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

담당 교수 / 분반 : 박성용 교수님 / 1반

이름 / 학번 : 백인찬 / 20150195

개발 기간 : 2020.11.03 ~ 2020.11.16

1. **개발 목표**

* 저번 프로젝트에서 구현한 System call을 바탕으로 이전에 구현하지 않았던 System call인 create, remove, open, close, seek, tell, filesize를 구현한다. 또 Reader/Writer 문제를 해결하기 위해 여러 thread들 간의 synchronization을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
   * Files to be modified and referenced

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

1. **File Descriptor**

핀토스를 포함해 유닉스 계열의 시스템에선 프로세스가 파일을 다룰 때 File Descriptor를 사용한다. 각 thread는 독립적인 File Descriptor를 갖고 관리하기 때문에 각 thread 구조체에 File Descriptor를 관리할 수 있는 멤버 변수를 추가한다.

1. **System Calls**

이번 프로젝트에서 구현해야 할 파일과 관련된 System call인 create, remove, open, close, filesize, seek, tell을 구현한다. 또 이전에 구현했던 read와 write에 대해 STDIN, STDOUT 뿐 아니라 인자로 받은 File Descriptor에 대해서도 입출력을 수행할 수 있도록 수정한다.

**3. Synchronization in Filesystem**

실행 중인 파일에 대해 다른 thread에서의 변경을 막는 Synchronization을 위해 file\_deny\_write(), file\_deny\_allow()를 적절히 사용해 구현한다. 또 여러 thread가 생성되었을 경우 여러 원인으로 인한 orphan / zombie thread의 발생을 semaphore를 통해 막도록 구현한다.

* 1. **개발 내용**

1. **File Descriptor:**

* File Descriptor(이하 FD)를 구현하기 위해 thread 구조체의 정의부에 struct File\* files[128]이라는 배열을 추가하였다. 배열 자료구조를 사용한 것은 Linked List에 비해 메모리의 낭비가 있을 수 있지만 순회가 간단하고, free()를 적절히 하지 않음으로 인한 메모리 누수의 위험이 적기 때문이다. FD 0, 1, 2는 OS에 의해 STDIN, STDOUT, STDERR로 관리되고 있기 때문에 3번부터 사용할 수 있도록 한다.

1. **System Calls:**

* create() : 인자로 파일의 이름과 파일의 크기를 받아, filesys\_create를 호출해 해당 이름을 갖는 파일을 생성한다. 파일 생성의 성공 여부를 true/false로 반환한다.
* remove() : 인자로 파일의 이름을 받아, filesys\_remove를 호출해 해당 이름을 갖는 파일을 삭제한다. 파일 삭제의 성공 여부를 true/false로 반환한다.
* open() : 인자로 파일의 이름을 받고 해당 이름을 갖는 파일을 filesys\_open을 호출해 연다. 해당 파일을 thread의 FD 배열의 빈 곳에 할당하고, 해당 FD를 반환한다.
* close() : 인자로 받은 fd에 해당하는 파일을 file\_close를 호출해 닫는 함수이다. 해당 파일을 닫은 뒤 thread의 FD 배열에서 fd에 해당하는 배열의 인덱스를 null을 대입해 닫아준다.
* filesize() : 인자로 받은 fd에 해당하는 파일의 크기를 file\_length를 호출해 반환한다.
* seek() : 인자로 fd와 위치를 받아, fd에 해당하는 파일을 찾아 그 위치를 두번째 인자의 위치로 옮긴다.
* tell() : 인자로 받은 fd에 해당하는 파일을 찾아 그 위치를 반환한다.
* read() : 지난 프로젝트에서는 인자인 fd가 0인 STDIN에 대해서만 입력을 받고 나머지에는 -1을 반환하도록 구현했는데, 특정 파일을 지정하는 fd에 대해서도 입력을 수행하도록 구현한다.
* write() : 지난 프로젝트에서 인자인 fd가 1인 STDOUT에 대해서만 출력을 하고 나머지에는 -1을 반환하도록 구현했는데, 특정 파일을 지정하는 fd에 대해서도 출력을 수행하도록 구현한다.

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

Synchronization을 위해 semaphore를 사용한다. 지난 프로젝트에서 구현한 semaphore 이외에 추가의 semaphore를 추가해 thread가 생성한 자식 thread들이 완전하게 종료할 수 있도록 한다.

