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

담당 교수 / 분반 : 박수용 교수님

이름 / 학번 : 20181202 김수미

개발 기간 : 2021.11.1 ~ 2021.11.14

1. **개발 목표**Project 1에서는 구현하지 않았던 file system 관련 System calls 들을 구현한다.File system 관련 System call은 운영체제에서 파일의 생성, open, close등에 관여하는 기능을 관리하며 create, remove ,open, close, filesize, read, write, week, tell 등의 system call이 이에 해당한다. 해당 System call 들을 구현함과 동시에 File system 에서 필요한 File descriptor와 Synchronization 기능을 함께 구현한다.
2. **개발 범위 및 내용**
   1. **개발 범위**

**1) File Descriptor**File Descriptor(FD)란 process가 특정 file에 접근할 때 사용하는 값을 의미한다. 특정 파일을 open 하는 상황을 가정했을 때, kernel은 process의 file descriptor 중 사용하지 않는 가장 작은 값을 파일에 할당해 주는데, 이 때 다른 프로세스가 open 되어있는 파일에 system call로 접근하는 경우 이 fd 값을 이용해서 파일을 access 할 수 있다. Proj2 에서는 File Descriptor의 역할을 수행하는 자료구조를 직접 작성해 주어야 한다.

**2) System Calls**User가 kernel 수준에서 제공하는 서비스를 이용하기 위해 system call 명령어를 입력했을 때, 해당 명령어는 argument passing에 의해 stack에 저장되므로 system call handler를 통해 stack에 저장된 값들을 읽어 user가 요청한 서비스를 실행시킨다. Proj 1에서 한 것과 동일하게 user level과 Kernel API 사이에서의 system call handler를 작성하면 된다.

**3) Synchronization in Filesystem**Executable file 또는 running program이 delete 되거나, 동시에 여러 개의 process가 write 등과 같은 작업을 수행하려고 한다면 운영체제상에서 문제가 발생할 수 있다. 특히 두번째 상황은 여러개의 Process가 동시에 shared data를 사용하려고 할 때 발생한다. 이 때 발생할 수 있는 문제를 해결하기 위해 Critical Section 관리를 해 주어야 한다. 각 Process는 critical section이라고 불리는 code segment를 가지는데, 이 영역을 shared data가 access 한다. 따라서 어떤 process가 critical section에서 작동중에 있을 때 다른 process가 그 critical section에서 작동할 수 없도록 제한을 두어야 한다. Proj2 에서 해당 부분을 구현하게 된다.

* 1. **개발 내용**

**1) File Descriptor**File Descriptor를 구현하기 위한 자료구조로는 Array를 사용하였다. Array는 index를 이용한 random access가 가능한 자료구조이기 때문이다. 각 Process는 저마다의 file descriptor를 가지고 있어야 하기 때문에 thread 자료구조 내에 file구조체 배열 fd를 선언해 두었다. 핀토스 매뉴얼에서 process 한개에 최대 128개의 open files 개수 limit을 두는 것을 지향하고 있으므로, 배열의 크기는 128로 지정해 주었다.

**2) System Calls**

1. **create** : file 이름과 initial\_size를 입력 받아 해당 이름과 initial\_size(byte) 만큼의 크기를 가지는 새로운 file을 생성한다. 파일 생성에 성공했으면 true 값을, 실패했으면 false 값을 return 한다.
2. **remove** : 파일 이름을 입력 받으며 그 이름에 해당하는 파일을 삭제한다. 삭제에 성공하면 true를, 실패하면 false 값을 return 한다. 파일의 open/close 상태 여부에 관계 없이 삭제되며 open되어 있는 파일을 삭제하는 것을 그 파일을 close 하는 작업을 포함하지 않는다.
3. **open** : 파일 이름을 입력받아 그 이름에 해당하는 파일을 open 하며 파일에 할당되는 fd 값(0이상의 정수)을 return 한다. 파일 open에 실패한 경우 -1의 값을 return 한다. fd 값 중 0은 standard input을 의미하며 1은 standard output을 의미하는데, open system call은 0이나 1의 값은 반환하지 않는다.
4. **close** : 주어진 file의 file descriptor를 close 한다.
5. **filesize** : 주어진 fd로 open되는 file의 size를 byte 단위로 return 한다.
6. **read** : file descriptor와 buffer 변수, size 값을 입력 받으며, file descriptor가 가리키는 file 에서 size 만큼을 읽어 buffer에 저장한다. 실제로 읽은 byte 수를 반환한다. fd 값이 0이면 input\_getc( ) 함수를 이용해 keyboard로부터 입력을 받는다.
7. **write** : file descriptor와 buffer 변수, size 값을 입력 받으며, buffer에서 size 만큼의 값을 file descriptor가 가리키는 file에 write 한다. fd 값이 1이면 console창에 write 작업을 수행한다.
8. **seek** : open file fd에서 read 혹은 write를 하는 위치를 다음 byte로 이동시킨다.
9. **tell** : read 혹은 write의 대상이 되는 다음 byte의 position을 return 한다.

