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

담당 교수 / 분반 : 박성용 교수님 / 1반

이름 / 학번 : 황동준 / 20181703

개발 기간 : 2020-11-10 ~ 2020-11-16

1. **개발 목표**

이번 프로젝트에서는 파일 시스템과 관련된 시스템 콜을 구현할 예정이다. 즉, 추가적인 시스템 콜을 구현하는 것이 목적인데, create, remove, open, close, filesize, read, write, seek, tell이라는 시스템 콜을 만들고 이를 구현하여 OS에서 파일을 열거나 삭제하고 읽고 쓰는 역할을 담당하는 내용을 담을 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

File Descriptor을 만들면, 이는 해당 Thread에서 파일이 open되거나 close 될 때 파일을 받고 저장하는 역할을 수행하게 된다. 연 파일은 해당 Thread의 structure에 들어있는 file descriptor에 저장이 된다. 그리고 이 file descriptor을 구현하게 되면, STDIN, STDOUT 뿐만 아니라 다른 open 한 파일들을 읽고 쓸 때 쉽게 해당 Thread 안에서 파일을 찾을 수 있게 된다.

2. System Call

이번 System Call은 create, remove, open, close, filesize, read, write, seek, tell 총 9가지 이며 이를 구현할 경우 파일을 쉽게 관리하고 읽고 쓰게 될 수 있게 된다. Create는 파일을 불러서 새롭게 만들고 size만큼의 공간을 할당하고, open하지는 않은 상태로 file을 둘 것이다. Open이라는 시스템 콜을 통해 열리는 파일은 읽고 쓸 수 있는 형태가 된다. Remove는 말 그대로 file을 지울 수 있고, close는 파일을 닫고, filesize는 file의 크기를 return 해주며, read와 write는 파일의 내용을 보거나 쓸 수 있다. Seek은 응용프로그램이 파일 디스크립터의 포인터를 바꾸어 파일의 새 위치에서 파일을 읽고 쓸 수 있게 해주며, tell은 파일이 읽고 쓰게 될 다음 위치를 return할 수 있게 된다.

3. Synchronization file system

이번 파일 시스템에서는 서로 다른 process나 thread가 같은 파일에 접근하지 못하도록 하는 것이 큰 관건이다. 즉 read, write할 내용이 겹치거나 (read, read면 상관없음) 동시에 진입할 수 없는 Section인 Critical Section이 존재할 경우 Synchronization을 해주어야 한다는 의미이다. 이를 통해서 다른 Thread가 동시에 같은 파일에 접근할 수 없도록 만들 것이다.

* 1. **개발 내용**

1. File Descriptor

file.h에 이미 선언 되어있는 struct file을 이용할 것이며, 이는 inode를 통해서 file을 저장하게 되고, file이 지금 읽고 쓰고 있는 위치와 deny\_write를 선언하여 lock을 걸어주어 synchronization을 필요로 하는 시스템 콜에 이를 이용할 것이다. 또한 file Descriptor는 파일을 관리하는 Thread에 추가될 것이며, STDIN은 첫번째 index (0), STDOUT은 두번째 index(1)에 접근하게 되고, index (2)는 STD\_ERR의 것이므로 나머지 부분은 최대 128개의 file Descriptor가 open되어 이용 될 수 있다.

2. System call

(1) Create

Create은 받은 파일의 크기와 파일 struct를 받고 이를 생성하고 할당하는 역할을 한다. Create 했을 당시에는 open까지 한 상태가 아니며, 그냥 create만 한 상태이다.

(2) Remove

Remove는 해당 파일을 받고 이를 제거하는 형식으로 이루어 진다. Open file이 제거되었을 경우 또한 close하지않고 삭제하며, 만약 정상적으로 지워졌을 경우 successful, 그렇지 않다면 false를 return하게 된다.

(3) open

Open은 file을 불러와서 read 혹은 write가 가능한 상태로 만드는 것이다. 그러기 위해서는 해당 thread의 file descriptor에 추가를 해야 하며, 만약 음수 값(-1)이 return 될 경우에는 파일 open에 실패 한 것이다.

(4) filesize

open해서 file descriptor에 추가되었던 파일 중에서 index를 입력받은 파일의 크기를 return한다.

(5) read

파일을 읽게 된다. 입력 값으로는 file descriptor의 index와 메모리 buffer, 읽을 파일의 크기가 들어오게 된다. 해당 파일의 내용이 저장되어있는 buffer에 접근해 read를 수행하게 된다.

