

File Systems and Processes

Code Review Reflection
(Procs) due EOD Friday!

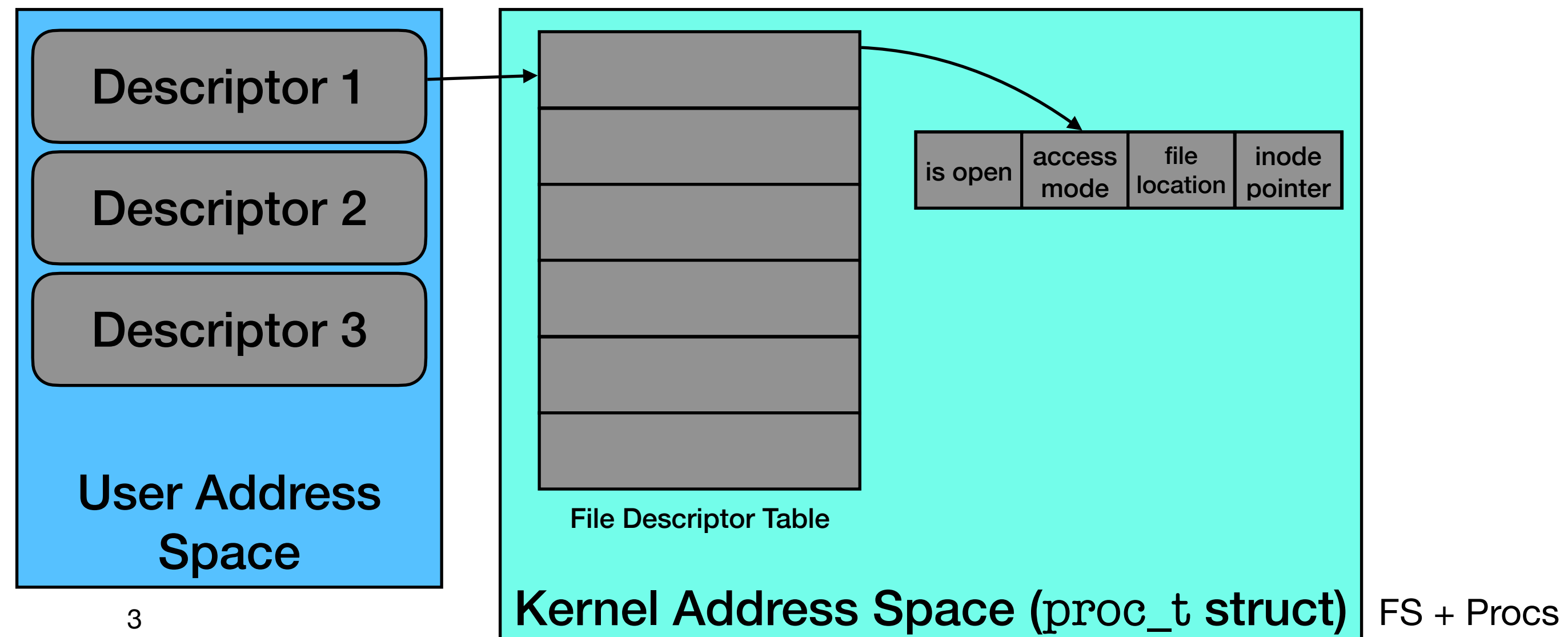
Outline

- The relationship between processes and the file system
- Procs: Code Review
- Maintaining and freeing vnodes

Processes and the File System

- To be able to interact with the file system, a process running in the OS kernel needs to have two distinct ways to interact with a file:
 - (1) having a notion of the current working directory in which it is running;
 - (2) being able to interact with various files (according to its permissions)

```
1 /* Process descriptor */
2 typedef struct proc
3 {
4     pid_t p_pid;           // Process ID
5     char p_name[MAX_STRING_LEN]; // Process name
6
7     list_t p_threads;      // Threads list
8     spinlock_t p_threads_lock; // lock for threads list
9     list_t p_children;     // Children list
10    spinlock_t p_children_lock; // lock for children list
11    struct proc *p_pproc;   // Parent process
12
13    list_link_t p_list_link; // Link of list of all processes
14    list_link_t p_child_link; // Link on parent's list of children
15
16    vnode_t *p_cwd;         // The current working directory
17
18    long p_status;          // Exit status
19    proc_state_t p_state;   // Process state
20 } proc_t;
```



Chat with your neighbor(s)!

Consider a program that opens the same file twice, what would this mean in the associated process' file descriptor table?