**3) Synchronization in Filesystem**

1. **Lock** : 하나의 file에 여러개의 process가 동시에 read 또는 write 작업을 수행하는 문제를 방지하기 위해 lock 구조체 변수와 lock\_acquire, lock\_release 함수를 사용할 수 있다. read, write, open 명령을 수행하기 전에 lock\_acquire 함수를 호출하고, 작업이 끝난 후 flock\_release 함수를 호출하면 process의 critical section에 여러개의 process가 동시에 접근하는 것을 방지할 수 있다.
2. **Semaphore** : Process 간의 Synchronization 구현을 위해 semaphore 구조체 변수를 사용할 수 있다. Process가 execute 또는 start 될 때 semaphore 변수를 사용해주면 된다. thread 구조체 내부에 semaphore 구조체 변수 필드인 file\_semaph를 추가했고, sema\_up, sema\_down 함수를 이용해 Synchronization in File system을 구현했다.
3. **추진 일정 및 개발 방법**
   1. **추진 일정**

**11/1 ~ 11/7** : File Descriptor 자료구조 build + System Call 구현(1)

**11/7~11/12** : System Call 구현(2) + Synchronization 구현 + 디버깅

**11/12~11/13** : 코드 간소화 작업 및 보고서 작성

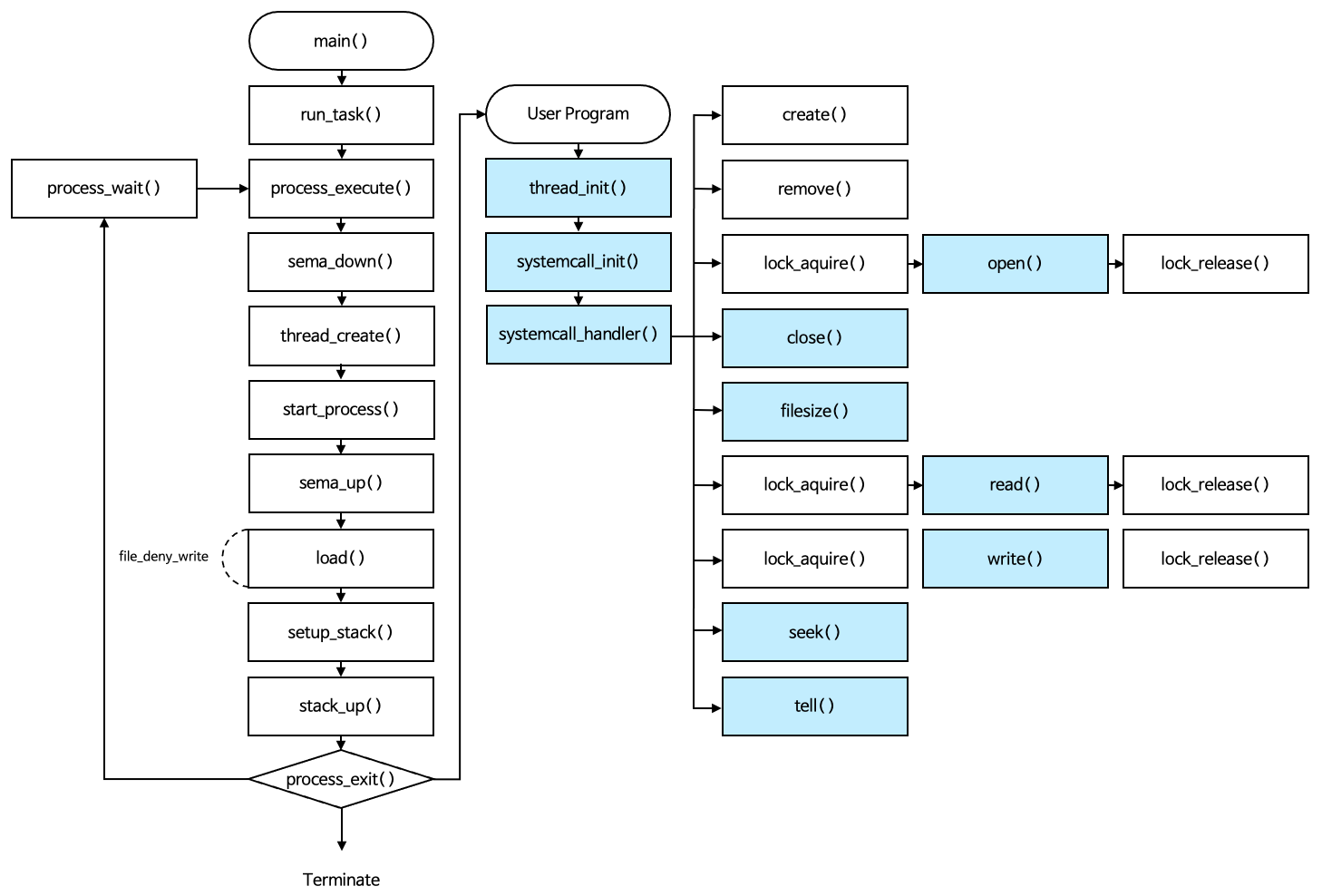
* 1. **개발 방법**

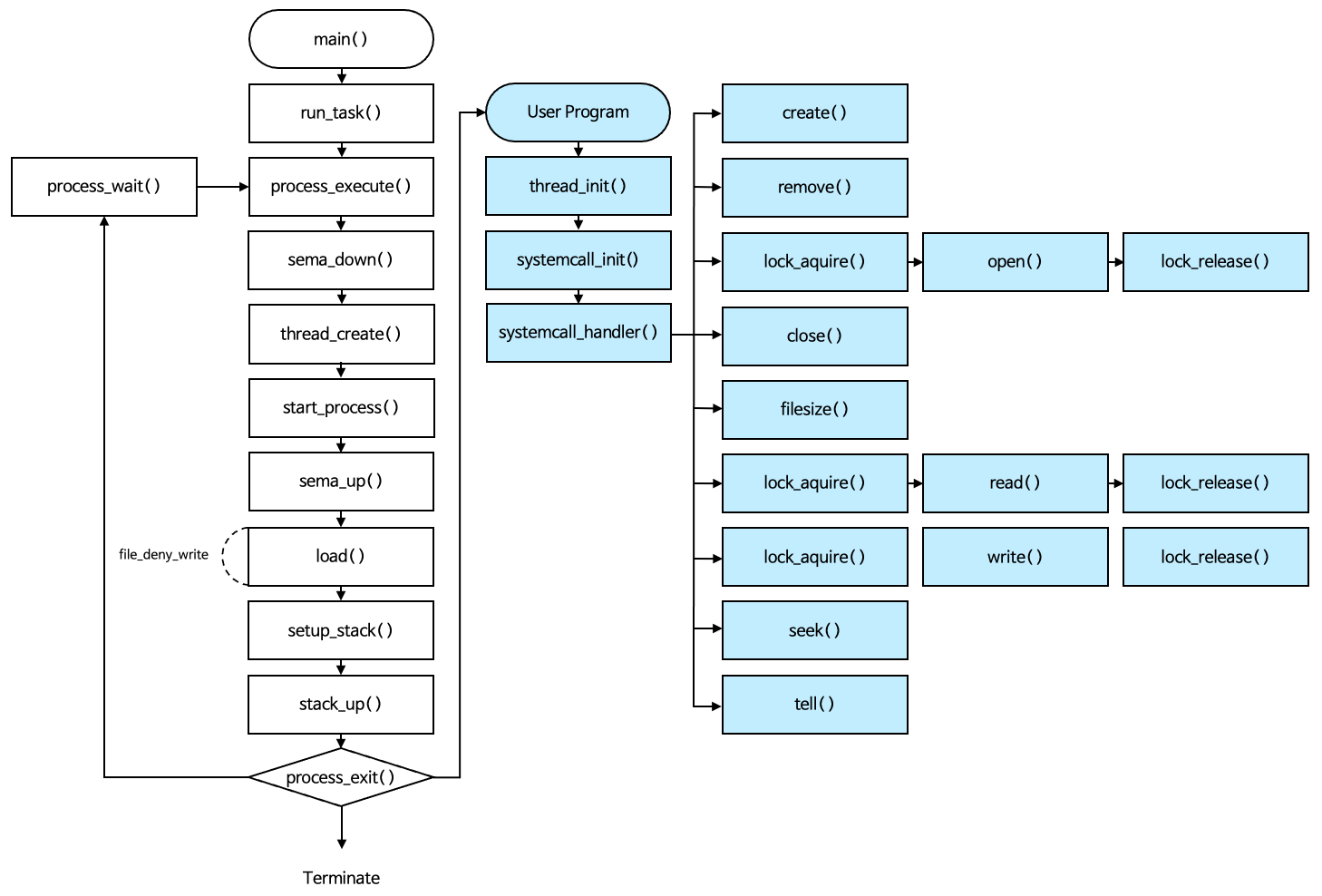
**1) File Descriptor**file.c, file.h, thread.h, thread.c 파일을 수정하여 File Descriptor를 구현할 수 있다.  
먼저 file.c에 선언되어 있는 file 구조체를 다른 파일에서도 사용할 수 있도록 file.c가 아닌 file.h에 선언해준다.  
그 다음 thread.h에서 thread 구조체 내부에 File Descriptor 구현을 위한 struct file \*fd[128] 필드를 추가해준다. 그리고 thread.c에서 init\_thread 함수 내부에 file descriptor를 NULL로 초기화 하는 코드를 추가한다.

