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

담당 교수 / 분반 : 김영재 / 02

이름 / 학번 : 이도안 / 20201610

개발 기간 : 24.10.20 ~ 24.11.04

1. **개발 목표**

**Project 1에서 개발한 시스콜, 파일 loading을 바탕으로, 더 필요한 System call들을 개발한다. 이 때 개발에 필요한 파일 시스템들에 대해 쓰레드, 프로세스 간 발생할 수 있는 Syncronization 문제를 해결을 목표로 하여 개발한다.**

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

Project 1 에서는 다른 파일을 읽어서 별도의 쓰레드 fd에 저장하지 않고 시스템의 입, 출력으로 FD 0과 1을 사용하는 방식으로 진행되었다. 하지만 이 프로젝트에서는 각 파일을 읽고, 이를 입,출력 구분할 수 있는 방법인 FD가 필요하다. PINTOS에서는 STDERR로 FD 2를 사용하지 않기 때문에 2부터 자유롭게 활용하면 된다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

OPEN, CLOSE, CREATE, REMOVE, FILESIZE, SEEK, TELL 가 추가된다. 또한 read, write에서 stdin, stdout 뿐만 아니라 다른 파일의 입, 출력까지 실행할 수 있어야 한다. 이러한 시스템 콜들이 파일 시스템을 argument로 받아들여 각자의 역할을 수행할 수 있어야 한다.

3. Synchronization in Filesystem

Pintos가 멀티 쓰레드 기반으로 진행되는 가정 하에, 한 파일 시스템에 대해 여러 쓰레드나 프로세스가 접근할 수 있는 상황이 발생한다. 접근되고 있는 한 파일 시스템에 대해서 다른 쓰레드가 접근하여 변경하거나 읽으면, 예상할 수 없는 버그가 발생할 수 있는데, 이를 수행하거나 종료할 때 적절한 Synchronization이 필요하다. 이를 위해 적절한 mutex 또는 semaphore를 선언하여 lock, unlock을 수행해야 한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

FD 에 대한 구조체는 별도로 없어 file 구조체를 배열로 선언한 struct thread\_fd[128]을 thread.h에 선언하여 각 thread 마다 최대 128개(매뉴얼을 참조하여)의 fd를 가질 수 있게 하였다.

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)

* + - 1. CREATE

파일 이름을 인자로 받아 해당 파일에 해당하는 새로운 파일을 만든다. 생성 성공 유무를 따지는Boolean 값을 반환한다.

* + - 1. REMOVE

파일 이름을 인자로 받아 해당 파일에 해당하는 새로운 파일을 만든다. 생성 성공 유무를 따지는Boolean 값을 반환한다.

* + - 1. OPEN

파일 이름을 인자로 받아 해당 파일에 해당하는 파일을 연다. 파일 열기에 성공하면 배정된 fd를 반환하고, 배정에 실패하거나 열 수 없으면 -1을 반환한다.

같은 파일을 또 여는 것을 방지하는 예외 처리를 해준다.

* + - 1. CLOSE

인자로 받은 fd값에 해당하는 열린 파일을 찾아 닫는다. 열려 있지 않은 파일을 닫으려고 시도한 경우에 -1을 반환하는 예외 처리를 한다.

* + - 1. FILESIZE

인자로 받은 fd값에 해당하는 열린 파일을 찾아 그 파일의 size를 출력한다. Fd에 해당하는 파일이 존재하지 않거나, 읽을 수 없는 경우 -1을 출력하는 예외 처리를 한다.

* + - 1. READ

Project 1에서 구현된 부분에 더해, fd가 0이 아닐 경우 fd에 해당하는 파일을 읽고, synchronization과 같은 예외 처리를 추가적으로 한다.

* + - 1. WRITE

Project 1에서 구현된 부분에 더해, fd가 1이 아닐 경우 fd에 해당하는 파일을 쓰고, synchronization과 같은 예외 처리를 추가적으로 한다.

* + - 1. SEEK

인자로 받은 pos만큼 현재 파일 디스크립터의 파일 포인터를 이동시킨다. File\_seek 함수를 응용한다. 반환값이 없다.

