**Pintos 프로젝트 2\_2. User Program Basic**

**(설계 프로젝트 수행 결과)**

**과목명 : [CSE4070-02] 임베디드시스템소프트웨어**

**담당교수 : 서강대학교 컴퓨터공학과 김 영 재**

**조원 : 28조 안진우**

**개발기간 : 11월6일 ~ 11월10일**

**최 종 보 고 서**

**프로젝트 제목: Pintos 프로젝트 2\_2. User Program Basic**

**제출일: 2017.11.10**

**참여조원: 28조 안진우**

**I. 개발 목표**

이번 프로젝트의 목표는 User program 이 pintos 내에서 파일 시스템 관련 시스템 콜을 문제없이 running되도록 하는 것이다. 이를 위해서는 우선 file system call에 관련된 함수를 구현하여 User program이 서비스 요청을 가능하도록 하고, file system 내에서의 critical section에 대한race condition을 방지하기 위한 synchronization을 적절하게 수행하며, 자식 process와 부모 process간의 적절한 처리가 필요하다. 이를 통해 핀토스를 안정화시켜 총 76개의 테스트케이스에 대해 모두 통과하는 것 또한 이번 프로젝트의 중요한 목표이다.

**II. 개발 범위 및 내용**

1. **개발 범위**

Userprogram이 file system에 access하여 file control하기 위한 부분을 개발해야 한다. 이를 위해 관련된 시스템콜 함수를 구현하고, file system 내에서의 동기화 이슈를 적절히 처리해야 한다. 또, 프로젝트2-1과 맞물려 여러 메모리 누수 등 예외처리 및 동기화문제를 control하여 핀토스의 user process가 여러 issue에서도 안정적으로 돌아가도록 보완해야 한다.

**나. 개발 내용**

1. file system관련 시스템콜 구현

User program이 system call을 통해 파일 시스템에서의 파일 처리를 할 수 있도록 개발하였다. filesystem과 관련된 create, open, close, filesize, read, write, seek, tell, remove 총 9개의 시스템콜을 구현하였다. 구현 과정에서 file descriptor table을 관리하기 위하여 thread structure에 fd table을 추가하였다. 또, Denying Writes to Excutable을 위해 thread structure에 해당 process가 저장된 file을 추가하였다.

2. file system 관련 동기화 구현

file system내에서는 inode list등의 메타데이터를 공유하고, 한번에 여러 process가 동일한 파일에 write, delete 등을 시도 할 수 있기 때문에 이를 critical section으로 지정하여 순차적으로 공유 데이터에 접근할 수 있도록 하였다. 특히, pintos 메뉴얼에서는 file system전체를 critical section으로 보라고 나와 있지만, efficiency를 위해 sparse하게 lock을 잡기 위해 inode lock과 filesystem lock을 따로 구현하였다.

3. parent 와 child 사이의 동기화 구현 및 안정화

프로젝트 2\_1에서 구현한 parent 와 child간의 exit status전달을 위한 동기화 처리에서, 여러 예외 상황에 대비하여 lock의 위치를 최적화 시키고, process\_execute에 lock을 추가하여 child process가 load 실패하는 경우에 효과적으로 대처하도록 구현하였다.

4. 메모리 누수 방지

test case 중 multi-oom을 통과하기 위해서는 모든 경우에 대한 메모리 누수를 최소화시켜야 한다. 따라서 기존에 malloc으로 할당한 동적 메모리를 모두 free() 처리 하였으며, process가 open한 모든 file에 대해서는 리턴하기 전에 close처리를 하였다.

5. 76개의 test case에 대한 통과

지속적으로 test case를 체크하여 프로그램에 내제된 에러를 고치고, pintos 의 안정성을 향상 시켰다. 특히, multi-oom 테스트케이스는 아주 복잡한 recursion을 통해 parent와 child의 예외적 상황, 메모리 누수 등을 시험하기에, 해당 test case를 통과하기 위해 2박3일 동안 지속적으로 에러를 수정하고 동기화를 안정화시켰다.

