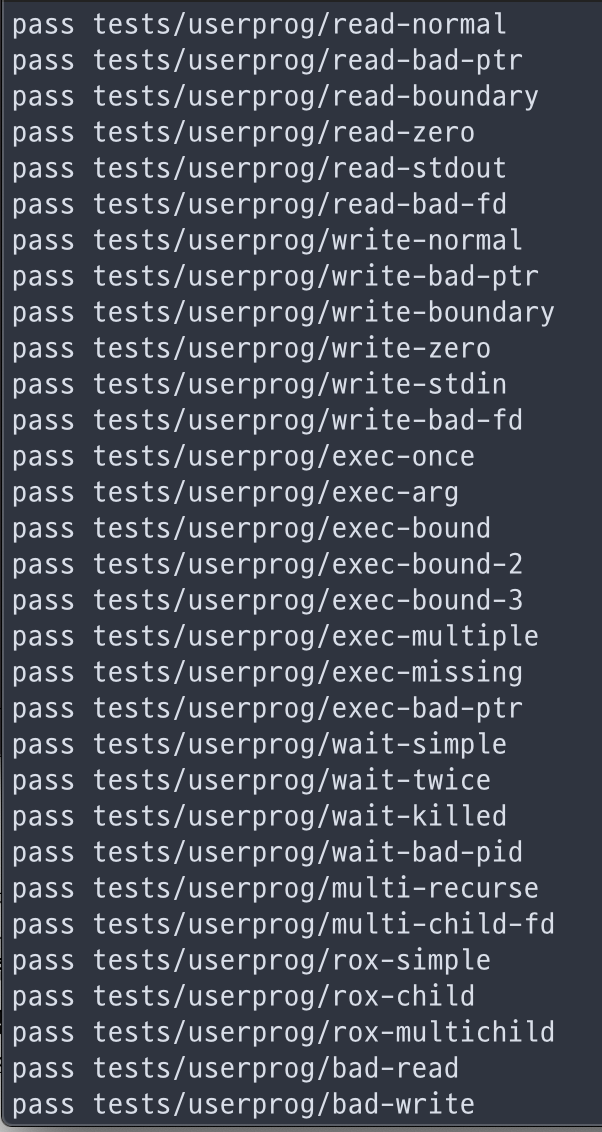
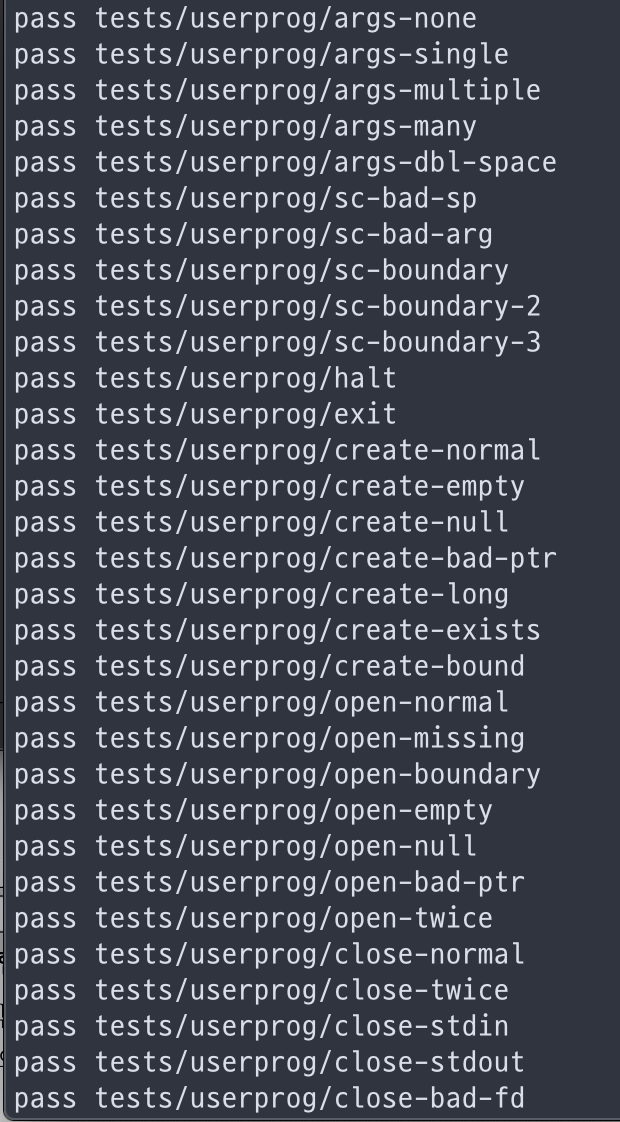
**2. File System Call**

|  |  |
| --- | --- |
| **userprog/syscall.c** | **설명** |
| static void  syscall\_handler (struct intr\_frame \*f UNUSED) {  (생략)  switch(\*(uint32\_t \*)(f->esp)){  (생략)  case SYS\_CREATE:  check\_user\_mem\_acc(f->esp + 16);  check\_user\_mem\_acc(f->esp + 20);  f-> eax = create((const char \*)\*(uint32\_t \*)(f->esp + 16), (unsigned)\*(uint32\_t \*)(f->esp + 20));  break;  case SYS\_REMOVE:  check\_user\_mem\_acc(f->esp + 4);  f-> eax = remove((const char\*)\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_OPEN:  check\_user\_mem\_acc(f->esp + 4);  f-> eax = open((const char\*)\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_CLOSE:  check\_user\_mem\_acc(f->esp + 4);  close((int)\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_FILESIZE:  check\_user\_mem\_acc(f->esp + 4);  f-> eax = filesize((int)\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_SEEK:  check\_user\_mem\_acc(f->esp + 16);  check\_user\_mem\_acc(f->esp + 20);  seek((int)\*(uint32\_t \*)(f->esp + 16), (unsigned)\*(uint32\_t \*)(f->esp + 20));  break;  case SYS\_TELL:  check\_user\_mem\_acc(f->esp + 4);  f-> eax = tell((int)\*(uint32\_t \*)(f->esp + 4));  break;  default:  break;  }  } | - system call handler에서 file system call에 해당하는 system call number을 switch문으로 처리하도록 한다.  -switch문을 사용해 각각의 system call number에 따라 처리해준다.  -인자가 있는 경우에는 system call number 다음으로 stack에 차례대로 저장되어 있기 때문에 4를 더해서 접근한다.  -esp로 접근하는 모든 메모리는 check\_user\_mem\_acc()함수를 사용해서 invalid memory access를 검사한다.  -create, remove, open, filesize, read, write, tell system call의 경우 반환 값을 f->eax에 저장하고 close와 seek 호출만 해준다.  -인자가 2개인 create, seek의 경우에는 user stack이 16byte씩 밀려 있어서 16을 더해 접근해준다. |
