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

담당 교수 / 분반 : 박성용 교수님 / 3반

이름 / 학번 : 김혜린 / 20190258

개발 기간 : 2021.11.01 ~ 2021.11.08

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술

이번 프로젝트에서는 pintos에 구현되어 있는 Base File System을 이용하여 file과 관련된 시스템콜을 처리할 수 있도록 하고 (create, remove, open, close, filesize, read, write, seek, tell), file이라는 shared data에 대해 pintos에 구현된 synchronization API와 file struct의 deny\_write 멤버를 이용해 Critical Section을 보호할 수 있도록한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

유닉스에서는 file descriptor를 이용해 file I/O 및 stdin/stdout을 처리한다. file descriptor는 inode table을 이용해 작업을 처리하는데 현재 pintos에서는 해당 작업을 처리하는 기능이 없기 때문에 이를 구현하기 위해 struct thread에 이와 관련된 기능을 추가해 file 입출력 및 console 입출력 기능을 완성한다.

* 1. System Calls

이번 프로젝트에서는 create, remove, open, close, filesize, read(file), write(file), seek, tell의 시스템 콜을 구현한다. 위의 시스템 콜 구현을 통해 파일을 만들고 열고 읽고 수정하고 삭제하는 일련의 file system 기능을 구현할 수 있다.

* 1. Synchronization in Filesystem

file system 구현을 통해 여러 user program이 동시에 하나의 파일에 접근할 경우 파일에 예기치 못한 손상이 생기는 문제가 발생할 수 있다. 따라서 lock을 이용해 read와 write를 critical section으로 취급해 synchronization을 해결한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

pintos에서 파일은 file.c에 구현된 struct file을 이용해 관리하기 때문에 해당 자료구조를 이용해 file descriptor를 구현할 예정이다. 각 프로세스는 자신만의 buffer cache를 가지기 때문에 struct thread에 struct file 포인터 배열을 선언해 배열의 index를 file handler로 반환할 것이다. 이때 fd가 2이하인 경우는 stdin&stdout과 관련된 index 3부터 disk에 저장된 file을 가져와 저장한다. 핀토스 매뉴얼에서 알 수 있듯이 최대 128개의 file을 file descriptor를 이용해 관리할 수 있으므로 배열을 선언할 때 배열의 크기는 131(128+3)으로 한다.

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)

1) create

file을 생성하는 시스템 콜로 파일명과 초기 사이즈를 받아와 disk에 이를 할당한다.

2) remove

file을 제거하는 시스템 콜로 User Program이 제거하고자하는 file을 인자로 넘기면 해당 파일을 찾아 삭제한다.

3) open

file을 열어 파일에 접근할 수 있는 handler를 반환하는 시스템 콜로 open을 통해 반환받은 handler를 이용해 read 혹은 write를 진행할 수 있다.

4) filesize

User Program이 인자로 넘긴 fd 값에 맞는 파일의 사이즈를 반환하는 시스템 콜이다.

5) read

User Program으로부터 접근하고자하는 file의 fd, read해 갈 buffer와 size를 받아와 fd에 맞게 file read를 수행한다.

6) write

User Program으로부터 접근하고자하는 file의 fd, write해 갈 buffer와 size를 받아와 fd에 맞게 file write를 수행한다.

7) seek

open된 파일을 읽거나 쓸 새로운 위치를 찾아주는 시스템 콜이다.

8) tell

파일을 읽고 쓰게 되는 다음 위치를 찾아준다.