**III. 추진 일정 및 개발 방법**

**가. 추진 일정**

|  |  |
| --- | --- |
| 11/6 ~ 11/6 | pintos 메뉴얼 분석 및 코드 이해 |
| 11/7 ~ 11/7 | 시스템 콜 구현 및 file system 관련 구현 |
| 10/8 ~ 11/9 | 76개의 test case에 대한 테스팅, 에러 분석 및 주석처리 |
| 11/9 ~ 11/9 | 최종 보고서 작성 |

**나. 개발 방법**

이번 프로젝트에서는 물론 file system call 함수 구현도 중요하지만, 핀토스 2\_1 프로젝트에서 구현한 내용과 통합하여, 여러 예외상황에 안정적으로 대처하는 내구성있는 user program을 완성하는 것이 가장 중요했다. 따라서 프로그램에 대한 각 테스트 케이스를 체크하여 예외 원인을 분석하고, 문제를 해결하여 재구현하여 프로그램을 안정화시키는 부분에 가장 많은 시간을 소모하였다. 지속적으로 에러를 체크하고 수정한 결과 76개의 테스트 케이스 전부에 통과하는 pintOS를 구현할 수 있었다.

1-1. file system 관련 system call 구현

파일 시스템 관련 시스템콜을 구현하여 user가 file 처리를 위한 요청을 가능하도록 하였다. create, remove, open, close, filesize, read, write, seek, tell 등 총 9개의 시스템콜을 구현하였다.

1-2. file system 내 동기화 처리

파일 시스템 내에서는 공유 데이터의 접근을 제어하기 위해 open시에 file system lock, write시에 inode lock을 디자인하였다. 또, excutable file에 대해서는write 또는 delete명령에 대해 deny\_write operation을 이용하여 해당 명령을 거부하도록 하였다.

2-1. memory 관리

메모리 누출 방지를 위해 동적으로 생성된 메모리는 반드시 해제하도록 하였다. 또, process가 종료되기 전에 open 된 file 을 전부 close시키도록 하여 file structure을 메모리 해제시켰다.

2-2. parent , child 간의 동기화

multi-oom 테스트케이스를 통과하는 과정에서 프로젝트 2\_1에서 작성한 parent, child간의 동기화 코드를 상당 부분 수정하였다. 이를 통해 child 의 load실패, 좀비 프로세스 등의 예외상황에 대해 적절한 처리가 가능하도록 구현하여 프로그램을 안정화시켰다.

