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

담당 교수 / 분반 : 박성용 / 1

이름 / 학번 : 정지석 / 20181687

개발 기간 : 10/11 ~10/22

1. **개발 목표**

Project 1에서 구현하지 않았던 file system 관련 system call들을 모두 구현한다(create, remove, open, close, file size, read, write). 또한 filed을 read, write 할 때 여러 개의 process가 동시에 접근하지 못 하도록 synchronization을 잘 맞춰주어야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. File Descriptor

Process는 file descriptor를 통해 file에 접근하여 read와 write를 수행한다. 이번 실습에서 이러한 file descriptor를 직접 구현하여 각 process들이 file에 access 할 수 있도록 한다.

1. (이번 프로젝트에서 구현해야 하는) System Calls

File system과 관련된 system call들을 직접 구현하여 해당 명령어들이 입력되었을 때에 제대로 실행되는지를 확인한다.

1. Synchronization in Filesystem

Synchronization in Filesystem을 구현하여 여러 process가 하나의 file을 동시에 read 혹은 write하는 것을 방지한다.

* 1. **개발 내용**

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

File descriptor를 구현하기 위해 file 구조체를 사용하였다(file 구조체는 file.c에 정의되어 있다.). 각각의 thread는 서로 독립적으로 작업을 수행해야 하므로 file descriptor를 배열의 형태로 선언해 주었다.

1. System Calls: 구현할 각 system call에 대해 간략히 서술 (하나의 system call 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

Create(): file의 주소와 size를 받아 새로운 file을 생성한다. File 생성 성공시에 true를, 실패했을 경우 false를 return한다.

Remove(): file의 주소를 받아 해당 file을 삭제한다. File 삭제 성공시에 true를, 실패했을 경우 false를 return한다.

Open(): file의 주소를 받아 file을 열어주는 역할을 한다. 이때 다른 process들이 file에 access할 수 없도록 lock\_acquire, lock\_release 함수를 이용해 locking을 해주어야 한다.

Filesize(): file descriptor를 받아 해당 file의 크기를 return해준다.

Read(): file descriptor와 buffer, file의 size를 받아와 file을 읽는 함수이다. Fd가 0일 경우 키보드로 직접 입력한 값을 읽고, fd가 2보다 큰 경우에는 file을 읽어서 return한다. File을 읽을 때에는 lock\_acquire, lock\_release 함수를 이용해 다른 process들의 접근을 막는다.

Write(): read 함수와 동일한 parameter를 받아와 file에 내용을 적는 함수이다. File에 write할 때에는 lock\_acquire, lock\_release 함수를 이용해 다른 process들의 접근을 막는다.

Seek(): file descriptor와 위치 값을 받아와 file에서 해당 위치로 이동한다.

Tell(): file descriptor를 받아 현재 file에서의 위치를 return한다.

Close(): file descriptor를 입력으로 받아 해당 fd를 close한다.

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

어떤 process가 file을 access하여 read나 write를 실행 중인 경우에는 file system에서 다른 process들이 file에 access하지 못하도록 해주어야 한다. System call에서 read나 write를 실행할 때 lock\_acquire, lock\_release 함수를 사용하여 현재 access한 process 이외의 다른 process 들은 access할 수 없도록 lock 해준다.

또한 children process가 memory에 load 되기 전에 parent process가 종료되는 것을 방지하기 위해 sema\_up, sema\_down 함수를 사용하여 children process가 종료된 후에 parent process가 실행될 수 있게 해준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/11 ~ 10/15: system call 구현

10/15 ~ 10/20: synchronization 문제 해결

10/21 ~ 10/22: 보고서 작성

* 1. **개발 방법**

/userprog/exception.c

*if* (!user || is\_kernel\_vaddr(fault\_addr) || not\_present) {

    exit(-1);

  }

If문에 not\_present를 추가해준다.

/threads/thread.h

   struct semaphore exit;

   struct semaphore wait;

   struct semaphore load;

   struct file \*fd[128];

*/\* Owned by thread.c. \*/*

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

File descriptor 배열과 synchronization을 위해 추가적인 semaphore를 선언해 주어야 한다.

/threads/thread.c

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

  sema\_init(&(t->exit), 0);

  sema\_init(&(t->wait), 0);

  sema\_init(&(t->load), 0);

  list\_init(&(t->child)); *// child list  initialize*

*for* (int i = 0; i < 128; i++)

    t->fd[i] = NULL;

init\_thread에서 thread를 initialization할 때 위에서 선언한 semaphore와 file descriptor 배열도 추가해 주어야한다.

/userprog/syscall.c

bool create(const char \**file*, unsigned *initial\_size*);

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

int open(const char \**file*);

int filesize(int *fd*);

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

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

void seek(int *fd*, unsigned *position*);

unsigned tell(int *fd*);

void close(int *fd*);

이번 file system 관련 system call 함수들을 구현해 준다(아래에서 자세히 설명할 예정).

