CSE 3211 : Operating System Assignment 2

Saif Mahmud
2015-116-815 & SH-54

Tauhid Tanjim
2015-716-819 & SH-58

January 30, 2019

1 Introduction

We have implemented the following system calls regarding file system in OS161: **open, close, read, write, lseek and dup2** as part of the assigned task. In order to accomplish the task of implementing syscalls, we have modified the following 7 files which are listed below :

- kern/syscall/file.c
- kern/include/file.h
- kern/include/syscall.h
- kern/arch/mips/syscall/syscall.c
- kern/include/uio.h
- kern/lib/uio.c
- kern/include/proc.h

2 Description of Implementation

2.1 kern/include/syscall.h

In this file we have included the header ***"file.h"***.

2.2 kern/arch/mips/syscall/syscall.c

In this C file, we have used the function calls needed for each system call related to the implemented file system. We have obtained the syscall numbers from the file ***kern/include/kern/syscall.h***. In these system calls discussed above, we have used the corresponding functions and they are SYS_open(), SYS_close(), SYS_write(), SYS_read(), SYS_lseek() and SYS_dup2(). We gained the arguments from the trapframe structure which is used as a parameter of the syscall functions. The result of the system calls is assigned in the ret_val variable that is passed through the v0 and v1 variables in the trapframe. In case of SYS_lseek() we used another type of variable in order to return the result. Here we have used an off_t type variable named retval_off.

2.3 kern/include/proc.h

In this header file, we have initialized a file descriptor table in the proc structure. We have included structure :

file_t *file_descriptor_table [MAX_PROCESS_OPEN_FILES];

Therefore, the file descriptor table is an array of "file_t" type and is of size MAX_PROCESS_OPEN_FILES.

2.4 kern/include/uio.h

In this file, we implemented a function to initialize input-output buffers.

2.5 kern/lib/uio.c

In this C file, we have added the function that initializes buffers to carry blocks of data after executing file system calls. It creates a buffer, pointer to the created buffer, its offset, its size and other information that construct the uio structure before allowing the transfer of data.

2.6 kern/include/file.h

In this header file for the system calls of OS161 file system, we have defined a structure named ***"file_t"***. This structure keeps all necessary pieces of information for a file. The variables of this structure are listed below :

- **off_t offset** : To keep track of the offset. It is modified after read, write and seek operation.
- **int openflags** : To store the flags of the file.
- **int32_t references** : To store the references count. Its purpose is same as unix file systems reference count.
- **struct lock *file_lock** : A lock type pointer used to provide mutually exclusive access to files.
- **struct vnode *v_ptr** : A vnode pointer required for vfs function calls.

In addition to this, we have declared the function prototypes we used in *file.c*.

2.7 kern/syscall/file.c

The basic requirement of the implementation was to initialize per process file descriptor tables with a global open file table. The implementation of the per process file descriptor tables was done by including a reference to a table in the process structure. Hence we were able to access each process' fd table through *curproc*. The table itself is simply an array of pointers to the entries in the global open file table and new slots are assigned through a linear search for empty slots. The entries of the global open file table contained :

- Reference to the vnode that was acquired from the vfs

- Lock for mutual exclusion when reading and writing. This would protect against two entries writing at once and also prevents two reads from messing with the offset file pointer
- File pointer to keep track of the current offset
- Flags that this file was opened with number of handles that reference this entry

In case of the implementation of the global file table, we decided to not keep the entries in any data structure but leave them in the kernel heap. We made this decision since we found that there was no need to keep the entries in a structure which would lead to unnecessarily complicate the implementation. If we simply keep references to them in the file descriptor tables.

The implementation details of the system calls function for the file system is described below :

- **sys_open** : This function essentially passes most of the work off to vfs_open and creates the new open_file entry in the process descriptor table and subsequently creates the lock, initializes offset to 0 etc. First thing that needed to be done to prevent security issues with the userptr to the filename is we used copyinstr to transfer the filename into kernel memory safely before passing to vfs_open. We also check the userptr filename isn't NULL first. We have also reused most of the code used by this function to bind STDOUT and STDERR to the 2 and 3 descriptors respectively.
- **sys_close** : For this function we had to consider what would happen if sys_close was called on a descriptor that had been cloned whether by fork() or dup2(). Since we need to free all the memory allocations only when we delete the last reference, we decided to keep a count of references on the open_file data structure, only freeing the memory when it reached 0.
- **sys_read** : We first do some user input checking like checking whether the file has not been opened in O_WRONLY mode. We then acquire the open file lock and create a uio in UIO_USERSPACE and UIO_READ mode and pass this to vop_read with the open file pointer to read the data directly into the userptr buffer. We then compute the amount read data by subtracting the initial file offset before reading from the new offset returned by the uio. The reading is done with the lock acquired to ensure mutual exclusion with multiple processes reading the same open file and advancing the offset file pointer.
- **sys_write** : Write is in fact the replica of read, for example call vop_write instead and initialize the UIO in UIO_WRITE mode, check that file hasn't been opened in O_RDONLY mode.
- **sys_lseek** : This system call requires more than 4 registers, since the 2nd argument is a 64 bit value, causing the arguments to be assigned to a2 and a3. This subsequently required us to fetch the last argument from the stack. Hence the implementation of the function incorporates modifying the offset field in the open_file entry and taking concurrency into account.

- **sys_dup2** : This function simply (after error checking) copies the reference in oldfd to newfd, calling sys_close on newfd if it was already populated. We also increment the reference counter, taking concurrency into account through lock.

3 Conclusion

According to above discussion, we have successfully implemented the basic system calls for the file system as per the assigned task in order to access and edit the files organised in the file system of OS161 with respect to our conceived design principle described above. It should be noted that we have encountered a number of system inconsistency while framing the syscalls which we did troubleshooted through debugger and fixed the issues regarding the file system calls implementation.