**2) System Calls**syscall.c 파일을 수정하여 System call 들을 구현할 수 있다.  
syscall.c 파일의 syscall\_handler 함수 안에 있는 switch 문에 create, remove, open, close, filesize, read, write, seek, tell system call 관련 부분을 추가해준다. 또한 Proj 1 에서 구현한 read, write system call 부분에 filesystem을 이용하는 코드를 추가적으로 작성해 주었다.

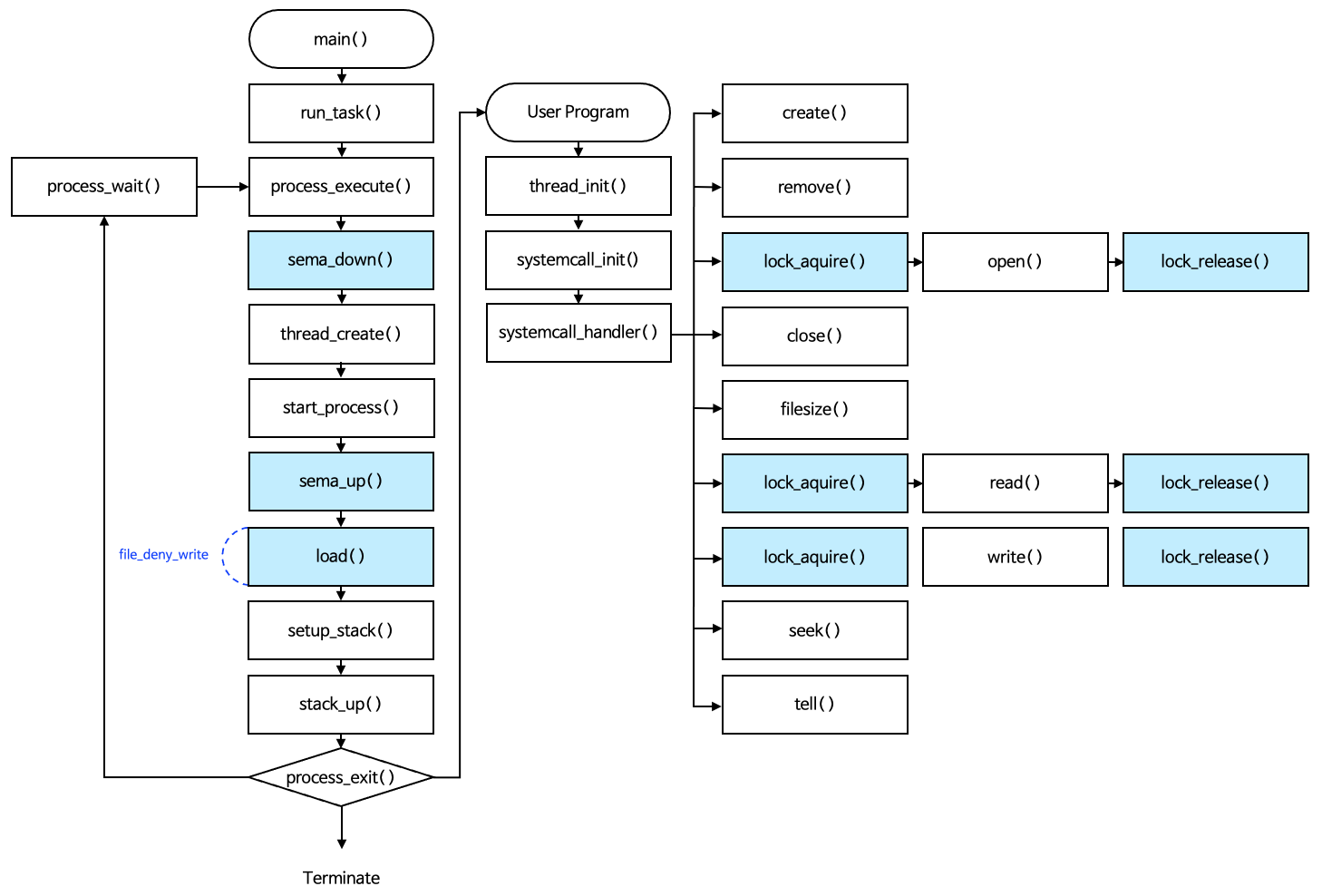
**3) Synchronization in Filesystem**thread.h, thread.c, exception.c, process.c, syscall.c 파일을 수정하여 Synchronization을 비롯한 각종 error 들로부터 시스템을 보호할 수 있다.  
먼저 thread.h 파일에서 lock\_acquire, lock\_relese 함수를 사용하기 위한 lock 구조체 변수 file\_lock을 선언해준다. 이 변수는 thread.c 의 thread\_init 함수 내에서 초기화된다.  
그 다음 thread.h 의 thread 구조체 내부에 현재 file process가 execution 상태에 있는지를 관리하는 semaphore 변수 file\_semaph와 해당 process의 parent process를 가리키는 thread 구조체 포인터 변수 parent, process가 비정상적으로 종료된 경우를 관리하는 check 값, 현재 실행중인 process의 file을 가리키는 current\_file 필드를 추가해준다. 여기서 추가해준 필드는 thread.c 파일의 init\_thread 함수 내부에서 초기화 시켜 주었다.  