Lock과 file\_deny\_write() 함수를 이용해서 실행중인 파일에 대해 출력을 수행해 변경할 수 없도록 한다. Lock을 통해 어떤 실행중인 파일에 어떤 thread가 특정 동작을 수행하고 있다면(Critical Section) 다른 thread는 그 파일에 접근해서 파일을 수정할 수 없도록 하고, 어떤 thread가 파일에 접근하고 있다면 file\_deny\_write을 통해 File 구조체의 deny\_write을 set해 해당 파일의 변경을 제한한다. 이 deny\_write은 thread가 종료될 때 file\_allow\_write을 통해 자동으로 unset된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11.3(화) ~ 11.10(화) : ppt / 강의자료 및 pintos manual 분석

11.11(수) ~ 11.12(목) : File Descriptor 및 System call 구현

11.13(금) ~ 11.15(일) : Synchronization in Filesystem 구현

11.16(월) : 디버깅 및 오류 수정

* 1. **개발 방법**
     1. **File Descriptor:**

**/threads/ thread.h thread.c**

* FD를 구현하기 위해 thread 구조체에 struct file\* files[128]이라는 배열을 추가했다. 이를 초기화하기 위해 thread를 초기화하는 함수 내에 각 배열을 NULL로 초기화하도록 했다.
  + 1. **System Calls:**

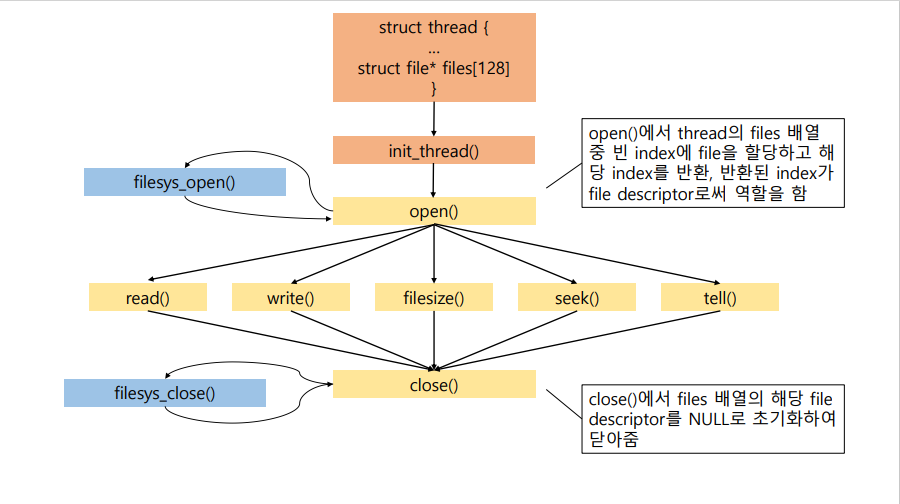
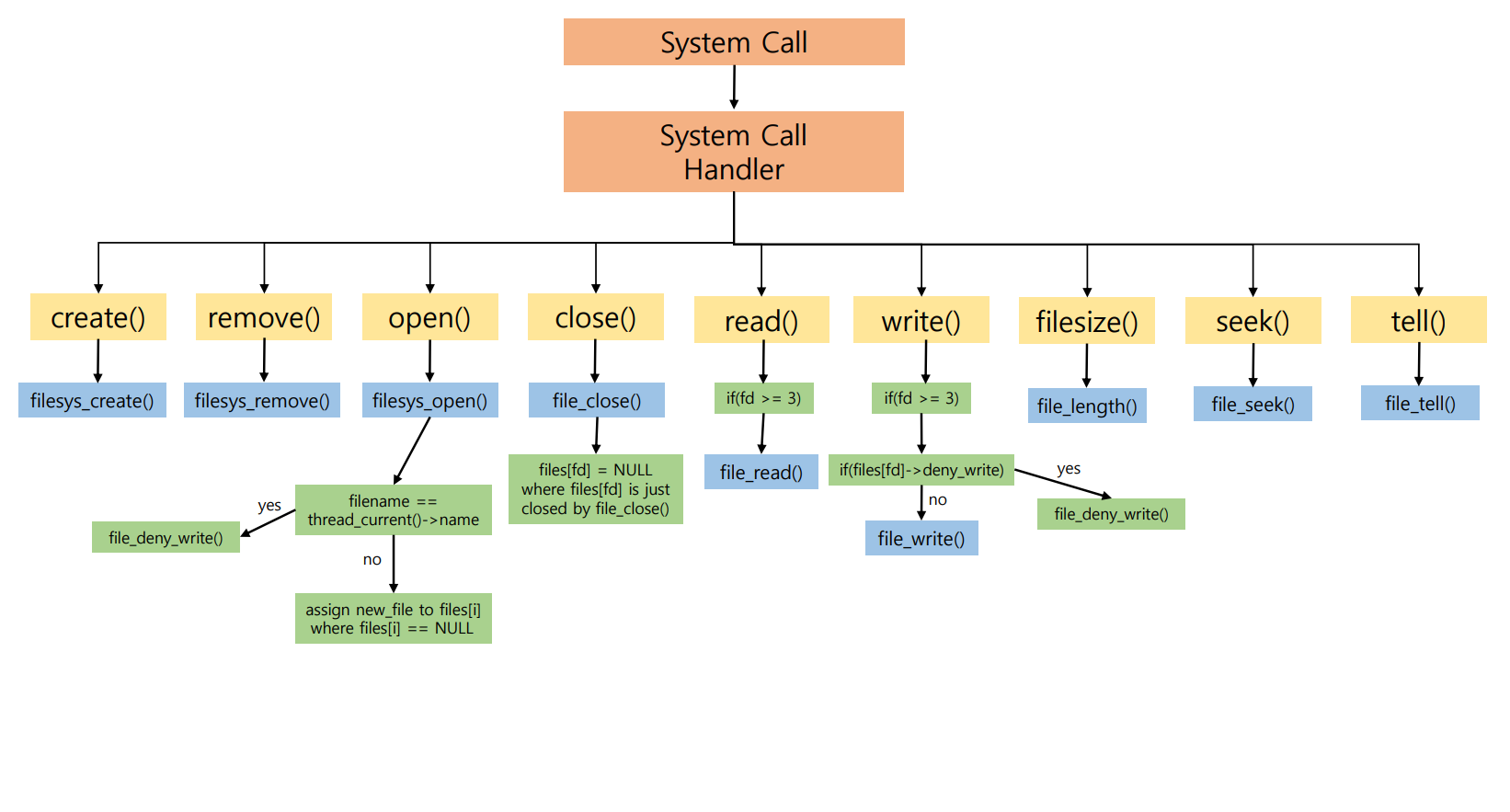
**/userprog/ syscall.c**

