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

담당 교수 / 분반 : 박수용 교수님 / 1반

이름 / 학번 : 나호영 / 20211531

개발 기간 : 23/10/10 ~ 23/10/25

1. **개발 목표**파일 시스템 관련 시스템 콜들인 open, close, create, remove, close, filesize, week, tell, read, write를 구현한다. 또한 구현 중에 생기는 Sychronization 문제를 적절히 해결하여 파일 시스템 관련 시스템 콜들이 정상적으로 작동될 수 있도록 한다.
2. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor  
파일 디스크립터는 프로세스가 파일들에 접근할 때 어떤 파일에 접근할지 정할 때 필요한 요소이다. 이 때, 128개의 파일 디스크립터가 존재하는데 0번은 STDIN, 1번은 STDOUT, 2번은 STDERR으로 리눅스에서는 정해져있으나 핀토스에서는 1번과 2번만 리눅스와 같이 정해져있다. 특정한 파일을 OPEN 시 커널은 사용되지 않고 있는 파일 디스크립터들 중 가장 작은 번호를 가지고 있는 파일 디스크립터를 할당해준다. 이후에 다시 이 파일에 접근할 때는 해당 번호의 파일 디스크립터 값을 이용해서 접근이 가능하다. 이를 위해 이번 프로젝트에서는 128개의 파일 디스크립터를 저장할 자료구조를 직접 구현해야 한다.

2. System Calls  
이전 USER PROGRAM1 프로젝트에서 구현했던 시스템 콜들과 마찬가지로 사용자가 커널이 제공하는 기능을 이용하기 위해서 필요한 파일 시스템과 관련된 시스템 콜들을 구현해야 한다. 이를 위해서 해당하는 시스템 콜 번호를 식별하여 처리하는 시스템 콜 핸들러에 이번 파일 시스템과 관련된 처리를 추가하는 작업이 필요하다.

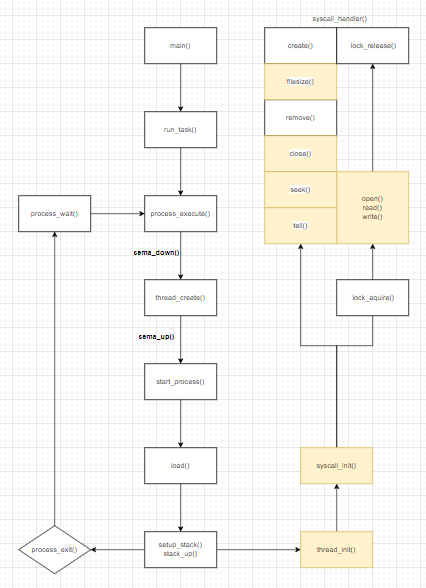
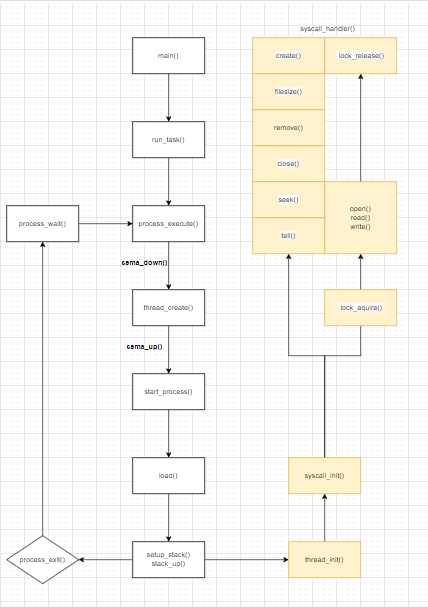
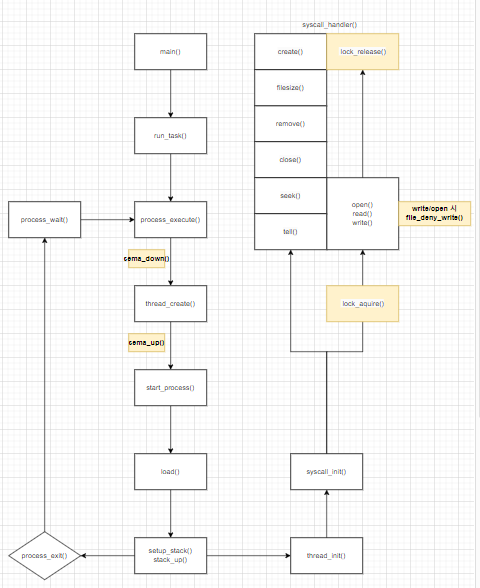
3. Synchronization in Filesystem  
특정 파일에 2개 이상의 프로세스가 write 작업을 수행하려고 하거나 executable file이 삭제되는 등의 문제는 해당 파일에 손상을 끼칠 수 있다. 이를 위해 critical section에 대한 알맞은 synchronization 해야한다. 따라서 특정 프로세스가 critical section에 있을 때 다른 프로세스가 침범하지 않도록 적절한 처리가 필요하다.