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

하나의 file에 대해 동시에 read/write가 일어나면 안되기 때문에 syscall.c에 struct lock file\_lock을 선언하여 해당 lock을 이용해 read와 write가 일어날 때 lock을 걸어주고, 해당 프로세스의 read/write가 종료되면 lock을 풀어주는 방식으로 read/write에서 일어날 수 있는 synchronization을 해결할 것이다. 또한 parent-child process에서 parent process가 child process가 죽어 status를 넘겨주기 전까지는 죽지 않도록 process\_execute와 start\_process 부분에 추가적으로 semaphore를 이용해 이를 구현할 것이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2021.11.08 : 프로젝트 1에서 구현하였던 syscall\_handler 내용을 바탕으로 이번 프로젝트에서 구현해야하는 system call에 대한 처리 틀 구축 (메모리 엑세스 및 sys\_num 처리)

2021.11.09~2021.11.13: file system 구축을 위한 자료구조를 추가하고 각 시스템 콜에 대해 핀토스에서 사용해야하는 함수를 연결하고 에러 처리 및 file synchronization 처리

2021.11.13~2021.11.14: process execute와 start process에서 parent-child process에 대한 semaphore 처리

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드

1. thread.h/thread.c

thread 구조체에 file descriptor와 관련된 변수와 parent-child process 관리를 위한 기타 변수들을 추가할 예정이므로 init\_thread함수에서도 이를 반영해 thread 구조체를 초기화해주어야 한다.

1. process.c/process.h

process\_execute 함수에서 child process를 생성하고 현재 thread의 children을 돌며 thread 구조체에 새로 추가한 wait\_status를 확인해 아직 실행중인 child가 있다면 exec\_lock을 걸어 child가 종료되면 sema\_up을 걸어 parent process가 다시 실행될 수 있도록 한다. 또한 start\_process에서 load가 실패했을 경우 wait\_status를 1로 변경해줌으로써 parent process가 비정상적으로 죽은 child process를 처리할 수 있도록 해준다.

1. syscall.c

추가적으로 구현해야하는 create, remove, open, close, filesize, seek, tell에 대한 switch-case문을 추가하고 해당 시스템콜을 처리할 각각의 함수들을 선언한다. 또한 지난 프로젝트에서 구현했던 READ와 WRITE sys\_num에 대한 시스템콜 함수인 read와 write함수에 fd가 2보다 큰 경우에 대한 read/write를 추가로 처리해주어야 한다.

* + 수정하거나 추가해야 하는 자료구조

|  |
| --- |
| struct thread |
| // 2: 20190258  struct file \*fd\_table[BUF\_MAX];  struct thread \*parent;  struct semaphore exec\_lock;  int wait\_status; |

struct thread에 다음과 같은 네 가지 멤버 변수를 추가하였다. fd\_table을 file descriptor를 관리하는 배열로 BUF\_MAX는 131로 선언되어 있다. 배열 형식으로 구현했기 때문에 빠르게 fd에 접근할 수 있다는 장점이 있다.

parent의 경우 child process가 exec\_lock이 걸린 parent process의 lock을 풀어주려면 해당 child가 parent를 기억하고 있어야 하기 때문에 추가해주었고 exec\_lock은 아직 실행중인 child\_process를 기다리기 위해 parent process가 실행한 lock이다. child process가 wait이 필요한지 아닌지의 여부는 해당 프로세스의 wait\_status를 통해 확인할 수 있다.

