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

담당 교수 / 분반 : 박성용 / 1

이름 / 학번 : 이은지 / 20170101

개발 기간 : 2020/11/10~2020/11/15

1. **개발 목표**

* 파일 관련 시스템콜 추가 구현
* 각종 예외 처리
* 동기화 및 lock을 통한 파일 시스템 접근 제어

1. **개발 범위 및 내용**
   1. **개발 범위**

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

1. File Descriptor – 현재 프로세스에서 열려있는 모든 파일 엔트리들을 관리하기 위함이다. 이를 통하여 파일을 접근할 수 있다.

2. (이번 프로젝트에서 구현해야 하는) System Calls – 파일과 관련된 시스템콜을 활용하여 안전하게 파일에 접근하고, 파일을 열고 닫거나, 읽고 쓴다.

3. Synchronization in Filesystem – 현재 구현된 파일시스템은 내부 동기화가 없기 때문에, 최대 하나의 프로세스만 파일 시스템에 접근하는 것을 보장하기 위해 파일 접근을 하는 시스템콜 전후로 Lock을 제어해야한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

텍스트이(가) 표시된 사진

자동 생성된 설명

File descriptor table의 엔트리인 구조체 file\_desc는 process.h에 정의되어있다.

- int id : fd번호

- struct list\_elem elem: file\_descriptors list에 담기 위한 list element

- file\* file : 커널에서 관리하는 (유저 프로그램의) 파일 객체

한 프로세스에 여러 파일이 존재할 가능성을 고려하여 리스트 형식으로 파일을 연결하기 위한 list\_elem을 활용한다.

그리고 효율적인 접근을 위하여 별도의 fd라는 id를 설정한다.

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)

**open()** : file\_desc 객체가 새로 생성된다. 열고자 하는 파일을 filesys\_open()을 통해 open한다. fd번호를 할 때는 단순하게 file\_descriptors 리스트에 존재하는 file descriptor 중 가장 큰 fd + 1을 새로운 번호로 사용하여 모든 번호를 고유하게 만든다. 0, 1, 2는 이미 stdin, stdout,stderr에 예약되었으므로 3부터 할당되도록 하였다.

**close()** :인자로 주어지는 fd를 통해, file descriptors 테이블에서 해당 file desciptor엔트리를 순차적으로 탐색한다. 열려있는 file을 닫고, 리스트에서 제거한 후 메모리를 해제한다.

**create()** :filesys\_create()를 호출한다. 만약 파일시스템 동작이 실패할 경우에는 시스템콜의 반환값으로 처리한다.

**remove() :** filesys\_remove()를 호출한다. 만약 파일시스템 동작이 실패할 경우에는 시스템콜의 반환값으로 처리한다.

**filesize()** :해당 thread에서 원하는 file\_descriptor에 맞는 파일을 찾아 file\_length()를 호출한다. 만약 파일시스템 동작이 실패할 경우에는 -1을 리턴한다.

**read()** : 유저 메모리 영역의 버퍼가 올바른지 체크한 후 fd가 0인 경우 표준 입력이므로 input\_getc() 함수를 통해 size 바이트만큼 버퍼를 입력으로 채운다. fd가 1인 경우 file\_descriptor를 찾아 file 객체를 얻고, file\_read()함수를 통해 버퍼를 채운다.

**write()** : 버퍼가 올바른지 체크한 후 buffer가 올바른지 검사한다. fd가 1인 경우, 표준 출력이므로 putbuf()를 사용하여 콘솔에 버퍼 내용을 출력한다. 이 경우가 아니라면, fd로 file descriptor와 file 객체를 찾고, file\_write()함수를 통해 버퍼의 내용을 파일에 기록한다.

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

파일 접근을 위한 lock은 syscall.c에 filesys\_lock으로 선언되어 있으며, exec()을 비롯한 여러 다른 파일시스템 관련 시스템콜을 수행하기 전에는 lock을 acquire하고 이후에는 lock을 release하도록 구현하였다.

