CSE 3211: Operating System Lab
Assignment 2: System Calls and Processes
Design Document

Tanjid Hasan Tonmoy
Roll : 09

Syeed Abrar Zaoad
Roll: 23

1 File System System Calls

In this assignment, we were required to implement the following file-based system calls.

open(), read(), write(), lseek(), close(), and dup2()

To accomplish that we implemented per process file descriptor tables and a global file table for open files. Process table has been included in the definition of the process structure with a reference to the aforementioned table. The file table for a process is an array of pointers which reference the entries in the global open file table. We use simple linear search to find empty position to assign new slots to the table.

As per general convention we defined function prototypes, data structures etc in various header files as required and implemented these functions in .c files. The main c files that have been modified are file.c, proc.c, arch/mips/syscall/syscall.c, uio.c.

The global file table has not been maintained using any separate data structure. Rather we utilize the kernel heap, since we simply keep references to the entries in the file descriptor tables.

The entries of the global open file includes :

- reference to the vnode acquired from vfs
- lock for mutual exclusion
- file pointer to keep track of the current offset
- flags

- reference count

The implementation of various syscalls are described as follows-

1.1 sys_open

This system call has been implemented using the vfs_open function under the hood, so vfs handles much of the work. For convenience we defined a function open() which actually calls the vfs_open(). This function is also reused to bind STDOUT and STDERR to the descriptors. A file_handle entry is created in the process descriptor table which initializes offset to 0, creates lock and performs other related tasks. We used copyinstr to move the filename safely into kernel memory before passing it to vfs_open for security. At the beginning, we made sure that the userptr filename is not null. We also checked to ensure that no more than maximum number of files were opened.

1.2 sys_close

We have kept count of the number of references in the file_handle data structure. The memory would have to be freed only when that count is zero e.g. the descriptor is not cloned anywhere else. So we finally free the memory while removing the last reference using vfs_close. This also has implication regarding concurrency and synchronization since one handle may have multiple reference. We used lock while decreasing reference counts to ensure mutual exclusion. Other regular checks were also performed as in sys_open.

1.3 sys_read

We validated the arguments passed to the function before performing the read operation. For example checking which mode the file is opened in and ensuring it is not O_WRONLY. We acquire a lock initialize uio in UIO_USERSPACE and UIO_READ mode and finally use vop_read function. This takes with the open file pointer to read the data directly into the userptr buffer safely. The amount read is calculated by subtracting the initial file offset from the new offset returned by uio. The lock is required to prevent multiple processes advancing the offset file pointer while reading the same open file.

1.4 sys_write

We checked if the file has been opened in read only mode(O_RDONLY) and if not then we proceed with the write. Necessary validation such as more than maximum allowed files not being opened has been performed and entry in file table was done prior to performing write.

For this system call, we relied on the implementation of read for virtual file system vop_write(). UIO has been initialized in UIO_WRITE mode and mutual exclusion was ensured during the operation by using lock.

1.5 sys_dup2

We perform some error checking and copy the reference in oldfd to newfd, using sys_close on newfd if it was already opened. We also increment the reference count, taking concurrency issues into account.

1.6 sys_lseek

This system call was implemented by modifying the offset field open_file entry after performing some checks for possible errors as in the other syscalls. This is the only system call we implemented that needed more than 4 registers, since the 2nd argument is a 64 bit value. It was assigned to a2 and a3 registers. This required fetching the last argument from the user stack. We also took concurrent access into account and used lock.

2 conclusion

The fork system call has not been implemented. We have implemented all the required file system system calls as described in this document and these seem to work as expected.