* create() : 파일 이름을 인자로 받고 그 인자가 NULL인지를 확인하여 잘못된 수행에 대해 exit(-1)을 호출하여 종료하도록 한다. 유효한 인자를 받았을 경우엔 filesys\_create함수를 호출하고 파일 이름과 파일의 크기를 인자로 넘겨 filesys\_create의 성공 여부인 Boolean 값을 return한다.
* remove() : 파일 이름을 인자로 받고 그 인자가 NULL인지를 확인하여 잘못된 수행에 대해 exit(-1)을 호출하여 종료하도록 한다. 유효한 인자를 받았을 경우엔 filesys\_remove함수를 호출하고 파일 이름을 인자로 넘긴다. filesys\_remove는 인자로 받은 파일 이름에 해당하는 파일을 찾아 삭제한 후 성공 여부인 Boolean 값을 반환하면 그 값을 remove()함수에서 받아 return한다.
* open() : 인자로 파일의 이름을 받아 그 인자가 NULL인지를 확인하여 잘못된 수행에 대해 exit(-1)을 호출하여 종료하도록 한다. 그 후 파일의 이름을 인자로 받는 filesys\_open을 호출하고, 해당 파일이 현재 열려 있는 파일인지 확인한다. 현재 열려 있는 파일이라면, 해당 파일이 변경되지 않게 하기 위해 file\_deny\_write 함수를 호출하여 변경을 제한한다. 그리고 현재 thread의 files 배열 중 NULL인 인덱스를 순회하며 찾아 그 인덱스에 새로 연 파일을 추가해준다.
* close() : 인자로 fd값을 받아 현재 thread의 files 배열내의 fd에 해당하는 파일이 유효한지 검사하고, 잘못된 인자일 경우 exit(-1)을 호출하여 종료하도록 한다. 유효한 인자라면 file\_close 함수를 호출해 해당 파일을 close한다. 또 현재 thread의 files 배열의 fd 인덱스에 있는 파일이 close 되었음을 명시하기 위해 해당 인덱스에 NULL 값을 대입한다.
* filesize() : 인자로 fd값을 받아 현재 thread의 files 배열내의 fd에 해당하는 파일이 유효한지 검사하고, 잘못된 인자일 경우 exit(-1)을 호출하여 종료하도록 한다. 유효한 인자라면 file\_length 함수를 호출해 file의 size를 받아 반환한다.
* seek() : 인자로 fd값과 위치를 받아 현재 thread의 files 배열내의 fd에 해당하는 파일이 유효한지 검사하고, 잘못된 인자일 경우 exit(-1)을 호출하여 종료하도록 한다. file\_seek 함수를 호출해 fd에 해당하는 파일의 현재 위치를 인자로 받은 위치로 옮긴다.
* tell() : 인자로 fd값을 받아 현재 thread의 files 배열내의 fd에 해당하는 파일이 유효한지 검사하고, 잘못된 인자일 경우 exit(-1)을 호출하여 종료하도록 한다. 유효한 인자라면 file\_tell 함수를 호출해 fd에 해당하는 파일의 현재 위치를 반환한다.
* read() : 지난 프로젝트에 추가로 파일에 대한 입력을 구현하기 위해 fd 값이 3~127 (STDIN, STDOUT, STDERR 제외)의 범위에 있다면 file\_read 함수를 호출해 이 함수로부터 반환된 값을 read함수에서 반환한다. 인자로 받은 fd가 3 ~ 127 사이에 있을 때 현재 thread의 files 배열내의 fd에 해당하는 파일이 유효한지 검사하고, 잘못된 인자일 경우 exit(-1)을 호출하여 종료하도록 한다.
* write() : 지난 프로젝트에 추가로 파일에 대한 출력을 구현하기 위해 fd 값이 3~127 (STDIN, STDOUT, STDERR 제외)의 범위에 있다면 file\_write 함수를 호출해 이 함수로부터 반환된 값을 write 함수에서 반환한다. 인자로 받은 fd가 3 ~ 127 사이에 있을 때 현재 thread의 files 배열내의 fd에 해당하는 파일이 유효한지 검사하고, 잘못된 인자일 경우 exit(-1)을 호출하여 종료하도록 한다. 또 이 파일을 누가 읽거나 접근하고 있다면 변경하면 안되므로 lock을 통해 접근 권한을 얻고, 추가로 해당 파일의 deny\_write을 검사해 쓸 수 있을 경우에만 출력을 수행한다. 또 함수를 종료할 때 lock을 release하여 다른 thread에서 접근할 수 있도록 한다.
  + 1. **Synchronization in Filesystem:**

