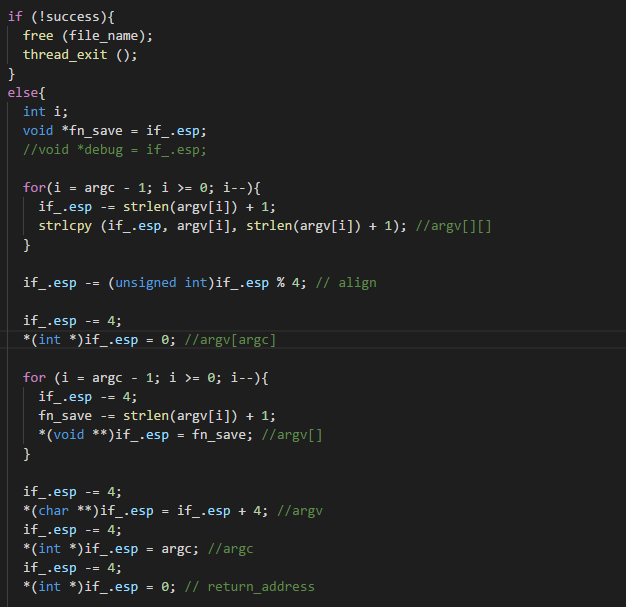
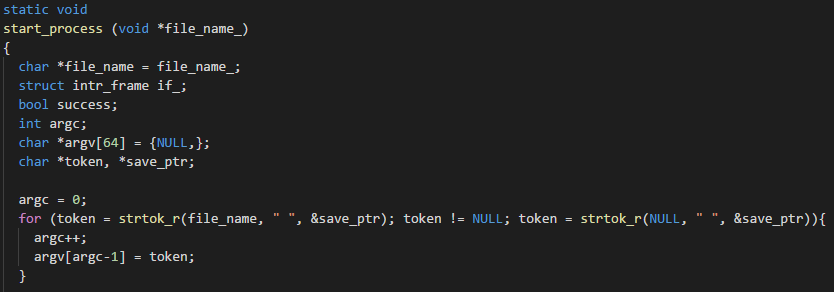
**Project 2 Final Report**

성해빈, 송창근

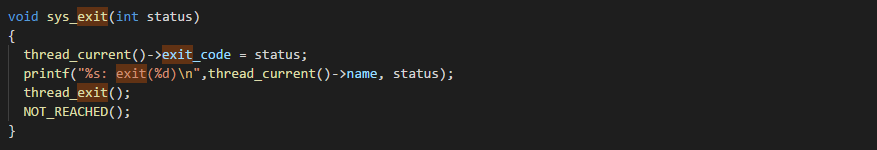
**1. 문제 해결**

1. User Process
   1. Argument Passing



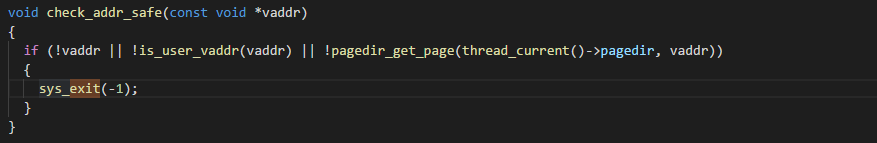
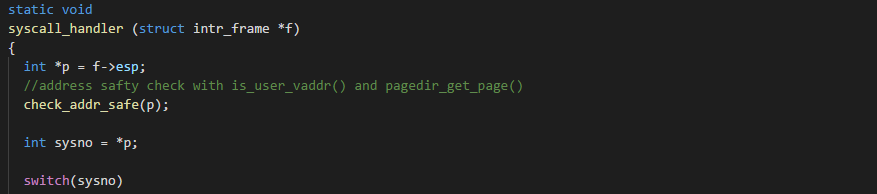
process.c 에 있는 process\_execute 함수는 user program의 thread creation을 담당한다. 이때 argument를 따로 분리하지 않고 단순히 fn\_copy으로만 명령을 통째로 따내서 thread\_create의 aux, 즉 실행된 function의 인자로 전달한다. 이렇게 되면 실행할 각 user program function마다 argument parsing을 진행해야 하므로, 상당히 번거로워질 것이다. 미리 start\_process에서 strtok\_r 을 이용하여 파싱을 한 뒤, 스택에 넣어주는 과정을 진행한다. esp를 빼가면서 매뉴얼에 나온 순서대로 받은 데이터를 저장했고, word\_align도 구현하였다.