* + - 1. TELL

현재 열린 파일의 현재 파일 포인터 위치를 반환한다.

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

이 프로젝트에서는 프로세스 간 싱크로나이제이션을 위해 semaphore를 사용하고, syscall에서 파일 입출력에 여러 프로세스가 동시 접근하는 것을 막기 위해 lock을 사용한다.

Semaphore는 init\_thread에서 초기화되며, 부모가 수행을 마친 후 child보다 일찍 종료되어, reaping 전에 종료되는 것을 막기 위해 부모에서 sema\_down을 걸어놓고, child에서 수행 종료 후 sema\_up을 수행하는 방식으로 구현하였다.

Syscall read와 write에서 각 파일에 접근할 때, lock\_acquire을 걸고, 접근이 완료된 후 lock\_release를 통해 파일 접근에 안정성을 강화하였다. Lock은 syscall\_init에서 초기화되어야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10.20 ~ 10.26 : syscall 및 fd 구현

10.27 ~ 10.30 : syscall과 file에 대한 synchronization 문제 해결

11.01 : 보고서 작성

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

1. FD

FD 에 대한 구조체는 별도로 없어 file 구조체를 배열로 선언한 struct thread\_fd[128]을 thread.h에 선언하여 각 thread 마다 최대 128개(매뉴얼을 참조하여)의 fd를 가질 수 있게 하였다.

fd구현에 대하여 별도의 함수는 선언하지 않았다.

2. System calls

[userprog/syscall.c]에서 read, write, seek, tell, create, remove, open, close, filesize에 해당하는 system call을 구현하였다. 또한, Exception.c 에서 page\_fault 함수에 다음을 추가해주었다.

if (is\_kernel\_vaddr(fault\_addr) || not\_present || user == NULL) {

    sys\_exit(-1);

  }

3. Synchronization in Filesystem: Lock, Semaphore

[userprog/syscall.c]에 f\_lock 의 이름을 가진 lock을 추가 선언하고, 이를 syscall\_init에서 lock\_init으로 초기화한다.

[userprog/thread.h]에서 loading\_sema 변수명의 semaphore를 추가 선언하여, 부모와 child의 synchronization문제를 해결하는 데에 활용한다. Process\_execute와 process\_start 함수에서 적절히 sema\_up, down을 추가해준다.

1. **연구 결과**
   1. **Flow Chart**

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

FD 에 대한 구조체는 별도로 없어 file 구조체를 배열로 선언한 struct thread\_fd[128]을 thread.h에 선언하여 각 thread 마다 최대 128개(매뉴얼을 참조하여)의 fd를 가질 수 있게 하였다.

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;                  /\* Page directory. \*/

    /\* Proj 1 \*/

    int exit\_status;

    struct list child\_list;

    struct list\_elem child\_elem;

    /\* Proj 2 \*/

    struct semaphore loading\_sema;

    struct semaphore mem\_lock;

    struct semaphore child\_lock;

   //  struct thread\* parent; 폐기

   //  struct semaphore parent\_lock; 폐기

    struct file\* thread\_fd[MAX\_FD];           /\* File Descriptor \*/

#endif

    /\* Owned by thread.c. \*/

    unsigned magic;                     /\* Detects stack overflow. \*/

    /\* Owned by syscall.c\*/

  };

위 보이는 것과 같이 struct file\* thread\_fd[MAX\_FD]; 로 선언해주었고, 여기서 매뉴얼에 의거해서 MAX\_FD는 128로 define 하였다.

2. System calls

[userprog/syscall.c] 에서 syscall\_handler() 함수안의 switch 문에 다음 내용을 추가한다.