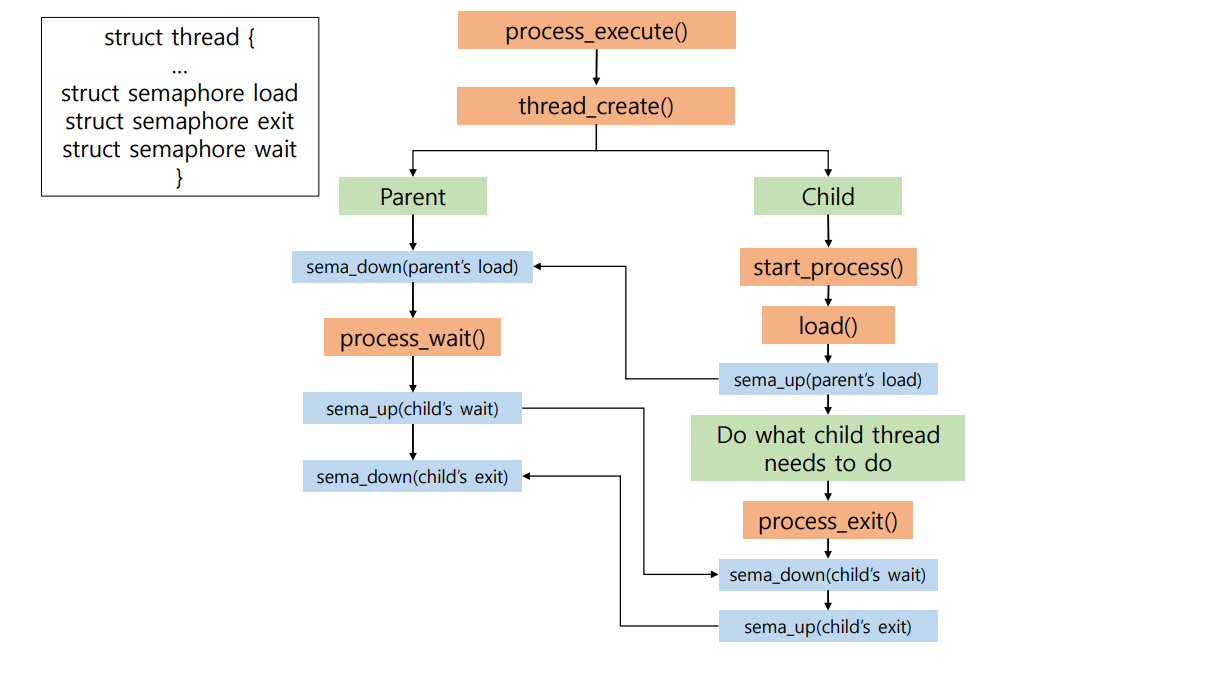
**/userprog/ process.c**

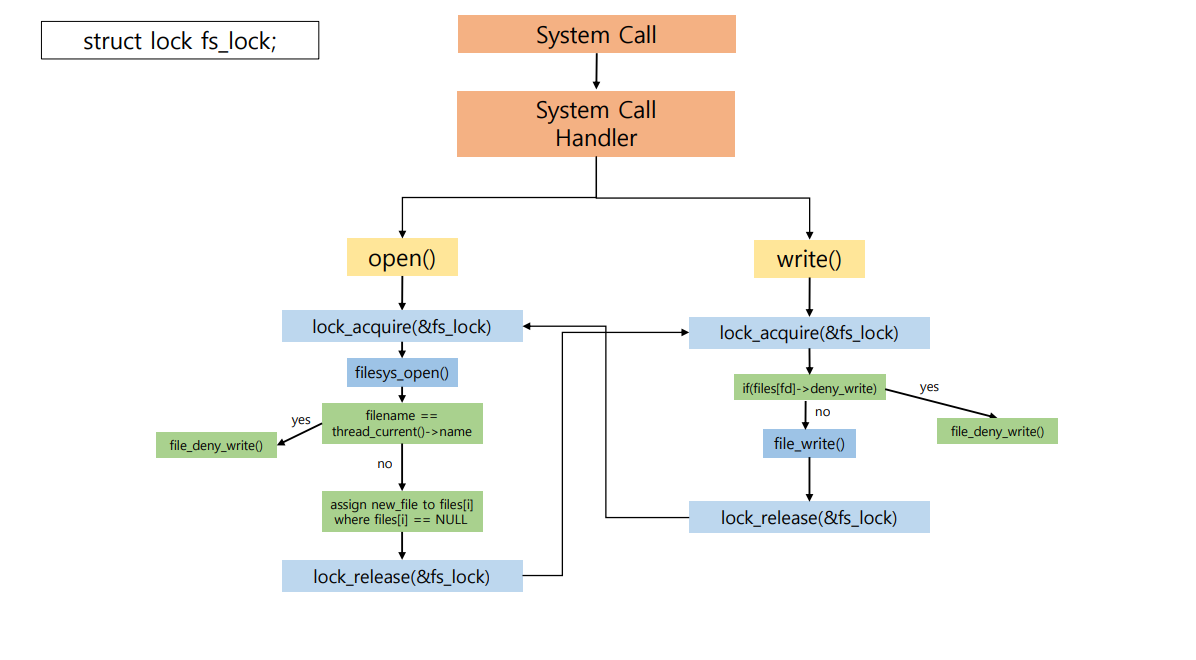
synchronization를 구현하기 위해 semaphore를 사용한다. semaphore를 sema\_up과 sema\_down 함수를 통해 제어하는데, sema\_down은 인자로 받은 semaphore의 value가 0이 아니게 될 때까지 thread를 spinlock 하게 하여 정지시켜 놓는 함수이고, sema\_up은 그 value를 증가시켜 thread를 다시 활성화시키는 함수이다. 이 함수들의 특징을 활용해 thread가 Critical Section에 접근하기 전후에 사용한다. 또 부모 thread가 자식 thread의 종료를 기다리는 process\_wait 함수에서 semaphore를 사용해 자식 thread가 종료되면 다시 활성화되도록 한다.

여러 thread에서 파일에 대해 접근할 때에도 synchronization이 필요하다. 이를 구현하기 위해 lock을 사용하는데, 어떤 thread가 파일을 여는 수행이 끝나지 않았는데 다른 thread가 이 파일을 열어 다른 작업을 하거나, 어떤 thread가 파일을 쓰는 수행을 하고 있는데 다른 thread에서 이 파일에 대해 쓰는 작업을 하려 하는 것을 막기 위해 open()과 write()의 시작/종료 부분에 lock을 건다. 또 어떤 thread가 파일을 읽고 있을 때 다른 thread에서 파일을 변경하면 안되므로 file\_deny\_write을 호출해 이 파일에 대해 쓰는 작업을 제한한다.

1. **연구 결과**
   1. **Flow Chart**
      1. **File Descriptor**
      2. **System Calls**
      3. **Synchronization in Filesystem**

* Thread 간 Synchronization



* Deny to executing file
  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**

