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

담당 교수 / 분반 : 김영재 교수님 , 2분반

이름 / 학번 : 김하늘, 20181618

개발 기간 : 2020-11-11 ~ 2020-11-15

1. **개발 목표**

* 이전 프로젝트에선 Pintos OS상의 user program 실행이 가능하도록 하는 여러 기능을 구현하였다. 이번 프로젝트는 file system 관련 system call을 추가로 구현하고 synchronization이 안정적으로 수행될 수 있게끔 하는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목을 구현해야 하는 이유, 혹은 구현 시 기대되는 결과를 간략히 서술

1. File Desciptor

파일 디스크립터는 프로세스에서 특정 파일에 접근할 때 사용하는 값이다. 일반적으로 0, 1, 2의 값은 각각 stdin, stdout, stderr에 매핑되어있으며 나머지 정수 값은 특정 파일에 매핑될 수 있다. 프로세스가 실행 중 파일을 open하면 커널은 해당 프로세스의 파일 디스크립터 숫자 중 사용하지 않는 가장 작은 값을 할당해주고, 그 다음 프로세스가 열려있는 파일에 시스템 콜을 사용해 접근할 때 파일 디스크립터 값을 이용해서 파일을 지칭할 수 있게 된다.

2. System calls

파일 시스템에 관련한 system call을 구현해야했다. 파일을 오픈하기 전 새로 만드는 create, 파일 디스크립터를 매핑시켜 파일을 여는 open, 파일을 닫는 close, 파일 관련 내용을 삭제하는 remove, 기존의 표준입출력 기능에 파일 입출력 기능을 추가하고 synchronization이 이루어지도록 구현하는 read & write, 파일의 크기를 반환하는 filesize, 파일의 커서 위치를 알려주는 tell, 파일의 커서를 특정 위치로 옮겨주는 seek을 구현하였다.

3. Synchronization in Filesystem

이번 프로젝트에서 구현하는 시스템 콜인 read, write에서 파일을 읽는 중 각 프로세스에 대해 critical section을 침범하지 않도록 swmaphore와 lock 기능을 사용하여 synchronization을 구현해야한다.

* 1. **개발 내용**

1. File Descriptor

thread 구조체에 struct file\* file\_info[128]을 추가하여 해당 배열 변수에 파일에 대한 정보를 저장할 수 있도록 하였다. 이를 통해 현재 thread에서 열려있는 파일을 다룰 수 있으며, 배열의 index를 각 파일 디스크립터 넘버와 매핑하여 접근할 수 있다.

2. System calls

- create : 파일의 이름, 파일의 초기 사이즈 값을 받아 새로운 파일을 만드는 명령어이다.

- open : 파일의 이름을 받아 파일을 열고, 해당 파일의 포인터를 파일 디스크립터에 매핑하여 저장한다.

- close : 파일 디스크립터 넘버를 받아 해당 테이블에 저장된 파일을 close하고 그 위치에는 NULL을 넣어 종료한다.

- remove : 파일 이름을 받아 해당 파일을 삭제한다.

- read : 파일 디스크립터 넘버를 받아 3 이상의 값일 경우 해당 파일 디스크립터 넘버를 이용해 테이블에 접근하여 파일을 읽는다.

- write : read와 마찬가지로 파일 디스크립터 넘버를 받아 3이상의 값일 경우 그 값으로 테이블에 접근하여 파일에 쓰는 작업을 한다.

- filesize : 파일 디스크립터 넘버를 받아 해당 파일의 크기를 반환한다.

- tell : 파일 디스크립터 넘버를 받아 현재 파일의 커서 위치를 반환한다.

- seek : 파일 디스크립터 넘버와 position값을 받아 해당 파일의 커서를 해당 position으로 이동시킨다.

3. Synchronization in Filesystem

프로세스가 하나의 파일에만 접근하여 읽고 쓸 수 있도록 전역변수로 struct lock 변수를 선언하고, 시스템 콜의 open, read, write에서 critical section을 해당 lock 변수를 이용하여 막았다. 또한 struct semaphore 변수를 추가하여 부모 쓰레드가 여러개의 자식 쓰레드를 만들어 접근할때 sychronization이 정상적으로 작동될 수 있게끔 제한을 걸어주었다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2020.11.11 ~ 2020 11.12 매뉴얼 정독 및 system call 틀 작성

2020.11.12 ~ 2020 11.13 synchronization 구현 및 디버깅을 통한 system call 추가 구현

2020.11.13 ~ 2020.11.15 synchronization 디버깅 후 보고서 작성

* 1. **개발 방법**

1. File Descriptor

우선 파일에 대한 정보를 저장하기 위해 thread.h의 thread 구조체에 struct file\* file\_info[128]을 추가하였다. 또한 int fd를 추가하여 해당 쓰레드에서 파일 디스크립터를 얼마나 사용하였는지 저장할 수 있도록 하였다. 이를 통해 syscall.c에서 파일을 이용해 filesys 디렉토리의 함수를 사용하여 파일을 적절히 다룰 수 있도록 하였다.