* 1. Process Termination Messages



Thread termination이 일어날 때 메세지를 출력하는데, 이는 그 역할을 담당하는 함수인 sys\_exit 함수에 삽입하였다. 출력하는 내용은 쓰레드의 이름과 종료시 받은 인자이다. halt system call이 발동된 경우는 메시지를 출력하지 않는다. 이렇게 sys\_exit 함수만 유별나게 따로 분리한 이유는 system call 내부에서 sys\_exit을 부를 일이 많은데, system call 내부에서 exit system call 을 부르면 모양새가 이상해진다고 생각했기 때문에 많이 쓰는 기능인 저 부분을 따로 뺐다.

1. System calls

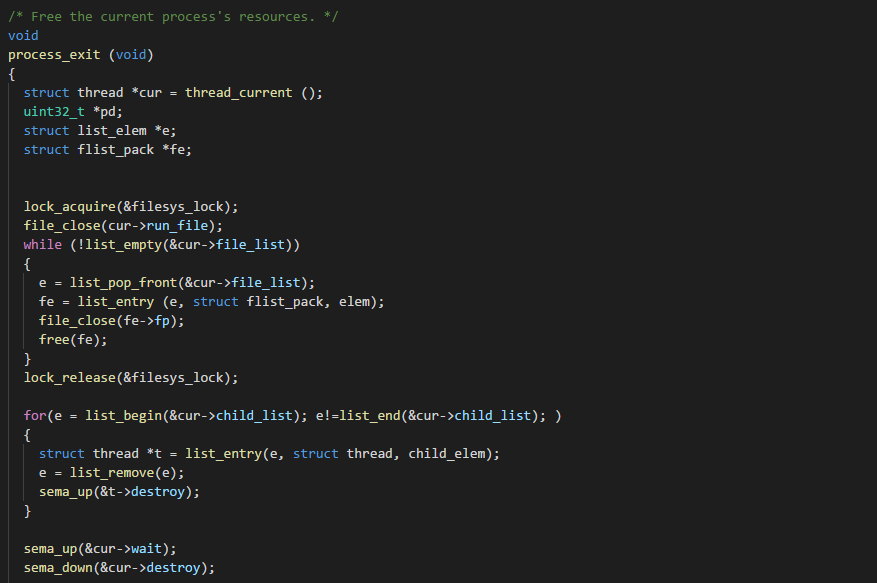


System call 은 userprog디렉토리에 있는 syscall.c와 syscall.h에 있는 syscall\_handler를 수정하여 구현하였다. Syscall은 각자에 해당하는 숫자가 있는데, syscall-nr.h에 번호들이 enum으로 저장돼있고, lib/user/syscall.c에 syscall의 인자 개수마다 prototype이 만들어져 있어 그 곳에 함수를 구현하였다. interrupt frame을 해석해 무슨 syscall을 부른 상황인지 파악해내고, 거기 필요한 인자를 받아와야 하는데, 이 과정에서 안전한 메모리 참조를 위해 포인터 위치가 valid한지 검사한다. valid를 검사하는 것은 is\_user\_vaddr와 pagedir\_get\_page를 통해 kernel 쪽을 침범했는지와 할당된 페이지 안에 있는지를 검사한다.

