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

담당 교수 / 분반 :김영재 교수님 1반

이름 / 학번 : 박성우/ 20171637

개발 기간 :10.01~10.08

1. **개발 목표**

* 프로젝트 1에서는 구현하지 않았던 system call들 중 read, write, create, remove, open, close, seek, tell, filesize 를 구현한다. 또한 기존의 process\_wait 의 waiting 방식을 thread\_yield() 에서 semaphore 를 사용하는 방식으로 바꾸어 준다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1. File Descriptor

- 각 thread 는 file descriptor 라는 것을 통해 file 을 열고 읽고 쓰고 닫고 등등을 할 수 있다. 그러므로 thread 구조체에 filesys/file.c 에 있는 file 구조체를 추가해주어 각 thread가 파일 관련 일을 처리할 수 있도록 한다.

2. (이번 프로젝트에서 구현해야 하는) System Calls

- file system과 관련된 system call handler와 함수들을 구현하여 기존에 있는 API들을 바탕으로 file system이 제대로 실행 될 수 있도록 해준다. 이번에 구현해야 할 system call 들은 create, remove, open, close, read, write, seek, tell, filesize 이다.

3. Synchronization in Filesystem

- 특정 file 에 관하여 여러 개의 thread 가 동시에 system call 을 하게 되면 얘기치 못한 상황이 생긴다.. 따라서 file 을 열고 일고 쓸 때 lock 을 사용하여 synchronize 할 것이고 열었을 때 file\_deny\_write 을 이용해 다른 thread 들이 write 하지 못하게 막는 부분을 구현한다.

* 1. **개발 내용**

1. File Descriptor: 구현에 이용할 자료구조와 선택한 이유를 서술

- thread 구조체에 file 구조체를 member로 추가하였다. 매뉴얼에는 하나의 프로세스 당 128개의 파일을 열 수 있다하여 기본 지정된 0,1,2 를 포함해서 총 131 칸의 file 형 배열을 선언하였다. 배열로 선언한 이유는 배열이 더 익숙해서 였다.

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)

- create: 원하는 name 과 size를 가지고 filesys\_create 을 호출하여 진짜로 file 을 생성한다. 생성에 성공하면 true, 실패하면 false 를 반환한다.

- remove: 특정한 이름을 갖는 file 을 삭제해준다. 성공하면 true, 실패하면 false 를 반환한다

- open: 특정한 이름을 갖는 file을 열어주는 함수로 filesys\_open 함수를 호출하여 file을 open한다. 파일을 여는데 실패하면 -1을 반환하고 성공하면 file descriptor 번호를 반환한다.

- close: 특정한 이름을 갖는 file을 닫아 주는 함수로 files\_close를 호출하여 file 을 닫는다. 닫기전에 현재 thread 의 file 구조체에서 file 이 열려있던 번호에 가서 null 로 바꾸어 주고 닫는다.

- filesize: 받아온 file descriptor 값에 해당하는 file을 찾아 그 file의 size 를 반환한다.

- seek: 받아온 file descriptor 값에 해당하는 file을 찾아서 해당 file 이 다음으로 읽히거나 쓰일 위치를 새로 지정해준다.

- tell : 받아온 file descriptor 값에 해당하는 file을 찾아서 해당 file의 어느 부분이 참조되고 있는 지 반환한다.

- read: 받아온 file descriptor 값에 해당하는 file에서 size 만큼 buffer로 읽어 들인다. 실재로 읽은 byte 를 반환하거나 어떠한 이유로 실패하면 -1 을 반환한다.

- write: 받아온 file descriptor 값에 해당하는 file에 buffer 에서 size 만큼의 byte 를 쓴다. 쓸 수 있을 만큼 쓰고 실재 쓴 byte 만큼을 반환한다. 어떠한 이유로 하나도 못쓰면 0을 반환한다.