| struct lock file\_lock;  void syscall\_init (void)  {  lock\_init(&file\_lock);  intr\_register\_int (0x30, 3, INTR\_ON, syscall\_handler, "syscall");  } | -동기화를 위해 사용할 lock을 선언한다.  -file\_lock을 초기화해준다. |
| bool create(const char \*file, unsigned initial\_size){  check\_user\_mem\_acc(file);  return filesys\_create(file, initial\_size);  }  bool remove(const char \*file){  check\_user\_mem\_acc(file);  return filesys\_remove(file);  }  int open(const char \*file){  int i, ans;  struct file\* f;  check\_user\_mem\_acc(file);  lock\_acquire(&file\_lock);  f = filesys\_open(file);  if (f == NULL) ans = -1;  else{  for(i = 3; i < 128; i++)  if (thread\_current()->fd[i] == NULL) {  if (!strcmp(thread\_current()->name, file))  file\_deny\_write(f);  thread\_current()->fd[i] = f;  ans = i;  break;  }  }  lock\_release(&file\_lock);  return ans;  }  void close(int fd){  struct file\* f = thread\_current()->fd[fd];  if (f == NULL) exit(-1);  thread\_current()->fd[fd] = NULL;  return file\_close(f);  }  int filesize(int fd){  struct file\* f = thread\_current()->fd[fd];  if (f == NULL) exit(-1);  return file\_length(f);  }  int read(int fd, void \*buffer, unsigned size){  int ans, i;  struct file\* f;  check\_user\_mem\_acc(buffer);  if (fd == 0){ //stdin  lock\_acquire(&file\_lock);  for(i=0; i<size; i++)  if(input\_getc() == '\0') break;  lock\_release(&file\_lock);  return i;  }  else if (fd >= 3){  f = thread\_current()->fd[fd];  if (f == NULL) exit(-1);  lock\_acquire(&file\_lock);  ans = file\_read(f, buffer, size);  lock\_release(&file\_lock);  