//filesys

    //1

    case SYS\_READ:

      check\_addr(f->esp + 4);

      check\_addr(f->esp + 8);

      check\_addr(f->esp + 12);

      f->eax = sys\_read(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8), \*(uint32\_t \*)(f->esp + 12));

      break;

    case SYS\_WRITE:

      check\_addr(f->esp + 4);

      check\_addr(f->esp + 8);

      check\_addr(f->esp + 12);

      f->eax = sys\_write(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8), \*(uint32\_t \*)(f->esp + 12));

      break;

    //2

    //OPEN, CLOSE, CREATE, REMOVE, FILESIZE, SEEK, TELL

    case SYS\_OPEN:

      check\_addr(f->esp + 4);

      f->eax = sys\_open(\*(uint32\_t\*)(f->esp+4));

      break;

    case SYS\_CLOSE:

      check\_addr(f->esp + 4);

      sys\_close(\*(uint32\_t \*)(f->esp + 4));

      break;

    case SYS\_CREATE:

      check\_addr(f->esp + 4);

      check\_addr(f->esp + 8);

      f->eax = sys\_create(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8));

      break;

    case SYS\_REMOVE:

      check\_addr(f->esp + 4);

      f->eax = sys\_remove(\*(uint32\_t \*)(f->esp + 4));

      break;

    case SYS\_FILESIZE:

      check\_addr(f->esp + 4);

      f->eax = sys\_filesize(\*(uint32\_t \*)(f->esp + 4));

      break;

    case SYS\_SEEK:

      check\_addr(f->esp + 4);

      check\_addr(f->esp + 8);

      sys\_seek(\*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8));

      break;

    case SYS\_TELL:

      check\_addr(f->esp + 4);

      f->eax = sys\_tell(\*(uint32\_t \*)(f->esp + 4));

      break;

[3. B]에서 열거한 시스템 콜에 대한 수행들을 추가하였다. 또한, 매 시스템 콜마다 check\_addr함수를 실행하여 memory 접근에 대한 타당성을 체크해주었다.

int sys\_read (int fd, void \*buffer, unsigned size){

  int ret = -1; // default

  if (fd == 1 || fd < 0) sys\_exit(-1);

  if(thread\_current()->thread\_fd[fd] == NULL) sys\_exit(-1);

  if(buffer == NULL) sys\_exit(-1);

  check\_addr(buffer);

  if (fd == 0) {

    lock\_acquire(&f\_lock);

    for (int i = 0; i < size; i++) {

      \*((uint8\_t \*)buffer + i) = input\_getc();

    }

    lock\_release(&f\_lock);

    return size;

  }

  else{

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

    if(fp == NULL) sys\_exit(-1);

    lock\_acquire(&f\_lock);

    ret = file\_read(fp, buffer, size);

    lock\_release(&f\_lock);

    return ret;

}

  return ret;

}

**Sys\_read 함수에서는 file decriptor를 인자로 받아, buffer에 size만큼 파일을 읽는다. Fd가 0인 경우에는 stdin을 실행하며, 그 외 정상적인 경우에는 파일에 대해 입력을 실시한다. 이 때, [filesys/file.c]의 file\_read 함수를 사용한다. Synchronization에 관련해서는 3번에 서술하였다.**

int sys\_write (int fd, void \*buffer, unsigned size){

  int ret = -1; // default

  check\_addr(buffer);

  check\_addr(buffer + size - 1);

  if(buffer == NULL) sys\_exit(-1);

  if (fd < 1) return -1;

  if (fd == 1) {

    lock\_acquire(&f\_lock);

    putbuf(buffer, size);

    lock\_release(&f\_lock);

    return size;

  }

  else{//fd>=2 -- stdin out 아닐 때

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

    if(thread\_current()->thread\_fd[fd] == NULL) sys\_exit(-1);

    lock\_acquire(&f\_lock);

    ret = file\_write(thread\_current()->thread\_fd[fd], buffer, size);

    lock\_release(&f\_lock);

    return ret;

  }

  return ret;

}

**Sys\_write 함수에서는 file decriptor를 인자로 받아, buffer에 size만큼 파일을 읽는다. Fd가 1일 경우에는 stdout을 시행해주며, 그 외 정상적인 경우(fd 2이상)일 때는 파일에 대한 출력을 한다. 이 때, [filesys/file.c]의 file\_write 함수를 사용한다. 이 함수는 쓴 바이트 수만큼 리턴하는데, sys\_write함수의 리턴값도 같아진다.**

