Fs_create(char *filename, int mode):

-we first check the mode, if the mode is O_CREAT we proceed further or else we throw a SYSERR stating that the mode is wrong

-then we check the number of entries in the system if theres no file in the system then simply set the flag to 1 if there are other files in the system then we check if a file with the same name exists in the system, if it does, we return a SYSERR stating that the file with the filename already exist after setting the flag to 0, if there is no conflicting filename we set the flag to 1 and proceed further

-we set the metadata of the inode struct where id is set to the number of inodes used in the system, type is file, nlink is set to 1 and the device is set to dev0 with the size being 0 till now

-we initialize all the blocks of that inode to -1 and put the inode in the system by calling the function fs_put_inode_by_num

-then we set the metadata of the fsystem inode entry, filename, and numentries.

Fs_open(char *filename, in

t flags):

-we first check the number of entries in the directory if the directory has 0 entries that means it has no file in the system then we return a SYSERR stating that there are no files in the system to open currently

-then we check in the file system directory if the file actually exists in the system, if it does then we set the flag to 1 if it doesn't then we set the flag to 0 and return a SYSERR stating that the file requested to open does not exist in the system

- we then check the next_open_fd value if its not -1 then the file in that system exist and a valid file state

- if the state of the file is already FSTATE_OPEN then that file is already open in the system and we cant open that file again if not we update the state of that file from the system directory to FSTATE_OPEN, the fileptr is set to 0 and the name of the dentry is changed to the current filename

-we then change the value of the next_open_fd to the next FSTATE_CLOSED file, if no file is closed w change the next_open_fd to -1 since next_open_fd should point to a file which is in the closed state in the system.

Int fs_write(int fd, void *buf, int nbytes):

-we first check the state of the file, if the file is in the closed_state, then we return a SYSERR stating that the file is in closed state and we cannot write in it.

-calculate the block_index, block_number and the inode offset, these are used to write the blocks using the bs_write function, then we set the mask bit for that block_number. If the block write does not return a SYSERR then we increase the block index for the next block to be written and simultaneously decreasing the number of bytes that are to be written in the file from the buffer.

-the final step of file write is to increase the size of the inode by the buffer_counter which in case of my program is counter that points to each block using an integer indexed value

Int fs_read(int fd, void*buf, int nbytes):

-we first calculate the block_index, block_nunber and the Inode offset similarly like we did in the file read case and also check the state of the file, if the file state is closed we cannot read the file and return a SYSERR

-then we start reading bytes by bytes from the blocks, in this function we just have to return the starting index of the buffer from where we are supposed to read our files from so we return an integer variable. The block_index, block_number and the offset are calculated in order to help us read the blocks in the system.

-then similarly the file write we keep decreasing the number of bytes to be read during each iteration and check if the block_number is -1 if its -1 then we have reached the end of the block and should return a SYSERR, which means that the data from that block cannot be read

Int fs_seek(int fd, int offset):

-we start by setting the pointer as the filesystem fileptr and increase it by the offset

-then if the pointer is greater than the total size of the file in the directory then we set the fileptr of the file as the pointer and return an OK or else we return a SYSERR

Int fs_close(int fd):

-we free the memory of the position(fd) passed by the fclose function, the memory is freed from the file system dentry and the state of that file is then changed to FSTATE_CLOSED.

The things learned from these implementations are:

-the ext2 file system involves in memory fragmentation due to blocks allocated to each file not all of them are filled completely which involves internal memory fragmentation.