또한 프로세스 수행에 있어서는 semaphore를 활용하여 프로세스 간 동기화 및 커뮤니케이션을 하였다. 이는 pcb 객체에 담겨있어 총 2개의 세마포어를 사용하였다. sema\_init은 start\_process()가 수행되는 자식 쓰레드에서 초기화를 수행 후 실패 혹은 성공 시 이 세마포어를 release함으로써 그 이후에 부모 프로세스가 다시 reschedule 될 수 있도록 한다. 그리고 sema\_wait는 자식프로세스의 세마포어로, 이를 통하여 부모 프로세스는 자식 프로세스가 exit()을 호출하여 종료할 때까지 기다리도록 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 시스템개발 범위를 포함하여 구현 내용에 대한 일정 작성

11.10 ~ 11.12 : Project1 시스템콜 개선 및 파일 시스템 관련 시스템콜 구현

11.13~ 11.15 : 파일시스템의 동기화

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

- syscall.c 핸들러 함수에서 시스템콜 함수 호출

|  |
| --- |
| case SYS\_CREATE: //4  {  const char \*file\_name;  unsigned initial\_size;  read\_user(f->esp + 4,&file\_name,sizeof(file\_name));  read\_user(f->esp + 8,&initial\_size,sizeof(initial\_size));  bool return\_code = sys\_create(file\_name,initial\_size);  f->eax = return\_code;  break;  }  case SYS\_REMOVE: //5  {  const char \*file\_name;  bool return\_code;  read\_user(f->esp + 4,&file\_name,sizeof(file\_name));  return\_code = sys\_remove(file\_name);  f->eax = return\_code;  break;  }  case SYS\_OPEN: //6  {  const char \*file\_name;  int return\_code;  read\_user(f->esp + 4,&file\_name,sizeof(file\_name));  return\_code = sys\_open(file\_name);  f->eax = return\_code;  break;  }  case SYS\_FILESIZE: //7  {  int fd;  read\_user(f->esp+4,&fd,sizeof(fd));  int return\_code = sys\_filesize(fd);  f->eax = return\_code;  break;  }  case SYS\_READ: //8  {  int fd;  void \*buffer;  unsigned size;  read\_user(f->esp + 4, &fd, sizeof(fd));  read\_user(f->esp + 8, &buffer, sizeof(buffer));  read\_user(f->esp + 12, &size, sizeof(size));  int return\_code = sys\_read(fd,buffer,size);  f->eax = (uint32\_t) return\_code;  break;  }  case SYS\_WRITE: //9  {  int fd;  void \*buffer;  unsigned size;  read\_user(f->esp + 4, &fd, sizeof(fd));  read\_user(f->esp + 8, &buffer, sizeof(buffer));  read\_user(f->esp + 12, &size, sizeof(size));  int return\_code = sys\_write(fd,buffer,size);  f->eax = (uint32\_t) return\_code;  break;  }  case SYS\_SEEK: //10  {  int fd;  unsigned position;  read\_user(f->esp+4, &fd, sizeof(fd));  read\_user(f->esp+8, &position, sizeof(position));  sys\_seek(fd,position);  break;  }  case SYS\_TELL://11  {  int fd;  unsigned return\_code;  read\_user(f->esp+4,&fd,sizeof(fd));  return\_code = sys\_tell(fd);  f->eax = (uint32\_t) return\_code;  break;  }  case SYS\_CLOSE://12  {  int fd;  read\_user(f->esp+4, &fd, sizeof(fd));  sys\_close(fd);  break;  } |