그리고 exception.c 파일의 page\_fault 함수 내부에 not\_present 값이 1로 set 되는 경우 sys\_exit(-1) 함수를 호출하는 부분을 추가해 주었다. 이 부분을 추가해야 not-present page를 발생시키는 경우의 test 들을 통과할 수 있다.  
그 다음 process.c 파일에서는 제일 먼저 process\_execute 함수와 start\_process 함수 내부에thread 구조체의 file\_semaph 필드를 이용하여 현재 process가 load 되기 전 parent process가 끝나는 경우, 또는 현재 process가 문제없이 작업을 끝내는 경우에만 메모리를 정리하도록 처리해 준다. 또한 현재 process가 비정상적으로 종료되는 경우의 처리를 위해 thread 구조체의 check 값 필드를 확인하여 process\_wait를 실행시킬지의 여부를 결정하는 부분도 추가해 준다. load 함수 내부에는 process loading이 성공적으로 수행되었는지를 표시하는 success 변수를 검사하는 부분을 추가해 success 값이 1인 경우 실행중인 thread에 write작업이 수행되는 것을 방지하기 위해 file\_deny\_write(file) 함수를 호출하는 부분을 추가해준다.  
syscall.c 파일에서도 system call 관련 부분을 작성하면서 synchronization 기능도 구현해 주어야 한다. 여기서는 lock\_acquire, lock\_release 함수를 이용하여 open, read, write system call 을 수행하는 중에 다른 process로 부터 중복된 system call이 발생하는 것을 방지해 줄 수 있다. 또한 Proj 1에서 작성한 sys\_exit 함수 내부에 저장한 thread의 current\_file 변수에 file\_close 함수를 호출해주는 코드를 추가한다.