A Basic vnode Organizational Structure

```
1 /* File System descriptor */
2 typedef struct fs
3 {
4     vnode_t *fs_root;           // the root vnode of the file system
5
6     list_t vnode_list;          // a structure to track all vnodes in the system
7     spinlock_t vnode_list_mutex; // for synchronization of the vnode list
8 } fs_t;
```

```
1 /* vnode descriptor */
2 typedef struct vnode
3 {
4     vnode_ops_t *vn_ops;        // a pointer to a struct of functions that may
5                                 // be performed on a vnode (FS agnostic)
6
7     fs_t *vn_fs;                // pointer to the file system (parent)
8
9     long vn_len;                // the size of the file (in bytes)
10    long vn_mode;                // indicates various features about the file
11
12    mobj_t vn_mobj;              // the memory object associated with the file
13
14    list_link_t vn_link;         // for the link in the fs list of vnodes
15 } vnode_t;
```

A vnode is a generic kernel object to represent a file (in memory) in a file system agnostic manner

The file system maintains all active vnodes, which live in kernel memory *outside* of individual process memory

A Basic vnode Organizational Structure

```
1 /* vnode operations */
2 typedef struct vnode_ops
3 {
4     /*
5      * Read the data associated with a file
6      */
7     long (*read)(vnode_t *file, long pos, void *buf, long count);
8
9     /*
10     * Write the provided data to a file
11     */
12     long (*write)(vnode_t *file, long pos, const void *buf, long count);
13
14     /*
15     * Create a new regular file
16     */
17     long (*mknod)(vnode_t *dir, const char *name, long namelen, vnode_t **ret);
18
19     /*
20     * Create a new directory
21     */
22     long (*mkdir)(vnode_t *dir, const char *name, long namelen, vnode_t **ret);
23
24     // ...
25 } vnode_ops_t;
```

```
static vnode_ops_t svfs_dir_vops = {.read = NULL,
                                     .write = NULL,
                                     .mknod = svfs_mknod,
                                     .mkdir = svfs_mkdir,
                                     .lookup = svfs_lookup,
                                     .readdir = svfs_readdir,
                                     .read_pframe = svfs_read_pframe,
                                     .write_pframe = svfs_write_pframe,
                                     .flush_pframe = NULL};

static vnode_ops_t svfs_file_vops = {.read = svfs_read,
                                       .write = svfs_write,
                                       .mknod = NULL,
                                       .mkdir = NULL,
                                       .lookup = NULL,
                                       .readdir = NULL,
                                       .read_pframe = svfs_read_pframe,
                                       .write_pframe = svfs_write_pframe,
                                       .flush_pframe = NULL};
```

When implementing the System V File System operations, we can define different types of vnodes that the FS can use according to expected operations

Chat with your neighbor(s)!

Consider our file system implementation, how would we implement user space operations such as “ls” or “cd” using vnodes? Think about the types of attributes of the various objects and the how they are stored in the file system overall.

