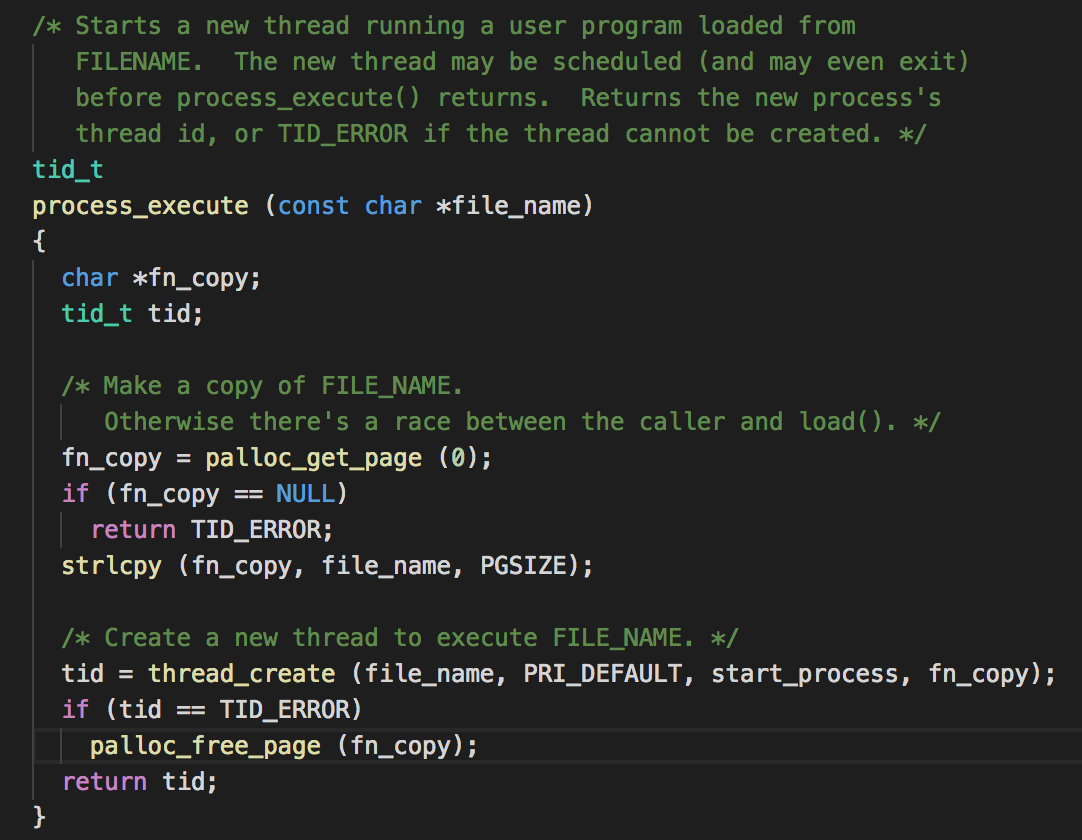
**Project 2 Design Report**

성해빈, 송창근

**1. 문제 해결**

1. User Process
   1. Argument Passing



process.c 에 있는 process\_execute 함수는 user program의 thread creation을 담당한다. 이때 argument를 따로 분리하지 않고 단순히 fn\_copy으로만 명령을 통째로 따내서 thread\_create의 aux, 즉 실행된 function의 인자로 전달한다. 이렇게 되면 실행할 각 user program function마다 argument parsing을 진행해야 하므로, 상당히 번거로워질 것이다. 미리 process\_execute function에서 char\*\* 타입을 만들어, argv를 따로 담는 2차원 배열을 만들자. “ls –l foo”가 커맨드라고 할 때, Fn\_copy[0] = “ls”, fn\_copy[1] = “-l”, fn\_copy[2] = “foo”가 되는 식으로 spacebar를 이용한 parsing을 수행한다. String parsing에는 string.h 에 있는 strtok\_r 을 이용하도록 한다. parsing 한 내용물(  
2차원 배열)은 start\_process에다 인자로 넣어준다.(file\_name) 또 argument 에 맞춰 stack을 조정하지 않으면 page fault가 난다. intr\_frame에 있는 esp로 스택 상황을 조정해주자. 이 스택 조정 부분은 process.c 에 있는 start\_process에다 구현한다.

* 1. Process Termination Messages

Thread termination이 일어날 때 메세지를 출력하는데, thread를 종료할 때 호출하는 thread\_exit 함수에 삽입하는 것이 적절해 보인다. 여기서 출력하는 thread name은 process\_execute()에 전달된 full string에서 argument만 뺀 것이다. 한마디로 fn\_copy[0]에 해당하는 부분이다. 여기서 user process 에 해당하는 것만 메세지를 출력하고, halt system call이 발동된 경우라면 메시지를 출력하지 않는 것에 조심하자. Process load failure에도 메시지 출력을 하도록 한다.