* 1. **개발 내용**

1. File Descriptor  
파일 디스크립터를 저장할 자료구조로 배열을 사용한다. 총 128개의 파일 디스크립터를 저장하는 파일 구조체 포인터 배열 fd를 thread 구조체 내부에 선언한다. 이유는 각각의 프로세스들이 고유한 파일 디스크립터 배열을 가지고 있어야 하기 때문이다.

2. System Calls  
(1) create : 파일 이름과 사이즈를 받아서 해당하는 이름과 사이즈를 가지는 새로운 파일을 생성하는 함수이다. 성공 시 true, 실패 시 false를 반환한다.  
(2) remove : 파일 이름을 받아서 해당하는 이름을 가지고 있는 파일을 삭제한다. 성공 시 true, 실패 시 false를 반환한다.  
(3) open : 파일 이름을 받아서 해당하는 이름을 가지고 있는 파일을 열고 해당 파일에 할당되는 파일 디스크립터 번호 값을 반환한다. 실패 시 -1을 반환한다.  
(4) close : 해당하는 파일의 파일 디스크립터를 close한다.  
(5) filesize : 해당하는 파일 디스크립터의 파일 사이즈를 반환한다.  
(6) read : 파일 디스크립터, 버퍼, 사이즈를 받아서 해당하는 파일에서 사이즈만큼 읽고 버퍼에 저장하는 함수이다. 읽어들인 바이트 수를 반환한다. 파일 디스크립터의 값이 0, 즉 STDIN이면 키보드로부터 입력받는다.  
(7) write : 파일 디스크립터와 버퍼, 사이즈를 받아서 버퍼로부터 사이즈만큼의 데이터를 파일에 쓴다. 파일 디스크립터의 값이 1, 즉 STDOUT이면 콘솔에 출력한다.  
(8) seek : open 된 파일의 파일 디스크립터에서 read/write 하는 위치(오프셋)를 다음 바이트로 이동시킨다.  
(9) tell : read/write 하는 위치(오프셋)를 반환한다.

3. Synchronization in Filesystem  
(1) Lock : 2개 이상의 프로세스들이 하나의 파일에 동시에 read/write 작업을 수행해 발생하는 문제를 예방하기 위해 Lock을 이용한다. open/read/write 함수를 호출하여 동작을 수행하기 이전에 lock\_acquire 를 호출하여 뮤텍스를 설정하고 동작이 종료되면 lock\_release 를 호출하여 뮤텍스를 해제한다. 이러한 방법으로 프로세스가 Critical Section에 있을 때 다른 프로세스가 침범하는 것을 예방할 수 있다.  
(2) Semaphore : 프로세스들 간의 동기화 문제를 예방하기 위하여 세마포어를 이용한다. 프로세스가 excute 될 때 쓰레드가 create되면 sema down, 프로세스가 load될 때 sema up을 한다. 이러한 방법으로 프로세스가 로드 되기 전에 부모 프로세스가 끝나거나 하는 문제등을 예방할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**23/10/10 ~ 23/10/15 : 프로젝트 강의 수강, 전체적인 코드 구조 설계  
      23/10/16 ~ 23/10/23 : system call 함수 구현, synchronization 구현  
      23/10/24 ~ 23/10/25 : 디버깅 및 테스트, 보고서 작성
   2. **개발 방법**  
        
      (1) File Descriptor  
      file.c에 존재하는 file 구조체를 include 하기 위해 file.h에 선언한다. 또한 파일 디스크립터 배열을 구현하기 위해 thread.h에 존재하는 thread 구조체에 struct file\* fd[128]을 선언한다. 추가한 파일 디스크립터 배열 fd를 초기화하기 위해 thread.c의 init\_thread 함수에서 파일 디스크립터 배열을 반복문을 통해 모두 NULL값으로 바꾸는 코드를 작성한다.  
        