1. **연구 결과**
   1. **Flow Chart**

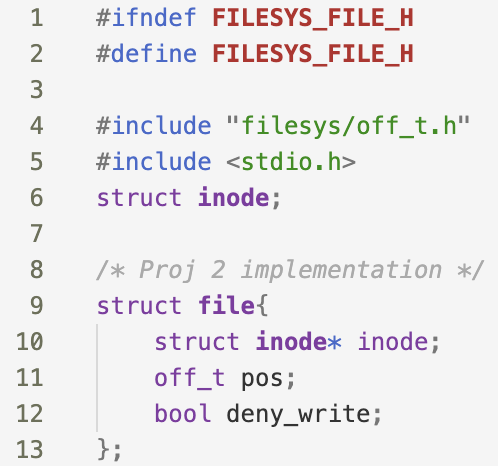
**1) File Descriptor  
**

**2) System Calls  
**

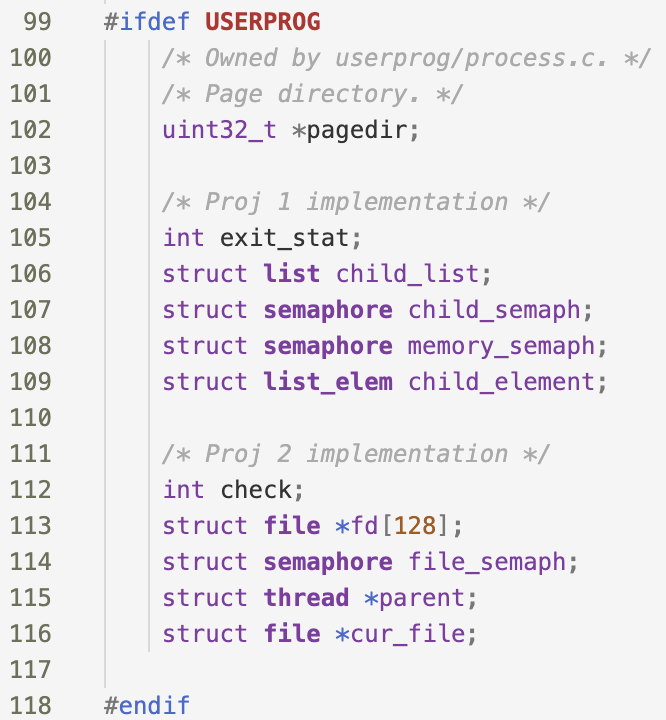
**3) Synchronization in Filesystem**

****

* 1. **제작 내용**

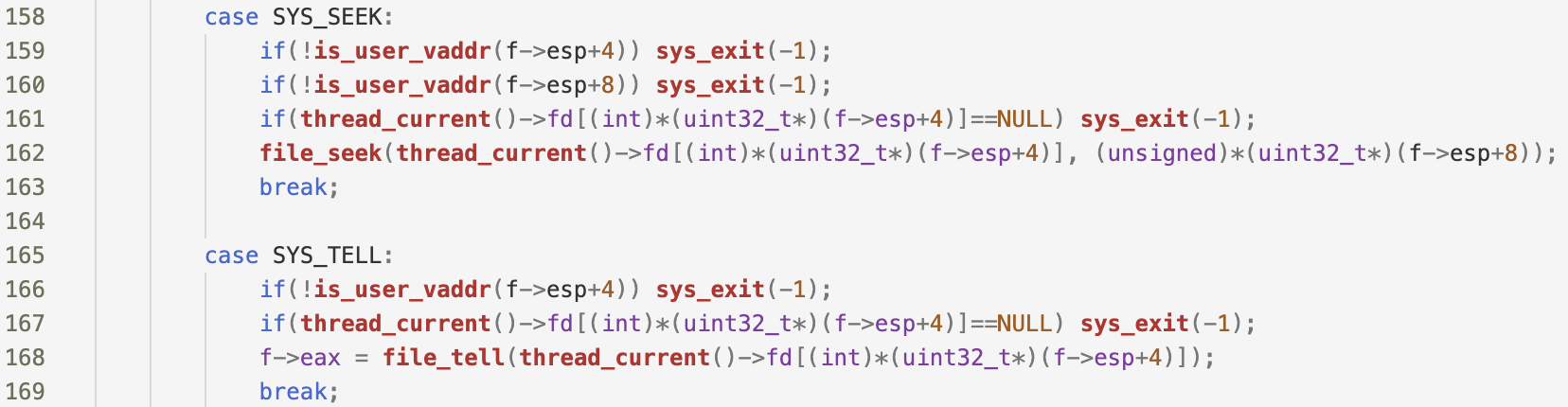
**1) File Descriptor  
**

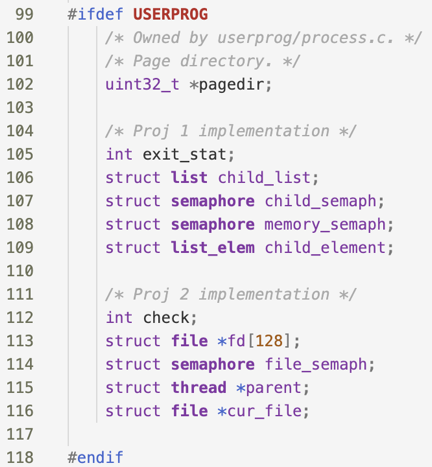
먼저 file.c 에 선언되어 있던 file 구조체를 file.h 에 옮겨서 선언해준다.

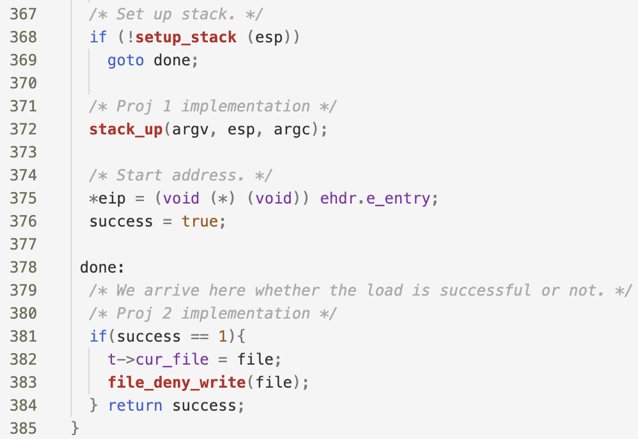


그 다음 thread.h 파일 내부의 thread 구조체에서 USERPROG 조건부 컴파일 부분에 file descriptor 배열 file \*fd[128]을 선언해준다.

  
선언한 배열은 thread.c 파일의 init\_thread 함수 내부에서 각 값을 NULL로 초기화 시켜준다.

**2) System Calls  
**SYS\_CREATE system call을 구현한 부분이다. 입력받은 인수를 filesys\_create 함수의 인자로 넘겨주면 된다.  
  
SYS\_OPEN system call을 구현한 부분이다. 입력 받은 인수를 filesys\_open 함수를 이용하여 open 하고, open한 file이 NULL이 아닌 경우 파일에 fd 값(0이상의 정수)을 할당하여 해당 fd 값을 return 한다. open 작업을 수행하는 중 다른 process가 access하는 것을 막기 위해 lock\_acquire, lock\_release 함수를 open 작업의 시작과 끝에 호출해 주었다.  
  
SYS\_CLOSE system call을 구현한 부분이다. 입력받은 인수의 file descriptor를 fd 배열에서 찾아 file\_close 함수의 인자로 넘겨주면 된다. close 작업을 마친 다음에는 thread\_current의 fd 값을 NULL 로 초기화 해 주어야 한다.  
SYS\_FILESIZE system call을 구현한 부분이다. 입력받은 인수의 file descriptor를 fd 배열에서 찾아 file\_length 함수의 인자로 넘겨주면 된다.  
  