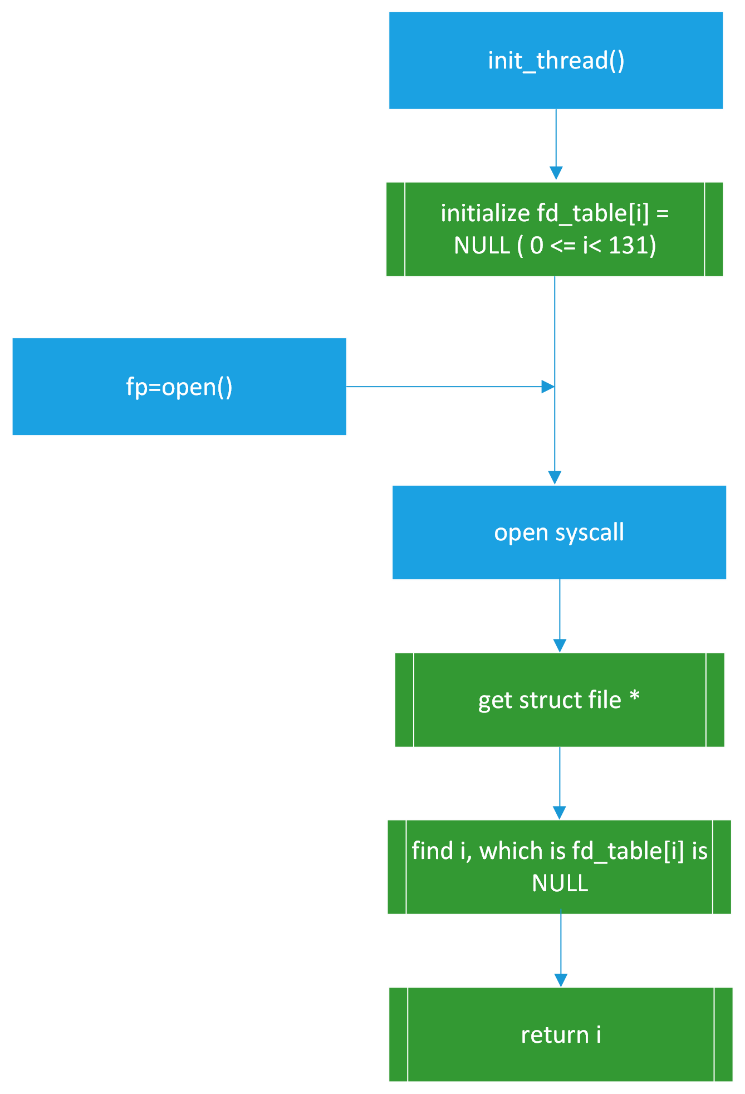
* + 수정하거나 추가해야 하는 함수

|  |
| --- |
| static bool create (const char \*file, unsigned initial\_size);  static bool remove (const char \*file);  static int open (const char \*file);  static int filesize (int fd);  static void seek (int fd, unsigned position);  static unsigned tell (int fd);  static void close (int fd); |

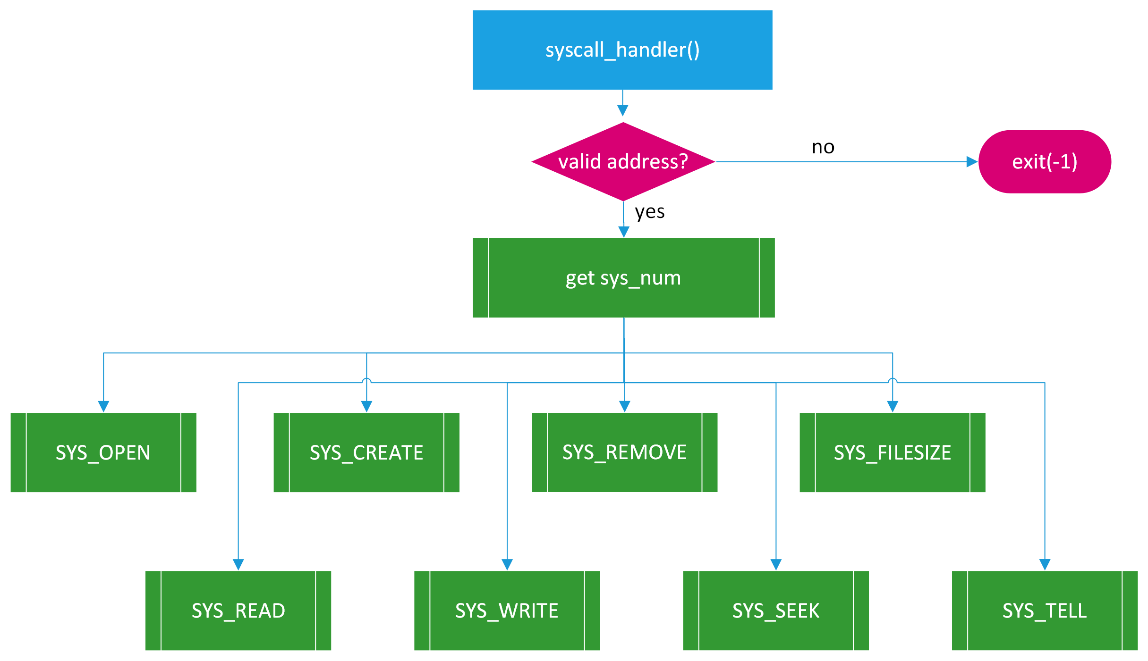
syscall.c에 위의 함수들이 추가로 구현된다. 각 함수들은 user에서 넘기는 각 시스템 콜의 인자와 동일한 인자를 가지고 있으며 각 함수에는 system call에 맞는 내용을 구현한다.