(6) write

Open한 즉 file descriptor에 추가된 것들 중에서 write할 file의 index를 받아와 file의buffer안의 내용을 수정한다. 또한 쓸 부분의 크기가 들어오게 되며 이 크기보다 input이 작을 경우 return 값에 쓰지 않아서 남은 부분의 크기가 들어가게 된다.

(7) seek

위에서도 말했다시피 연 파일이 읽거나 쓸 새로운 위치를 찾아주게 된다.

(8) tell

파일이 읽고 쓰게 되는 다음 위치(byte)를 찾아준다.

3. Synchronization File System

전 프로젝트에서 했던 Semaphore을 이용하여 Child가 EXECUTE 할 때 부모가 기다리는 방식과 비슷하게 이를 이용할 것이다. Open과 read, Write에 lock을 걸어줄 것인데, 이때 lock은 Semaphore이 0과 1의 값만 가질 수 있을 때를 의미하므로 나는 이를 구현할 때 lock을 걸어주어 서로 다른 process나 Thread가 같은 파일에 접근할 수 없도록 할 것이다. 또한 우리는 전 프로젝트에서 걸어주지 못했던 parent load lock에 대해서도 구현하게 될 것이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2020-11-10 pintos manual을 읽어보고 개발 구현에 대해 comment를 각각 남긴다.

2020-11-11 ~ 2020-11-13 file descriptor와 system call중 create, remove, open, close, filesize, seek, tell을 만든다. 이때 open에는 아직 lock을 걸어주지 않는다.

2020-11-13 ~ 2020-11-15 read와 write를 만들고 여기에 open까지 lock을 걸어주어 synchronize가 될 수 있도록 system call을 구현한다.

2020-11-15 ~ 2020-11-16 모든 과제를 해결한 후 보고서를 작성한다.

* 1. **개발 방법**
* 수정해야하는 소스코드

1. 첫번째 과정인 file Descriptor와 read, write를 제외한 system call을 만들 때에는 먼저 file descriptor을 저장하는 변수 131개를 선언해준다. 3개는 Standard in, out, error고 나머지는 최대로 허용되는 file descriptor 128개이다. 이미 create, filesize(file\_length), seek, tell, remove, close 같은 경우 받아온 pointer가 valid 한지 check하고 기존에 있던 함수를 이용하면 쉽게 구현할 수 있을 것이다. 실행하여 얻은 return 값들을 f->eax에 잘 넣어 주기만 하면 구현이 가능하다.
2. 두번째 과정은 read, write를 구현하고 여기에 open까지 lock을 걸어주어야 한다. 따라서, 이 또한 filesys와 file 파일에 주어진 함수를 이용하면 되지만, 만약 READ와 WRITE의 경우 Standard in, out이 들어왔을 때는 각각 input\_getc, putbuf를 이용하여 키보드에서 받은 정보를 직접 읽거나 쓰는 것이고, file descriptor의 index가 들어왔을 경우를 처리해 주기 위해서 index의 valid를 check하고 이를 함수에 넣은 후 return값을 f->eax에 넣어주기만 하면 된다. Open, Read, Write는 각각 lock을 걸어주어야 하는데, 즉 file과 관련된 함수를 이용하기 전에 lock\_acquire라는 주어진 함수를 이용하여 lock을 걸어주어 한 thread가 한 파일에 대해 open, read, write과정을 처리해 주고 있을 때 다른 thread가 접근을 할 수 없도록 해준다. 또한 file과 관련된 일이 끝났을 때에는 lock\_release라는 주어진 함수를 이용하여 lock을 풀어준다. Lock이라는 struct를 정의하고 init할 때에는 system call 내부에서만 일어나면 되므로, file\_lock이라는 lock struct를 선언하여 이를 syscall\_init에서 초기화 할 수 있도록 한다. 또한 parent라는 struct thread \* 변수를 추가하고 이를 이용해서 child가 create되고 start process를 할 때 까지 parent에 lock을 걸어주어 전 프로젝트에서 구현했던 execute를 추가 점검 할 수 있도록 한다. 그리고 write에서는 주어진 함수인 file\_deny\_write를 이용하여 write들 끼리는 절대 관여할 수 없도록 flag를 설정한다.