**/threads/ thread.h**

**struct thread**

thread가 열어서 작업하는 파일들을 담은 struct file\* files[128]을 선언했다. 배열의 index가 해당 파일의 file descriptor로써 역할을 한다.

|  |
| --- |
| struct thread  {  …..  struct file\* files[128];  …..  }; |

**/threads/ thread.c**

**static void init\_thread(struct thread\* t, const char\* name, int priority)**

thread의 files 배열의 모든 index를 NULL로 초기화한다. 해당 index가 NULL이라는 것은 해당 index를 fd로 갖는 파일이 없다는 의미이고 즉, 모든 index가 NULL이라는 것은 열려 있는 파일이 없다는 의미로 초기화를 한다.

|  |
| --- |
| static void  init\_thread (struct thread \*t, const char \*name, int priority)  {  ….  for(int i = 0; i< 128; i++) t->files[i] =NULL;  …  } |

* + 1. **System Call & Synchronization**

**/userprog/ syscall.c**

**struct lock fs\_lock**

실행 중인 파일에 대해 출력을 수행하거나 여는 작업을 수행하는 도중 다른 thread가 동시에 여는 작업을 수행하는 것을 막기 위해 fs\_lock이라는 lock 구조체 전역변수를 선언하였다.

|  |
| --- |
| #include “threads/synch.h”  #include “filesys/filesys.h”  #include “filesys/file.h”  struct lock fs\_lock; |

**void syscall\_init(void)**

fs\_lock을 초기화하여 lock\_acquire, lock\_release 등의 함수를 사용할 수 있도록 lock\_init을 호출한다.