/userprog/process.c

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

*while*(search!=list\_end(&(thread\_current()->child)))

  {

    tc = list\_entry(search, struct thread, child\_elem);

*if*(tc->exit\_stat == -1)

*return* process\_wait(tid);

    search = list\_next(search);

  }

Process\_execute 함수에서 자식 process가 모두 종료될 때까지 부모 process가 기다리도록 코드를 추가해준다.

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

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

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

start\_process 함수에서는 자식 process가 모두 load된 후에 sema\_up을 통해 부모 process를 unblock 해준다.

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

  int ExitStatus  = -1;

  sema\_down(&(child\_thread->wait));

  ExitStatus = child\_thread->exit\_stat;

  list\_remove(&(child\_thread->child\_elem));

  sema\_up(&(child\_thread->exit));

*return* ExitStatus;

process wait 함수는 부모 thread에게 자식 thread가 있는지 확인하고, 있다면 자식 thread를 wait하고 이후 자식 thread가 종료되면 그 exit status를 부모 thread에게 넘겨주는 역할을 수행한다.

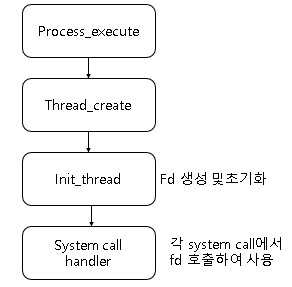
process\_exit (void)

    sema\_up(&(thread\_current()->wait));

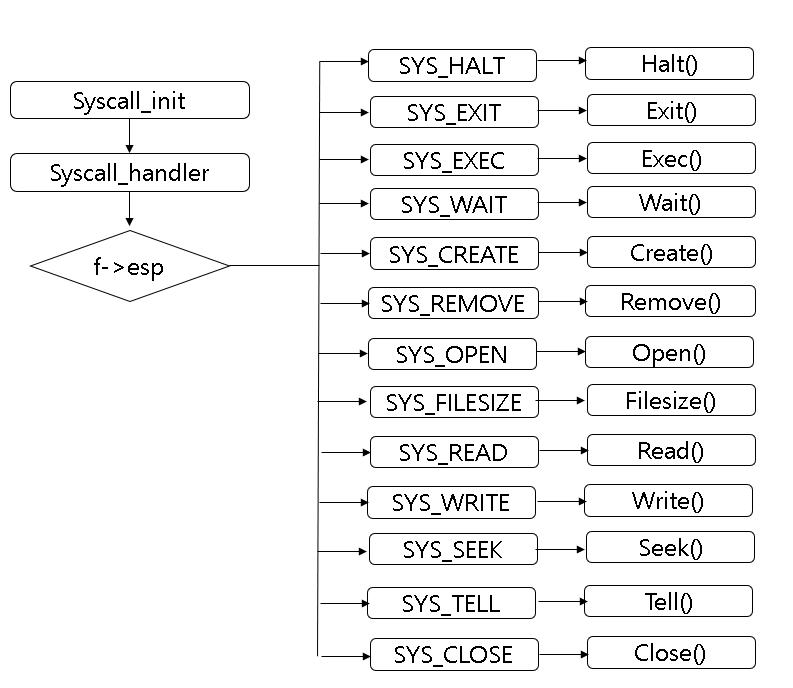
    sema\_down(&(thread\_current()->exit));

process\_wait 함수에는 thread를 wait한 뒤 exit하는 부분을 추가해주었다.

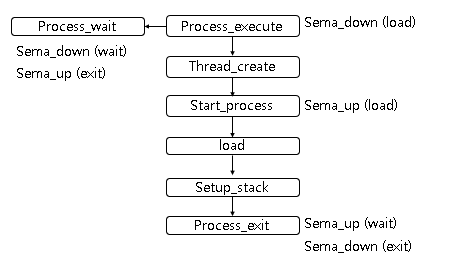
1. **연구 결과**
   1. **Flow Chart**
      1. File descriptor

S

* + 1. System call



* + 1. Synchronization

SS

* 1. **제작 내용**

1. File descriptor

threads/thread.h

   struct file \*fd[128];

thread를 생성할 때에 그에 대한 file descriptor 또한 함께 생성해준다. 각 thread는 independent하기 때문에 file descriptor는 여러 개가 필요하다. 따라서 fd를 선언할 때 배열을 사용하게 된다.

Threads/thread.c

*for* (int i = 0; i < 128; i++)

    t->fd[i] = NULL;

thread가 배열의 형태를 띄기 때문에 initialization 해줄 때에도 for과 같은 반복문을 통해 모든 fd 배열을 initialize 해주어야 한다.

Userprog/syscall.c

    lock\_acquire(&f\_lock);

    tmp = file\_read(thread\_current()->fd[fd], buffer, size);

    lock\_release(&f\_lock);

위에서 선언한 fd는 위와 같이 system call을 처리할 때 사용된다. 이때, 다른 process의 접근을 막기 위해 lock\_acquire, lock\_release 함수를 사용한다.