**다. 연구원 역할 분담**

1. 핀토스 메뉴얼 분석 및 코드 이해 : 안진우 100%

2. 프로그램 생성 및 구현 : 안진우 100%

3. 테스팅, 에러 처리 및 주석, 보고서 : 안진우 100%

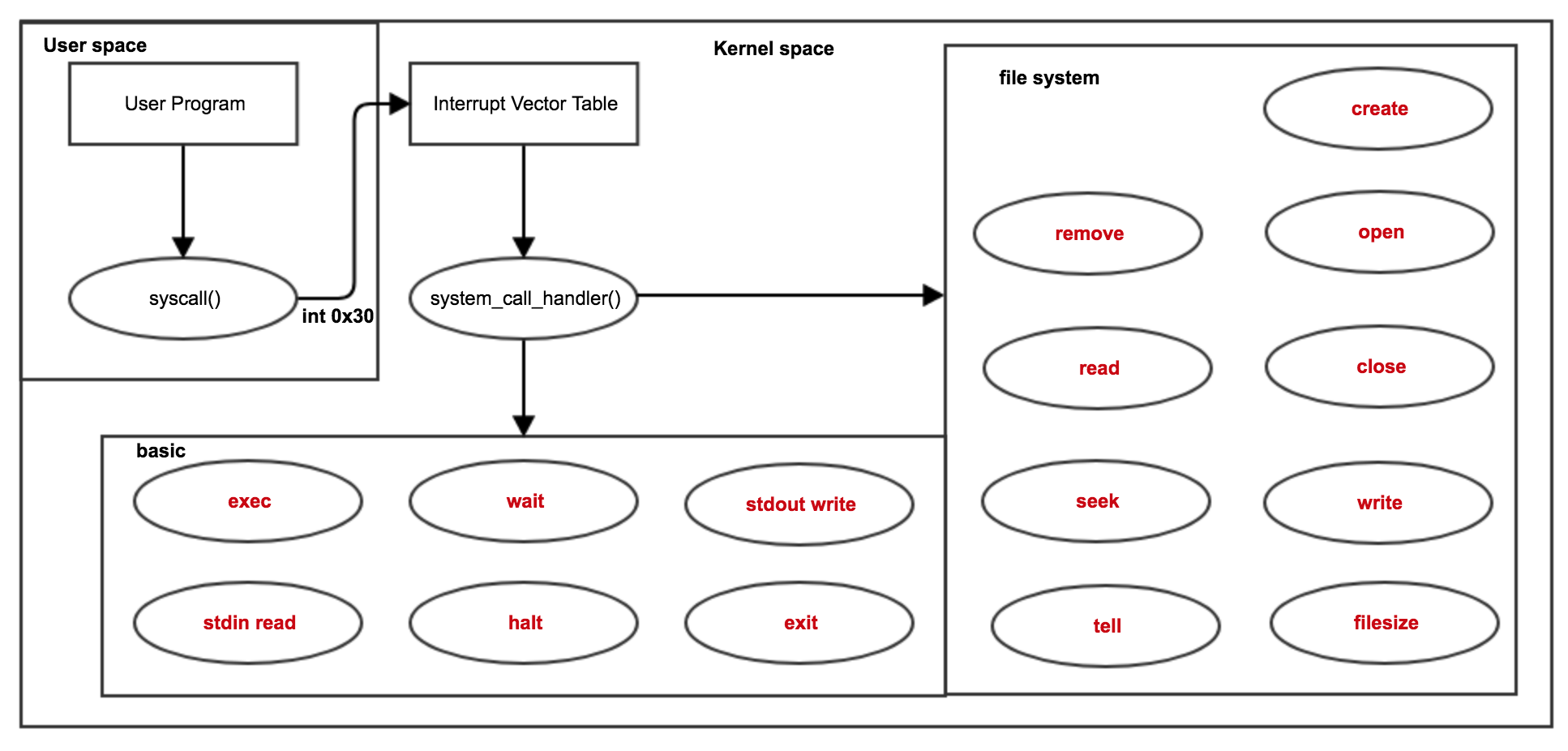
**IV. 연구 결과**

1. 합성 내용:

1.1. 프로그램 실행 구성도

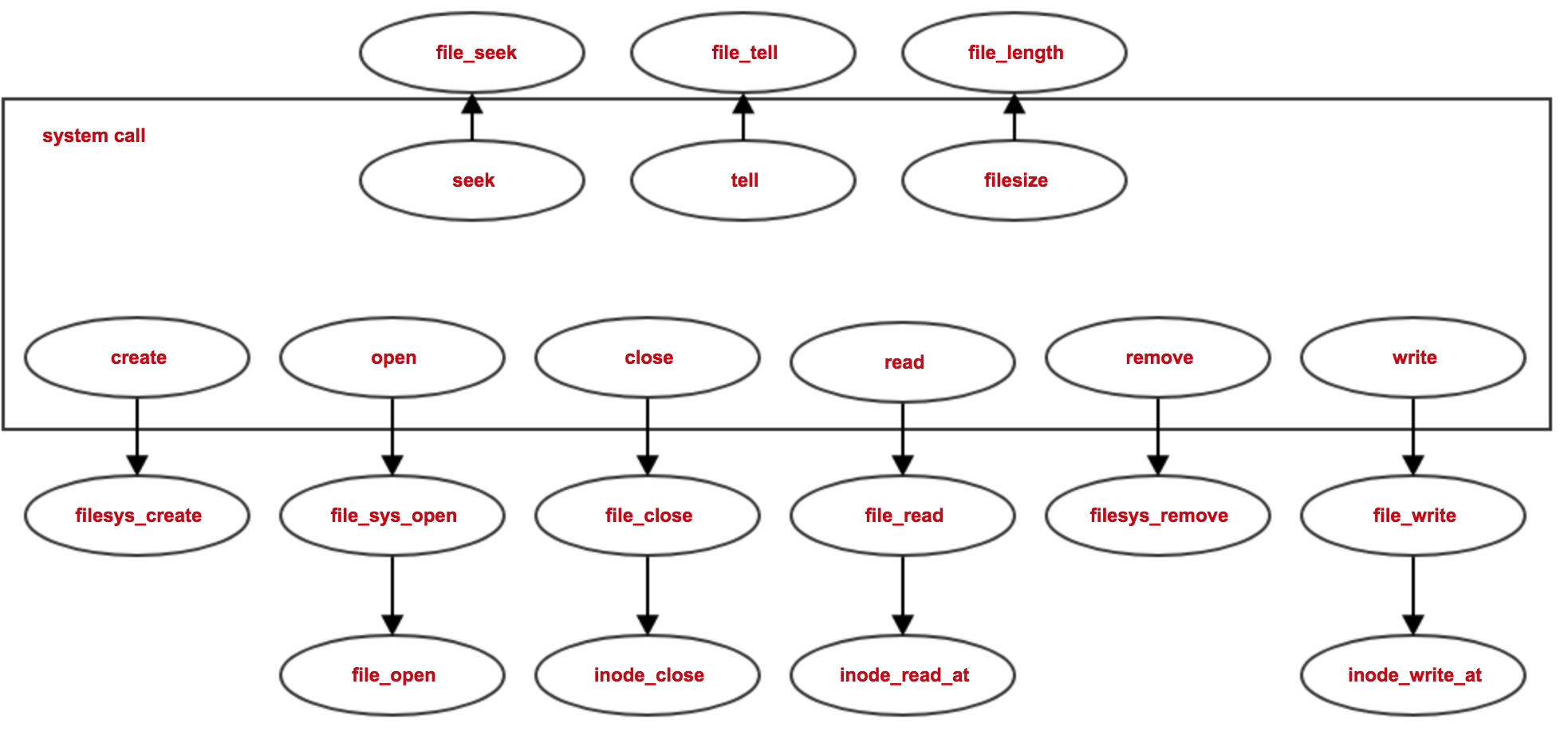
1.1 system call 함수 구상도

User program이 system call을 호출하면 다음과 같은 flow로 서비스가 수행된다. 이번에는 file system과 관련된 9개의 system call을 구현하였다.



2.1 Base file system structure

사용자가 호출한 filesystem 관련 system call에서는 다음과 같은 flow 로 file system의 module이 수행된다.



2. 제작 내용: 개발 결과

1. Userprogram filesystem

Userprogram에서 filesystem에서 제공하는 모듈을 사용하기 위해서는 우선 system call을 정의하고, user가 사용하는 도중에 문제가 발생하지 않도록 동기화를 취해주어야 한다.

1-1. system call(file system)

|  |
| --- |
| 이번 프로젝트에서는 User program이 file 처리를 위한 서비스를 요청할 수 있도록 file system 관련된 system call을 구현하였다. 유저는 이 system call을 통해서 실제 storage에 저장된 file들을 create, read, write 등의 contol할 수 있다.  bool create(const char \*file, unsigned initial\_size)  : size만큼의 byte를 할당하여 새로운 파일을 생성한다. 성공시 true, 실패시 false를 리턴한다.  int open(const char \*file)  :file을 open 하고 file descriptor을 리턴한다. user는 fild descriptor을 통해 해당 file에 대한 write/ read등 각종 연산을 수행할 수 있다. fild descriptor은 process내부에서 독립적으로 관리되며, file descriptor table을 통해 file과 맵핑된다.  void close(int fd)  :open된 file을 close한다. 이를 통해 할당된 file descriptor이 사라지고, 맵핑된 file은 메모리에서 해제된다.  int read(int fd, void \*buffer, unsigned size)  :file descriptor을 통해 파일에 접근하여 파일로부터 size만큼의 내용을 buffer에 read한다. read에 성공한 바이트수를 리턴한다. fd가 0이면 std\_in으로 input을 받는다.  int write(int fd, const void \*buffer, unsigned size)  : file descriptor을 통해 파일에 접근하여 파일에 size만큼의 내용을 buffer로부터 write한다. write에 성공한 바이트수를 리턴한다. fd가 1이면 std\_out으로 buffer 내용을 출력한다.  int filesize(int fd)  : file의 size를 리턴한다.  bool remove(const char \*file)  : file을 delete한다. 만일 file이 open되어 있으면 close될때까지 삭제되지 않아야 한다.  void seek(int fd, unsigned position)  : 현재 파일의 position을 재설정한다.  unsigned tell(int fd)  : 현재 파일의 position을 리턴한다. |