      (2) System Calls  
      syscall.c에 존재하는 syscall\_handler 함수를 수정한다. 이 때, 이번 프로젝트에서 구현해야하는 create, remove, open, close, filesize, seek, tell 을 처리하기 위한 case를 switch 문 내부에 추가적으로 구현하고 파일 시스템 구현을 위해 이미 존재하는 read, write 함수에 파일 디스크립터를 통해 파일을 읽고 수정할 수 있도록 받아온 파일 디스크립터 번호를 통해 STDIN/STDOUT과 다른 파일을 구분하여 read, write할 수 있도록 코드를 추가한다.  
        
      (3) Synchronization in Filesystem   
      공유 데이터에 여러 프로세스들이 접근하여 문제를 일으키는 것을 방지하기 위해 thead.h에 존재하는 lock과 관련된 함수들을 사용해야한다. 따라서 lock 구조체 변수 f\_lock을 추가한다. 또한 이 변수를 초기화하기 위하여 thread.c의 thread\_init 함수에서 f\_lock을 초기화하는 코드를 추가한다. 그 후, syscall.c의 syscall\_handler에서 open/read/wirte 가 실행될 때, 위에 서술한 문제를 예방하기 위하여 작업 전에 lock\_acquire, 작업 후에 lock\_release를 호출하는 코드를 작성한다.  
      또한 thread.h에 존재하는 thread 구조체에서 현재 프로세스가 정상적으로 실행되고 있는지 조절하는 세마포어 변수 f\_semaphore와 해당 프로세스의 부모 프로세스를 의미하는 parent, 프로세스가 비정상적으로 종료되어 문제를 일으킬 경우를 체크하기 위한 플래그 변수 terminate\_flag, 현재 실행되고 있는 프로세스의 파일을 의미하는 current\_file 변수를 추가한다. 그 후, thread.c의 init\_thread 함수에서 초기화 작업을 추가한다. exit() 시스템 콜 함수가 호출될 때 file\_close 함수를 호출하여 current\_file이 가르키는 파일을 close할 수 있도록 한다.  
      process.c에 존재하는 process.execute 함수와 start\_process 함수에 이전에 추가했던 thread 구조체의 세마포어 f\_semaphore를 이용해 정상적으로 종료될 때에만 메모리를 정리할 수 있도록 하고 프로세스가 load 되기전에 부모 프로세스가 끝나는 경우를 방지한다. 또한 load 함수에서 프로세스가 정상적으로 로드되었는지 체크하는 success 변수를 확인하여 정상적으로 로드되었을 시 file\_deny\_write 함수를 사용해 write 작업이 수행되지 않도록 하는 조건문을 추가한다. 추가적으로 thread 구조체에 선언했던 플래그 변수 terminate\_flag를 체크하여 process\_wait을 실행할지 말지 결정하는 조건문을 추가한다.  
      exception.c 에 존재하는 page\_fault 함수에서 not-present page가 발생하는 경우의 테스트들을 위하여 not\_present가 1일 때 exit하는 코드를 추가한다.
2. **연구 결과**
   1. **Flow Chart**(1) File Descriptor  
        
        
        
        
      (2) System Calls  
        
        
        
        
        
        
        
        
        
      (3) Synchronization in Filesystem   
       
   2. **제작 내용**(1) File Descriptor

#ifdef USERPROG

  /\* Owned by userprog/process.c. \*/

  uint32\_t \*pagedir;          /\* Page directory. \*/

  struct semaphore clocking;

  struct list child;

  struct list\_elem child\_elem;

  int exit\_stat;

  struct semaphore mlocking;

  /\* user program 2 \*/

  bool terminate\_flag;

  struct semaphore f\_semaphore;

  struct file\* current\_file;

  struct thread\* parent;

  struct file\* fd[128]; /\* user prog 2 \*/

**먼저 위와 같이 thread.h 의 thread 구조체에 파일 디스크립터를 저장할 배열인 fd 배열을 선언하였다.**#ifdef USERPROG

 ......

 for(int i=0; i < 128; i++){ /\* user prog 2 \*/

  t->fd[i] = NULL;

 }

.......