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

**int open(const char\* filename)**

인자로 받은 filename이 NULL인지, user address 위에 있는지 유효성을 검사한다. 유효하지 않을 경우 exit(-1)을 수행한다. open()의 수행을 Critical Section 내에서의 수행이라고 보고, synchronization을 위해 lock\_acquire를 수행한다. lock에 대한 권한을 얻은 후 filesys\_open을 호출한다.

filesys\_open은 pintos가 실행된 디렉터리부터 탐색해 filename과 해당하는 파일을 호출하고 해당 파일을 반환한다. 실행 중인 파일의 내용이 변경되지 않도록 file\_deny\_write을 호출하여 파일의 deny\_write을 true로 set한다.

이후 현재 thread의 files배열을 순회하며 빈 index에 방금 연 파일을 추가하고, lock\_release를 수행해 lock에 대한 권한을 돌려주고 해당 index를 반환한다.

|  |
| --- |
| int open(const char\* filename) {  if(filename == NULL) exit(-1);  if(!is\_user\_vaddr(filename)) exit(-1);  lock\_acquire(&fs\_lock);  int i, ret = -1;  struct file\* new\_file = filesys\_open(filename);  if(new\_file == NULL) ret= -1;  else {  if(strcmp(thread\_current()->name, filename) == 0) file\_deny\_write(new\_file);  for (i=3; i< 128; i++) {  if(thread\_current()->files[i] == NULL) {  thread\_current()->files[i] = new\_file;  ret = i;  break;  }  }  }  lock\_release(&fs\_lock);  return ret;  } |

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

인자로 받은 fd에 해당하는 파일에 대해 입력을 수행한다. fd 값은 0~127에 해당하는 정수 값이고, fd가 0일 경우엔 STDIN으로부터 입력을 받는다. 3~127에 해당하는 fd 값에 대해 file\_read를 호출한다. 인자로 받은 buffer와 buffer+size가 user address 위에 있는지를 검사하고, 현재 thread의 files 배열 중 fd에 해당하는 파일이 열려 있는지 확인한다. 검사에서 true가 아니라면 exit(-1)을 수행한다.

|  |
| --- |
| int read(int fd, void\* buffer, unsigned size){  if(!is\_user\_vaddr(buffer)) exit(-1);  if(!is\_user\_vaddr(buffer+size)) exit(-1);  int ret = -1;  if(fd == 0) {  //call input\_getc()  //prototype : uint8\_t input\_getc(void)  //return a char from keyboards  unsigned i;  for(i=0;i<size;i++){  \*(uint8\_t\*)buffer = input\_getc();  if(((char\*)buffer)[i] == '\0') break;  }  ret = i;  } else if (fd >= 3) {  if(thread\_current()->files[fd] == NULL) {  exit(-1);  }  ret = file\_read(thread\_current()->files[fd], buffer, size);  }  return ret;  } |

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

인자로 받은 fd에 해당하는 파일에 대해 출력을 수행한다. fd가 1일 경우엔 STDOUT에 출력을 수행한다. 3~127에 해당하는 fd 값에 대해 file\_write를 호출한다. 인자로 받은 buffer와 buffer+size가 user address 위에 있는지를 검사하고, 현재 thread의 files 배열 중 fd에 해당하는 파일이 열려 있는지 확인한다. 검사에서 true가 아니라면 exit(-1)을 수행한다.

또 write()의 수행을 Critical Section으로의 접근으로 보고 lock\_acquire를 수행해 lock에 대한 권한을 얻는다. fd에 해당하는 파일의 deny\_write이 set되어있는지 확인한 후 file\_write을 호출한다. 출력 작업을 모두 완료하고 lock\_release를 수행해 lock에 대한 권한을 돌려주며 file\_write으로부터 받은 그 결과값을 반환한다.

|  |
| --- |
| int write(int fd, const void\* buffer, unsigned size){  if(!is\_user\_vaddr(buffer)) exit(-1);  if(!is\_user\_vaddr(buffer+size)) exit(-1);  lock\_acquire(&fs\_lock);  int ret = -1;  if(fd==1) {  putbuf(buffer, size);  ret = size;  } else if (fd >= 3){  if(thread\_current()->files[fd] == NULL){  lock\_release(&fs\_lock);  exit(-1);  }  if(thread\_current()->files[fd]->deny\_write) file\_deny\_write(thread\_current()->files[fd]);  ret = file\_write(thread\_current()->files[fd], buffer, size);  }  lock\_release(&fs\_lock);  return ret;  } |

**int filesize(int fd)**

인자로 받은 fd에 해당하는 파일의 size를 반환한다. file\_length를 호출하여 그 결과값을 반환한다.