1-2. file system 내에서의 동기화

|  |
| --- |
| file system에서는 inode등의 shared data에 동시에 접근할 경우 의도에 맞지 않는 수행이 발생할 수 있기에, 이를 critical section으로 분리하여 동기화하여야 한다. pintos menual에 따르면 file system을 접근할 때에는 모든 영역을 critical section으로 판단하여 coarse하게 lock을 걸라고 나와있다. 하지만, 이 방식은 efficiency 에서 많은 손해를 보기 때문에 performance를 위해 sparse하게 lock을 구현하였다.  우선 구현한 lock은 다음과 같다.  -static struct lock lock\_filesys  file open시에 여러 process에서 접근하여 동시에 open하는 것을 방지하기 위해 전역변수로 lock\_filesys를 할당하였다. open시에는 inode를 추가하는 등 shared data를 접근하기 때문에 이를 critical section으로 판단하여 동시에 수행되지 못하도록 막아야 한다.  -struct semaphore sema  file write시에는 동일한 파일에 둘 이상의 프로세서가 동시에 접근하여 write를 수행할 시에 의도와 다른 write가 수행될 위험이 있다. 따라서 inode structure 내에 semaphore를 할당하여 같은 inode(파일)에 둘 이상의 프로세스가 동시에 write하지 못하도록 막았다. |

1-3. Denying write to Excutables

|  |
| --- |
| pintos에서는 실행중인 프로그램의 excutable file에 대해서는 write, delete를 막아야 한다. 따라서 프로세스의 life cycle, 즉 프로세스 생성부터 exit까지는 프로세스가 직접 file 에 대한 접근을 차단해야 한다.  프로세스에서 이를 직접 제어하기 위하여 thread strucuture 내에 다음과 같은 변수를 선언하였다.  struct file\* file\_cur  file\_cur은 프로세스의 excutable file을 저장하는 변수이다. 프로세스는 load될때 이 file을 open하여 read하는데, 이 때 다음과 같은 함수를 취해주면 다른 프로세스에서의 write / delete가 금지된다.  fild\_deny\_write()  그런데, 프로세스가 종료되면 다른 프로세스는 해당 파일에 대해 write/ delete할 권한을 돌려받을 수 있어야 한다. 따라서 thread\_exit 함수에서는 다음과 같은 함수를 취해준다.  fild\_allow\_write()  이를 통해 process는 life cycle동안만 다른 프로세스의 write / delete를 제어할 수 있다. |

2. 이번 과제의 핵심은 76개의 테스트 케이스에 통과하여 안정성 있는 pintos를 만드는 것이었다. 따라서 안정성 있는 프로그램을 만드는 것에 이번 기간 전력을 기울였다. 특히 이번 구현에 있어서 critical한 테스트케이스는 다음과 같다.

- rox-simple, rox-child, rox-multichild

Denying write to Excutable에 대한 테스트 케이스이다.

- syn-read, syn-remove, syn-write

file system 내의 synchronization에 대한 테스트 케이스이다.

- multi-oom

multi-oom은 이번 과제에서 가장 난제인 테스트케이스이다. child를 recursive하게 호출하는데, 30개 이상의 child process가 문제없이 생성되고 수행되어야 success된다. 이 테스트 케이스에서는 parent와 child간의 synchronization 문제, process의 비정상적인 종료문제, 메모리 누수 등 많은 예외상황을 하나 하나 고려해야 한다. 따라서 이번 프로젝트의 절반이상의 시간을 사용했을 정도로 많은 고민을 하였으며, 이를 통해 그동안 pintos구현상의 문제점을 다량 발견하고, 이를 수정하여 안정적인 pintos를 구현할 수 있었다.