#endif

} **또한 위는 thread.c의 init\_thread 부분으로 모든 파일 디스크립터를 NULL 값으로 초기화하였다.**struct file

 {

  struct inode \*inode;     /\* File's inode. \*/

  off\_t pos;          /\* Current position. \*/

  bool deny\_write;       /\* Has file\_deny\_write() been called? \*/

 }; **file 구조체를 사용하기 위하여 file.c에 존재하는 file 구조체의 정의를 file.h에 복사하여 file.h를 include해 file 구조체를 사용할 수 있게 만들었다.  
  
  
  
  
  
  
  
  
(2)System Calls**case SYS\_CREATE: /\* user prog 2\*/

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 4));

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 8));

    f->eax = create((const char\*)\*(uint32\_t\*)(f->esp + 4), (unsigned)\*(uint32\_t\*)(f->esp + 8));

    break;

  case SYS\_REMOVE:

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 4));

    f->eax = remove((const char\*)\*(uint32\_t\*)(f->esp + 4));

    break;

  case SYS\_OPEN:

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 4));

    f->eax = open((const char\*)\*(uint32\_t\*)(f->esp + 4));

    break;

  case SYS\_FILESIZE:

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 4));

    f->eax = filesize((int)\*(uint32\_t\*)(f->esp + 4));

    break;

  case SYS\_READ:

    /\* 수정 for read-bad-ptr \*/

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 8));

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 4));

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 12));

    f->eax = read((int)\*(uint32\_t\*)(f->esp+4), (void\*)\*(uint32\_t\*)(f->esp+8) , (unsigned)\*(uint32\_t\*)(f->esp+12));

    break;

  case SYS\_WRITE:

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 20));

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 24));

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 28));

    f->eax = write((int)\*(uint32\_t\*)(f->esp + 20), (void\*)\*(uint32\_t\*)(f->esp + 24), (unsigned)\*((uint32\_t\*)(f->esp + 28)));case SYS\_SEEK:

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 4));

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 8));

    seek((int)\*(uint32\_t\*)(f->esp + 4), (unsigned)\*(uint32\_t\*)(f->esp + 8));

  case SYS\_TELL:

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 4));

    f->eax = tell((int)\*(uint32\_t \*)(f->esp + 4));

    break;

  case SYS\_CLOSE:

    checking\_valid\_address((void\*)\*(uint32\_t\*)(f->esp + 4));

    close((int)\*(uint32\_t \*)(f->esp + 4));

    break; **위는 user program 2에서 추가적으로 구현한 시스템 콜들을 위해 syscall\_handler에 case를 추가한 코드이다. checking\_valid\_address를 활용하여 들어온 주소가 유효한지 검사하고 각각 해당하는 함수를 호출한다. 필요하면 eax에 반환값을 저장한다. 각각의 시스템 콜을 구현한 함수를 자세히 살펴보자.**bool create (const char\* file, unsigned size) {

  if(!file){

    exit(-1);

  }else{

    checking\_valid\_address(file);

    return filesys\_create(file, size);

  }

} **먼저 create 함수이다. 파일이 NULL값인지 체크하고 아니라면 checking\_valid\_address 함수를 호출하여 유효한 주소인지 체크한 뒤, filesys\_create 함수를 호출하여 구현하였다.**bool remove (const char\* file) {

  if(!file){

    exit(-1);

  }else{

    checking\_valid\_address(file);

    return filesys\_remove(file);

  }

}

**다음은 remove 함수이다. 파일이 NULL값인지 체크하고 아니라면 checking\_valid\_address 함수를 호출하여 유효한 주소인지 체크한 뒤, filesys\_remove 함수를 호출하여 구현하였다.**

int open (const char\* file) {

  if(!file){

    exit(-1);

  }else{

    checking\_valid\_address(file);

    lock\_acquire(&f\_lock);

    struct file\* fp = filesys\_open(file);

    if (!fp) {

      lock\_release(&f\_lock);

      return -1;

