CSE-3201 Operating System

Assignment 2

System Calls and Processes(Filesystem)

Submitted by:

Shahrear Bin Amin, Roll-055

Al Mohammad Aladin, Roll-059

We implemented `open(), read(), write(), lseek(), close(), and dup2()` funcitons. We tried to implement *fork()* but failed.

We changed these files:

- `asst2/src/kern/syscall/file.c`
- `asst2/src/kern/include/file.h`
- `asst2/src/kern/include/proc.h`
- `asst2/src/kern/include/syscall.h`
- `asst2/src/kern/arch/mips/syscall/syscall.c`
- `asst2/src/kern/proc/proc.c`
- `asst2/src/kern/include/uio0.h`

Each function described below:

1. **open()**

Function prototype*: int sys_open(userptr_t filename, int flags, int *ret);*

vfs_open performed most of the work. But before that, we checked that Is filename a valid pointer? Is flags valid? flags can only contain exactly one of O_RDONLY, O_WRONLY and O_RDWR

After these, we allocated fd to the opened file: just to scan the curthread->t_fdtable and find a available slot (NULL). Then we need to actually open the file using vfs_open. We copied filename into kernel buffer using copyinstr that vfs_open may destroy the pathname passed in.

Once vfs_open successfully returns, we initialized a struct fdesc. We paid special attention to fdesc->offset. Without O_APPEND, it should be zero. But with O_APPEND, it should be file size. So we checked it and use VOP_STAT to get file size if necessary. We also checkd the userptr filename isn't NULL first. We also reused most of the code

used by this function to bind STDOUT and STDERR to the 2 and 3 descriptors respectively

2. read()

Function prototype: *int sys_read(int filehandler, userptr_t buf, size_t size, int *ret);*

The main work here is using VOP_READ with struct iovec and struct uio. kern/syscall/loadelf.c is a good start point. However, initialized the uio for read/write for user space buffers. That means the uio->uio_segflg should be UIO_USERSPACE.

Note thatuio->uio_resid is how many bytes left after the IO operation. That's how we calculated how many bytes are actually read/written by len - uio->uio_resid.

3. write()

Function prototype: *int sys_write(int filehandler, userptr_t buf, size_t size, int *ret);*

Write is essentially the mirror of read. We used uio and iovec for this task. We checked if the file is read-only, in that case we threw an error. VOP_WRITE is used for writing into the file and we returned the result if it is successful. Concurrency is controlled using lock.

4. close()

Function prototype : *int sys_close(int filehandler);*

For this function we had to consider what would happen if sys_close() was called on a descriptor that had been cloned. This could have been done either by fork or by dup2. We also needed to free all the memory allocations after deleting the last reference, we decided to keep a count of references on the open() data structure, and we freed the memory when it reached 0. We also took some considerations based on concurrency issue since two processes had descriptor that could point to the same open_file which could potentially lead to memory leaks.

5.lseek()

Function prototype: *int sys_lseek(int fd, off_t pos, userptr_t whence_ptr, off_t *ret);*

Let's first see the definition of lseek off_t lseek (int fd, off_t pos, int whence) . From $OS161_SRC/kern/include/types.h, we can see that off_t is type-defined as 64-bit integer (i64). From the comment in $OS161_SRC/kern/arch/mips/syscall/syscall.c, we can see that, fd should be in $a0, pos should be in ($a2:$a3) ($a2 stores high 32-bit and $a3 stores low 32-bit**)**, and whence should be in sp+16. Here, $a1 is not used due to alignment. So in the switch branch of sys_lseek, we should first pack ($a2:$a3) into a 64-bit variable, say sys_pos. Then we use copyin to copy whence from user stack (tf->tf_sp+16).

Also from the comment, we know that a 64-bit return value is stored in ($v0:$v1) ($v0 stores high 32-bit and $v1 stores low 32-bit). And note that after the switch statement, retval will be assigned to $v0, so here we just need to copy the low 32-bit of sys_lseek's return value to $v1, and high 32-bit to retval.

6. dup2()

Function prototype: *int sys_dup2(int oldfd, int newfd);*

We can see that in dup2(oldfd, newfd): After dup2, oldfd and newfd points to the same file. But we called close on any of them and do not influence the other. After dup2, all read/write to newfd will be actually performed on oldfd. If newfd is previous opened, it is closed in dup2. We also increment the references counter to handle concurrency issues