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

이번 프로젝트에서는 여러 개의 프로세스가 동시에 파일을 열고 읽고 쓰는 것을 막기 위해 lock 을 통해 synchronize 를 하였다. Open, read, write system call 수행 중에서 시작할 때 lock\_acquire 함수를 불러서 다른 프로세스의 접근을 막고 끝나면 lock\_release 함수를 호출하여 lock 을 풀었다. 그리고 한 프로세스가 파일을 열었을 때 file\_deny\_write 를 호출하여서 다른 프로세스가 그 파일에 write 하는 것을 방지하였다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 10/1 : background와 file descriptor 에 대한 전반적 이해\
* 10/2~10/7: system call 구현 및 file system synchronization 구현 그리고 끊임없는 수정과 수정
* 10/7~10/8: 보고서 작성
  1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드

<Process.c>

process\_execute()에서는 쓰레드의 자식이 생성되면 자식 쓰레드의 load가 완전히 끝날때 까지 sema\_down하여 잠들어 있다가 자식 쓰레드의 load가 끝난 후엔 sema\_up하여 다시 깨게 된다. 추가로 혹시나 load 가 fail 되어 그대로 thread\_exit() 되어서 process\_exit() 의 sema\_down 에서 막혀있는 자식 thread 들이 있다면 exit status 를 회수하고 목숨을 끊어줘야 한다.

start\_process() 에서는 load 함수가 끝나면 부모가 계속 진행될 수 있게 sema\_up 해주고 load 에 실패하면 flag 를 새우고 thread\_exit() 을 통해 과감히 죽는다.

Process\_wait() 에서는 프로젝트1에서 thread\_yield() 를 통해 wait 한 것을 semaphore 를 통해 기다리는 것으로 바꾼다. Semaphore를 2개를 사용해서 부모 process 는 child process 가 죽기 직전 까지 멈춰 있다가 child process가 죽기 직전에 일어나서 child process의 exit status를 받고 child process를 죽게 하고 계속 진행해 나간다.

Process\_exit() 에서는 process\_wait 과 티키타카하는 semaphore 2개를 적절히 사용해주고 이 프로세스가 죽기전에 기존에 하는 일에 더해서 열려 있던 file들을 다 닫아줘야 한다.

<exception.c>

Unmapped virtual memory를 참조하는 것을 추가로 막아야 한다. 여기서 not\_present 변수가 true 일 때에 대해 exit(-1)로 처리를 해준다.