* 수정해야 하는 자료구조

|  |
| --- |
| struct thread |
| |  | | --- | | struct thread | |  | { | |  | /\* Owned by thread.c. \*/ | |  | tid\_t tid; /\* Thread identifier. \*/ | |  | enum thread\_status status; /\* Thread state. \*/ | |  | char name[16]; /\* Name (for debugging purposes). \*/ | |  | uint8\_t \*stack; /\* Saved stack pointer. \*/ | |  | int priority; /\* Priority. \*/ | |  | struct list\_elem allelem; /\* List element for all threads list. \*/ | |  |  | |  | /\* Shared between thread.c and synch.c. \*/ | |  | struct list\_elem elem; /\* List element. \*/ | |  |  | |  | #ifdef USERPROG | |  | /\* Owned by userprog/process.c. \*/ | |  | uint32\_t \*pagedir; | |  | struct semaphore child\_lock; | |  | struct semaphore mem\_lock; | |  | struct semaphore load\_lock; | |  | struct thread \*parent; | |  | struct list child; | |  | struct list\_elem child\_elem; | |  | int exit\_status; | |  | int wait\_status; | |  | struct file \*fn[131]; | |  | /\* Page directory. \*/ | |  | #endif | |  |  | |  | /\* Owned by thread.c. \*/ | |  | unsigned magic; /\* Detects stack overflow. \*/ | |  | }; | |

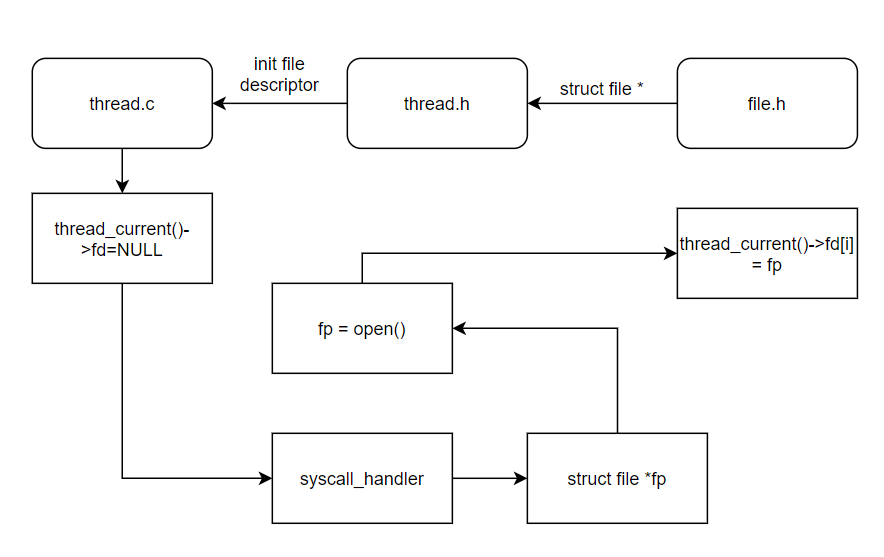
* 수정해야하는 함수

init\_thread, syscall\_handler, exit & (page\_fault, process\_execute, start\_process)