return ans;  }  return -1;  }  int write(int fd, const void \*buffer, unsigned size){  int ans;  struct file\* f;  check\_user\_mem\_acc(buffer);  if(fd ==1 ){ //stdout  lock\_acquire(&file\_lock);  putbuf(buffer, size);  lock\_release(&file\_lock);  return size;  }  else if (fd >= 3){  f = thread\_current()->fd[fd];  if (f == NULL) exit(-1);  lock\_acquire(&file\_lock);  ans = file\_write(f, buffer, size);  lock\_release(&file\_lock);  return ans;  }  return -1;  }  void seek(int fd, unsigned position){  struct file\* f = thread\_current()->fd[fd];  if (f == NULL) exit(-1);  file\_seek(f, position);  }  unsigned tell(int fd){  struct file\* f = thread\_current()->fd[fd];  if (f == NULL) exit(-1);  return file\_tell(f);  }  void exit(int status){  printf("%s: exit(%d)\n", thread\_name(), status);  thread\_current() -> exit\_status = status;  for (int i = 3; i < 128; i++)  if (thread\_current()->fd[i] != NULL) close(i);  thread\_exit();  } | -create시스템 콜로 file의 invalid memory access를 검사한다. filesys\_create()를 호출하고 그 반환값을 반환한다.  -remove시스템 콜로 file의 invalid memory access를 검사한다. filesys\_remove()를 호출하고 그 반환값을 반환한다.  -open시스템 콜로 file의 invalid memory access를 검사한다.  -file\_lock을 acquire해서 다른 프로세스가 파일을 open하지 못하도록 한다.  -주어진 이름의 file를 열고 f에 할당한다  -만약 파일을 여는데 실패하면 반환 값을 -1로 설정한다.  -for문을 통해서 배열을 탐색하면서 사용되지 않은 file descriptor를 찾는다.  -open하려는 file과 현재 thread의 실행 파일 이름이 같다면 이 파일에 write하는 것을 방지한다.  -file\_deny\_write을 호출해서 file구조체의 deny\_write값을 바꿔준다.  -사용되지 않은 file descriptor를 찾았다면 thread의 fd의 원소가 f를 가르킬 수 있도록 한다. 또한 fd 값을 반환값으로 설정한다.  -모든 동작이 끝나면 file\_lock을 release해서 다른 프로세스들이 open을 할 수 있도록 한다.  -close시스템 콜로 닫으려는 fd의 파일에 접근한다. 이 파일이 유효한지 확인하고 않다면 프로세스를 종료한다.  -파일을 닫기 전에 thread의 file descriptor를 비워준다.  -file\_close()를 호출해 파일을 닫는다.  -filesize 시스템 콜로 주어진 fd의 파일에 접근한다. 이 파일이 유효한지 확인하고 않다면 프로세스를 종료한다.  -file\_length()를 호출해서 파일의 길이를찾고 그 값을 반환한다.  -read 시스템 콜로 buffer의 invalid memory access를 검사한다  -만약 fd가 0이라면 표준 입력이기 때문에 input\_getc()를 호출해서 size만큼 표준입력으로 읽는다. 입력이 없을 때 읽기를 중단한다.  -표준 입력에서 read하는 동작을 동기화하기 위해서 읽기 전에 file\_lock을 acquire하고 읽은 후에 file\_lock을 release한다.  -읽은 데이터의 길이를 반환한다.  -fd가 3이상의 정수라면 주어진 fd의 파일에 접근한다. 이 파일이 유효한지 확인하고 않다면 프로세스를 종료한다.  -파일을 읽기 전에 read동작을 동기화하기 위해서 file\_lock을 acquire한다. 다른 프로세스가 open, write, read를 할 수 없도록 했다.  -file\_read()를 호출해서 파일을 읽고 그 반환 값을 ans 저장한다.  -파일을 읽은 후에 file\_lock을 release해서 다른 프로세스가 open, write, read를할 수 있도록 한다.  -ans를 반환한다.  -fd가 1, 2인 경우에는 -1를 반환한다.  -write 시스템 콜로 buffer의 invalid memory access를 검사한다  -만약 fd가 1이라면 표준 출력이기 때문에 putbuf()를 호출해서 size만큼 표준출력으로 쓴다.  -표준 출력으로 write하는 동작을 동기화하기 위해서 쓰기 전에 file\_lock을 acquire하고 쓴 후에 file\_lock을 release한다.  -쓴 데이터의 길이를 반환한다.  -fd가 3이상의 정수라면 주어진 fd의 파일에 접근한다. 이 파일이 유효한지 확인하고 않다면 프로세스를 종료한다.  -파일에 쓰기 전에 write동작을 동기화하기 위해서 file\_lock을 acquire한다. 다른 프로세스가 open, write, read를 할 수 없도록 했다.  -file\_write()를 호출해서 파일에 쓰고 그 반환 값을 ans 저장한다.  -파일에 쓴 후에 file\_lock을 release해서 다른 프로세스가 open, write, read를할 수 있도록 한다.  -ans를 반환한다.  -fd가 1, 2인 경우에는 -1를 반환한다.  -seek 시스템 주어진 fd의 파일에 접근한다. 이 파일이 유효한지 확인하고 않다면 프로세스를 종료한다.  -file\_seek()를 호출해 해당 파일의 읽거나 쓸 위치를 변경  -tell 시스템 콜로 주어진 fd의 파일에 접근한다. 이 파일이 유효한지 확인하고 않다면 프로세스를 종료한다.  -file\_tell()를 호출해서 파일에서 읽거나쓸 위치를 찾고 그 반환값을 반환한다.  -프로젝트 1에서 구현한 exit시스템 콜에 thread가 종료될 때 open했던 모든 파일을 close해주도록 했다.  -for문을 사용해서 모든 fd에 대해 open된 파일이 있다면 close를 호출해서 닫아준다. (메모리 누수 방지) |

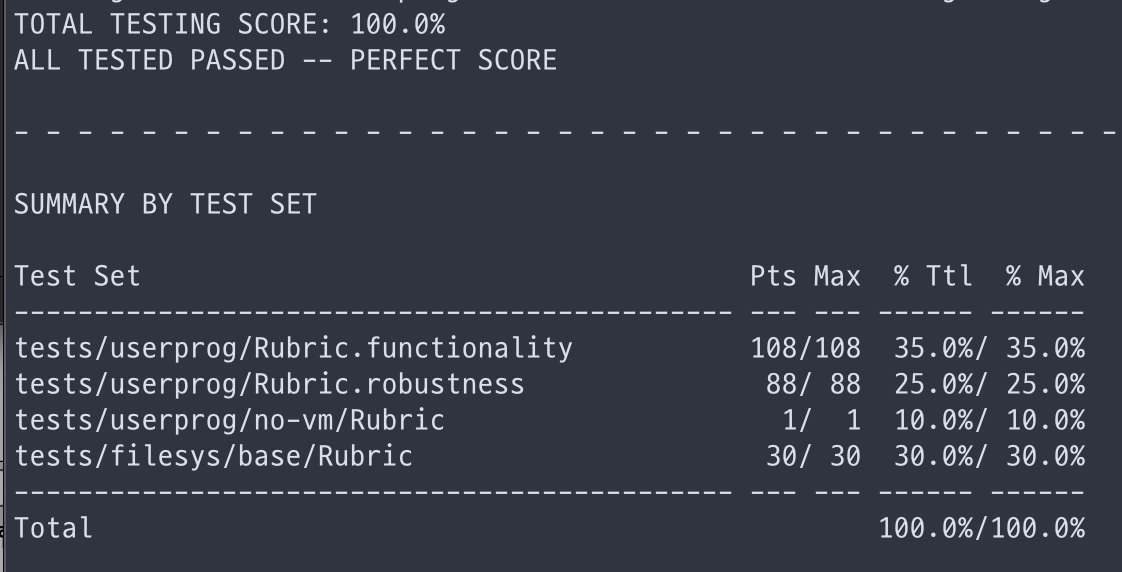