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

**bool create(const char\* filename, unsigned size)**

인자로 받은 파일의 이름과 파일의 크기에 해당하는 파일을 filesys\_create를 호출해 생성하고 생성에 성공하면 true 실패하면 false를 반환한다. 인자로 받은 파일의 이름이 NULL인지 검사하고 NULL이라면 exit(-1)을 수행한다.

|  |
| --- |
| bool create(const char\* filename, unsigned size) {  if(filename == NULL) exit(-1);  if(!is\_user\_vaddr(filename)) exit(-1);  return filesys\_create(filename, size);  } |

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

인자로 받은 파일의 이름에 해당하는 파일을 filesys\_remove를 호출해 제거하고 제거에 성공하면 true 실패하면 false를 반환한다. 인자로 받은 파일의 이름이 NULL인지 검사하고 NULL이라면 exit(-1)을 수행한다.

|  |
| --- |
| bool remove(const char\* filename) {  if(filename == NULL) exit(-1);  if(!is\_user\_vaddr(filename)) exit(-1);  return filesys\_remove(filename);  } |

**void close(int fd)**

인자로 받은 fd에 해당하는 현재 thread의 파일을 file\_close를 호출해 해당 파일에 대한 메모리를 해제하고, fd에 해당하는 파일의 인덱스에 NULL을 할당해 비워준다. 인자로 받은 fd에 해당하는 파일이 NULL인지 검사하고 NULL이라면 exit(-1)을 수행한다.

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

**void seek(int fd, unsigned pos)**

인자로 fd값과 위치를 받아 현재 thread의 files 배열내의 fd에 해당하는 파일이 유효한지 검사하고, 잘못된 인자일 경우 exit(-1)을 호출하여 종료하도록 한다. file\_seek 함수를 호출해 fd에 해당하는 파일의 현재 위치를 인자로 받은 위치로 옮긴다.

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

**unsigned tell(int fd)**

인자로 fd값을 받아 현재 thread의 files 배열내의 fd에 해당하는 파일이 유효한지 검사하고, 잘못된 인자일 경우 exit(-1)을 호출하여 종료하도록 한다. 유효한 인자라면 file\_tell 함수를 호출해 fd에 해당하는 파일의 현재 위치를 반환한다.

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

**void exit(int status)**

thread가 exit할 때, thread에 열려 있는 파일을 닫지 않으므로 별도로 해당 thread의 files배열을 순회하며 NULL이 아닌(열려 있는) 파일을 닫아준다.

|  |
| --- |
| void exit(int status){  thread\_current()->exit\_status = status;  printf("%s: exit(%d)\n", thread\_current()->name, status);  for(int i = 3; i < 128; i++)  if(thread\_current()->files[i] != NULL) close(i);  thread\_exit();  } |

**/threads/ thread.h**

**struct thread**

synchronization을 위해 다른 semaphore를 추가했는데, 부모 thread가 자식 thread가 load를 수행하는 것을 기다리도록 struct semaphore load를, 부모 thread가 wait을 수행하기 전에 자식 thread가 종료하는 (zombie thread) 것을 막기 위해 struct semaphore wait을 선언했다.

|  |
| --- |
| struct thread  {  …..  #ifdef USERPROG  /\* Owned by userprog/process.c. \*/  uint32\_t \*pagedir; /\* Page directory. \*/  tid\_t parent\_tid;  tid\_t waiting;  struct semaphore exit; // same with sync in prj #1  struct semaphore wait;  struct semaphore load;  int exit\_status;  struct file\* files[128];  #endif  /\* Owned by thread.c. \*/  unsigned magic; /\* Detects stack overflow. \*/  }; |

**/threads/ thread.c**

**static void init\_thread(struct thread\* t, const char\* name, int priority)**

새로 thread 구조체에 추가된 wait과 load를 sema\_init을 통해 초기화한다.

|  |
| --- |
| static void  init\_thread (struct thread \*t, const char \*name, int priority)  {  ….  #ifdef USERPROG  sema\_init(&(t->exit), 0);  sema\_init(&(t->wait), 0);  sema\_init(&(t->load), 0);  for(int i = 0; i< 128; i++) t->files[i] =NULL;  t->waiting = t->tid;  #endif  } |

**/userprog/ process.c**

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

자식 thread를 생성하여 start\_process를 load하게 한 후, 현재 thread(부모 thread)의 load를 sema\_up하여 기다리게 한다. 또 load를 성공적으로 수행하지 못해 강제 종료되었다는 일종의 flag인 tid가 -1인 thread를 찾고 해당 thread가 NULL이 아닌 경우 process\_wait을 호출해 수거하도록 한다.