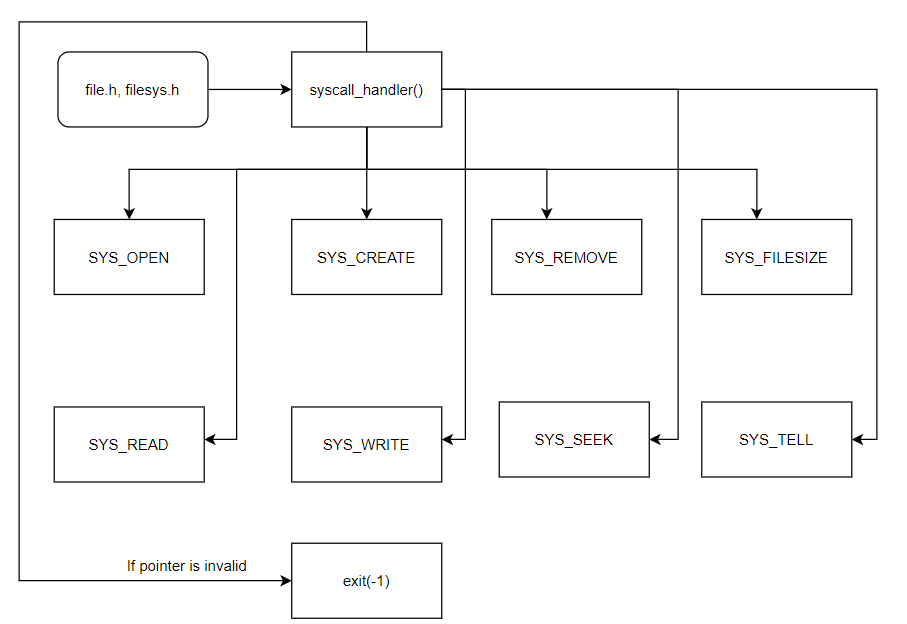
● 추가해야하는 함수

없음

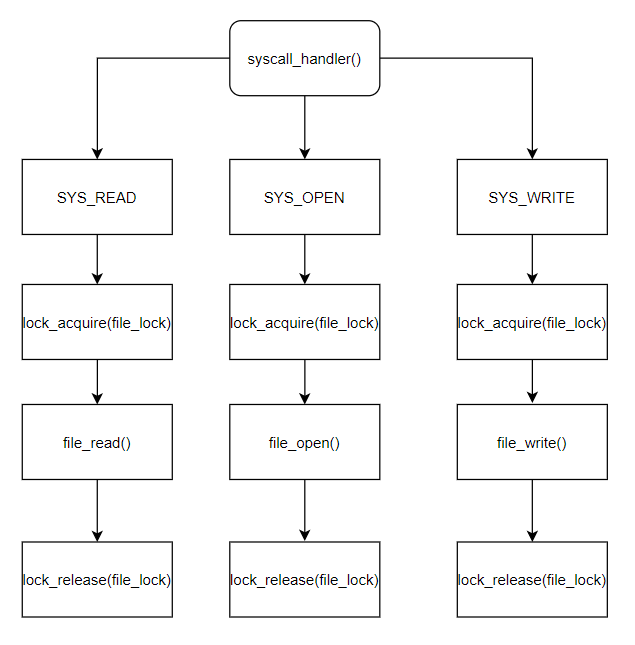
1. **연구 결과**
   1. **Flow Chart**
2. File Descriptor

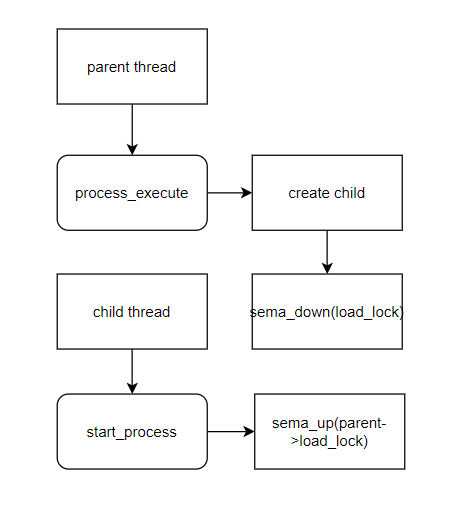


1. System Call



1. Synchronization File System





* 1. **제작 내용**

1. file descriptor

|  |
| --- |
| thread.h |
| struct thread  {  /\* Owned by thread.c. \*/  tid\_t tid; /\* Thread identifier. \*/  enum thread\_status status; /\* Thread state. \*/  char name[16]; /\* Name (for debugging purposes). \*/  uint8\_t \*stack; /\* Saved stack pointer. \*/  int priority; /\* Priority. \*/  struct list\_elem allelem; /\* List element for all threads list. \*/  /\* Shared between thread.c and synch.c. \*/  struct list\_elem elem; /\* List element. \*/  #ifdef USERPROG  /\* Owned by userprog/process.c. \*/  uint32\_t \*pagedir;  struct semaphore child\_lock;  struct semaphore mem\_lock;  struct semaphore load\_lock;  struct thread \*parent;  struct list child;  struct list\_elem child\_elem;  int exit\_status;  int wait\_status;  struct file \*fn[131];  /\* Page directory. \*/  #endif  /\* Owned by thread.c. \*/  unsigned magic; /\* Detects stack overflow. \*/  }; |
| struct file \*fn[131]을 선언하여 file descriptor을 저장할 공간 131개를 만들어 주었다. 또한 이를 초기화 하는 내용은 아래의 코드에 나와있다. |

|  |
| --- |
| static void  init\_thread (struct thread \*t, const char \*name, int priority) |
| for(int i=0;i<131;i++)  t->fn[i] = NULL; |
| struct thread \* t를 thread\_current로 선언하여 현재 thread의 file descriptor 상태를 NULL로 초기화 해주었다. |

|  |
| --- |
| syscall\_handler() |
| struct file \*fp;  //open  fp = filesys\_open(\*(char \*\*)(f->esp+4));  thread\_current()->fn[i] = fp;  //filesize, read, write, seek, tell, close  fp = thread\_current()->fn[\*(int \*)(f->esp+4)]; |
| 다음과 같이 file descriptor을 이용하여 struct file을 해당 함수의 전달인자로 넣게 되었다. |

* 이용한 라이브러리 및 함수

|  |
| --- |
| file.h |
| struct file  {  struct inode \*inode; /\* File's inode. \*/  off\_t pos; /\* Current position. \*/  bool deny\_write; /\* Has file\_deny\_write() been called? \*/  }; |
| 정해진 struct를 이용하여 file descriptor을 선언하였다. |