2. System calls

각 시스템 콜은 메뉴얼에 명시된 대로 syscall.c에 함수를 추가하여 구현하였다.

- create : syscall.c에 bool create(const char \*file, unsigned initial\_size)을 추가하였고, filesys.c의 filesys\_create()함수를 사용하여 구현하였다.

- open : syscall.c에 int open(const char \*file)을 추가하였고, synchronization을 위해 전역변수로 선언한 struct lock lk을 사용하였고, filesys.c의 filesys\_open()을 이용해 기능을 구현하였다.

- close : syscall.c에 void close(int fd)을 추가하였고, file.c의 file\_close()를 이용해 기능을 구현하였다.

- remove : syscall.c에 bool remove(const char \*file)을 추가하였고, filesys.c의 filesys\_remove()를 이용하여 구현하였다.

- read : syscall.c에 int read(int fd, void\* buffer, unsigned size)를 추가하였고, synchronization을 위해 struct lock lk를 사용하여 lock을 걸어주었다. 그리고 file.c의 file\_read를 이용하여 기능을 구현하였다.

- write : read와 마찬가지로 syscall.c에 int write(int fd, const void \*buffer, unsigned size) 를 추가하였고, synchronization을 위해 struct lock lk를 사용하여 lock을 걸어주었다. 그리고 file.c의 file\_write를 이용하여 기능을 구현하였다.

- filesize : syscall.c에 int filesize(int fd)을 추가하였고, file.c의 file\_length()를 이용해 기능을 구현하였다.

- tell : syscall.c에 unsigned tell(int fd)을 추가하였고 file.c의 file\_tell()을 이용해 기능을 구현하였다.

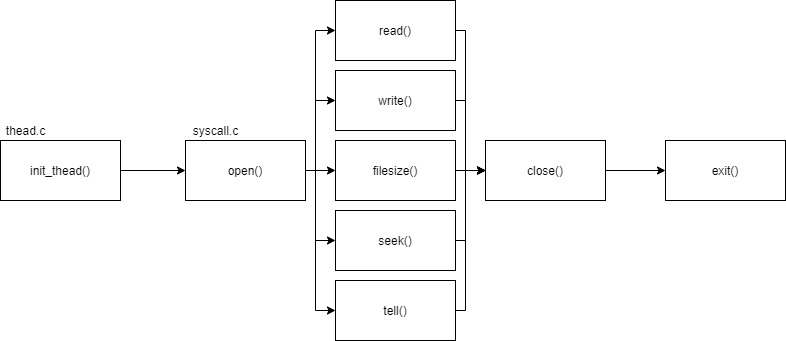
- seek : syscall.c에 void seek(int fd, unsigned position)을 추가하였고 file.c의 file\_seek()을 이용해 기능을 구현하였다.

3. Synchronization in Filesystem

thread.h의 thread 구조체에 struct semaphore load\_lock을 추가하여 프로세스 시작부터 실행이 끝나기 전까지 sema로 제한을 걸어두었다. 또한 syscall.c에 struct lock lk을 선언하여 open, read, write의 citical section에 lock으로 묶어두어 race condition이 발생하지 않도록 하였다.

