24.05.2019

CS342 OPERATING SYSTEMS
PROJECT 2

Elif Gülşah Kaşdoğan – 21601183
Sıla İnci – 21602415
SECTION 2

We traversed the list of active processes which are type of task_struct, we found the process with the given id. We took process id as parameter from the terminal. After finding the desired process we found fs_struct and files_struct to obtain desired information.

```
struct files_struct *current_files;
struct fs_struct *current_fs;
struct fdtable *files_table;
if (task->pid == processid)
{
        current_fs = task->fs;
        current_files = task->files;
        files_table = files_fdtable(current_files);
}
```

Each file has a files_table which represents open files of the process.
File table keeps index and respective pointers, we printed index number into table as descriptor number.
File object has inode object in it. Each file has inode which contains information such as user id, inode mode, file mode, inode number. We realized that we can retrieve some of these information also from dentry object. Dentry object has dentry inode which stands for the same inode held for a particular file.

```
files_path = files_table->fd[i]->f_path; //path of file
files_path.dentry->d_iname
files_path.dentry->d_inode->i_ino
```

From inode we can reach to number of blocks allocated and file length by reaching the”i_size” portion of the inode.

```
current_inode->i_size
current_inode->i_blocks
uid_t uid = i_uid_read(current_inode);// USER ID
umode_t       i_mode = current_inode->i_mode;
fmode_t f_mode = files_table->fd[i]->f_mode;
unsigned long i_ino = current_inode->i_ino;
cwd = d_path(&files_path,buf,100*sizeof(char)); //convert to string
```

From fs_struct we can obtain current directory of the process. Additional to the working directory of process we can also reach to file path of the open files by using f_path. Inode has a address_space struct which stands for the buffer cache(page cache) asked in 4th question. Adress_space is stored in inode with name “i_mapping”.

```
struct address_space *inode_adr_space = current_inode->i_mapping;
struct xarray array = inode_adr_space→i_pages;
```

xarray is a abstract data structure which holds pointers to actual memory locations. We obtained an array of type xarray and converted it to a void* array using extract function implemented in xarray.h. We traversed blocks and obtained block addresses using void* array.