- syscall.c에서 시스템콜 함수 구현 및 lock

(기존에 표준 입력과 출력만 되던 read(), write()수정)

|  |
| --- |
| bool sys\_create(const char \*file\_name, unsigned initial\_size){  if(file\_name == NULL)  sys\_exit(-1);  check\_user((const uint8\_t\*)file\_name);  lock\_acquire(&filesys\_lock);  bool success = filesys\_create(file\_name,initial\_size);  lock\_release(&filesys\_lock);  return success;  }  bool sys\_remove(const char \*file\_name){  if(file\_name == NULL)  sys\_exit(-1);  check\_user((const uint8\_t\*)file\_name);  lock\_acquire(&filesys\_lock);  bool success = filesys\_remove(file\_name);  lock\_release(&filesys\_lock);  return success;  }  int sys\_open(const char \*file\_name){  struct file\_desc\* fd;  struct file \*file;  if(!file\_name)  sys\_exit(-1);    check\_user((const uint8\_t\*)file\_name);  fd = palloc\_get\_page(0);  if(!fd)  return -1;  lock\_acquire(&filesys\_lock);  file = filesys\_open(file\_name);  if(file == NULL){  palloc\_free\_page(fd);  lock\_release(&filesys\_lock);  return -1;  }  if(strcmp(thread\_name(),file\_name) == 0){  file\_deny\_write(file);  }  //save the file to the file descriptor  fd->file = file;  //no directory  //fd\_list  struct list\* fd\_list = &thread\_current()->file\_descriptors;  if(list\_empty(fd\_list)){  fd->id = 3;  }  else{  fd->id = (list\_entry(list\_back(fd\_list),struct file\_desc,elem)->id) + 1;  }  list\_push\_back(fd\_list,&(fd->elem));  lock\_release(&filesys\_lock);  return fd->id;  }  int sys\_filesize(int fd){  struct file\_desc\* desc;  lock\_acquire(&filesys\_lock);  desc = find\_file\_desc(thread\_current(),fd);  if(desc == NULL){  lock\_release(&filesys\_lock);  return -1;  }  int ret = file\_length(desc->file);  lock\_release(&filesys\_lock);  return ret;  }  int sys\_read(int fd, void \*buffer, unsigned size){  check\_user((const uint8\_t \*)buffer);  check\_user((const uint8\_t \*)buffer + size -1);  lock\_acquire(&filesys\_lock);  int ret;  if(fd == 0){  unsigned i;  for(i = 0;i<size;i++){//STDIN  if(!input\_getc()){  sys\_exit(-1);  }  }  ret = size;  }  else{  struct file\_desc\* desc = find\_file\_desc(thread\_current(),fd);  if(fd && desc->file){  ret = file\_read(desc->file,buffer,size);  }  else{  ret = -1;  }  }  lock\_release(&filesys\_lock);  return ret;  }  int sys\_write(int fd, const void \*buffer,unsigned size){  //printf("%d\n",size);  int ret;  check\_user((const uint8\_t\*)buffer);  check\_user((const uint8\_t\*)buffer + size -1);  lock\_acquire(&filesys\_lock);  if(fd == 1){  putbuf(buffer,size);  ret = size;  }  else {  struct file\_desc \*desc = find\_file\_desc(thread\_current(),fd);  if(desc && desc->file){  ret = file\_write(desc->file,buffer,size);  }  else{  ret = -1;  }  }  lock\_release(&filesys\_lock);  return ret;  }  void sys\_seek(int fd, unsigned position){  lock\_acquire(&filesys\_lock);  struct file\_desc\* desc = find\_file\_desc(thread\_current(),fd);  if(desc && desc->file){  file\_seek(desc->file,position);  }  else  return;  lock\_release(&filesys\_lock);  }  unsigned sys\_tell(int fd){  lock\_acquire(&filesys\_lock);  struct file\_desc\* desc = find\_file\_desc(thread\_current(),fd);  unsigned ret;  if(desc && desc->file){  ret = file\_tell(desc->file);  }  else  ret = -1;  lock\_release(&filesys\_lock);  return ret;  }  void sys\_close(int fd){  lock\_acquire(&filesys\_lock);  struct file\_desc \*desc = find\_file\_desc(thread\_current(),fd);  if(desc && desc->file){  file\_close(desc->file);  list\_remove(&(desc->elem));  palloc\_free\_page(desc);  }  lock\_release(&filesys\_lock);  } |

* + 수정하거나 추가해야 하는 자료구조

텍스트이(가) 표시된 사진

자동 생성된 설명

thread의 정보를 담고 있는 자료구조에 lock 리스트와, file\_descriptors 리스트 추가, 현재 실행중인 파일을 보여주는 excuting\_file을 활용한다.