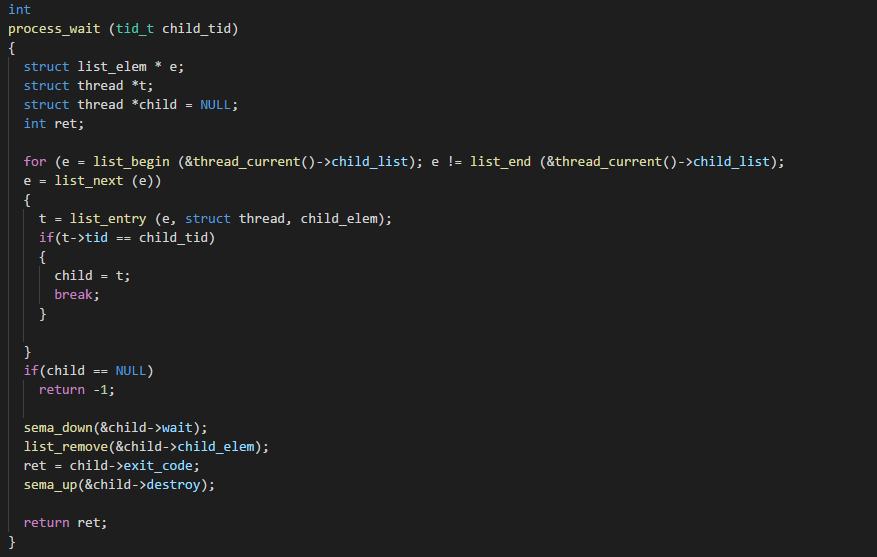
* 1. User process manipulation

System call 중 user process 와 관련된 system call이다. halt, exit, exec, wait 등이 해당된다. halt는 상당히 간단하여 OS를 종료하는 함수를 호출하기만 하면 되지만, 다른 것들은 상당히 복잡하다.

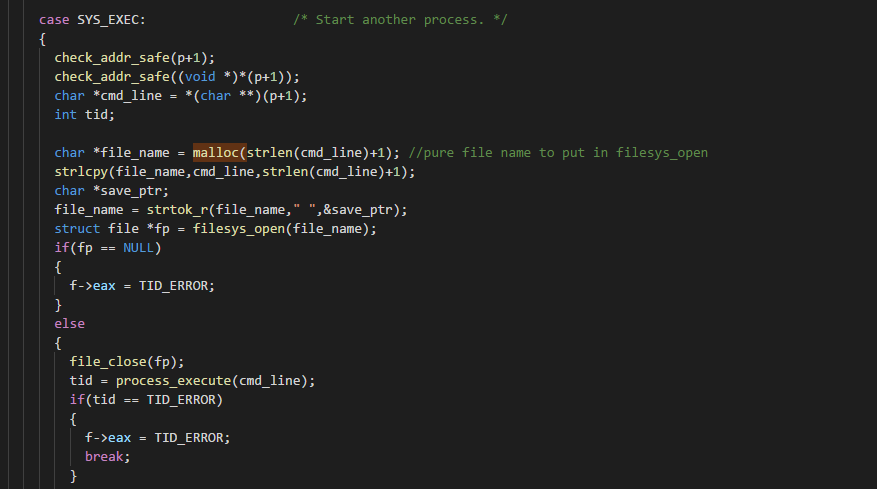
exit는 sys\_exit을 거쳐 thread\_exit, process\_exit을 호출한다. 주로 작성한 부분은 process\_exit 부분이다.



먼저 해당 프로세스가 점유하고 있는 파일을 전부 종료시키고 할당 해제한다. 그 뒤에 자식 쓰레드들을 처리해준다. wait와의 sync를 맞추기 위해 semaphore를 사용한다. 자식 쓰레드들은 전부 exit 단계에서 부모가 destroy semaphore를 올려주기까지 기다리고, 부모가 exit code를 받아내면 destroy semaphore를 올려 페이지를 파괴하고 dying thread로 만든다. 또 어떤 쓰레드가 exit할 때 만약 자식이 있다면, 전부 destroy sema up을 해준다.(부모가 wait을 하지 않으면 exit code를 받지 않아도 되기 때문에 바로 파괴해도 문제없다)



wait에서는 목표 tid를 가지는 child를 찾아 그 프로세스의 처리가 끝날 때까지 대기한다. exit\_code를 받아 오고 page destroy를 명령하는 것이 포인트다.