|  |
| --- |
| tid\_t process\_execute (const char \*file\_name)  {  …..  tid = thread\_create (cmd, PRI\_DEFAULT, start\_process, fn\_copy);  find\_thread(tid)->parent\_tid = thread\_current()->tid;  sema\_down(&thread\_current()->load);  if (tid == TID\_ERROR)  palloc\_free\_page (fn\_copy);  if(find\_thread(-1)!=NULL) return process\_wait(-1);  return tid;  } |

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

자식 thread가 load를 끝냈을 때, 부모 thread의 load를 sema\_up하여 기다리고 있는 부모 thread를 깨워준다. 또 load를 성공적으로 끝내지 못했을 경우 해당 thread (자식 thread)의 tid를 -1로 set한다.

|  |
| --- |
| static void start\_process (void \*file\_name\_)  {  …  success = load (cmd, &if\_.eip, &if\_.esp);  if (success) {  stack\_argument(file\_name, &if\_.esp);  }  /\* If load failed, quit. \*/  palloc\_free\_page (file\_name);  sema\_up(&find\_thread(thread\_current()->parent\_tid)->load);  if (!success) {  thread\_current()->tid = -1;  exit(-1);  }  ….  } |

**int process\_wait(tid\_t child\_tid)**

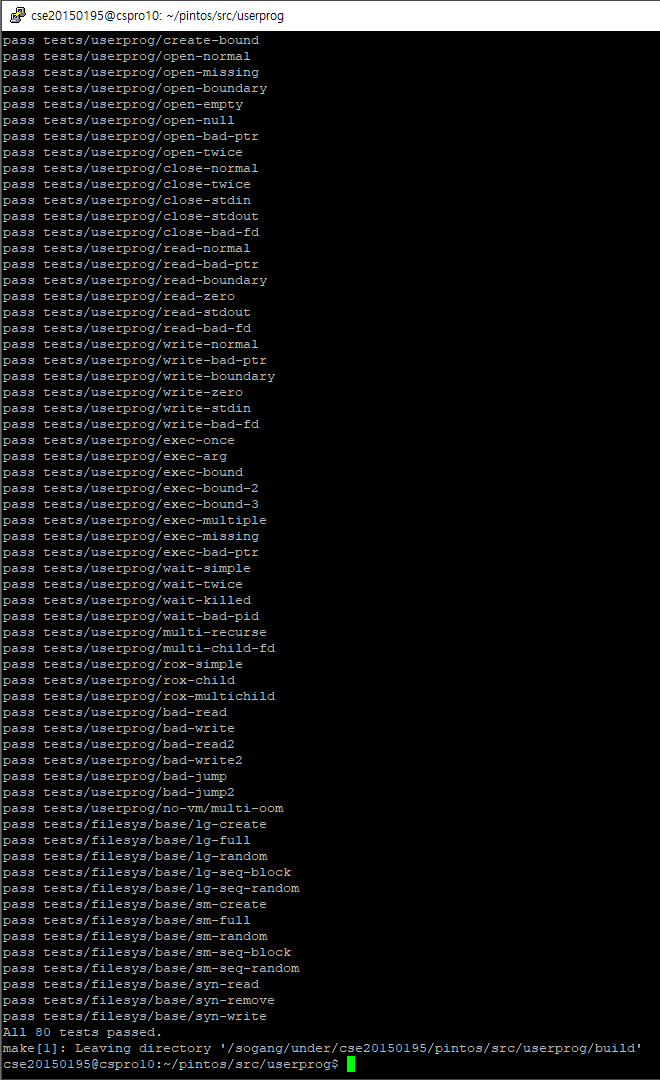
부모 thread가 process\_wait에 도착하는 걸 기다린 자식 thread를 sema\_up(자식 thread의 wait)을 통해 깨워준다.

|  |
| --- |
| int process\_wait (tid\_t child\_tid)  {  struct thread\* t = find\_thread(child\_tid);  if(t == NULL) return -1;  sema\_up(&(t->wait));  thread\_current()->waiting = child\_tid;  sema\_down(&(t->exit)); // wait for child to exit  thread\_current()->waiting = thread\_current()->tid;  return thread\_current()->exit\_status;  } |

**void process\_exit(void)**

부모 thread가 아직 process\_wait에 도착하지 않았을 경우 sema\_down을 통해 기다린다.

|  |
| --- |
| void  process\_exit (void)  {  struct thread \*cur = thread\_current ();  uint32\_t \*pd;  if(find\_thread(cur->parent\_tid)->waiting != cur->tid) sema\_down(&(cur->wait));  …..  } |

* 1. **시험 및 평가 내용**
* 
* 