● 자체제작 함수

없음

● 개발 중 발생한 이슈

없음

1. System Call

|  |
| --- |
| syscall\_handler() |
| //2 : create, remove, open, close, filesize, read, write, seek, tell  case SYS\_CREATE:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr((\*(char\*\*)(f->esp+4))) || pagedir\_get\_page(t->pagedir, (\*(char \*\*)(f->esp+4))) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(\*(char \*\*)(f->esp+4) == NULL) exit(-1);  f->eax = filesys\_create(\*(char \*\*)(f->esp+4), \*(off\_t \*)(f->esp+8));  break;  case SYS\_REMOVE:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr((\*(char\*\*)(f->esp+4))) || pagedir\_get\_page(t->pagedir, (\*(char \*\*)(f->esp+4))) == NULL)  exit(-1);  if(\*(char \*\*)(f->esp+4) == NULL) exit(-1);  f->eax = filesys\_remove(\*(char \*\*)(f->esp+4));  break;  case SYS\_OPEN:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(\*(char\*\*)(f->esp+4)) || pagedir\_get\_page(t->pagedir, \*(char \*\*)(f->esp+4)) == NULL)  exit(-1);  if(\*(char \*\*)(f->esp+4) == NULL) exit(-1);  f->eax = -1;  lock\_acquire(&file\_lock);  fp = filesys\_open(\*(char \*\*)(f->esp+4));  if(fp == NULL) f->eax = -1;  else{  for(int i=3;i<131;i++){  if(thread\_current()->fn[i] == NULL){  if(!strcmp(thread\_current()->name, \*(char \*\*)(f->esp+4))) file\_deny\_write(fp);  thread\_current()->fn[i] = fp;  f->eax = i;  break;  }  }  }  lock\_release(&file\_lock);  break;  case SYS\_FILESIZE:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  fp = thread\_current()->fn[\*(int \*)(f->esp+4)];  if(fp == NULL) exit(-1);  else f->eax = file\_length(fp);  break;  case SYS\_READ:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+12) || pagedir\_get\_page(t->pagedir, f->esp+12) == NULL)  exit(-1);  if(!is\_user\_vaddr((\*(char \*\*)(f->esp+8))) || pagedir\_get\_page(t->pagedir, (\*(char \*\*)(f->esp+8))) == NULL)  exit(-1);  f->eax = -1;//initialize  lock\_acquire(&file\_lock);  if((int)\*(uint32\_t \*)(f->esp+4) == 0){  for(int i = 0;i< \*(int \*)(f->esp+12);i++)  (\*(char \*\*)(f->esp+8))[i] = input\_getc();  f->eax = \*(int \*)(f->esp+12);  }  else if((int)\*(uint32\_t \*)(f->esp+4) > 2){  if(thread\_current()->fn[\*(int \*)(f->esp+4)] == NULL){  lock\_release(&file\_lock);  exit(-1);  }  f->eax = file\_read(thread\_current()->fn[\*(int \*)(f->esp+4)], \*(void \*\*)(f->esp+8), \*(off\_t \*)(f->esp+12));  }  lock\_release(&file\_lock);  break;  case SYS\_WRITE:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+12) || pagedir\_get\_page(t->pagedir, f->esp+12) == NULL)  exit(-1);  if(!is\_user\_vaddr((void \*)\*(uint32\_t \*)(f->esp+8)) || pagedir\_get\_page(t->pagedir, (void \*)\*(uint32\_t \*)(f->esp+8)) == NULL)  exit(-1);  f->eax = -1;  lock\_acquire(&file\_lock);  if((int)\*(uint32\_t \*)(f->esp+4) == 1){  putbuf((void \*)\*(uint32\_t \*)(f->esp+8), \*(size\_t \*)(f->esp+12));  }  else if((int)\*(uint32\_t \*)(f->esp+4) > 2){  if(thread\_current()->fn[\*(int \*)(f->esp+4)] == NULL){  lock\_release(&file\_lock); exit(-1);  }  if(thread\_current()->fn[\*(int \*)(f->esp+4)]->deny\_write)  file\_deny\_write(thread\_current()->fn[\*(int \*)(f->esp+4)]);  f->eax = file\_write(thread\_current()->fn[\*(int \*)(f->esp+4)], \*(void \*\*)(f->esp+8), \*(off\_t \*)(f->esp+12));  }  lock\_release(&file\_lock);  break;  case SYS\_SEEK:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(thread\_current()->fn[\*(int \*)(f->esp+4)] == NULL) exit(-1);  file\_seek(thread\_current()->fn[\*(int \*)(f->esp+4)], \*(off\_t \*)(f->esp+8));  break;  case SYS\_TELL:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(thread\_current()->fn[\*(int \*)(f->esp+4)] == NULL) exit(-1);  f->eax = file\_tell(thread\_current()->fn[\*(int \*)(f->esp+4)]);  break;  case SYS\_CLOSE:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  //if((\*(int \*)(f->esp+4))<0||(\*(int \*)(f->esp+4))>=131) exit(-1);  if(thread\_current()->fn[\*(int \*)(f->esp+4)] == NULL) exit(-1);  //lock\_acquire(&file\_lock);  file\_close(thread\_current()->fn[\*(int \*)(f->esp+4)]);  thread\_current()->fn[\*(int \*)(f->esp+4)] = NULL;//close file  //lock\_release(&file\_lock);  break; |
| open과 read, write를 제외한 나머지 영역에서는 거의 return값이 있는 경우 f->eax에 저장해 주고, 함수는 filesys, file에 이미 선언 되어있는 함수들을 이용하였다. 대부분 함수에 들어갈 esp(pointer)는 invalid 검사를 하여 만약 쓸 수 없는 메모리 영역에 침범하는 경우 exit(-1)로 thread를 강제종료 시키도록 하였고 이는 모든 system call에 해당한다.  open과 read, write는 synchronization 방법을 아래에서 설명할 것이다. Read, write은 standard input이 들어오는 것에 대비하여 if문을 만들어 standard input, output이 들어가는 경우를 따로 처리해주고 나머지는 file descriptor의 값을 넣었으며, 이때 또한 filesys, file의 기존에 있던 함수를 이용하여 file을 읽고 썼다. |