* + 수정하거나 추가해야 하는 자료구조

<thread/thread.h>

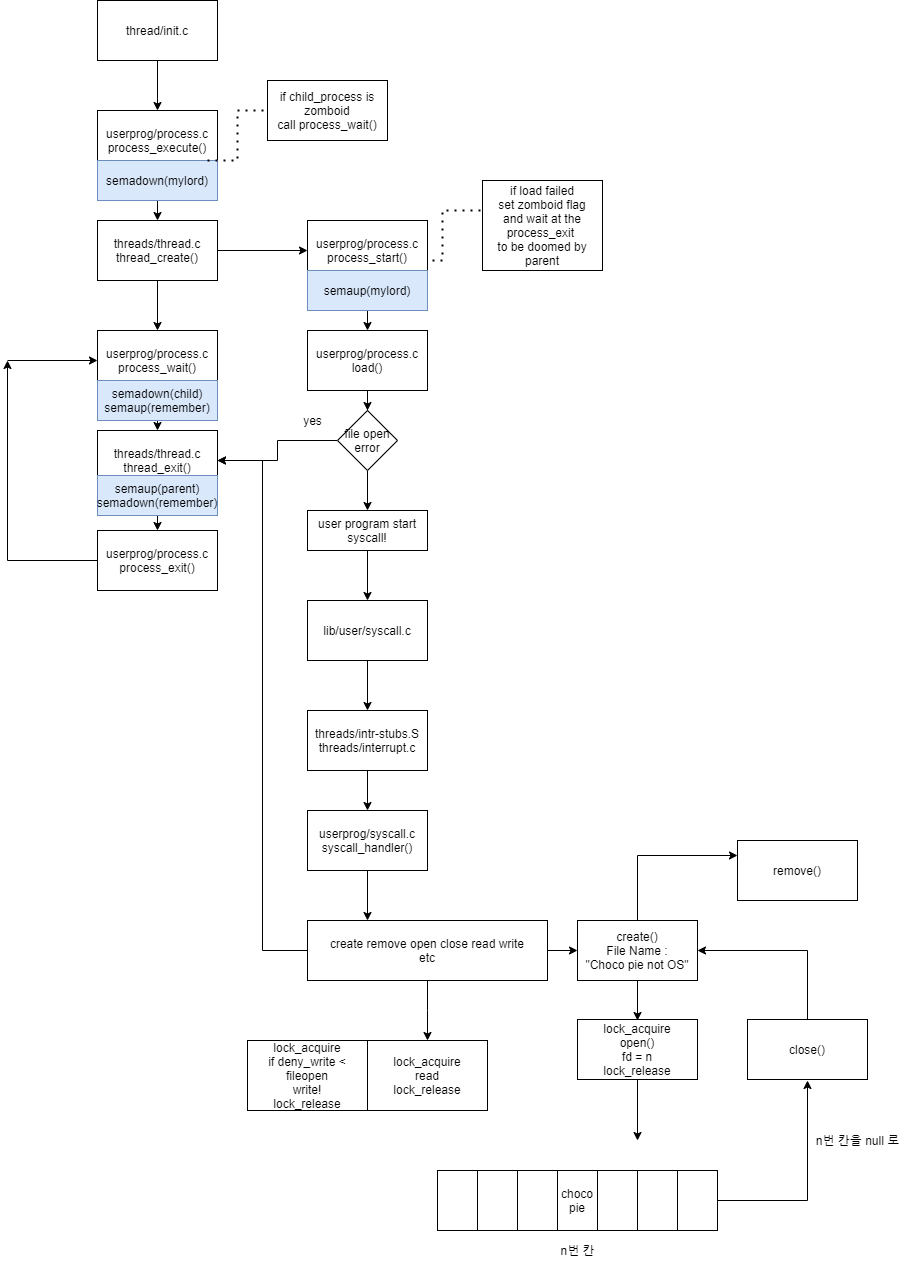
file descriptor를 구현하기 위해 thread struct에 struct file\* file\_des[131]를 추가한다. 그리고 process1 에서 만든 item 구조체는 이제 필요 없으니 삭제한다..

* + 수정하거나 추가해야 하는 함수

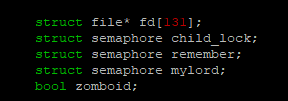
file system과 관련된 system call들을 handle할 수 있도록 필요한 함수를 추가한다.

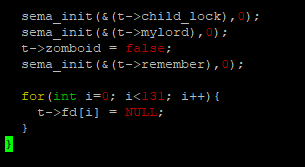
그리고 필요한 경우 lock 을 통해 동기화를 한다.

1. **연구 결과**
   1. **Flow Chart**

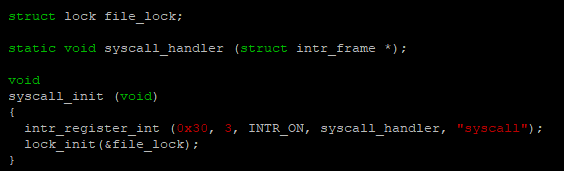
* 
  1. **제작 내용**

filesys/file.c의 file struct를 사용할 수 있도록 threads/thread.h의 struct thread에 아래의 내용을 추가해 주고 thread/thread.c 의 초기화를 꼭 해주자. 하나의 thread 에서 최대 128개 까지 파일을 열 수 있다하여 기본 지정 3개에 128개를 더해서 131칸의 배열을 선언하였다. 그리고 동기화를 위해 사용할 semaphore 3개도 선언하고 초기화 해주었다.



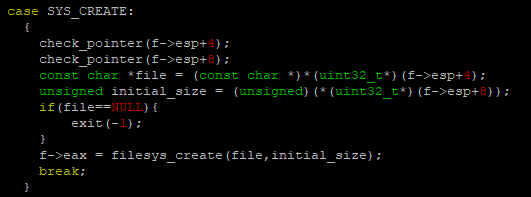


이제 userprog/syscall.c 로 가서 구현해야 할 system call 들을 살펴봐야한다. 우선 나중에 file synchronization 을 위한 lock 을 전역변수로 선언하고 syscall\_init 이 실행될 때 lock\_init 함수로 초기화를 해주었다.