    }

    for (int i = 3; i < 128; i++) {

      if (!(thread\_current()->fd[i])) {

        if (!strcmp(thread\_current()->name, file)) {

          file\_deny\_write(fp);

          }

        thread\_current()->fd[i] = fp;

        lock\_release(&f\_lock);

        return i;

      }

    }

    lock\_release(&f\_lock);

    return -1;

  }

} **다음은 open 함수이다. 마찬가지로 들어온 파일이 NULL 값인지 체크한다. 아니라면 checking\_valid\_address 함수를 호출하여 유효한 주소인지 체크한 뒤, filesys\_open 함수를 호출하여 파일을 열어준다. 그 후, 해당 디스크립터가 NULL인지 체크하고 아니라면 thread의 파일 디스크립터 배열을 확인하여 가장 작은 번호를 가지는 빈 곳에 할당한다. synchronization과 관련된 부분은 (3)에서 설명하겠다.**

int filesize (int fd) {

  if(!(thread\_current()->fd[fd])){

    exit(-1);

  }else{

    return file\_length(thread\_current()->fd[fd]);

  }

} **다음은 filesize 함수이다. 들어온 파일 디스크립터를 확인해 만약 null값이면 종료하고 아니라면 file\_length 함수를 호출하여 구현하였다.**

int read(int fd, const void\* buff, unsigned size) {

  // if (size == 0){ // read-zero

  //   return 0;

  // }

  lock\_acquire(&f\_lock);

  int result;

  if(fd==0){

    for(int i=0; i<size; i++){

      uint8\_t c = input\_getc();

      if(c=='\0') {

        result = i;

        break;

      }

    }

  } else if(fd > 2){

    if(!thread\_current()->fd[fd]){

      lock\_release(&f\_lock);

      exit(-1);

    }

    lock\_release(&f\_lock);

    return file\_read(thread\_current()->fd[fd], buff, size);

  } else {

    lock\_release(&f\_lock);

    return -1;

  }

} **다음은 read 함수이다. 들어온 파일 디스크립터 fd 의 값에 따라 만약 STDIN이라면 input\_getc()함수를 활용하여 입력을 받는다. 아니라면 파일 디스크립터 배열 fd에서 해당 파일 디스크립터를 찾아서 file\_read 함수를 호출하여 구현하였다. synchronization과 관련된 부분은 (3)에서 설명하겠다.**

int write(int fd, const void\* buff, unsigned size) {

  checking\_valid\_address(buff);

  lock\_acquire(&f\_lock);

  if (fd == 1) {

    putbuf(buff, size);

    lock\_release(&f\_lock);

    return size;

  }else if (fd > 2){

    if (!(thread\_current()->fd[fd])) {

      lock\_release(&f\_lock);

      exit(-1);

    }

    if (thread\_current()->fd[fd]->deny\_write) {

      file\_deny\_write(thread\_current()->fd[fd]);

  }

    lock\_release(&f\_lock);

    return file\_write(thread\_current()->fd[fd], buff, size);

  }

  lock\_release(&f\_lock);

  return -1;

} **다음은 write 함수이다. read 함수와 비슷하게 들어온 파일 디스크립터 fd가 STDOUT이면 putbuf를 활용해 출력한다. 아니라면 파일 디스크립터 배열 fd에서 해당하는 파일 디스크립터를 꺼내어 file\_write 함수를 호출하여 구현하였다. synchronization과 관련된 부분은 (3)에서 설명하겠다.**void seek (int fd, unsigned place) {

  if(!(thread\_current()->fd[fd])){

    exit(-1);

  }else{

    file\_seek(thread\_current()->fd[fd], place);

  }

} **다음은 seek 함수이다. 들어온 파일 디스크립터 fd가 NULL 값인지 확인하고 아니라면 file\_seek 함수를 호출하여 구현하였다.**unsigned tell (int fd) {

  if(!(thread\_current()->fd[fd])){

    exit(-1);

  }else{

    return file\_tell(thread\_current()->fd[fd]);

  }

} **다음은 tell 함수이다. 들어온 파일 디스크립터 fd가 NULL 값인지 확인하고 아니라면 file\_tell 함수를 호출하여 구현하였다.**

void close (int fd) {

  if(!(thread\_current()->fd[fd])){

    exit(-1);

  }else{

    struct file\* fp = thread\_current()->fd[fd];

    thread\_current()->fd[fd] = NULL;

    return file\_close(fp);

  }

} **마지막으로 close 함수이다. 들어온 파일 디스크립터 fd가 NULL 값인지 확인하고 아니라면 파일 디스크립터를 꺼내어 NULL 로 초기화한 뒤, file\_close 함수를 호출하여 구현하였다.  
  