* 이용한 라이브러리 및 함수

|  |
| --- |
| filesys.h |
| bool filesys\_create (const char \*name, off\_t initial\_size);  struct file \*filesys\_open (const char \*name);  bool filesys\_remove (const char \*name); |
| create나 remove같은 경우 파일이 만들어지거나 제거하는 것에 성공했을 때 true값을 return하고 아닐 시 false값을 return하게 된다. Filesys\_open은 입력받은 파일의 name을 가지고 해당 이름을 가진 파일을 dir\_lookup(open)하고 이를 반환한다. 이때 file\_open이라는 또 다른 함수를 함수 내에서 쓰게 된다. |

|  |
| --- |
| file.h |
| struct file \*file\_open (struct inode \*);  void file\_close (struct file \*);  off\_t file\_read (struct file \*, void \*, off\_t);  off\_t file\_write (struct file \*, const void \*, off\_t);  void file\_seek (struct file \*, off\_t);  off\_t file\_tell (struct file \*);  off\_t file\_length (struct file \*); |
| 다음은 해당하는 system call에서 쓰게 되는 라이브러리 내장 함수이다. Struct file \* 부분에서는 file descriptor가 들어가게 되고, 나머지 부분에는 pointer에서 받아온 값들이 적절히 들어가야 한다. File\_open은 filesys\_open 안의 함수에서 쓰게 되는 것이고, file\_close 또한 close에서도 이용하지만 나중에 개발 중 발생한 multi-oom issue 때문에 이용하는 부분이다. (이 issue는 synchronization에서 설명할 것이다.) |

● 자체제작 함수

없음

● 개발 중 발생한 이슈

Page\_fault에 관련된 내용으로, 쉽게 page fault가 작동하여 thread를 킬하고 커널 패닉이 발생하면서 생기는 문제를 exception.c에서 몇가지 조건을 추가하여 해결하였다.

|  |
| --- |
| exception.c |
| if(!user || is\_kernel\_vaddr(fault\_addr) || not\_present) exit(-1); |
| 만약 page\_fault가 일어났을 경우 우리는 fault\_addr에서 page fault가 일어난 address를 찾을 수 있고, interrupt frame의 error\_code를 이용해서 write나 user의 문제인지 확인할 수 있다. 보통 일어나는 문제가 page fault가 떠서 정상종료하지 않고 강제로 kill을 해버리니까 file을 제대로 close하지 못하고 종료되거나 status를 -1로 제대로 띄우지 않고 종료하는 경우가 많았다. 따라서 page fault가 나서 강제 종료하기 직전에 위와 같이 오류가 난 부분을 catch하여 정상종료 시킨다. |