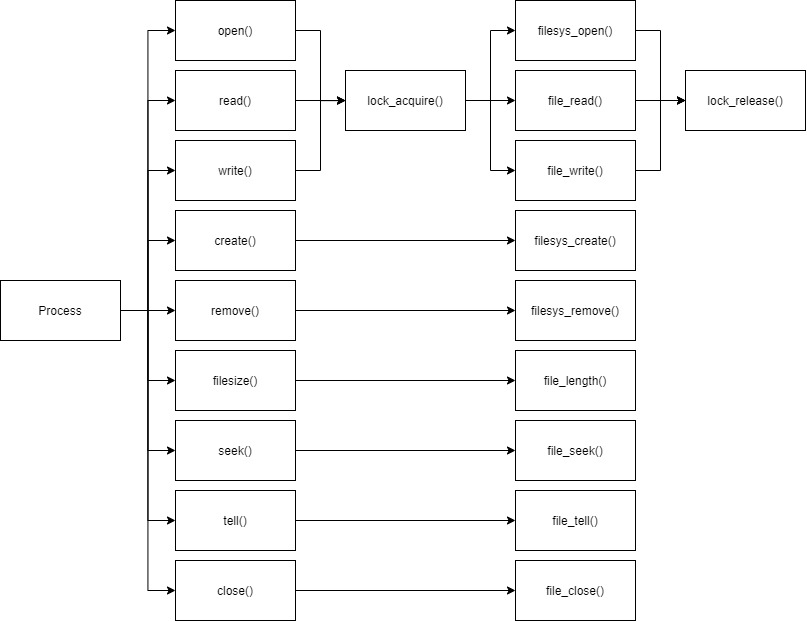
1. **연구 결과**
   1. **Flow Chart**

**1. File Descriptor**

****

file descriptor 배열을 초기화한뒤 open()에서 파일 포인터를 저장하고, system call 명령어에 따라 해당 파일에 접근하여 작업을 수행한 후 close() 및 exit()을 통해 file descriptor에 NULL값을 넣어 종료한다.

**2. System call & Synchronization**

****

프로젝트 1에서 구현한 system call은 제외하였다. file 관련 system call을 구현하였고, open, read, write에선 lock을 사용하여 syncronization을 맞춰주었다.

* 1. **제작 내용**

1. File Descriptor

프로세스가 open하는 파일은 각자 파일 디스크립터 넘버를 가진다. 이 디스크립터에 맞게 개별적으로 저장될 수 있게끔 thread의 구조체에 아래와 같은 파일 구조체 배열을 추가하여 파일 디스크립터를 구현하였다. 또한 현재 쓰레드에서 사용한 마지막 파일 디스크립터 넘버를 int fd에 저장할 수 있게 선언하였다.



이 파일 변수는 이번 프로젝트에서 구현할 system call의 파일 관련 명령어들을 처리하기 위해 파일에 접근할 때 사용되며 자세한 구현은 다음 system call에서 설명되어있다.

2. System calls

|  |
| --- |
| bool create(const char \*file, unsigned initial\_size) |
| if(file == NULL) exit(-1);  if(check\_address(file, thread\_current())) exit(-1);  return filesys\_create(file, initial\_size); |

입력받은 파일 파라미터 및 현재 쓰레드가 적절한 위치에 접근하고 있는지 확인한 후 filesys.c의 filesys\_create()함수를 사용하여 파일을 생성하였다.

|  |
| --- |
| int open(const char \*file) |
| struct file \*cur\_file;  struct thread \*cur\_thread;  int temp;  if(file == NULL) exit(-1);  if(check\_address(file, thread\_current())) exit(-1);  lock\_acquire(&lk);  cur\_file = filesys\_open(file);  cur\_thread = thread\_current();  if(cur\_file == NULL){  temp = -1;  }  else{  if((cur\_thread->fd) >= 128) {  lock\_release(&lk);  return -1;  }    cur\_thread->file\_info[cur\_thread->fd++] = cur\_file;  temp = (cur\_thread->fd)-1;  }  lock\_release(&lk);  return temp; |

파일 파라미터 및 쓰레드의 접근이 정상적인지 확인한 후 파일을 open하기 전 race condition을 막기 위해 lock을 걸어주었다. 그 다음 filesys.c의 filesys\_open()으로 파일을 열어 파일의 포인터를 반환 받고, 이 포인터가 NULL값이거나 파일 디스크립터의 접근이 비정상적일 경우 -1을 return하였다. 아무 이상이 없을 경우 open한 파일의 포인터를 file\_info에 저장하고 해당 디스크립터 넘버 값을 return한다. 모든 경우에서 return하기 직전에 lock을 풀어주었다.

|  |
| --- |
| void close(int fd) |
| struct thread \*cur\_thread;  cur\_thread = thread\_current();  if(fd < 3 || fd >= cur\_thread->fd) exit(-1);  if(cur\_thread->file\_info[fd] == NULL) exit(-1);  file\_close(cur\_thread->file\_info[fd]);  cur\_thread->file\_info[fd] = NULL; |