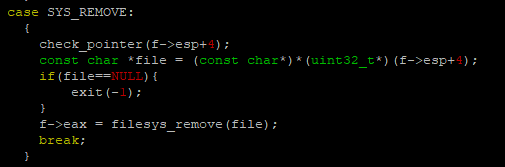
System call handler의 file 관련 system call 을 확인하자.

Create()



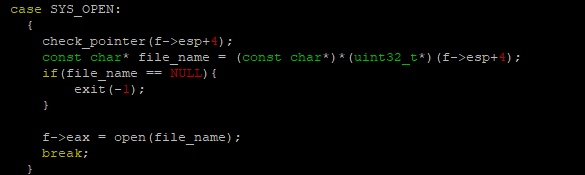
Create system call 이다. File 의 이름과 size를 받은 후 혹시 file 이 null 이면 exit 하고 아니면 filesys\_create 을 통해 파일을 만들어보고 성공했는지 실패했는지 eax 에 넣어준다.

Remove()

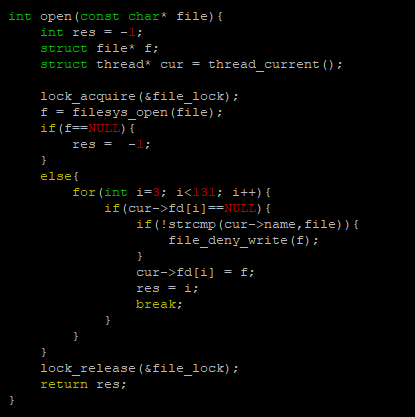


Remove system call 이다. File 의 이름을 받아서 혹시 file 이 null 이면 exit 하고 아니면 filesys\_remove 함수로 file 을 제거한다. 그리고 성공했는지 여부를 eax 저장한다.

Open()

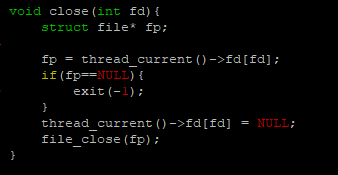


File 이름을 받아와서 file 이 null 이면 exit 하고 아니면 내가 만든 open 함수를 실행하고 그 결과값을 eax 에 넣어준다. 아래가 open 함수이다.



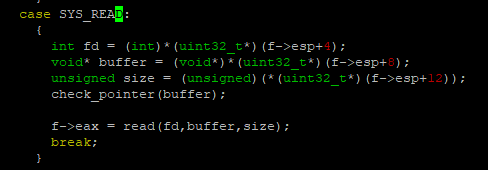
여기서 synchronization 이 사용된다. File 을 열 때 다른 간섭이 없어야 한다. 따라서 lock\_acquire 와 lock\_release 함수를 사용해서 다른 간섭이 없게 동기화를 하였다. Pintos 명세서에 적힌대로 무언가가 잘못되면 -1, 제대로 열리면 file descriptor 배열에 그 file을 추가하고 몇 번 칸에 추가하였는지를 반환한다. 만약 현재 load된 파일, 즉 실행파일이라면 이 파일에 대해서는 write가 불가능하게 해주어야한다. 이 경우엔 파일이름이 쓰레드의 이름과 동일 하기 때문에 이와 비교하여 동일한 경우엔 file\_deny\_write를 호출하여 해당 파일에 write하는것을 막아준다

Close()

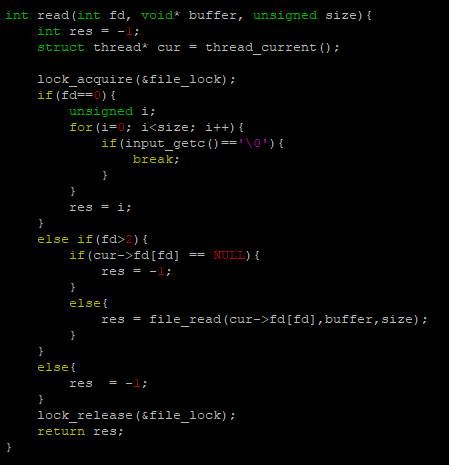


Close system call 이다. 현재 thread 의 file descriptor 에 이 파일이 들어있던 칸을 null 로 바꾸어 메모리 누수를 막고 file\_close 를 불러 file을 닫아버린다.