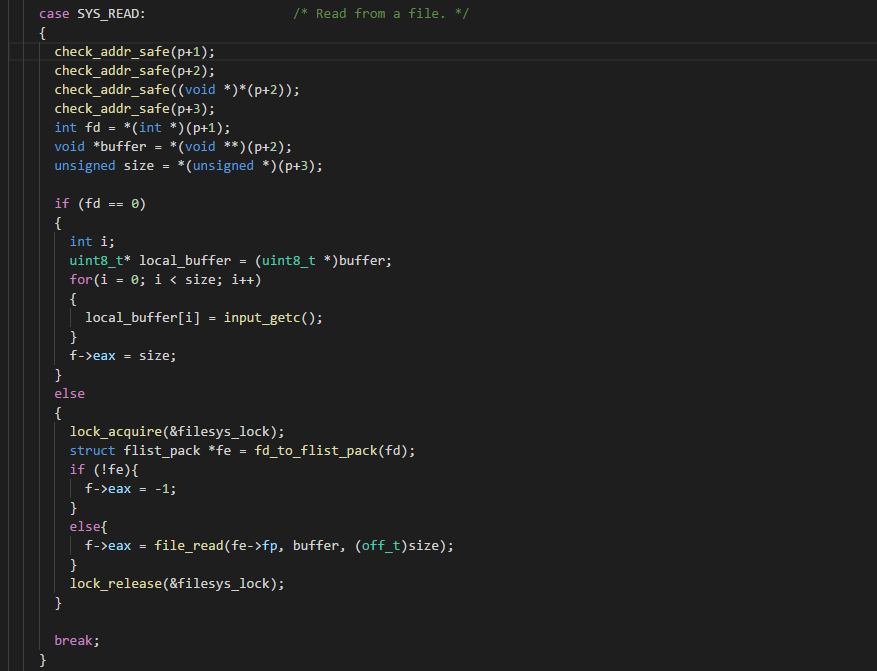
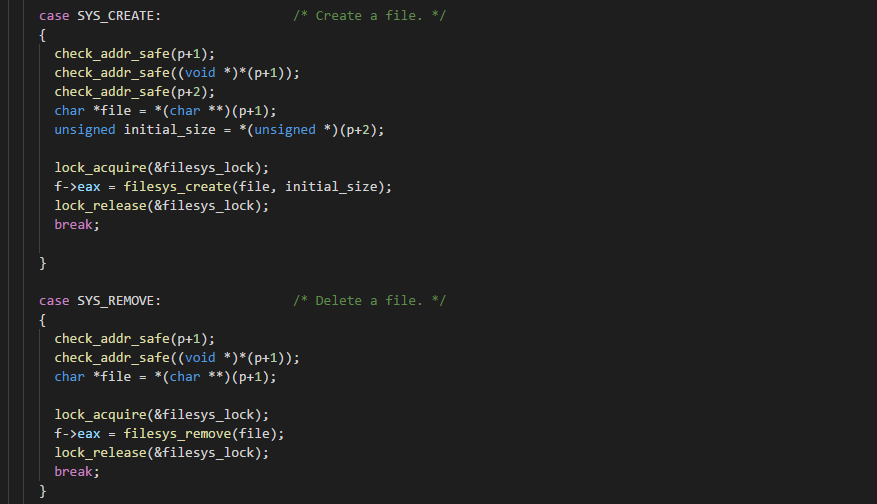
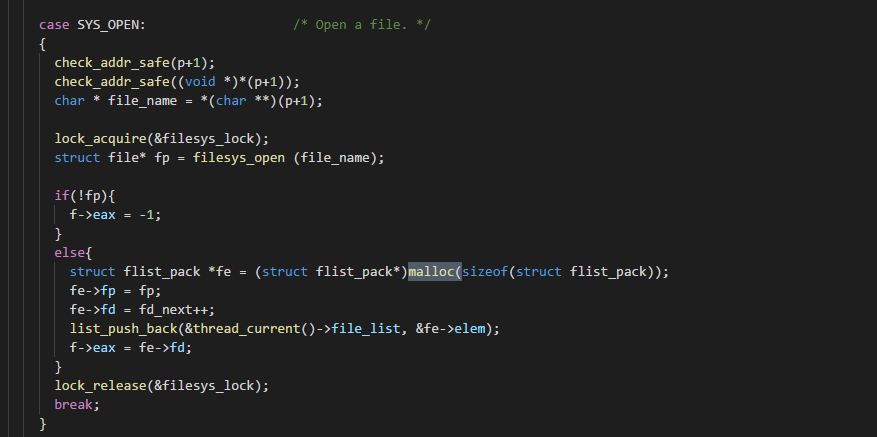


exec에서는 커맨드 라인 입력을 받아 그에 해당하는 프로세스를 실행한다. 해당하는 파일을 연 뒤, process\_excute를 호출한다. sys\_exec은 start\_proces와 load sema와 load\_succeed 여부를 공유하는데, 이는 load가 제대로 실행됐는지 확인해야 하기 때문이다.