**Synchronization에 관련해서는 3번에 서술하였다.**

// OPEN, CLOSE+, CREATE+, REMOVE+, FILESIZE+, SEEK+, TELL+ << ADD

// chech\_thread 빼기.

bool sys\_remove (const char \*file) {

  if(file == NULL) sys\_exit(-1);

  return filesys\_remove(file);

}

**Sys\_remove 함수는 말 그대로 파일을 지우는 함수다. Filesys\_remove를 리턴해주는데, 성공 여부를 알려주는 boolean 값에 해당한다.**

int sys\_open(const char \*file){//fd 리턴

  //예외필요 open empty, missing --> (f->eax) 반환 안해줘서 그랬음.

  if(file == NULL) sys\_exit(-1);

  check\_addr(file);

  lock\_acquire(&f\_lock);

  struct file \*open\_f = filesys\_open(file);

  if(open\_f == NULL){

    lock\_release(&f\_lock);

    return -1;

  }

  struct thread\* cur = thread\_current();

  for(int i=2;i<128;i++){//빈 fd 찾기

    if(cur -> thread\_fd[i] == NULL){ // 빈 fd 찾음

      if(strcmp(cur->name, file) == 0) file\_deny\_write(open\_f);//열린 파일에 쓰기 방지

      thread\_current()->thread\_fd[i] = open\_f;

      lock\_release(&f\_lock);

      return i;

    }

  }

  //fd못찾을 때, default

  file\_close(file);

  lock\_release(&f\_lock);

  return -1;

}

**Sys\_open 함수는 open할 파일 이름을 인자로 받아 이를 filesys\_open() 함수를 통해 열어주는 system call이다. 이 함수는 file struct를 반환하고, open에 실패하면 null 값을 반환하여 -1을 반환하며 예외처리 해준다.**

**만약 open에 성공하면, 현재 thread의 fd array에서 빈 fd를 찾아 할당해준다. 이미 fd가 꽉 차있으면 어쩔 수 없이 다시 파일을 닫고 -1을 반환하며, fd를 찾는 데 성공했을 경우에는 해당 fd를 반환한다.**

int sys\_close(int fd){

  struct thread\* cur = thread\_current();

  if(fd==NULL || cur->thread\_fd[fd] == NULL) sys\_exit(-1);

  file\_close(cur->thread\_fd[fd]);

  cur->thread\_fd[fd] = NULL;

}

**Sys\_close 함수는 인자로 받은 fd에 해당하는 파일을 닫고, thread에서 그 fd를 NULL로 삭제한다. 반환값이 없는 void형 함수다.**

void sys\_seek (int fd, unsigned pos) {

  if(thread\_current()->thread\_fd[fd] == NULL) sys\_exit(-1);

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

}

**Sys\_seek 함수는 [filesys/file.c]의 file\_seek 함수를 사용하여 pos만큼 파일 포인터를 이동해주는 함수이다.**

unsigned sys\_tell (int fd) {

  if(thread\_current()->thread\_fd[fd] == NULL) sys\_exit(-1);

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

}

**Sys\_tell 함수는 [filesys/file.c]의 file\_tell 함수를 사용하여 pos만큼 파일 포인터를 이동해주는 함수이다.**

int sys\_filesize (int fd) {

  if(thread\_current()->thread\_fd[fd] == NULL ) sys\_exit(-1);

  if (fd == 0) return -1;

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

}

**Sys\_filesize 함수는 [filesys/file.c]의 file\_length 함수를 사용하여 인자로 전달받은 fd의 파일 사이즈를 리턴해주는 함수이다.**

bool sys\_create (const char \*file, unsigned size) {

  if(file == NULL) sys\_exit(-1);

  return filesys\_create(file, size);

}

Sys\_create 함수는 [filesys/filesys.c]의 filesys\_create 함수를 활용하여 인자로 전달받은 file의 이름과, size에 해당하는 파일을 생성하는 함수이다.

위 모든 함수에서 인자로 전달받은 file struct가 null인 경우, fd에 해당하는 thread의 파일이 null일 경우 모두 sys\_exit를 호출하여 예외처리하였다.