2-1 메모리 누수 문제

|  |
| --- |
| multi-oom에서 가장 민감하게 detecting하는 것은 메모리 누수이다. child process가 recursive하게 생성되기 때문에, 메모리 관리를 제대로 하지 않으면 에러가 발생한다.  우선적으로 malloc을 통해 할당한 동적 메모리에 대한 free를 시켰다. 기존 2-1 프로젝트 때에는 malloc에 대해 free를 시키지 않았는데, 함수가 리턴되면서 동적 메모리의 주소를 잃어버리면서 영영 free할수 없게 되는 구조였다. 특히, 다음과 같은 함수구현은 커다란 문제를 가지고 있었다.  스크린샷%202017-11-10%20오후%208.50.42.png |
| 프로젝트 2-1에서 구현한 다음과 같은 함수는 함수 내에서 malloc으로 argv를 구현하고 이를 직접 return한다. 따라서 이 함수를 호출한 모든 함수에서는 argv를 직접 free시켜야 하는데, malloc하지도 않은 호출 함수에서 free를 시키는 것은 이는 모듈화 , transparency측면에서 매우 적절치 못한 구현이었다. 따라서 해당 함수를 다음과 같이 수정하여 호출한 함수에서 malloc, free를 함께 수행하도록 구현하여 modularity, transparency를 회복할 수 있었다.  스크린샷%202017-11-10%20오후%208.54.16.png  또, 다음 함수의 경우에는 아예 free가 불가능한 구조이다.  스크린샷%202017-11-10%20오후%208.52.13.png  이 함수역시 프로젝트 2-1에서 구현하였는데, 함수 내에서 tp\_ptr free를 시키면 \*file\_name이 손상되어 사용하지 못하게 된다. 따라서 다음과 같이 수정하여 malloc과 free를 호출부에서 수행하도록 하였다.  스크린샷%202017-11-10%20오후%208.55.26.png  두번째로 thread strucuture에 저장된 file descriptor table은 file open갯수만큼 file structure을 갖고 있다. 이를 close시키지 않고 리턴시키면 메모리에 영구히 남아있는 문제점이 생기기에 thread가 종료될때 이를 반드시 해제시켜야 한다. 이를 위해 thread\_exit함수에서 다음과 같이 열린 file을 모두 close시키도록 하였다.  스크린샷%202017-11-10%20오후%208.58.35.png |

2-2 parent, child간의 동기화

|  |
| --- |
| multi-oom은 가능한 많은 child를 recursive하게 호출한다. 뿐만 아니라 각 child는 잘못된 주소 접근 등 비정상적인 행동을 수행한다. 따라서 parent 와 child의 동기화를 정밀하게 다루지 않으면 결코 multi oom은 성공적으로 수행되지 않는다.  특히 가장 큰 문제는 child가 load fail을 할 경우의 수이다. 다른 모든 테스트케이스로는 이 경우를 잡지 못하는데, pintos menual에 따르면 자식 프로세스의 load가 fail하면 이를 process\_execute()함수에서 잡아서 -1로 리턴해야 한다. 문제는 process\_execute랑 load가 수행되는 start\_process중 누가 먼저 수행될지 확신할 수 없다는 것이다. 따라서 lock을 통해 process\_execute는 start\_process에서 load가 완료될 때 까지 wait해야 한다. 이를 위해 thread strucuture에 새로운 semaphore인 sema\_exec을 할당하였다. 각 동기화는 다음과 같이 구현하였다.  process\_execute  스크린샷%202017-11-10%20오후%209.08.08.png  start\_process  스크린샷%202017-11-10%20오후%209.08.37.png  우선 process\_execute에서는 자식 프로세스가 load될때까지 wait한다. (sema의 초기값이 0이다.) 이동안 자식 프로세스는 load를 수행하며, 실패할 경우 exit\_soon에 true를 할당하고 process\_execute을 깨운다. 깨어난 process\_execute은 exit\_soon을 확인하여 load실패를 확인하고 -1을 리턴한다. |

3. 시험 및 평가 내용:

-평가 방법

1. 76개의 테스트 케이스 수행하기 (make clean 상태일때)

1. $cd pintos/src/userprog
2. $make
3. $cd build
4. $make check
5. 76개의 테스트케이스가 통과하였는지 확인한다.