1. **연구 결과**
   1. **Flow Chart**

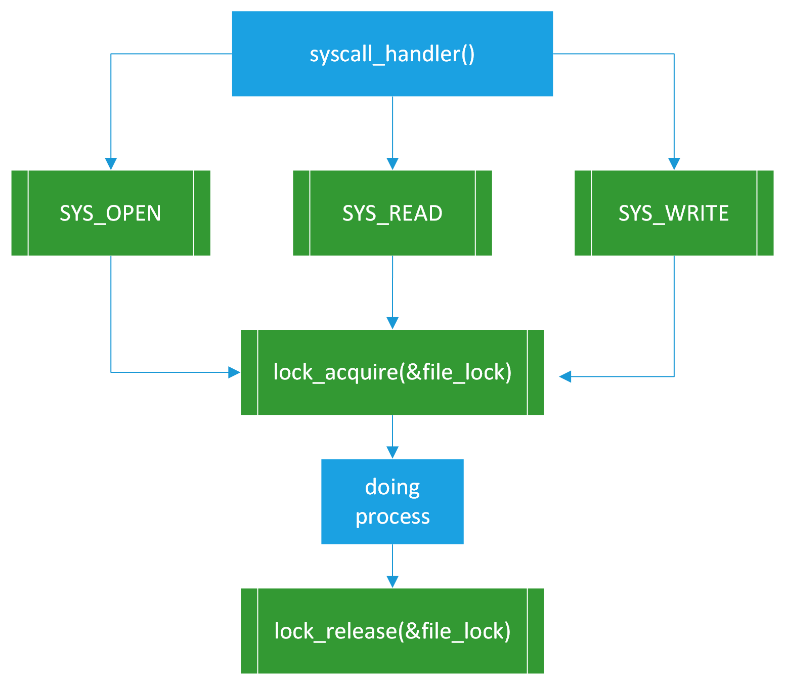
* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
  + 1. file descriptor



* + 1. system call



* + 1. synchronization file system



* 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
  + 1. file descriptor

|  |
| --- |
| static void init\_thread (struct thread \*t, const char \*name, int priority) |
| sema\_init(&(t->exec\_lock), 0);  t->parent = running\_thread();  for(int i=0;i<131;i++)  t->fd\_table[i] = NULL;  t->wait\_status = 0; |

앞에서 수정한 자료구조에서도 언급했듯이 thread 구조체에 exec\_lock, parent, fd\_table, wait\_status를 추가해주었으므로 thread.c에 구현된 init\_thread에서도 이를 반영해 thread 구조체를 초기화 해주어야 한다.

fd\_table의 각 fd는 NULL로 초기화되며 file open시 앞에서부터 탐색하며 NULL을 가지고 있는 테이블 중 가장 작은 fd에 file이 할당된다. fd\_table은 filesys/file.h에 선언된 struct file의 포인터 배열로 선언되어 있으며 exec\_lock의 타입인 struct semaphore의 경우 threads/synch.h에 선언되어 있는 struct semaphore 구조체를 사용하였다. file이 fd\_table에 할당되는 자세한 과정은 밑의 open system call에서 자세히 다룰 예정이다.

|  |
| --- |
| void exit (int status) |
| for(i=3; i<BUF\_MAX; i++){  if(t->fd\_table[i] != NULL){  close(i);  }  } |

또한 thread에 fd\_table이 추가되었기 때문에 프로세스 종료시 아직 close되지 않은 파일이 있는지 확인해준 뒤, 프로세스를 종료시켜주었다.