1. System calls

System call 은 userprog디렉토리에 있는 syscall.c와 syscall.h에 있는 syscall\_handler를 수정하면 된다. Syscall은 각자에 해당하는 숫자가 있는데, syscall-nr.h에 번호들이 enum으로 저장돼있고, lib/user/syscall.c에 syscall의 인자 개수마다 prototype이 만들어져 있고, 우리는 각 함수를 만들어주면 되는 식이다. Stack push와 ret으로 prototype이 정의되어있다. interrupt frame을 해석해 무슨 syscall을 부른 상황인지 파악해내고, 각자에 맞는 행동을 해주는 것이 syscall handler의 임무다. system call은 user memory access가 필요하기 때문에 dereference 안전장치가 필요한데, 먼저 포인터 위치가 valid한지 검사하고 dereference를 수행하는 방법을 채택하겠다.

* 1. User process manipulation

System call 중 user process 와 관련된 system call이다. 우리가 구현해야하는 Halt, exit, exec, wait 등이 해당된다.

* 1. File manipulation

System call 중 file 과 관련된 system call이다. 우리가 구현해야하는 File create, file remove, file open, file size, read, write, seek, tell, close등이 포함된다. Filesys directory와 filesys.h, file.h에 있는 함수들을 수정 없이 그대로 사용하라고 매뉴얼에 적혀있다.

1. Denying writes to executables
   1. 필요한 이유

어떤 thread가 현재 특정 파일을 이용하고 있는데 다른 thread가 파일을 수정하게 내버려둔다면 큰 문제가 생길 것이다. 이를 막기 위해 thread가 파일을 사용할 때 다른 thread가 수정하는 것을 방지해야하며, 이는 semaphore이나 lock을 사용할 필요가 있는 전형적인 상황이다. file\_deny\_write() 와 file\_allow\_write()를 활용하도록 한다.

**2. 시스템 분석**

1. Process execution procedure

user program을 실행할 때, process.c에 있는 process\_execute()로 새로운 쓰레드를 생성한다. 여기서 user stack은 가장 위(high)에 argv string이 역순으로 쌓여야 하고, argv string 주소가 다시 역순으로 쌓이고, argc가 쌓이고 return address 가 마지막으로 쌓인다. 이 형식을 맞추고 esp를 return address쪽에 맞춘 후, user process를 실행해야 제대로 실행된다.

1. System call procedure (for each system call)

system call는 internal interrupt의 일종으로, interrupt frame을 만들고 인자로 전달해 syscall\_handler 가 syscall을 실질적으로 실행한다.

* halt : shutdown\_power\_off() 를 부른다. PintOS를 꺼버린다.
* exit : user program을 종료한다.
* exec : executable (ELF)를 실행한다.
* wait : 특정 pid의 프로세스가 exit할때까지 기다리고 exit status를 return한다.
* create : file을 생성한다.
* remove : file을 삭제한다.
* open : file open을 하고 file descriptor을 return 한다.
* filesize : 인자로 받은 file descriptor의 file size를 가늠한다.
* read : file -> buffer로 값을 복사한다.
* write : buffer -> file로 값을 복사한다.
* seek : file 의 cursor을 원하는 position으로 변경한다.
* tell : 현재 cursor의 위치에 있는 값을 리턴한다.
* close : file을 닫는다.

1. File system

PintOS는 file system을 구현하는 것이 아니므로 기본적으로 filesys에 필요한 file system이 구현되어있다. 이번 프로젝트에서는 그 폴더의 함수를 이용하기만 하면 되며, 수정을 요하지 않는다. 따라서 구현되어있지 않은 virtual disk를 만들어내는 기능이나 동기화 기능은 PintOS에서 구현해야 한다. 프로젝트 4에서 파일 시스템을 개선하기 전까지, 주어진 파일 시스템은 다음과 같은 제약들을 가진다.

* 내부 동기화가 없다. 직접 동기화를 해주어야 한다.
* 파일 크기가 생성시에 고정된다. 루트 디렉토리도 파일로 표시되기에, 만들 수 있는 파일의 개수가 제한된다.
* 파일 데이터가 저장될 시 단일 범위에서 할당된다. 연속된 섹터에서만 저장된다.
* 하위 디렉토리가 없다
* 파일 이름이 14자로 제한된다.

또 다음과 같은 중요한 특징을 가진다.

* 파일이 제거될 때, 열려 있었다면 파일이 최종적으로 닫힐 때까지는 해당 블록이 할당 해제되지 않고, 다른 스레드에서도 액세스할 수 있다.