Each address_space has an inode named host. According to documentation, host is either the inode or the block device file. We used host to retrieve information wanted in 5. host holds count for read and write and additional count which is use count, we printed all three operations.

```
struct inode *current_host = inode_adr_space->host; //storage device the block is in
atomic_t count = current_host->i_count;
 atomic_t read_count = current_host->i_readcount;
atomic_t write_count = current_host->i_writecount;
```

```
[ 2670.081623] Open file with fd 2  pipe:[68898]
[ 2670.081624] Current file position pointer: 00000000941f1e4f
[ 2670.081624] Current file position pointer: 0
[ 2670.081624] Descriptor number: 2
[ 2670.081625] User id: 1000
[ 2670.081625] i_mode: 4480
[ 2670.081625] f_mode: 786434
[ 2670.081625] File name:
[ 2670.081626] Inode no: 68898
[ 2670.081626] File length in bytes: 0
[ 2670.081626] Number of blocks allocated to file: 0
[ 2670.081627] Current directory of process: pipe:[68898]
[ 2670.081627] Host(inode or block device): 68898
[ 2670.081627] Use Count: 1
[ 2670.081627] Read Count: 1
[ 2670.081628] Write Count: 0
```

```
[ 2670.081662] Open file with fd 3  anon_inode:[eventpoll]
[ 2670.081662] Current file position pointer: 0000000047f7f82f
[ 2670.081662] Current file position pointer: 0
[ 2670.081663] Descriptor number: 3
[ 2670.081663] User id: 0
[ 2670.081663] i_mode: 384
[ 2670.081664] f_mode: 524291
[ 2670.081664] File name: [eventpoll]
[ 2670.081664] Inode no: 11882
[ 2670.081664] File length in bytes: 0
[ 2670.081665] Number of blocks allocated to file: 0
[ 2670.081665] Current directory of process: anon_inode:[eventpoll]
[ 2670.081665] Host(inode or block device): 11882
[ 2670.081666] Use Count: 690
[ 2670.081666] Read Count: 106
[ 2670.081666] Write Count: 0
```

```
[ 2670.083713] Block No 0: 00000000b4433ae3
[ 2670.083714] Block No 1: 0000000091b85711
[ 2670.083715] Block No 2: 00000000b452f81b
[ 2670.083716] Block No 3: 00000000bd6a3838
[ 2670.083718] Block No 4: 000000002f85d72d
[ 2670.083719] Block No 5: 000000005c505c49
[ 2670.083720] Block No 6: 0000000007761b15f
[ 2670.083721] Block No 7: 000000000bb1df62
[ 2670.083722] Block No 8: 000000005519b23b
[ 2670.083723] Block No 9: 000000001d59a131
[ 2670.083724] Block No 10: 00000000150db6e4
[ 2670.083725] Block No 11: 0000000005d1a8ab
[ 2670.083726] Block No 12: 00000000eb52c522
[ 2670.083728] Block No 13: 00000000e775b2d3
[ 2670.083729] Block No 14: 00000000d79d0512
[ 2670.083730] Block No 15: 00000000a11603e0
[ 2670.083731] Block No 16: 00000000c98a8226
[ 2670.083732] Block No 17: 000000006697d41e
[ 2670.083733] Block No 18: 000000003d808518
[ 2670.083734] Block No 19: 000000009d6f2bc1
[ 2670.083735] Block No 20: 000000006c49d8ec
[ 2670.083736] Block No 21: 00000000204cfe4b
[ 2670.083738] Block No 22: 000000003ff3a8c6
[ 2670.083739] Block No 23: 000000001ef19220
[ 2670.083740] Block No 24: 00000000f15604b6
[ 2670.083741] Block No 25: 00000000fee6fb98
[ 2670.083742] Block No 26: 00000000c0a264a3
[ 2670.083743] Block No 27: 00000000eb3dc423
[ 2670.083744] Block No 28: 000000004792913c
[ 2670.083745] Block No 29: 000000000370dbeb
[ 2670.083747] Block No 30: 000000005de817c5
[ 2670.083748] Block No 31: 0000000085712f31
[ 2670.083749] Block No 32: 00000000f6ee63f4
[ 2670.083750] Block No 33: 00000000f9f0f346
[ 2670.083751] Block No 34: 0000000050c6076e
[ 2670.083752] Block No 35: 000000008d9f285f
[ 2670.083753] Block No 36: 00000000791b95f8
[ 2670.083754] Block No 37: 00000000ad35121f
[ 2670.083755] Block No 38: 00000000c2b7d0d6
[ 2670.083756] Block No 39: 0000000092f0ee4b
[ 2670.083758] Block No 40: 00000000fd552335
[ 2670.083758] Block No 41: 00000000a2c1d69d
[ 2670.083759] Block No 42: 000000000c544fa6
[ 2670.083759] Block No 43: 00000000466015d6
[ 2670.083759] Block No 44: 0000000023bd5e1f
[ 2670.083760] Block No 45: 00000000b714b6c6
[ 2670.083760] Block No 46: 0000000025771716
[ 2670.083760] Block No 47: 00000000bc9d810b
[ 2670.083761] Block No 48: 00000000ad265357
[ 2670.083761] Block No 49: 00000000a6c29071
[ 2670.083761] Block No 50: 000000006e0fd5e8
[ 2670.083762] Block No 51: 0000000090675cb5
[ 2670.083762] Block No 52: 00000000924b7f14
[ 2670.083762] Block No 53: 00000000a0822dc6
[ 2670.083763] Block No 54: 00000000afffc33c
```

In our application we fork the parent process and with this process, we open a new file and write to it, then we close and open it again. We read from this file in batches and then close the file in the end. We print the pid of that process in order to test our module. We give this pid as a parameter to our module as below,

```
slanci@slanci-XPS-15-9560:~/Modules$ sudo insmod project.ko processid=9991
```

When we give this pid to our module, it prints the information of this process, the information wanted in the project assignment. Since we are opening the file in user mode, the kernel prints the information that the User id of the process file is 1000. (0 for kernel, 1000 for user mode) 4 blocks are allocated. Since we are moving down the tree, the first information printed by our module is related to the closest one to the end of the red-black-tree used by the kernel.