* + 수정하거나 추가해야 하는 함수
* - 원하는 파일을 찾기위한 추가 함수

|  |
| --- |
| static struct file\_desc\* find\_file\_desc(struct thread \*t,int fd){  ASSERT(t!=NULL);  if(fd < 3){  return NULL;  }  struct list\_elem \*e;  if(!list\_empty(&t->file\_descriptors)){  for(e = list\_begin(&t->file\_descriptors);e !=list\_end(&t->file\_descriptors);e = list\_next(e)){  struct file\_desc \*desc = list\_entry(e, struct file\_desc, elem);  if(desc -> id == fd){  return desc;  }  }  }  return NULL;  } |

**-** 이미 lock이 걸린 파일에 acquire\_lock을 시도하는 예외 처리

|  |
| --- |
| static void is\_invalid(void){  if(lock\_held\_by\_current\_thread(&filesys\_lock))  lock\_release(&filesys\_lock);  sys\_exit(-1);  } |

-process.c에서 semaphore 설정

프로세스를 생성할 때 process status를 담는 pcb 구조체를 할당하고, 초기화한다. 그리고 thread\_create 실행 후에 sema\_down을 통하여 해당 start\_process 실행이 종료될 때까지 기다리도록 한다. 그리고 pcb 리스트를 통하여 연결시킨다.

|  |
| --- |
| pid\_t  process\_execute (const char \*file\_name)  {  char \*fn\_copy = NULL;  char \*cmd\_copy = NULL;  struct process\_control\_block \*pcb = NULL;  tid\_t tid;  char \*save\_ptr = NULL;  /\* Make a copy of FILE\_NAME. for PCB  Otherwise there's a race between the caller and load(). \*/  fn\_copy = palloc\_get\_page (0);  if (fn\_copy == NULL)  goto failed;  strlcpy (fn\_copy, file\_name, PGSIZE);    //make thread name  cmd\_copy = palloc\_get\_page(0);  if(cmd\_copy == NULL)  goto failed;  strlcpy(cmd\_copy,file\_name,PGSIZE);  cmd\_copy = strtok\_r(cmd\_copy," ",&save\_ptr);  //Before to create a thread,  //create a pcb along with the name and pass it into the new thread  pcb = palloc\_get\_page(0);  if(pcb == NULL)  goto failed;  pcb -> pid = PID\_INIT;  pcb -> parent\_thread = thread\_current();  pcb -> cmdline = fn\_copy;  pcb -> waiting = false;  pcb -> exited = false;  pcb -> orphan = false;  pcb -> exitcode = -1;  //Initialization  sema\_init(&pcb->sema\_init,0);  sema\_init(&pcb->sema\_wait,0);  /\* Create a new thread to execute FILE\_NAME. \*/  tid = thread\_create (cmd\_copy, PRI\_DEFAULT, start\_process, pcb);  if (tid == TID\_ERROR)  goto failed;  //wait until initialization inside start\_process() is completed  sema\_down(&pcb->sema\_init);  if(cmd\_copy){  palloc\_free\_page(cmd\_copy);  }  //printf("%d\n",pcb->pid);  //in case of success of creating process, add child process to the list  if(pcb->pid>=0){  list\_push\_back(&(thread\_current()->child\_list),&(pcb->elem));  }  //palloc\_free\_page(fn\_copy);  return pcb->pid;  failed:  if(cmd\_copy)  palloc\_free\_page(cmd\_copy);  if(fn\_copy)  palloc\_free\_page(fn\_copy);  if(pcb){  palloc\_free\_page(pcb->cmdline);  palloc\_free\_page(pcb);  }  return PID\_ERROR;  } |