1. Synchronization File System

|  |
| --- |
| thread.c, thread.h, syscall.h, syscall\_init() |
| struct thread  int wait\_status;  struct semaphore load\_lock;  struct thread \*parent;  static void  init\_thread (struct thread \*t, const char \*name, int priority)  t->parent = running\_thread();  sema\_init(&(t->load\_lock), 0);  t->wait\_status = 0;  void  syscall\_init (void)  lock\_init(&file\_lock);  struct lock file\_lock; |
| synchronization을 위하여 선언된 변수와 struct들이며 전부 아래 소스코드에서 이용되게 된다. |

|  |
| --- |
| syscall\_handler() |
| case SYS\_OPEN:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(\*(char\*\*)(f->esp+4)) || pagedir\_get\_page(t->pagedir, \*(char \*\*)(f->esp+4)) == NULL)  exit(-1);  if(\*(char \*\*)(f->esp+4) == NULL) exit(-1);  f->eax = -1;  lock\_acquire(&file\_lock);  fp = filesys\_open(\*(char \*\*)(f->esp+4));  if(fp == NULL) f->eax = -1;  else{  for(int i=3;i<131;i++){  if(thread\_current()->fn[i] == NULL){  if(!strcmp(thread\_current()->name, \*(char \*\*)(f->esp+4))) file\_deny\_write(fp);  thread\_current()->fn[i] = fp;  f->eax = i;  break;  }  }  }  lock\_release(&file\_lock);  break;  case SYS\_READ:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+12) || pagedir\_get\_page(t->pagedir, f->esp+12) == NULL)  exit(-1);  if(!is\_user\_vaddr((\*(char \*\*)(f->esp+8))) || pagedir\_get\_page(t->pagedir, (\*(char \*\*)(f->esp+8))) == NULL)  exit(-1);  f->eax = -1;//initialize  lock\_acquire(&file\_lock);  if((int)\*(uint32\_t \*)(f->esp+4) == 0){  for(int i = 0;i< \*(int \*)(f->esp+12);i++)  (\*(char \*\*)(f->esp+8))[i] = input\_getc();  f->eax = \*(int \*)(f->esp+12);  }  else if((int)\*(uint32\_t \*)(f->esp+4) > 2){  if(thread\_current()->fn[\*(int \*)(f->esp+4)] == NULL){  lock\_release(&file\_lock);  exit(-1);  }  f->eax = file\_read(thread\_current()->fn[\*(int \*)(f->esp+4)], \*(void \*\*)(f->esp+8), \*(off\_t \*)(f->esp+12));  }  lock\_release(&file\_lock);  break;  case SYS\_WRITE:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+12) || pagedir\_get\_page(t->pagedir, f->esp+12) == NULL)  exit(-1);  if(!is\_user\_vaddr((void \*)\*(uint32\_t \*)(f->esp+8)) || pagedir\_get\_page(t->pagedir, (void \*)\*(uint32\_t \*)(f->esp+8)) == NULL)  exit(-1);  f->eax = -1;  lock\_acquire(&file\_lock);  if((int)\*(uint32\_t \*)(f->esp+4) == 1){  putbuf((void \*)\*(uint32\_t \*)(f->esp+8), \*(size\_t \*)(f->esp+12));  }  else if((int)\*(uint32\_t \*)(f->esp+4) > 2){  if(thread\_current()->fn[\*(int \*)(f->esp+4)] == NULL){  lock\_release(&file\_lock); exit(-1);  }  if(thread\_current()->fn[\*(int \*)(f->esp+4)]->deny\_write)  file\_deny\_write(thread\_current()->fn[\*(int \*)(f->esp+4)]);  f->eax = file\_write(thread\_current()->fn[\*(int \*)(f->esp+4)], \*(void \*\*)(f->esp+8), \*(off\_t \*)(f->esp+12));  }  lock\_release(&file\_lock);  break; |
| 먼저 이 syscall\_handler에서 synchronization을 위해 처리해준 부분은 lock과 file\_deny\_write 부분이다. Lock은 file에 접근하는 부분 전 후에 선언을 해주어 다른 thread가 서로 다른 syscall을 선언할 때 같은 파일에 접근하지 못하도록 한다. 그래서 file에 접근하는 영역이 Critical Section이 되는 것이다. 그리고 또한 서로 다른 thread에서 같이 write에 접근할 수도 있으므로 만약 deny\_write부분이 활성화가 되어 있는 경우 즉, 실행가능한 thread에 write하는 것을 막아야 하는 경우에는 deny\_write 함수를 실행한다. 실행가능한 thread라는 것은 open이 된 file의 thread인 것이므로, 이 thread의 해당 file descriptor에 file\_deny\_write을 걸어주며, 만약 이 write 작업을 시작하기 전에 deny\_write 변수의 값이 true인지를 check하고 file\_deny\_write 함수를 실행시킨다. |