* Sys\_open에서 f->eax에 리턴값을 제대로 전달해주지 않아 open\_empty, missing의 오류가해결되지 않는 현상이 있었다. 이는 open 함수가 오류 출력인 -1을 전달해주지 않아 생기는 현상이라고 판단하였는데, 왜인지 모르게 fd 를 찾아 return 해주는 구문도 전달이 되지 않았을텐데 정상적인 테스트 케이스는 통과하였다.
* 추가 : Exception.c 파일에

if (is\_kernel\_vaddr(fault\_addr) || not\_present || user == NULL) {

    sys\_exit(-1);

  }

조건문을 추가해주어 page fault로 인한 커널 패닉을 방지하였다.

Fault\_addr가 커널의 가상 주소 공간에 있는지 확인하고, 메모리 페이지가 현재 존재하는지, user포인터가 NULL인지 확인하여 비정상인 경우 -1 로 비정상 종료한다.

3. Synchronization

int sys\_open(const char \*file){//fd 리턴

  //예외필요 open empty, missing --> (f->eax) 반환 안해줘서 그랬음.

  if(file == NULL) sys\_exit(-1);

  check\_addr(file);

  lock\_acquire(&f\_lock);

  struct file \*open\_f = filesys\_open(file);

  if(open\_f == NULL){

    lock\_release(&f\_lock);

    return -1;

  }

  struct thread\* cur = thread\_current();

  for(int i=2;i<128;i++){//빈 fd 찾기

    if(cur -> thread\_fd[i] == NULL){ // 빈 fd 찾음

      if(strcmp(cur->name, file) == 0) file\_deny\_write(open\_f);//열린 파일에 쓰기 방지

      thread\_current()->thread\_fd[i] = open\_f;

      lock\_release(&f\_lock);

      return i;

    }

  }

  //fd못찾을 때, default

  file\_close(file);

  lock\_release(&f\_lock);

  return -1;

}

**위 sys\_open 함수에서, 파일에 접근하는 filesys\_open을 하기 전에 여러 프로세스가 한번에 같은 파일에 접근하여 오류를 발생하는 것을 방지하기 위해 미리 선언해두었던 f\_lock에 대해 lock\_acquire를 실행한다. 그 후 open 처리가 완료되어 리턴하기 전, release를 수행한다.**

**또한, 현재 실행중인 파일에 대하여 쓰기 작업을 수행할 경우 오류를 일으킬 수 있으므로 open하려던 파일이 현재 파일과 같으면 [filesys/file.c]의 file\_deny\_write 함수를 불러와 현재 열린 파일에 쓰는 것을 방지해준다.**

int sys\_read (int fd, void \*buffer, unsigned size){

  int ret = -1; // default

  if (fd == 1 || fd < 0) sys\_exit(-1);

  if(thread\_current()->thread\_fd[fd] == NULL) sys\_exit(-1);

  if(buffer == NULL) sys\_exit(-1);

  check\_addr(buffer);

  if (fd == 0) {

    lock\_acquire(&f\_lock);

    for (int i = 0; i < size; i++) {

      \*((uint8\_t \*)buffer + i) = input\_getc();

    }

    lock\_release(&f\_lock);

    ret = size;

  }

  else{

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

    if(fp == NULL) sys\_exit(-1);

    lock\_acquire(&f\_lock);

    ret = file\_read(fp, buffer, size);

    lock\_release(&f\_lock);

  }

  return ret;

}

**Sys\_read 에서는 fd == 0 일때 stdin을 받고, fd 2 이상일 때 file\_read를 수행한다. 그 수행을 하는 동안 다른 프로세스의 접근을 막기 위해 syscall.c에서 선언한 f\_lock에 대하여 락을 걸었다가 수행을 끝낸 후 다시 락을 풀어준다.**

int sys\_write (int fd, void \*buffer, unsigned size){

  int ret = -1; // default

  check\_addr(buffer);

  check\_addr(buffer + size - 1);

  if(buffer == NULL) sys\_exit(-1);

  if (fd < 1) return -1;

  if (fd == 1) {

    lock\_acquire(&f\_lock);

    putbuf(buffer, size);

    lock\_release(&f\_lock);

    ret = size;