Read()

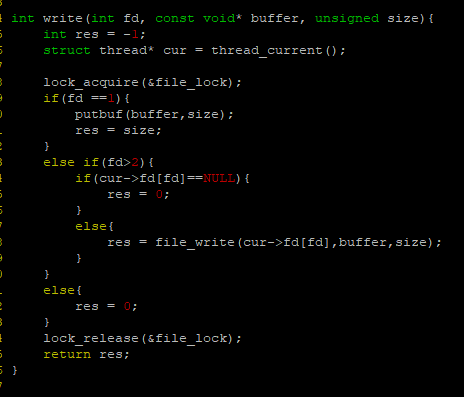


Read system call 이다. File, buffer, size 를 받아오고 buffer 가 올바른 위치인지 꼭 확인한다. 그리고 read 함수를 불러 파일을 읽는다. 아래가 read 함수이다.



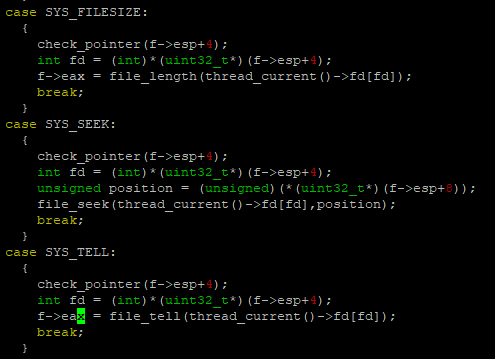
Read 할 때도 동기화를 해주어야 하므로 lock\_acquire 와 lock\_release 를 해준다. Fd 가 stdin 일 경우 프로젝트1 때 했던 그대로이고 만약 stdin이 아니면 여기서 파일을 읽는다. 그런데 파일이 null 이면 -1 을 리턴하고 제대로 열려있으면 file\_read 로 읽고 실제 읽은 byte 수를 리턴한다. 그 외에 모든 경우는 무언가가 잘못된 것이므로 역시 -1 을 리턴한다.

Write()



Write 할 때도 동기화를 해주어야 하므로 lock\_acquire 와 lock\_release 를 해준다. Fd 가 stdout 일 경우 프로젝트1 때 했던 그대로이고 만약 stdout이 아니면 여기서 파일에 write한다. 명세서에 적힌 그대로 코드를 짠다. 파일이 null 이면 0 을 리턴하고 제대로 열려있으면 file\_write 으로 쓰고 실제 쓴 byte 수를 리턴한다. 그 외에 모든 경우는 무언가가 잘못되어 한 개도 write 하지 못하는 경우이므로 0을 리턴한다.

Filesize, seek, tell



나머지 3가지 system call 은 주어진 kernel api 들을 그냥 가져다 쓰면 된다.

Userprog/exception.c



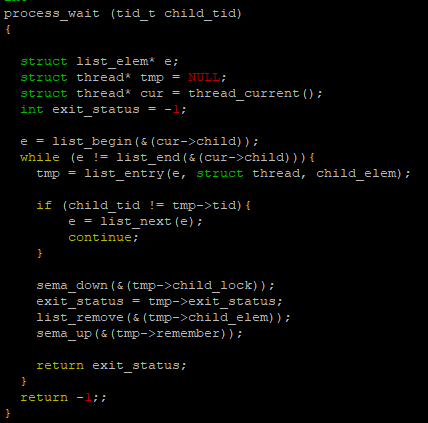
Test case 중에 unmapped virtual memory 를 참조하려고 하는 test case가 있다. 그것을 방지하기 위해 exception.c 에 not\_present 라는 미리 설정되어있는 변수를 이용해 이 것이 참이면 exit 해버린다.