(3) Synchronization in Filesystem**

struct lock f\_lock; **thread.h 에 프로세스의 임계 구역에 다른 프로세스가 침범하는 것을 방지하는 synchronization을 위하여 f\_lock 구조체 변수를 선언하였다.**

void

thread\_init (void)

{

.....

 lock\_init (&f\_lock);

.....  
 initial\_thread = running\_thread ();

 init\_thread (initial\_thread, "main", PRI\_DEFAULT);

 initial\_thread->status = THREAD\_RUNNING;

 initial\_thread->tid = allocate\_tid ();

}

**이 구조체는 위와 같이 thread.c의 thread\_init 함수에서 초기화하였다.**int open (const char\* file) {

  if(!file){

    exit(-1);

  }else{

    checking\_valid\_address(file);

    lock\_acquire(&f\_lock);

    struct file\* fp = filesys\_open(file);

    if (!fp) {

      lock\_release(&f\_lock);

      return -1;

    }

    for (int i = 3; i < 128; i++) {

      if (!(thread\_current()->fd[i])) {

        if (!strcmp(thread\_current()->name, file)) {

          file\_deny\_write(fp);

          }

        thread\_current()->fd[i] = fp;

        lock\_release(&f\_lock);

        return i;

      }

    }

    lock\_release(&f\_lock);

    return -1;

  }

} **open 함수의 동기화 부분을 살펴보자. 이전에 선언하였던 f\_lock을 filesys\_open 하기 전에 lock\_acquire 함수를 호출하여 락을 걸어준다. 그 후, 반환되기전에 lock\_release 함수를 호출하여 락을 풀어준다. 이로써 임계 구역 침범에 대한 문제를 해결할 수 있었다. 또한 rox test를 통과하기 위해서는 write 작업이 executable of running thread에 실행되는 것을 방지해야 한다. 따라서 이 부분의 동기화 문제를 해결하기 위해 어차피 write하려면 open해야하니까 파일 디스크립터 배열을 반복문을 이용해 순회하면서 thread의 name과 일치하면 file\_deny\_write를 호출하여 write를 하지 않게 하였다.**int write(int fd, const void\* buff, unsigned size) {

  checking\_valid\_address(buff);

  lock\_acquire(&f\_lock);

  if (fd == 1) {

    putbuf(buff, size);

    lock\_release(&f\_lock);

    return size;

  }else if (fd > 2){

    if (!(thread\_current()->fd[fd])) {

      lock\_release(&f\_lock);

      exit(-1);

    }

    if (thread\_current()->fd[fd]->deny\_write) {

      file\_deny\_write(thread\_current()->fd[fd]);

  }

    lock\_release(&f\_lock);

    return file\_write(thread\_current()->fd[fd], buff, size);

  }

  lock\_release(&f\_lock);

  return -1;

} **이어서 write 함수에서 아까 하던 설명을 이어나가보자. open에서 file\_deny\_write를 호출하였는데 이를 확인하는 작업이 필요하여 현재 쓰레드의 이용하고 있는 파일 디스크립터의 deny\_write를 체크하여 file\_deny\_write를 호출하는 코드를 추가하였다. write 함수 역시 마찬가지로 동기화를 위해 lock\_acquire과 lock\_release를 작업 전과 후에 추가하였다. 이 때 주의할 점은 exit될 때 역시 락을 반드시 풀어야한다는 점이다.**int read(int fd, const void\* buff, unsigned size) {

  // if (size == 0){ // read-zero

  //   return 0;

  // }

  lock\_acquire(&f\_lock);

  int result;

  if(fd==0){

    for(int i=0; i<size; i++){

      uint8\_t c = input\_getc();

      if(c=='\0') {

        result = i;

        break;

      }

    }

  } else if(fd > 2){

    if(!thread\_current()->fd[fd]){

      lock\_release(&f\_lock);

      exit(-1);

    }

    lock\_release(&f\_lock);

    return file\_read(thread\_current()->fd[fd], buff, size);

  } else {

    lock\_release(&f\_lock);

    return -1;

  }

} **read함수 역시나 lock\_acquire과 lock\_release를 작업 전과 후에 추가하여 동기화 문제를 방지하는 것을 위의 코드를 통하여 확인할 수 있다.**

#ifdef USERPROG

  ...