-자식 프로세스가 끝날 때까지 기다리도록 sema\_down 함수를 활용한다.

|  |
| --- |
| int  process\_wait (tid\_t child\_tid)  {  //자식 프로세스가 모두 종료될 때까지 대기  //자식 프로세스가 올바르게 종료됐는지 확인  if(child\_pcb == NULL){//NOT found  //printf(“wait() : child not found, pid = %d\n”,child\_tid);  return -1;  }  if(child\_pcb->waiting){//already waiting  //printf(“wait() : child found, but it is already ‘waiting’, pid = %d\n”,child\_tid);  return -1;  }  else{  child\_pcb -> waiting = true;  }  //wait() until child terminates  //signaling this semaphore  if(!child\_pcb->exited){  sema\_down(&(child\_pcb->sema\_wait));  }  ASSERT(child\_pcb->exited == true);  //remove the child from the list  ASSERT(child\_pcb != NULL);  list\_remove(&(child\_pcb -> elem));  //return the exit code of child process  int ret = child\_pcb->exitcode;  //free pcb  palloc\_free\_page(child\_pcb);  return ret; |

이외에도 정상 종료되지 않는 프로세스를 처리해야한다. (pcb 해제, file\_descriptor해제)

- 자식 프로세스보다 먼저 종료하는 경우

1) 부모 프로세스가 자식 프로세스에 대해 wait()을 호출한 적이 없는 경우 : 추후에 부모 프로세스는 process\_exit()루틴에서 자신의 자식 프로세스들 중 terminate 된 것이 있으면 해제하고, pcb도 해제를 한다.

(process.c / process\_exit())

|  |
| --- |
| /\* Erase PCB \*/  struct list \*pcb\_list = &cur -> child\_list;  while(!list\_empty(pcb\_list)){  e = list\_pop\_front(pcb\_list);  pcb = list\_entry(e, struct process\_control\_block,elem);  if(pcb->exited == true){  palloc\_free\_page(pcb->cmdline);  palloc\_free\_page(pcb); //already signed  }  else{  //the child process became an orphan, so they should have pcb yet.  pcb->orphan = true;  pcb->parent\_thread = NULL;  }  }  //Unblock the waiting parent process from wait(), and then free the resources  cur->pcb->exited = true;  bool cur\_orphan = cur->pcb->orphan;  sema\_up(&cur->pcb->sema\_wait);  if(cur\_orphan){  palloc\_free\_page(&cur->pcb->cmdline);  palloc\_free\_page(&cur->pcb);  } |