* + 1. system call

각각의 시스템 콜 넘버에 대한 처리는 이전 프로젝트와 마찬가지로 switch case문을 이용해 처리하였다. 각 시스템 콜에 대해 해당 시스템 콜의 이름을 가진 함수를 선언하여 시스템 콜을 처리하였다. 각각에 해당하는 함수를 호출하기 전에 stack address가 유효한지 check\_memory\_access함수를 이용해 확인하였다.

|  |
| --- |
| static void syscall\_handler (struct intr\_frame \*f UNUSED) |
| case SYS\_OPEN:  check\_memory\_access(f->esp, t, 1);  f->eax = open(\*(char\*\*)(f->esp+4));  break;  case SYS\_FILESIZE:  check\_memory\_access(f->esp, t, 1);  f->eax = filesize(\*(int\*)(f->esp+4));  break;  case SYS\_READ:  … |

1. create

|  |
| --- |
| static bool create (const char \*file, unsigned initial\_size) |
| {  struct thread \*t = thread\_current();  if(!is\_user\_vaddr(file) || pagedir\_get\_page(t->pagedir, file) == NULL)  exit(-1);  if(file == NULL) exit(-1);  return filesys\_create(file, (off\_t)initial\_size);  } |

filesys/filesys.h에 선언된 file을 disk에 할당해주는 filesys\_create함수를 이용해 create 시스템 콜을 구현하였다. 이때 해당 함수를 콜하기 전에 인자로 넘어온 file이 저장된 address가 유효한 address인지 확인해주고 file값이 NULL일 경우 status를 -1로 해 해당 프로세스를 종료시켰다.

1. remove

|  |
| --- |
| static bool remove (const char \*file) |
| {  struct thread \*t = thread\_current();  if(!is\_user\_vaddr(file) || pagedir\_get\_page(t->pagedir, file) == NULL)  exit(-1);  if(file == NULL) exit(-1);  return filesys\_remove(file);  } |

filesys/filesys.h에 선언된 filesys\_remove함수를 이용해 file을 삭제하는 시스템콜로 앞의 create systemcall과 마찬가지로 file변수 자체의 유효성 판단을 거친 후 해당 함수를 호출한다.

1. open

|  |
| --- |
| static int open (const char \*file) |
| {  struct thread \*t = thread\_current();  struct file \*f;  int i, fd = -1;  if(!is\_user\_vaddr(file) || pagedir\_get\_page(t->pagedir, file) == NULL)  exit(-1);  if(file == NULL) exit(-1);  lock\_acquire(&file\_lock);  f = filesys\_open(file);  if(f == NULL) fd = -1;  else{  for(int i=3;i<BUF\_MAX;i++){  if(t->fd\_table[i] == NULL){  if(!strcmp(t->name, file)) file\_deny\_write(f);  t->fd\_table[i] = f;  fd = i;  break;  }  }  }  lock\_release(&file\_lock);  return fd;  } |

앞의 char 포인터 타입을 인자로 받는 다른 시스템콜과 마찬가지로 포인터 값의 유효성을 확인하고 file이 NULL이 아닌지 확인한 후 시스템콜을 진행한다. 해당 프로세스가 file open을 진행하는 동안 다른 프로세스는 진행할 수 없도록 file\_lock을 걸어주었고 이는 file open이 모두 진행되고 나면 lock\_release를 통해 풀어준다. fd\_table에 file을 할당하는 과정은 for문을 통해 이뤄지며 STDI/O와 구분하기 위해 3부터 탐색을 시작해 NULL인 table의 fd에 file을 저장하고 해당 fd를 반환한다.