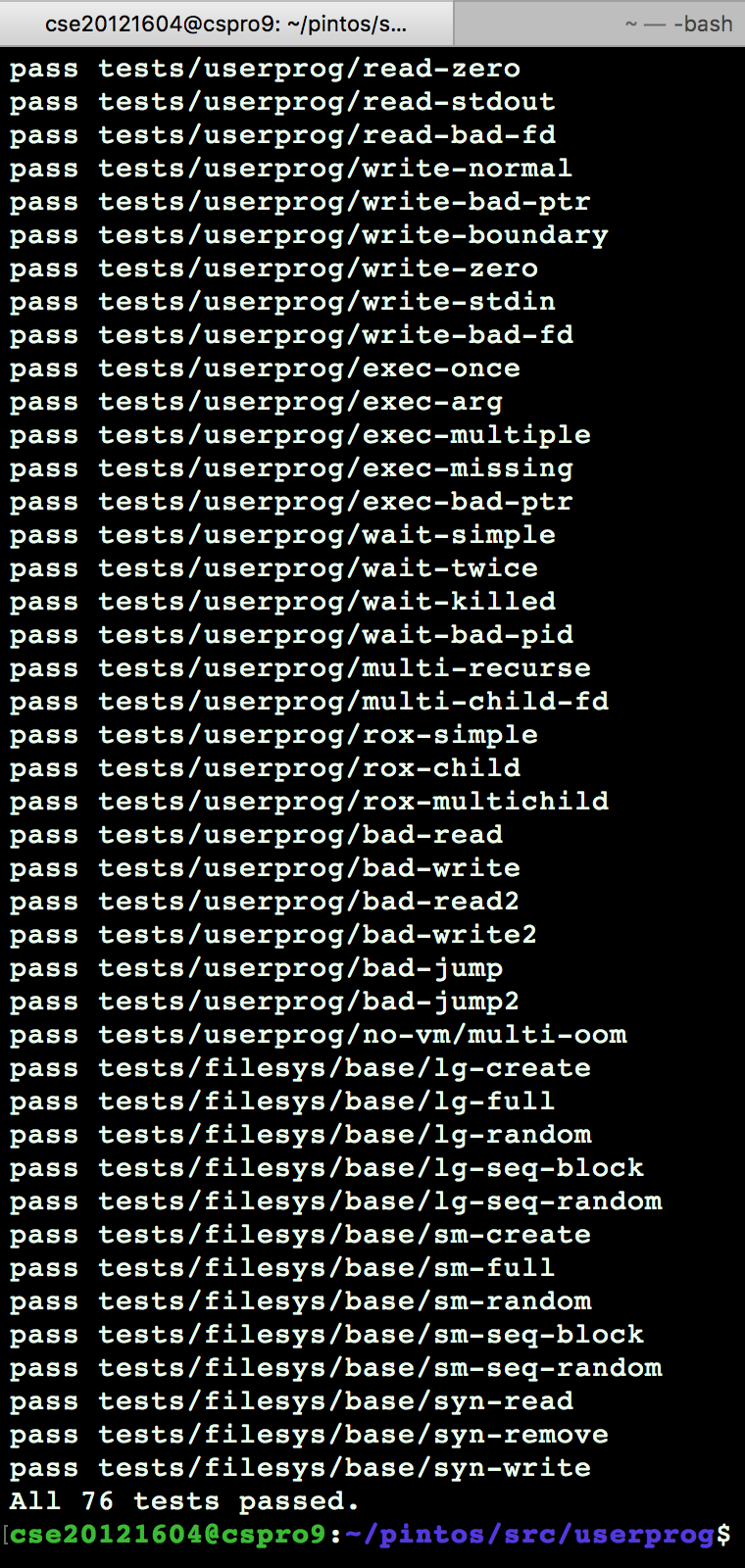
-보건 및 안정, 생산성과 내구성

이번 프로젝트는 Userprogram을 마무리하는 프로젝트로써 특히 안정성이 중요하였다. 테스트 케이스를 해결하는 과정에서 다양한 취약점 및 에러가 드러났으며, 많은 고민을 통해 차근 차근 이러한 문제들을 해결해 나갔다.

