Design of Operating System Assignment 2: System Calls and Processes

Kashob Kumar Roy
Roll:01

Md.Musfiqur Rahman
Roll: 05

January 27, 2019

Design and Implementation Issues:

In this assignment we have implemented the file-system-related system calls: open(), read(), write(), lseek(), close(), and dup2(). Actually here we have constructed the subsystem that implements the interface for the user-level programs by invoking the appropriate VFS and vnode operations.

Each process contains a File descriptor array of fixed size. For any given process the indexes 0,1 and 2 of file descriptor table are reserved. Index 0 , 1 and 2 are reserved for standard input (stdin), standard output (stdout) and standard error (stderr) respectively. Others indexes are used for the other file operations.

Here is the process structure as follows:

```
struct proc {
        char *p_name; /* Name of this process */
        struct spinlock p_lock;
        /* Lock for this structure */

        unsigned p_numthreads;
        /* Number of threads in this process */

        /* VM */
        struct addrspace *p_addrspace;
        /* virtual address space */

        /* VFS */
        struct vnode *p_cwd
        /* current working directory */

        struct File *process_file_table[MAX_PROCESS_OPEN_FILES];
```

```
        /* File Table For Each Process*/
        struct proc *parent_process;
        /* Pointer for Parent Processs */
        struct proc *child_process_list[MAX_CHILD_PROCESS];

        int count;
};
```

Here we are going to describe briefly each file operation.

The trapframe to handle system calls:
We got the system call no from the following line

```
callno = tf->tf_v0;
```

Then according to the system call we called different function for open(), close() , read(), write(), lseek() and dup2() and passed trapframe as argument. We did not implemented the fork() function which is a bit complex. After executing the system call , program counter is increased by 4 so that new system call can be executed.

syscall_open():
Main works of this function are: to create the new open_file entry in the process descriptor table, create the lock and initialize the offset to 0 etc. It essentially passes most of the work off to the vfs_open() function. The filename was received as parameter and an memory was allocated for opening a file. Then It was checked whether any index of the file descriptor table is available in the process file table.

```
for (; newfile_index < MAX_PROCESS_OPEN_FILES;
                                        newfile_index++) {
        if (curproc->process_file_table[newfile_index]
                                            == NULL) {
                break;
        }
}
```

Finally we can open the file with vfs_open() function since we included vfs.h library.

```
result = vfs_open(filename, flags, 0, &vn);
```

One 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 also reuse most of the code used by this function to bind STDOUT and STDERR to the 2 and 3 descriptors respectively.

syscall_close():

For this function we had to consider what would happen if syscall_close was called on a descriptor that had been cloned (whether by fork() or dup2()).we kept a count of references on the open_file data structure. When the count of references reached 0, we need to free all the memory allocations. That's mean when the last reference was deleted, we made all allocated memory free. Some considerations based on concurrency also needed to be made since if two processes had descriptors that point to the same open_file, we could potentially have memory leaks.

```
if(file->references<=0) {
            lock_release(file->flock);
            vfs_close(file->v_ptr);
            lock_destroy(file->flock);
            kfree(file);
    }
    else{
            lock_release(file->flock);
    }
```

syscall_read(): We will first check user input if the file is opened in O_WRONLY mode. After acquiring 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 safely. We get the read amount by subtracting initial file offset from new current offset which is returned by the uio. Lock is acquired to prevent concurrency issues with multiple processes when they try to read the same open file and so we advance the offset file pointer.

```
lock_acquire(file->flock);
        off_t old_offset = file->offset;
        uio_uinit(&iov, &myuio, buf, size,
            file->offset, UIO_READ);
        int result = VOP_READ(file->v_ptr, &myuio);
        if (result) {
                lock_release(file->flock);
                return result;
        }

        file->offset = myuio.uio_offset;
        *ret = file->offset - old_offset;
        lock_release(file->flock);
```

The file lock has been acquired first. Then the contents of the location of the file has been read. Finally it returns the next location of the file from where the contents of file will be read next .

syscall_dup2(): dup2() system call copies the file reference from oldfd to newfd. But if the newfd is already occupied then sys_close is called. Cosidering

concurrency, we also increment the reference counter. We initialize the newfd index of process file table with the data of the oldfd process table information.

```
struct File *file = curproc->process_file_table[newfd] =
                            curproc->process_file_table[oldfd];
        lock_acquire(file->flock);
        file->references++;
        lock_release(file->flock);
```

syscall_write(): Write syscall is mainly duplicate of read syscall. We will call vop_write instead and initialize the UIO in UIO_WRITE mode and also check that file hasn't been opened in O_RDONLY mode. We acquire a lock just before writing in the file.

```
    lock_acquire(file->flock);
    int result = VOP_WRITE(file->v_ptr, &fuio);
            if (result) {
                    lock_release(file->flock);
                    return result;
                }
```

syscall_lseek(): We will need four or more registers to store the values in this system call. So we store and load the last argument from user stack. We implemented the function considering the modification of the offset field in the open_file entry and also concurrency control. This function will return the current file position from where the content of the file will be shown.

```
case SEEK_CUR:
        lock_acquire(file->flock);
        if(file->offset + pos < 0){
                lock_release(file->flock);
                return EINVAL;
        }
        *ret = file->offset += pos;
        lock_release(file->flock);
        break;
```

To implement the Open(), Close(), Read(), Write() and lseek() functions, we have to change in the following files.

- asst2/src/kern/include/file.h
- asst2/src/kern/syscall/file.c
- asst2/src/kern/include/proc.h
- asst2/src/kern/proc/proc.c
- asst2/src/kern/include/syscall.h
- asst2/src/kern/arch/mips/syscall/syscall.c
- asst2/src/kern/include/uio.h