1. close

|  |
| --- |
| static void close (int fd) |
| {  struct thread\* t = thread\_current();  if(t->fd\_table[fd] == NULL) exit(-1);  file\_close((struct file\*)t->fd\_table[fd]);  t->fd\_table[fd] = NULL;  } |

fd에 해당하는 file handler를 할당해제하는 시스템콜로 fd\_table의 fd로 접근해 해당 file에 대해 filesys/file.h에 선언된 file\_close를 통해 file을 close해주고 fd에 해당하는 index에 NULL값을 할당해 fd\_table에서 해당 file을 제거한다. 이때 fd가 유효한 fd인지 t->fd\_table[fd] == NULL 구문을 통해 확인한다.

1. filesize

|  |
| --- |
| static int filesize (int fd) |
| {  struct thread \*t = thread\_current();  if(t->fd\_table[fd] == NULL) exit(-1);  return (int)file\_length((struct file\*)t->fd\_table[fd]);  } |

현재 프로세스의 fd\_table의 fd에 해당하는 파일의 사이즈를 반환하는 시스템 콜로 앞의 close 시스템 콜과 마찬가지로 fd의 유효성을 판단한 뒤 filesys/file.h에 선언된 file\_length함수를 호출해 해당 테이블에 저장된 파일의 크기를 반환해준다.

1. read

|  |
| --- |
| static int read (int fd, void \*buffer, unsigned size) |
| {  int i;  int ret = 0;  struct thread \*t = thread\_current();  if(!is\_user\_vaddr((char\*)buffer) || pagedir\_get\_page(t->pagedir, (char\*)buffer) == NULL)  exit(-1);  lock\_acquire(&file\_lock);  if(fd == 0){  for(i = 0;i< size;i++)  ((char\*)buffer)[i] = input\_getc();  ret = size;  }  else if(fd > 2){  if(thread\_current()->fd\_table[fd] == NULL){  lock\_release(&file\_lock);  exit(-1);  }  ret = file\_read(t->fd\_table[fd], buffer, size);  }  else {  lock\_release(&file\_lock);  exit(-1);  }  lock\_release(&file\_lock);  return ret;  } |

read system call의 경우 이전 프로젝트에서 0으로 fd가 들어올 경우 input\_getc로 console read를 처리하는 것까지 구현하였다. 이번 프로젝트에서는 fd가 2보다 클 때 해당 fd가 유효한 fd인지 확인하고 유효하다면 file\_read함수를 통해 read 시스템콜을 진행하고 filesys/file.h에 선언된 file\_read 함수가 반환하는 실제로 read된 크기를 시스템 콜 결과로 반환한다.

1. write

|  |
| --- |
| static int write (int fd, const void \*buffer, unsigned size) |
| {  int ret = 0;  struct thread \*t = thread\_current();  if(!is\_user\_vaddr(buffer) || pagedir\_get\_page(t->pagedir, buffer) == NULL)  exit(-1);  lock\_acquire(&file\_lock);  if(fd == 1){  putbuf(buffer, size);  ret = size;  }  else if(fd > 2){  if(t->fd\_table[fd] == NULL){  lock\_release(&file\_lock);  exit(-1);  }  if(t->fd\_table[fd]->deny\_write)  file\_deny\_write(t->fd\_table[fd]);  ret = file\_write(t->fd\_table[fd], buffer, size);  }  else{  lock\_release(&file\_lock);  exit(-1);  }  lock\_release(&file\_lock);  return ret;  } |

read system call과 마찬가지로 앞의 프로젝트에서 fd가 1일 경우 putbuf를 통해 console write를 진행했던 것에 추가로 fd가 2보다 클 경우 해당 fd에 대한 유효성을 판단한 뒤, 해당 file이 deny\_write가 아닐 경우에만 filesys/file.h에 선언된 file\_write함수를 이용해 시스템 콜을 처리하였다.