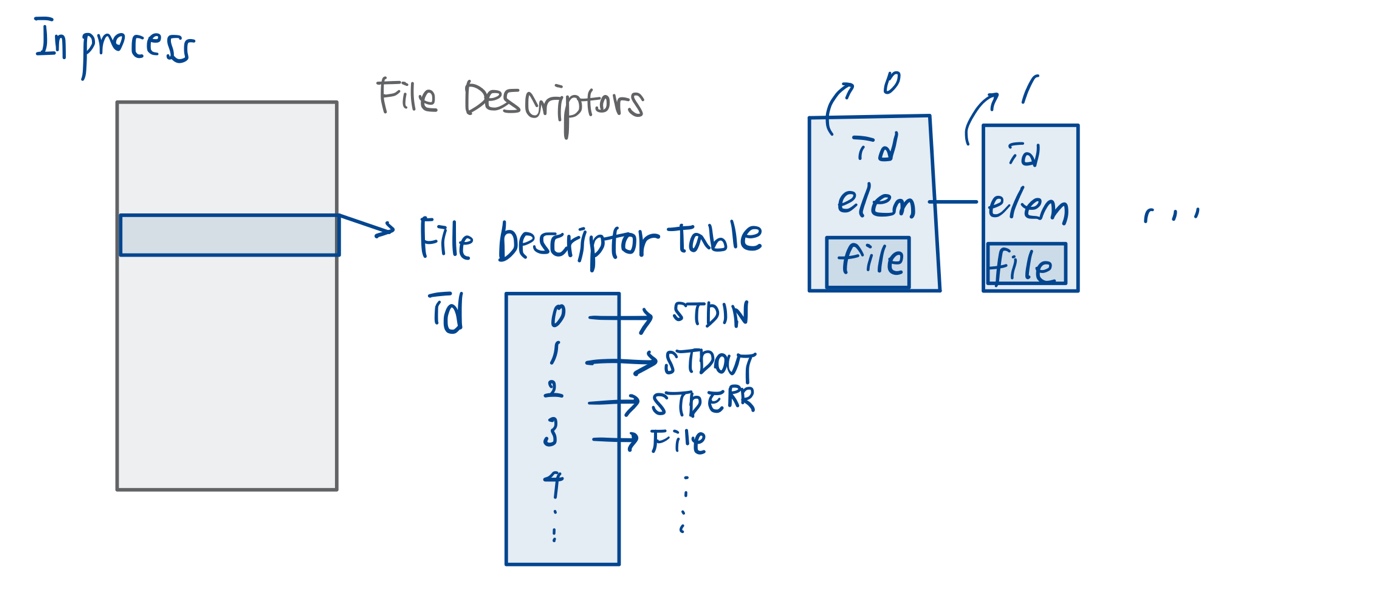
- 부모 프로세스가 자식 프로세스에 대해 wait()을 호출한 경우: 이 때에는 부모 프로세스의 wait()실행 중. 자식 프로세스가 종료한 이후에 부모 프로세스가 자식 프로세스를 child list에서 제거하고, 자식 프로세스의 pcb를 해제한다.

(process.c/ process\_wait())