2.System call

Userprog/syscall.c

create(const char \**file*, unsigned *initial\_size*)

{

*if* (file == NULL)

    exit(-1);

*return* filesys\_create(file, initial\_size);

}

Filesys\_create 함수를 사용해 새로운 file을 생성한다.

remove(const char \**file*)

{

*if* (file == NULL)

    exit(-1);

*return* filesys\_remove(file);

}

Filesys\_remove 함수를 사용해 해당하는 file을 삭제한다.

open(const char \**file*)

{

*if* (file == NULL || !is\_user\_vaddr(file))

    exit(-1);

  lock\_acquire(&f\_lock);

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

*if* (fp == NULL)

  {

    lock\_release(&f\_lock);

*return* -1;

  }

*else*

  {

    int tmp = -1, i = 3;

*while* (i < 128)

    {

*if* (thread\_current()->fd[i] == NULL)

      {

*if* (strcmp(thread\_current()->name, file) == 0)

          file\_deny\_write(fp);

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

        tmp = i;

*break*;

      }

      i++;

    }

    lock\_release(&f\_lock);

*return* tmp;

  }

}

Filesys\_open 함수를 사용해 file을 열어준다. File descriptor가 배열 형태이기 때문에 for문 같은 반복문을 사용해 각 fd를 할당해준다. 다른 process의 접근을 방지하기 위해 작업 중에는 lock을 걸어준다.

filesize(int *fd*)

{

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

}

File\_length 함수를 이용해 file의 size를 return한다.

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

{

*if* (!is\_user\_vaddr(buffer))

    exit(-1);

*if* (fd == 0)

  {

    int i = 0;

*for* (i = 0; i < (int)size; i ++)

*if*(((char \*)buffer)[i] == '\0')

*break*;

*return* i;

  }

*else* *if* (fd >= 3)

  {

    int tmp = -1;

*if* (thread\_current()->fd[fd] == NULL)

      exit(-1);

    lock\_acquire(&f\_lock);

    tmp = file\_read(thread\_current()->fd[fd], buffer, size);

    lock\_release(&f\_lock);

*return* tmp;

  }

}

File\_read 함수를 사용해 해당 file을 읽어준다. 다른 process의 접근을 방지하기 위해 작업 중에는 lock을 걸어준다.

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

{

*if* (!is\_user\_vaddr(buffer))

    exit(-1);

*if* (fd == 1)

  {

    putbuf(buffer, size);

*return* size;

  }

*else* *if* (fd >= 3)

  {

    int tmp = -1;

*if* (thread\_current()->fd[fd] == NULL)

      exit(-1);

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

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

    lock\_acquire(&f\_lock);

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

    lock\_release(&f\_lock);

*return* tmp;

  }

}

File\_write 함수를 사용해 해당 file에 원하는 내용을 입력한다. 다른 process의 접근을 방지하기 위해 작업 중에는 lock을 걸어준다.

seek(int *fd*, unsigned *position*)

{

*if* (thread\_current()->fd[fd] == NULL)

    exit(-1);

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

}

File\_seek 함수를 사용하여 찾고자 하는 위치로 이동한다.

tell(int *fd*)

{

*if* (thread\_current()->fd[fd] == NULL)

    exit(-1);

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

}

File\_tell 함수를 사용하여 file에서 현재 위치를 반환한다.

close(int *fd*)

{

*if* (thread\_current()->fd[fd] == NULL)

    exit(-1);

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

  thread\_current()->fd[fd] = NULL;

}

File\_close 함수를 사용하여 해당 fd를 닫아준다.

3.Synchronization

Threads/thread.h

   struct semaphore exit;

   struct semaphore wait;

   struct semaphore load;

부모 process가 자식 process를 기다리는 작업을 해주기 위해서는 semaphore가 여러 개 필요하다. load semaphore외에 추가적으로 exit, wait semaphore를 선언해 준다.

Threads/thread.c

  sema\_init(&(t->exit), 0);

  sema\_init(&(t->wait), 0);

  sema\_init(&(t->load), 0);

3개의 semaphore를 모두 initialize 해준다.

Userprog/process.c

  sema\_down(&(child\_thread->wait));

  ExitStatus = child\_thread->exit\_stat;

  list\_remove(&(child\_thread->child\_elem));

  sema\_up(&(child\_thread->exit));

process\_wait 함수에서 child가 모두 load될 때가지 wait한다. 이후 종료되면 exit status를 parent에게 넘겨준다. 그후 exit를 sema\_up하여 child를 종료한다.

  sema\_down(&thread\_current()->load);

process\_execute 함수에서 process를 실행할 때 sema\_down을 하여 child process가 load될 때까지 기다린다.

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

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

child process가 모두 load 된 것을 확인하면 start\_process 함수에서 sema\_up을 하여 parent process가 실행될 수 있도록 한다.

    sema\_up(&(thread\_current()->wait));

    sema\_down(&(thread\_current()->exit));

process\_exit 함수에서는 process가 종료되었을 때 wait를 sema\_up해주고, exit을 sema\_down해줘 종료를 기다린다.

* 1. **시험 및 평가 내용**
* 