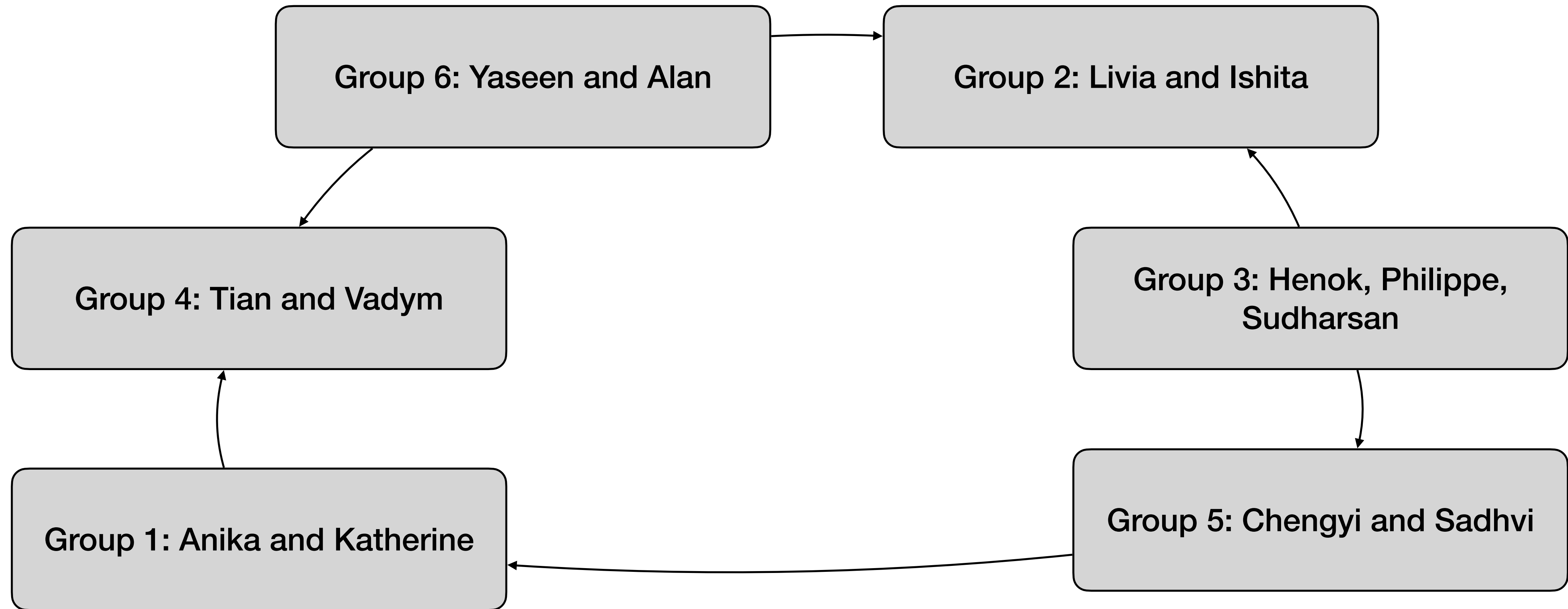
```
1 /* File System descriptor */
2 typedef struct fs
3 {
4     vnode_t *fs_root;           // the root vnode of the file system
5
6     list_t vnode_list;          // a structure to track all vnodes in the system
7     spinlock_t vnode_list_mutex; // for synchronization of the vnode list
8 } fs_t;
```

```
1 /* vnode descriptor */
2 typedef struct vnode
3 {
4     vnode_ops_t *vn_ops;        // a pointer to a struct of functions that may
5                                 // be performed on a vnode (FS agnostic)
6
7     fs_t *vn_fs;                // pointer to the file system (parent)
8
9     long vn_len;                // the size of the file (in bytes)
10    long vn_mode;                // indicates various features about the file
11
12    mobj_t vn_mobj;              // the memory object associated with the file
13
14    list_link_t vn_link;         // for the link in the fs list of vnodes
15 } vnode_t;
```

A Basic **vnode** Organizational Structure

- We use vnodes as a “base struct” on top of which a file system may implement the various functions necessary to have a wholly functional file system → vnodes do not necessarily describe regular files or directories, they instead provide the whole interface that could be used to access a file
- Often times, the file system implementation will utilize a different node type of which the associated vnode is one member — this is done to implement systems of caching, etc... to make the system more efficient or easier to implement
- Our file system acts as a container for the vnodes in our system... the naive choice of data structure is likely inefficient (but good enough for now)!

Procs: Code Review



Procs: Road Map

- Programming component is done 
- Code Review has been completed 
- Code Review Response due by Friday 
- If you would like to submit changes before I look at your code (for thoughts and feedback), you can continue to do so through Friday 
- Homework 3 released! Details on the course page (programming due October 26)

HW3 Programming Partners

Group 1: Ishita and Sadhvi

Group 2: Philippe and
Katherine

Group 3: Livia and Vadym

Group 4: Tian and Anika

Group 5: Alan and Sudharsan

Group 6: Yaseen, Chengyi,
and Henok

Chat with your neighbor(s)!

To create a `vnode` in the file system, we allocate the space in kernel memory and insert the created structure into the `vnode_list` in the file system. How do we know when a `vnode` can be destroyed? Think about what happens if one or more processes each have a file descriptor pointing to that file.

If a process refers to a `vnode`, we cannot destroy it

To track how many references to a `vnode` there are, the `vnode` itself needs to track a count of “references” pointing to it

When creating an entry in the process file descriptor table, we need to also increment the `vnode` reference count!

Summary

- We introduced the idea of a “vnode” which is a way for the file system to track all files currently in use (regardless of what type of file system those nodes belong to)
- To reason about the system of vnodes, we need to have a mechanism to implement the “file system tree” ➡ doing so in a list may or may not be the most efficient data structure to implement such a scheme
- To safely destroy a vnode, we need to maintain a reference counter to each vnode in the system ➡ this will allow us to know whether one or more processes is done with the file