  /\* user program 2 \*/

  bool terminate\_flag;

  struct semaphore f\_semaphore;

  struct file\* current\_file;

  struct thread\* parent;

  struct file\* fd[128]; /\* user prog 2 \*/

#endif#ifdef USERPROG

 sema\_init(&(t->clocking), 0);

 list\_init(&(t->child));

 sema\_init(&(t->mlocking), 0);

 sema\_init(&(t->f\_semaphore), 0);

 for(int i=0; i < 128; i++){ /\* user prog 2 \*/

  t->fd[i] = NULL;

 }

 t->parent = running\_thread();

 t->terminate\_flag = false;

 t->current\_file = NULL;

 list\_push\_back(&(running\_thread()->child), &(t->child\_elem));

#endif

} **또한, 프로세스 간의 동기화를 위하여 위와 같이 f\_semaphore 변수와 parent, 종료 state를 체크하는 플래스 변수 terminate\_flag, 현재 파일을 의미하는 current\_file 변수를 thread 구조체 내부에 추가하고 thread.c 의 init\_thread 함수에서 초기화하는 코드를 추가하였다.**tid\_t

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

{

.... 생략

 /\* Create a new thread to execute FILE\_NAME. \*/

 tid = thread\_create (command\_name, PRI\_DEFAULT, start\_process, fn\_copy);

 sema\_down(&thread\_current()->f\_semaphore);

 if (tid == TID\_ERROR)

  palloc\_free\_page (fn\_copy);

 for (struct list\_elem\* element = list\_begin(&thread\_current()->child); element != list\_end(&thread\_current()->child); element = list\_next(element)) {

  struct thread\* th = list\_entry(element, struct thread, child\_elem);

   if (th->terminate\_flag) {

    return process\_wait(tid);

   }

 }

 return tid;

} **위는 설명에 필요없는 부분을 생략한 process.c 의 process\_execute이다. 살펴보면 thread가 생성되고 난 후에 sema\_down을 호출하여 프로세스의 동기화 문제를 해결하기 위해 필요한 f\_semaphore을 내려준다. 이는 자식 쓰레드가 로드되기 전에 부모 프로세스가 끝나는 것과 같은 경우를 방지하여 준다. 또한 아래의 반복문은 terminate\_flag 변수를 확인하여 비정상적으로 종료된 자식 프로세스들을 wait하여 reap하기 위하여 추가하여 주었다.**static void

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

{

 .... 생략  
 success = load (command\_name, &if\_.eip, &if\_.esp);

 sema\_up(&thread\_current()->parent->f\_semaphore);

 if (!success) {

   thread\_current()->terminate\_flag = true;

   exit(-1);

 } **위의 코드를 확인해보면 위의 prcess\_execute에서 올려주었던 세마포어 변수를 load되고난 후에 올려주는 것을 확인할 수 있다. 아래의 load 부분에서 success 를 체크하여 이전에 선언한 terminate\_flag 플래그 변수를 true로 바꾸어준다. 이는 이전에 설명하였듯이 비정상적으로 종료된 자식 프로세스들을 체크하여 wait하기 위함이다.**

... 생략  
 if (!user) {

   exit(-1);

 }

 if (is\_kernel\_vaddr(fault\_addr)) {

   exit(-1);

 }

 if (not\_present) {

  exit(-1);

 }

... 생략 **마지막으로 위는 exception.c의 page\_fault 함수인데 not-present와 관련된 테스트들을 통과하기 위하여 exception.c 에서 not\_present를 확인하여 종료하는 코드를 추가하였다.**

* 1. **시험 및 평가 내용**