이번 프로젝트에서 보건성, 안정성으로 핵심을 둔 부분은 synchronization 문제였다. file system과 parent,child간에는 critical section 이 존재하였기에 lock을 걸어야만 했다. 특히 parent, child간 synchronization에서는 3개의 lock을 추가하여 다양한 조건에 대해서 효과적으로 동작하도록 만들었다. 우선 wait에서는 child가 terminate될때까지 기다리는 sema\_wait , child가 죽기 전 exit status를 parent에게 넘겨주기 위해 기다리는 sema\_exit, 마지막으로 execute에서 child의 load실패를 확인하기 위하여 load하기를 기다리는 sema\_exec을 각 thread structure에 할당하여 수행하였다. 특히 multi-oom 테스트케이스에서는 해당 락의 취약점을 다양한 방법으로 공격했는데, 이로 인해 dead lock 등의 문제가 발생하기도 하였다. 따라서 lock의 위치를 미세하게 조절하여, 가능한 모든 flow에 대해 대처할 수 있도록 구현하였다.

두번째로 내구성, 생산성을 보장하기 위해서 가장 중점에 둔 것은 메모리 누수이다. malloc등 동적 메모리 할당에 대해 반드시 free를 수행하고, thread가 종료될때 open된 모든 file을 종료시키고 file descriptor을 메모리 해제하여 주소를 잃어버린 메모리가 아예 생기지 않도록 만전을 가했다.

이번 프로젝트에서는 대부분의 시간을 이와 같이 프로그램의 안정성을 위해 투자하였다. 그 결과 multi oom을 포함한 테스트케이스 76개를 모두 통과하는 안정적이고 내구성있는 pintos를 구현하는 데 성공하였다.



**V. 기타**

1**. 연구 조원 기여도:**

안진우: 100%

2. 기타 본 설계 프로젝트를 수행하면서 느낀 점을 요약하여 기술하라. 내용은 어떤 것이든 상관이 없으며, 본 프로젝트에 대한 문제점 제시 및 제안을 포함하여 자유롭게 기술할 것.

이번 프로젝트는 2-1 프로젝트와 달리, 새로운 구현보다는 구현된 코드의 완성도에 집중하는 것이 포인트였다. 스스로 짠 코드의 취약점을 발견하고, 이를 보완하는 과정에서 스스로에게 많은 반성을 할 수 있었고, 덕분에 많은 성장이 되었다고 생각한다. 특히 마지막 multi-oom 을 해결하기 위해 심혈의 노력을 기울였다. 아마 서른번이상의 전략수립과 코드수정이 있었을 것이라 생각되는데, 이것을 고민하며 머리를 쥐고 밤을 새면서 많은 걸 배울 수 있었다. 우선 연구자에게 있어서 같은 자리에 앉아 ‘노력’만 기울이는 것은 효율을 오히려 떨어뜨림을 느꼈다. 디버깅도 제대로 안되는 multi-oom을 해결하기 위해선 순전히 추론력에 의존해야 하는데, 오히려 산책을 하거나 버스를 타면서 새로운 아이디어가 생각나고, 머리가 정리되어 문제가 차근차근 해결되는 것을 느꼈다. 밤을 새는 것 역시, 사람마다 다르겠으나, 머리가 돌아가지 않는 새벽에 졸음을 참아봐야 창의적인 생각이 떠오르지 않기에 거의 진전이 되지 않았다. 이에 반해, 가장 뇌가 활발한 낮 시간에 집중적으로 카페에서 코드를 들여다보면서 많은 문제점을 발견할 수 있었고, 이를 차근 차근 고칠 수 있었다. 이렇듯, 무작정 노력한다고 다 되는 것이 아니라, 스스로를 컨트롤 하면서 효율적인 순간에 효율적인 방식으로 연구하는 것이 훨씬 좋은 결과를 낳는 다는 것을 알게 되었다.

마지막으로 multi-oom의 메모리누수문제를 해결하면서, 그동안 매우 안좋은 코딩습관을 가지고 있었다는 것을 깨닫게 되었다. 이론적으로는 malloc에 free해야함을 알고 있지만, 이로 인해 실질적 문제가 발생하지 않았기에 free에 대한 생각없이 코드를 짠 결과, 절대로 해제가 불가능한 malloc을 나도 모르게 선언하고 있었던 것이다. 이렇듯, 프로그램을 안정화 시키면서 스스로 잘못된 습관을 바로잡을 수 있다는 점 역시 이번 프로젝트의 큰 수확이다.