파일 디스크립터 값 및 해당 파일 디스크립터와 매핑된 file\_info에 저장된 값이 정상적인 접근인지 확인한 후 file.c의 file\_close()함수를 이용하여 파일을 close하였고 해당 file\_info에는 NULL값을 넣어 종료한다.

|  |
| --- |
| bool remove(const char \*file) |
| if(file == NULL) exit(-1);  if(check\_address(file, thread\_current())) exit(-1);  return filesys\_remove(file); |

입력받은 파일 파라미터 및 현재 쓰레드가 적절한 위치에 접근하고 있는지 확인한 후 filesys.c의 filesys\_remove()함수를 사용하여 파일을 삭제시켰다.

|  |
| --- |
| int read(int fd, void\* buffer, unsigned size) |
| unsigned int len = 0;  char c = 0;  int temp;  if(check\_address(buffer, thread\_current())) exit(-1);  lock\_acquire(&lk);  if(fd == 0){  while(1){  c = input\_getc();  ((char\*)buffer)[len] = c;  len++;  if(len >= size) break;  }  temp = len;  }  else if(fd >= 3 && fd < 128){  if(thread\_current()->file\_info[fd] == NULL){  exit(-1);  }    temp = file\_read(thread\_current()->file\_info[fd], buffer, size);  }  else{  temp = -1;  }  lock\_release(&lk);  return temp; |

파라미터 및 쓰레드가 정상적인지 체크한 뒤 버퍼에 읽기 전 lock을 걸어주었다. fd = 0인 경우는 프로젝트 1에서 구현한 stdin이고, 3 이상일땐 file.c의 file\_read()함수를 이용하여 파일을 읽는다. 읽은 byte 수를 반환하며 반환하기 직전에 lock을 풀어주고 종료한다.

|  |
| --- |
| int write(int fd, const void \*buffer, unsigned size) |
| int temp;  if(check\_address(buffer, thread\_current())) exit(-1);  lock\_acquire(&lk);    if(fd == 1){  putbuf(buffer, size);    temp = size;  }  else if(fd >= 3 && fd < 128){    if(thread\_current()->file\_info[fd] == NULL) {  lock\_release(&lk);  exit(-1);  }  if(thread\_current()->file\_info[fd]->deny\_write)  file\_deny\_write(thread\_current()->file\_info[fd]);  temp = file\_write(thread\_current()->file\_info[fd], buffer, size);  }  else{  temp = -1;  }  lock\_release(&lk);  return temp; |

파일의 비정상적인 접근을 체크한 뒤 lock을 걸고, fd가 1인 경우는 프로젝트 1에서 구현한 stdout이므로 fd가 3 이상인 경우를 고려한다. 이때 파일 디스크립터와 매핑된 file\_info를 검사하고, 이미 실행중인 thread에 또 쓰려고 접근하는 것을 막기 위한 file\_deny\_write를 사용하였다. 이 deny함수가 끝나면 접근 가능한 상태가 되므로 file\_allow\_write함수는 사용하지 않았다. 정상적으로 접근 가능할때 file.c의 file\_write()함수를 호출하여 파일에 쓰고, 쓴 byte 크기를 반환받아 return한다. return하기 직전에 lock을 풀어주고 종료한다.

|  |
| --- |
| int filesize(int fd) |
| struct thread \*cur\_thread;  cur\_thread = thread\_current();  if(fd < 3 || fd >= cur\_thread->fd) exit(-1);  if(cur\_thread->file\_info[fd] == NULL) exit(-1);  return file\_length(cur\_thread->file\_info[fd]); |

파일의 비정상적인 접근을 체크한 뒤 file.c의 file\_length()를 사용하여 파일의 크기를 반환한다.

|  |
| --- |
| unsigned tell(int fd) |
| struct thread \*cur\_thread;  cur\_thread = thread\_current();  if(fd < 3 || fd >= cur\_thread->fd) exit(-1);  if(cur\_thread->file\_info[fd] == NULL) exit(-1);  return file\_tell(cur\_thread->file\_info[fd]); |

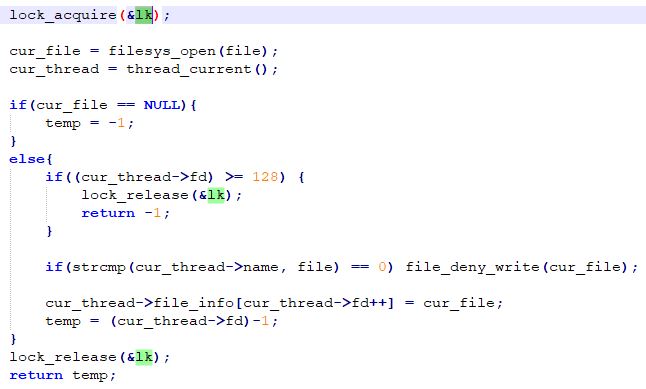
파일의 비정상적인 접근을 체크한 후 file.c의 file\_tell()을 사용하여 해당 파일의 커서 위치를 반환한다.