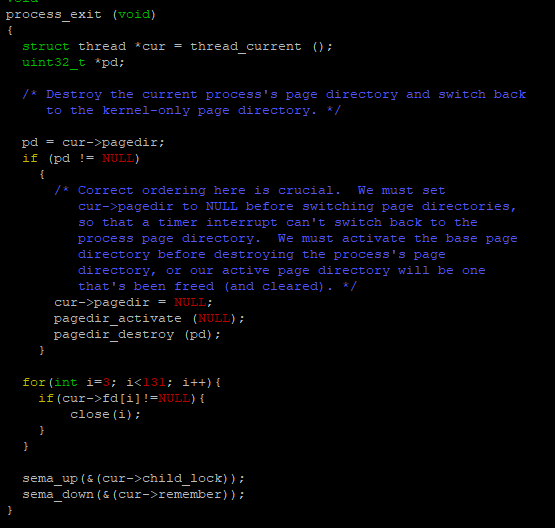
**프로세스 간 동기화**

여기서부터는 process.c 에 관한 수정이다. 굉장히 많은 부분이 변화하였다.

우선 프로젝트1에서 했던 process\_wait 과 process\_exit 을 semaphore 를 활용해서 수정을 하였다.

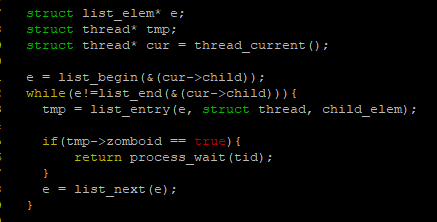
Process\_wait() / Process\_exit()

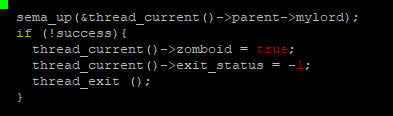




Semaphore 를 사용하는 것으로 바꾸었을 뿐 흐름은 같다. 우선 부모의 process 에서 sema\_down 으로 child 가 실행이 끝나기 직전까지 기다린다. Child process 에서는 할 거 하고 끝날 때 열었던 파일들을 우선 다 닫아주고 sema\_up을 하여 부모 프로세스가 진행하도록 한다. 그리고 나서 바로 sema\_down 을 한다! 왜나면 부모 process 에서 child process 의 exit status를 받을 때 까지 죽으면 안되기 때문이다. 그래서 부모 process 에서 child process의 exit\_status를 받고 그 다음 sema\_up 을 해주어 child process가 죽게 해준다.

여기서 동기화와 관련하여 생각하여야 할 것이 2가지가 있다. Process\_execute 와 start\_process 의 코드를 보자.





위가 process\_execute 이고 아래가 start\_process 이다. 우선 parent process 가 child process 를 만들 때 child가 load 되기도 전에 끝나는 불상사가 생긴다. 따라서 child process 가 제대로 load 가 되도록 sema\_down을 해주고 child process 는 load 가 끝나면 sema\_up 을 해준다. 그러나 한 가지 문제가 더 있다. Load 가 실패하는 경우이다. 이러면 원래 코드 상에서는 바로 thread\_exit 이 수행되고 그대로 process\_exit 이 수행되어 위 process exit 에서 file 을 다 닫고 child\_lock semaphore 의 count를 1 증가 시키고 remember semaphore 에서 sema\_down 되서 영원히 기다린다. 이런 아이들을 반드시 구제해주어야 한다. 따라서 만약 load 실패가 된 경우 zomboid flag 를 세우고 exit\_status 에 -1을 넣고 thread\_exit 하고 구제될 때 까지 기다린다. 그럼 parent 의 process\_execute 에서 child list 를 확인하면서 만약 zomboid flag 가 올라간 경우 구제하려고 process\_wait 을 실행한다. Process\_wait 이 실행되면 child\_lock semaphore 는 이미 child 에서 1 증가시켰으므로 1 감소시키고 통과되고 아까 넣은 child의 exit\_status (-1) 을 잘 회수하고 remember semaphore를 sema\_up 시켜 드디어 child가 죽을 수 있도록 한다.

이런 과정을 통해 child 들을 하나도 빠짐없이 전부 죽임으로써 메모리 누수가 생기지 않도록 하고 동기화를 해준다!

* 1. **시험 및 평가 내용**