  }

  else{//fd>=2 -- stdin out 아닐 때

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

    if(thread\_current()->thread\_fd[fd] == NULL) sys\_exit(-1);

    lock\_acquire(&f\_lock);

    ret = file\_write(thread\_current()->thread\_fd[fd], buffer, size);

    lock\_release(&f\_lock);

  }

  return ret;

}

**Sys\_write 에서는 fd == 1 일때 stdout을 하고, fd 2 이상일 때 file\_write 함수를 수행한다. 그 수행을 하는 동안 다른 프로세스의 접근을 막기 위해 syscall.c에서 선언한 f\_lock에 대하여 락을 걸었다가 수행을 끝낸 후 다시 락을 풀어준다.**

**Lock\_acquire를 수행하면 release되기 전에 다른 프로세스가 lock\_acquire을 수행했을 때, 해당 lock에 대해서 lock\_release되기까지 기다리게 된다. 이를 통해 synchronization을 맞춰줄 수 있다.**

**Synchronization을 구현하며 가장 어려웠던 마지막 문제 상황은 syn read, syn write와 multi-oom이었는데, 나는 multi-oom을 해결하는 것에 초점을 두고 연구하였다.**

**Multi-oom 테스트 케이스를 열어보니, 수많은 child를 실패할 때까지 생성하고 이를 통해 메모리 누수를 체크하는 상황이었다.**

**//process\_execute**

// oom 해결

  struct thread \*child;

  child = ret\_child(tid)//만든 tid랑 같은 child 찾아서, 100% 찾음.

  sema\_down(&(child->loading\_sema));// 차일드 락 걸어. 걸려있으면 기다려.

                                  // 차일드 실행중이면 기다리는 거

**//start\_process**

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

  /\* If load failed, quit. \*/

  palloc\_free\_page (file\_name);

  //execute에서 자식 프로세스 실행

  //start\_process에서 자식 로드 기다렸다가

  //로드 실패하면 -1 받아야 함.

  sema\_up(&(thread\_current()->loading\_sema)); // 나 실행 끝났으니까 부모 실행해도 돼

위 코드에 사용된 user-defined ret\_child 함수는 다음과 같다.

struct thread\* ret\_child(tid\_t tid){

  struct thread \*cur = thread\_current();

  for (struct list\_elem \*elem = list\_begin(&(cur->child\_list)); elem != list\_end(&(cur->child\_list)); elem = list\_next(elem)) {

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

    if (tid == child->tid) {

      return child;

    }

  }

}

Thread의 child 리스트를 탐색해 원하는 tid에 해당하는 child의 쓰레드 포인터를 리턴하는 함수.

**다소 주관적인 주석 처리이나, 봤을 때 가장 이해하기 쉬워 남겨놓았다.**

**Execute 함수에서는 현재 실행중인 파일이 parent가 되고, start되는 파일이 child가 된다. 따라서, 방금 실행시킨 pid와 같은 child를 찾고, 그 child가 load를 마치고 sema\_up을 해줄 때 까지 sema\_down에서 기다리게 된다. 이후 실패한 프로세스에 대해서는 wait을 통해 reaping 해주고, 그 외에는 pid를 리턴해준다.**

**이 과정을 통하여 load에 실패하여 execute에 대해 exit\_status가 -1인 reaping을 못하던 상황, 메모리 누수와 같은 상황을 방지하였다.**

**여기서 loading\_sema를 사용하는데, 각 쓰레드에 대해서 한 번 씩만 필요할 것 같아 lock으로 수행해보았으나 오류가 발생하였다. 이유는 과제 중 끝내 찾지 못했고 운영체제 수업을 통해, 그리고 더 공부를 해서 이유를 알아내보려 한다.**

**이렇게 load에 대한 synchronization을 해결해주니 read, write에 대한 synch도 맞춰져 나머지 모든 문제가 해결되었다.**

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

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷, 텍스트, 블랙이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

80개의 테스트 케이스 모두 cspro10 환경에서 잘 수행되는 것을 확인하였다.