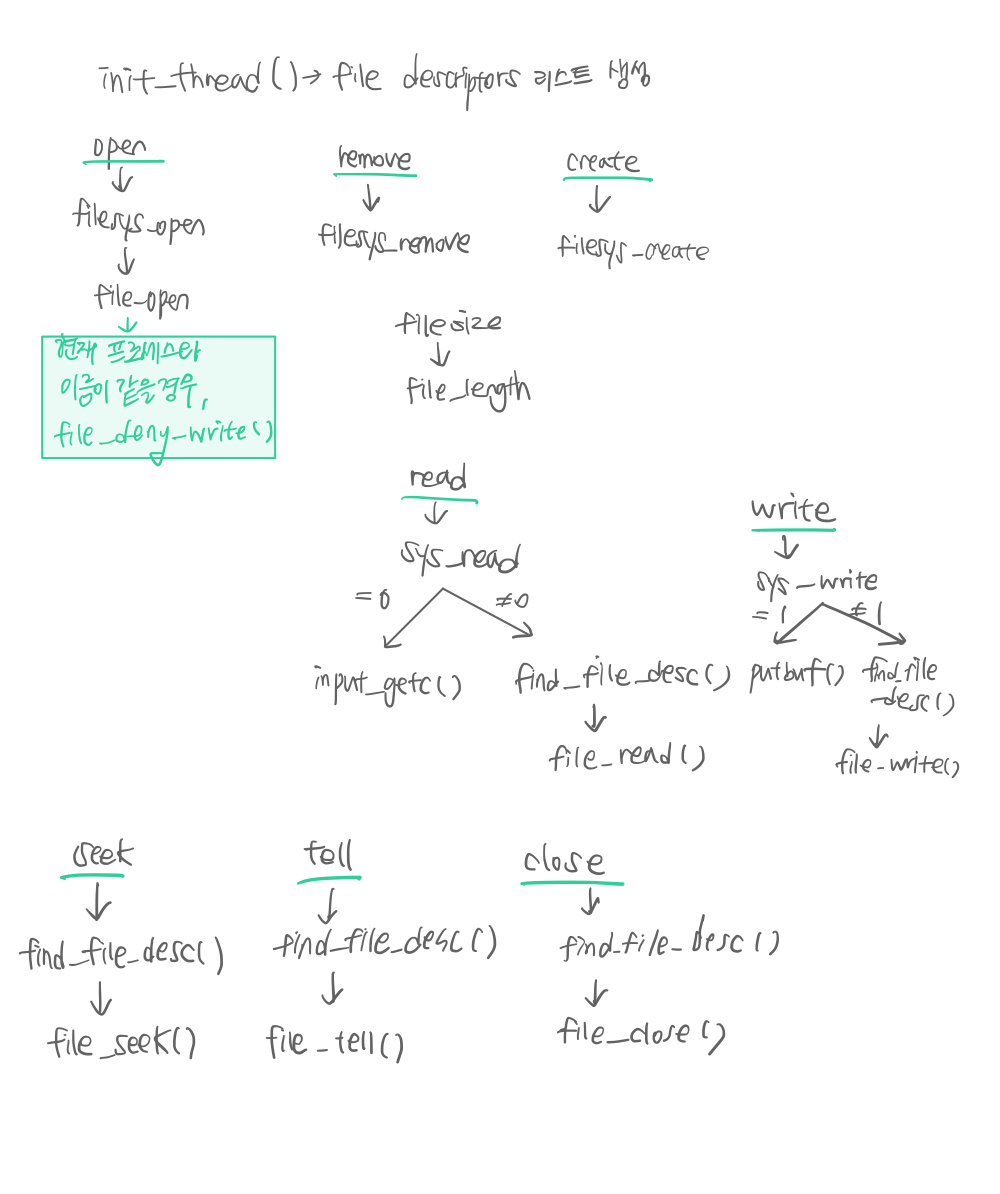
|  |
| --- |
| //remove the child from the list  ASSERT(child\_pcb != NULL);  list\_remove(&(child\_pcb -> elem));  //return the exit code of child process  int ret = child\_pcb->exitcode;  //free pcb  palloc\_free\_page(child\_pcb);  return ret; |

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 3가지 항목에 대하여 Flow Chart 작성
* **1. File Descriptor**

****

**2. System Call**

****

**3. Synchronization**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* 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
* 파일과 프로세스를 반복적으로 찾아야하는 경우가 존재하므로 이에 대해 별도의 함수를 제작하여 효율적으로 코드를 구현하였다.

- 메모리 누수를 막기 위해 palloc\_gt\_page()한 메모리는 반드시 palloc\_free\_page()를 수행하여야한다. 혹시 모를 메모리 누수를 막기 위해 process\_exit() 시에 리스트를 순회하며 파일 디스크립터와 pcb를 메모리 해제하였다.

|  |
| --- |
| /\* Erase FD\*/  struct list \*fdlist = &cur -> file\_descriptors;  while(!list\_empty(fdlist)){  struct list\_elem \*e = list\_pop\_front(fdlist);  struct file\_desc \*desc = list\_entry(e,struct file\_desc,elem);  file\_close(desc->file);  palloc\_free\_page(desc);  }  /\* Erase PCB \*/  struct list \*pcb\_list = &cur -> child\_list;  while(!list\_empty(pcb\_list)){  e = list\_pop\_front(pcb\_list);  pcb = list\_entry(e, struct process\_control\_block,elem);  if(pcb->exited == true){  palloc\_free\_page(pcb->cmdline);  palloc\_free\_page(pcb); //already signed  }  else{  //the child process became an orphan, so they should have pcb yet.  pcb->orphan = true;  pcb->parent\_thread = NULL;  }  } |

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

|  |
| --- |
| 텍스트이(가) 표시된 사진  자동 생성된 설명  텍스트이(가) 표시된 사진  자동 생성된 설명 |