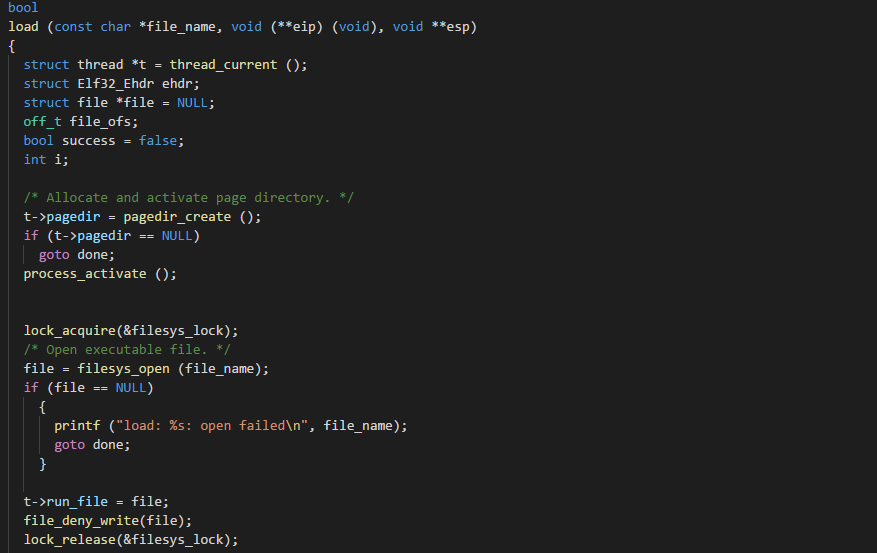
호출된 process\_execute에서는 입력을 파싱하여 쓰레드를 만든다. 여기서 load\_lock은 한 번에 많은 프로세스가 load되는 것을 방지하기 위함이다. thread\_create의 결과값으로 받은 tid 값을 반환한다.process\_execute에서 start\_process를 실행시키고, start\_process에서 load를 실행시킨다.

* 1. File manipulation



System call 중 file 과 관련된 system call이다. File create, file remove, file open, file size, read, write, seek, tell, close등이 포함된다. 이번 프로젝트는 File system을 구현하는 것이 주 목적이 아니기에 Filesys directory와 filesys.h, file.h에 있는 함수들을 수정 없이 그대로 사용하여 구현하였다. 파일 접근시 여러 프로세스가 동시에 접근하는 일을 막기 위하여 filesys\_lock을 만들어 이용하였다. 이외에 특기할 사항으로는 read와 write시에 standard IO를 고려해야 한다는 것 정도가 있다.

1. Denying writes to executables



실행되고 있는 executable 파일을 누군가가 수정하면 안 되기 때문에, load에서 파일을 불러오면 그 파일을 수정하지 못하게file\_deny\_write()를 설정해주고, thread\_exit을 부를 때 file\_close() 안에 있는 file\_allow\_write()를 실행해준다. executable file은 각 thread의 run\_file 이 가지고 있다.

**3. Conclusion**

이번 프로젝트는 user program와 system call을 구현하였다. process.c 와 syscall.c를 중점적으로 수정하였다. argument passing이 low-level을 건드리는 작업이어서 처음에는 상당히 혼란스러웠지만. 매뉴얼에 적힌대로 진행하니 수월하였다. 해당 작업이 끝난 뒤에 system call을 구현했는데, 파일 관련 호출은 이미 구현된 파일 시스템이 있어 구현하기 수월하였으나(중복처리 방지를 위한 lock정도가 문제였다) 프로세스 관리 부분은 상당히 어려웠다. 가장 고생한 부분은 process\_wait와 process\_exit 과정의 sync를 맞추어 주는 것이었는데, 결국 많은 조사 끝에 세마포어를 여러 개 사용해서 해결할 수 있었다. 이 부분에서 매우 많은 시간을 들였기에 sync와 메모리 누수를 체크하는 multi\_oom 테스트가 통과되었을 때는 순간 통과했다는 사실을 믿지 못할 정도였다. 이번 프로젝트를 통하여 OS가 프로세스를 어떻게 관리하고 사용자의 요청에 어떤 작업을 하는지에 대한 이해를 많이 높일 수 있었다. 결국 우여곡절 끝에 이번에도 올 패스를 이뤄내서 다행이다.