부모 thread가 자식 thread를 생성하고 자식 thread가 load되기 전 부모 thread가 종료된다면 자식 thread를 회수하지 못하게 된다. 이 경우에는 메모리 누수가 발생하여 multi-oom test case를 통과하지 못한다. 부모 thread가 자식 thread를 생성하고 자식 thread가 load될 때까지 기다려 주는 기능을 구현하였다.

|  |  |
| --- | --- |
| **userprog/process.c** | **설명** |
| tid\_t process\_execute (const char \*file\_name)  {  (생략)  struct thread \*parent = thread\_current();  struct list \*children = &(parent ->children);  struct list\_elem\* e;  struct thread\* tmp = NULL;    (생략)  /\* Create a new thread to execute FILE\_NAME. \*/  tid = thread\_create (token, PRI\_DEFAULT, start\_process, fn\_copy);    sema\_down(&(thread\_current()->l\_lock));  if (tid == TID\_ERROR)  palloc\_free\_page (fn\_copy);    for (e = list\_begin(children); e != list\_end(children); e = list\_next(e)) {  tmp = list\_entry(e, struct thread, child\_elem);  if (tmp->exit\_status == -1) return process\_wait(tid);  }  return tid;  } | -자식 thread를 생성하고 자식 thread가 load될 때까지 기다려주는 동작을 추가한다.  -자식 thread를 생성한다.  -l\_lock을 사용해서 자식 thread가 load가 될 때까지 기다려준다.  -for문을 이용해서 자식 thread의 list를 탐색한다.  -load를 실패해서 종료된 자식 thread를 wait()을 호출해서 처리한다. |
| static void  start\_process (void \*file\_name\_)  {  (생략)  success = load (file\_name, &if\_.eip, &if\_.esp);  /\* If load failed, quit. \*/  palloc\_free\_page (file\_name);    sema\_up(&(thread\_current()->parent->l\_lock));  if (!success)  exit(-1);  (생략)  } | -thread를 load한다.  -load가 끝나면 parent thread의 l\_lock를 깨워서 parent thread가 나머지 동작을 할 수 있도록 한다.  -load에 실패한 경우 프로세스를 종료하고 parent의 wait으로 회수되도록 한다. |

* 1. **시험 및 평가 내용**



위는 프로젝트 2에서 구현한 핀토스의 make check 결과이다. 80개의 test case 모두를 통과하였음을 알 수 있다.



위는 프로젝트 2에서 구현한 핀토스의 make grade 결과이다.