|  |
| --- |
| process.c |
| tid\_t  process\_execute (const char \*file\_name)  sema\_down(&thread\_current()->load\_lock);//for synchronization  //////////////////////////////////////  static void  start\_process (void \*file\_name\_)  sema\_up(&thread\_current()->parent->load\_lock); |
| thread\_create하였을 때 child를 load하기 전에 parent가 끝나버린 경우를 처리하는 부분이다. 이 때문에 syn-read, syn-wrtie에서 계속 child의 status가 이상하게 나온다는 것을 알 수 있었다. 즉 정상종료가 안되고 parent가 이미 끝나버려 create 된 채 그냥 계속 존재했던 것이다. 따라서 parent에서 child process를 create 할 때 load\_lock을 걸어주고 child process가 start process를 할 때 child의 parent load\_lock을 풀어준다. 그러면 child가 생성되고 나서 바로 parent가 죽어버려 아무것도 못하는 일이 발생하지 않는다. 이는 개발 중 발생한 이슈에 포함하려 했지만 그렇게 처리하지 않았다. |

* 이용한 라이브러리 및 함수

|  |
| --- |
| synch.h |
| struct semaphore  {  unsigned value; /\* Current value. \*/  struct list waiters; /\* List of waiting threads. \*/  };  void sema\_init (struct semaphore \*, unsigned value);  void sema\_down (struct semaphore \*);  void sema\_up (struct semaphore \*);  struct lock  {  struct thread \*holder; /\* Thread holding lock (for debugging). \*/  struct semaphore semaphore; /\* Binary semaphore controlling access. \*/  };  void lock\_init (struct lock \*);  void lock\_acquire (struct lock \*);  void lock\_release (struct lock \*); |
| 이 함수들을 이용하여 lock, semaphore를 걸어주고 synchronization하게 file system을 관리할 수 있었다. 이는 전부 위 소스코드에서 적절히 이용되었다. |

● 자체제작 함수

없음

● 개발 중 발생한 이슈

|  |
| --- |
| multi-oom 해결 1 🡪 process.c |
| static void  start\_process (void \*file\_name\_)  if (!success){  //thread\_exit ();  thread\_current()->wait\_status = 1;  exit(-1);  //thread\_exit();  }  tid\_t  process\_execute (const char \*file\_name)  for(e = list\_begin(&(thread\_current()->child)); e != list\_end(&(thread\_current()->child));e = list\_next(e)){  t = list\_entry(e, struct thread, child\_elem);  if(t->wait\_status == 1) return process\_wait(tid);  } |
| 먼저, child가 load될 때 제대로 load에 실패하였을 때 status가 -1로 뜨지 않는 현상은 start\_process에서 success가 child가 load에 성공했는지의 여부를 묻는 그런 변수인데, 만약 false일 경우 exit(-1)로 하여 정상적으로 status가 들어갈 수 있게 끔 해주었다.  두번째로 종료된 모든 child를 검사한 결과 wait의 status가 -1이 뜨지 않았는데 강제종료한 child가 wait를 통해서 정상종료되지 못하여 wait status가 계속 -1이 나온 것이다. 따라서 parent의 child list를 전부 돌면서 load에 실패하여 exit(-1)한 친구들을 전부 불러와 process\_wait를 시켜 wait status까지 정상적으로 만들어 준다. |

|  |
| --- |
| multi-oom 해결 2 🡪 process.c & syscall.c |
| tid\_t  process\_execute (const char \*file\_name)  char \*file\_name\_ptr, \*ptr\_tok;  char temp\_name[130];  //file\_name\_ptr = palloc\_get\_page (0);  strlcpy(temp\_name, file\_name, strlen(file\_name)+1);  file\_name\_ptr = strtok\_r(temp\_name, " ", &ptr\_tok);  if(filesys\_open(file\_name\_ptr) == NULL) return -1;  void exit(int status)  //multi-oom  for(int i=0;i<131;i++){  if(t->fn[i] != NULL){  file\_close(t->fn[i]);  t->fn[i] = NULL;  }  } |
| 위의 나머지는 다 해결이 되었는데, 계속 exit status가 51이 아니라 37이 뜨는 것이다. 그래서 비정상 종료된 친구들의 file을 닫아주지 않아 생긴거라고 생각해서 exit에 항상 모든 파일을 검사해서 file\_close를 해주었다. 그랬더니 exit status가 46이 계속 발생했다. 이때 argument passing에서 어떤 char형 포인터를 정적배열로 선언해주지 않고 palloc\_get\_page로 선언을 한 것이 기억났다. 이를 free조차 시켜주지 않았기에 그냥 정적으로 선언해 버려서 문제를 해결했다. |

* 1. **시험 및 평가 내용**