SYS\_SEEK system call을 구현한 부분이다. 입력 받은 인수의 file descriptor를 fd 배열에서 찾아 file\_seek 함수의 인자로 넘겨주면 된다.  
SYS\_TELL system call을 구현한 부분이다. 입력 받은 인수의 file descriptor를 fd 배열에서 찾아 file\_tell 함수의 인자로 넘겨주면 된다.

**3) Synchronization in Filesystem  
 **thread.h에서 Synchronization에 필요한 변수 및 thread 구조체 필드를 추가한 결과이다.  
먼저 system call 에서 lock\_acquire, lock\_release 함수 호출 시 사용되는 lock 구조체 변수 file\_lock을 선언해주고, thread.c 의 thread\_init 함수에서 이 변수를 초기화하는 부분을 추가해준다. ** **그 다음 thread 구조체 내부에 현재 file process가 execution 상태에 있는지를 관리하는 semaphore 변수 file\_semaph와 해당 process의 parent process를 가리키는 thread 구조체 포인터 변수 parent, process가 비정상적으로 종료된 경우를 관리하는 check 값, 현재 실행중인 process의 file을 가리키는 current\_file 필드를 추가해준다. 여기서 추가해준 필드들은 thread.c 파일의 init\_thread 함수 내부에서 초기화 시켜 주었다.  
  
exception.c 파일의 page\_fault 함수 내부에 not\_present 값이 1로 set 되는 경우 sys\_exit(-1) 함수를 호출하는 부분을 추가해 준 것이다. 이 부분을 추가해야 not-present page를 발생시키는 경우의 test 들을 통과할 수 있다.  
   
process.c 파일에서 process\_execute 함수와 start\_process 함수 내부에 thread 구조체의 file\_semaph 필드를 이용하여 현재 process가 load 되기 전 parent process가 끝나는 경우, 또는 현재 process가 문제없이 작업을 끝내는 경우에만 메모리를 정리하도록 sema\_down 함수와 sema\_up 함수를 이용하여 처리해 주었다. 현재 process가 비정상적으로 종료되는 경우의 처리를 위해 thread 구조체의 check 값 필드를 확인하여 process\_wait를 실행시킬지의 여부를 결정하는 부분도 start\_process 함수 내부에 추가해 주었다.

  
process.c 파일의 load 함수 내부에는 process loading이 성공적으로 수행 되었는지 표시하는 success 변수를 검사하는 부분을 추가해, success 값이 1인 경우 실행중인 thread에 write 작업이 수행되는 것을 방지하기 위해 file\_deny\_write(file) 함수를 호출하는 부분을 추가해 주었다.  
  
Proj 1에서 구현한 SYS\_READ system call 부분에 lock 구조체 변수를 이용한 synchronization 부분을 추가해 준 것이다. file descriptor 값이 0 인 경우 이는 standard input을 의미하므로 input\_getc 함수로 read 작업을 처리해 주었고, file descriptor 값이 3 이상인 경우에는 file\_read 함수로 read 작업을 처리해 주었다. read 작업의 시작과 끝을 lock\_acquire 와 lock\_release로 감싸 작업이 시행되는 도중 다른 process로 부터의 접근이 발생하는 것을 방지했다.  
  
Proj1에서 구현한 SYS\_WRITE system call 부분에 lock 구조체 변수를 이용한 synchronization 부분을 추가해 준 것이다. file descriptor 값이 1 인 경우 이는 standard output을 의미하므로 putbuf 함수를 이용하여 write 작업을 처리해 주었고, file descriptor 값이 3 이상인 경우에는 file\_write 함수로 write 작업을 처리해 주었다. write 작업의 시작과 끝을 lock\_acquire 와 lock\_release로 감싸 작업이 시행되는 도중 다른 process로 부터의 접근이 발생하는 것을 방지했다.  
  
sys\_exit 함수 내부에 parent process가 현재 process(child process)보다 먼저 종료되는 것을 방지하기 위해 현재 process에 대한 process\_wait를 호출하는 while문을 추가하여 parent process가 child process의 종료 signal을 기다리도록 처리했다. child process의 실행이 종료되면 현재 thread가 관할하는 file에 대해 file\_close작업을 수행함으로써 open된 file을 먼저 close해주고 해당 thread를 thread\_exit 시켜 주었다.

* 1. **시험 및 평가 내용**