|  |
| --- |
| void seek(int fd, unsigned position) |
| struct thread \*cur\_thread;  cur\_thread = thread\_current();  if(fd < 3 || fd >= cur\_thread->fd) exit(-1);  if(cur\_thread->file\_info[fd] == NULL) exit(-1);  file\_seek(cur\_thread->file\_info[fd], position); |

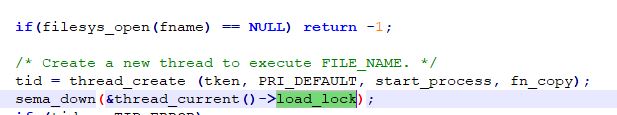
파일의 비정상적인 접근을 체크한 후 file.c의 file\_seek()을 사용하여 파라미터의 positon 위치로 커서를 이동시키고 해당 위치를 반환한다.

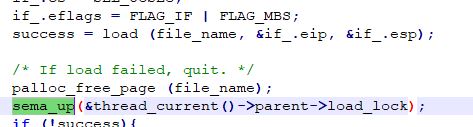
3. Synchronization in Filesystem

- 위의 system call파트에서 설명하였던것처럼 syscall.c에 struct lock lk를 선언하여 open(), read(), write()에서 lock을 acquire 및 release해주면서 critical section을 보호하였다.



- 그리고 thread 구조체에 load\_lock이라는 세마포를 추가해 load 전후의 synchronization을 맞춰주었다.

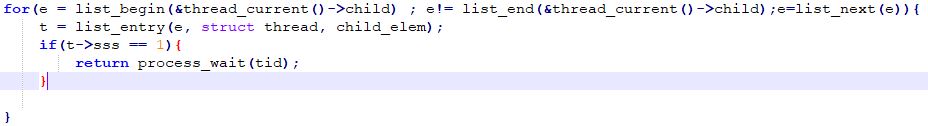
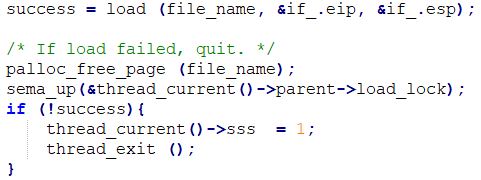
process\_execute()

start\_process()

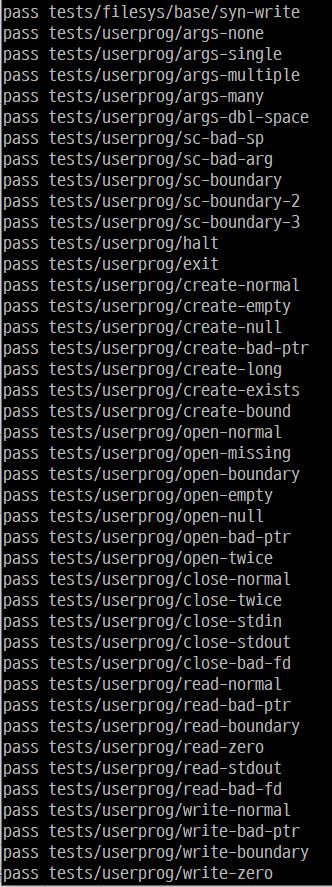
- 또한 write()에선 실행중인 쓰레드에 write하는 것을 막기위한 file\_deny\_write()를 사용하였다.



그리고 multi-oom 케이스를 해결하는데 있어서 메모리 누수를 막기 위해 start\_process()에서 load가 실패했을 경우 해당 프로세스에 wait을 걸어주어야 한다. 때문에 thread.h의 thread 구조체에 load 결과를 저장하는 int sss를 선언하여 start\_process()에서 load의 결과값이 반환된 success값을 확인하고, 이 값이 0일 경우 sss의 플래그를 세워주어 프로세스 실행시 이 플래그가 세워진 쓰레드는 wati이 되도록 제한을 걸어주었다.



* 1. **시험 및 평가 내용**



모든 case에 대해 PASS를 확인하였다.