1. seek

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

다른 fd를 인자로 받는 시스템콜과 마찬가지로 fd에 대한 유효성 검사 후 filesys/file.c에 선언된 file\_seek함수를 호출해 해당 시스템콜을 처리한다.

1. tell

|  |
| --- |
| static unsigned tell (int fd) |
| {  struct thread \*t = thread\_current();  if(t->fd\_table[fd] == NULL) exit(-1);  return file\_tell((struct file\*)t->fd\_table[fd]);  } |

앞의 fd를 인자로 받는 시스템콜과 같은 에러 처리를 해준 뒤, filesys/file.h에 선언된 file\_tell 함수를 호출해 해당 시스템 콜을 처리한다.

* + 1. synchronization file system

file\_lock에 대한 구현은 위 시스템 콜의 open과 read, write함수의 lock\_acquire함수와 lock\_release 함수의 호출 위치를 참고하면 된다.

동시에 file open, read, write가 이뤄지는 것을 막기 위해 프로세스가 open, read, write를 진행하는 동안을 file\_lock을 걸어서 다른 프로세스가 해당 시스템콜을 진행할 수 없게 하고 해당 프로세스가 file open, read, write를 완료할 경우 file\_lock을 풀어 다른 프로세스도 해당 시스템콜을 사용할 수 있도록 해준다.

file\_lock은 struct lock구조체를 가지며 해당 구조체는 thread/synch.h에서 확인할 수 있다.

|  |
| --- |
| struct lock  {  struct thread \*holder; /\* Thread holding lock (for debugging). \*/  struct semaphore semaphore; /\* Binary semaphore controlling access. \*/  }; |

* + - * 개발 중 발생한 이슈

synchronization file system을 구현하면서 처음에 open, read, write 진행 과정에서 발생할 수 있는 에러에 대해 바로 exit(-1)을 호출해 file\_lock에서 holding error발생하였다. file\_lock과 error control을 동시에 고려해야하는 상황에서 발생한 이슈였다.

|  |
| --- |
| tid\_t process\_execute (const char \*file\_name) |
| struct thread \*t = thread\_current();  sema\_down(&t->exec\_lock);  …  struct list\_elem \*e = list\_begin(&(t->children));  struct thread \*child;  while(e != list\_end(&(t->children))){  child = list\_entry(e, struct thread, child\_elem);  if(child->wait\_status == 1){  return process\_wait(tid);  }  e = list\_next(e); |

child가 load되기 전에 parent가 끝나버리는 경우가 발생하면 안되기 때문에 exec\_lock을 걸어주고 process\_execute에서 child 프로세스를 생성할 때 현재 thread t에 대해 해당 children을 돌면서 아직 비정상적으로 종료돼 아직 parent process로부터 처리되지 못한 애들을 process\_wait함수를 호출해 정상적으로 처리할 수 있도록 해주었다.

그리고나서 자식 프로세스가 start process를 할 때 해당 lock을 풀어줌으로써 자식 프로세스가 load되기 전에 부모 프로세스가 죽어 자식 프로세스가 고아가 되는 것을 방지하였다.

|  |
| --- |
| static void start\_process (void \*file\_name\_) |
| sema\_up(&thread\_current()->parent->exec\_lock);  if (!success){  //thread\_exit ();  thread\_current()->wait\_status = 1;  exit(-1);  } |

load가 완료되고 나면 parent의 exec lock을 풀어줘서 parent가 실행되어 죽어도 됨을 알려준다. 이때 load가 성공적으로 이뤄지지 않았다면 이를 parent에게 알려 처리할 수 있도록 해야하므로 해당 프로세스의 wait\_status를 1로 처리하고 exit(-1)을 호출해 비정상적인 종료를 알린다.

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

