OPERATING SYSTEMS – ASSIGNMENT 4

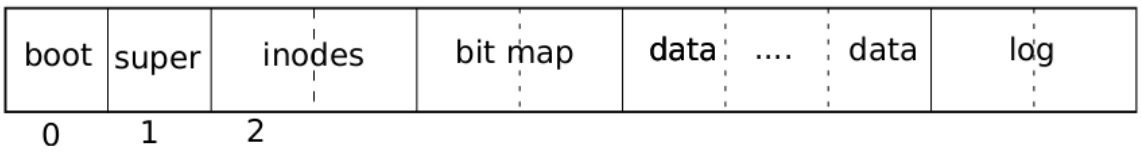
FILE SYSTEMS

Responsible TA – Matan Drory

The xv6 file system provides Unix-like files, directories, and pathnames, and stores its data on an IDE disk for persistence. The file-system addresses several challenges:

- The file system needs on-disk data structures to represent the tree of named directories and files, to record the identities of the blocks that hold each file's content, and to record which areas of the disk are free.
- Accessing a disk is orders of magnitude slower than accessing memory, so the file system must maintain an in-memory cache of popular blocks.
- Different processes may operate on the file system at the same time, and must coordinate to maintain invariants.
- The file system must support crash recovery. That is, if a crash (e.g., power failure) occurs, the file system must still work correctly after a restart. The risk is that a crash might interrupt a sequence of updates and leave inconsistent on-disk data structures (e.g., a block that is both used in a file and marked free).

To do so, xv6 divides the disk into several sections, as shown in the Figure below. The file system does not use block 0 (it holds the boot sector). Block 1 is called the superblock, it contains metadata about the file system (the file system size in blocks, the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold inodes. After those come bitmap blocks tracking which data blocks are in use. Most of the remaining blocks are data blocks. The blocks at the end of the disk hold the logging layer's log.



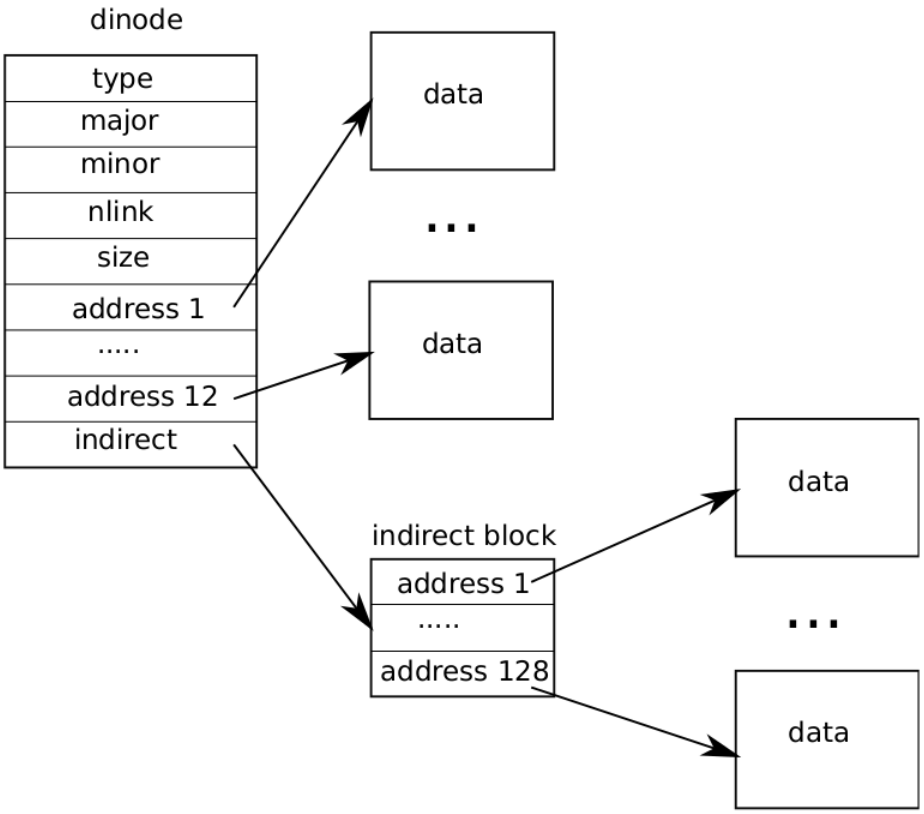
The term inode can have one of two related meanings. It might refer to the on-disk data structure (`struct dinode, fs.h`) containing a file's size and list of data block numbers. Or "inode" might refer to an in-memory inode (`struct inode, file.h`), which contains a copy of the on-disk inode as well as extra information needed within the kernel.

The on-disk inode is defined by a struct dinode. The type field distinguishes between regular files, directories, and special files (devices). A type of zero indicates that an on-disk inode is free. The nlink field counts the number of hard links to the file, that is, the number of directory entries that refer to this inode, in order to recognize when the on-disk inode and its data blocks should be

freed. The size field records the number of bytes of content in the file. The addrs array records the block numbers of the disk blocks holding the file's content.

```
// On−disk inode structure
struct dinode {
        short type; // File type
        short major; // Major device number (T_DEV only)
        short minor; // Minor device number (T_DEV only)
        short nlink; // Number of links to inode in file system
        uint size; // Size of file (bytes)
        uint addrs[NDIRECT+1]; // Data block addresses
};
```

The on-disk inode structure, struct dinode, contains a size and an array of block numbers (see Figure below). The inode data is found in the blocks listed in the dinode's addrs array. The first NDIRECT blocks of data are listed in the first NDIRECT entries in the array, these blocks are called direct blocks. The next NINDIRECT blocks of data are listed not in the inode but in a data block called the indirect block. The last entry in the addrs array stores the address of the indirect block.



The kernel keeps the set of active inodes in memory. struct inode is the in-memory copy of a struct dinode on disk. The kernel stores an inode in memory only if there are C pointers referring to that inode. The ref field counts the number of C pointers referring to the in-memory inode, and the kernel discards the inode from memory if the reference count drops to zero.

```
// in−memory copy of an inode
struct inode {
        uint dev; // Device number
        uint inum; // Inode number
        int ref; // Reference count
        struct sleeplock lock; // protects everything below here
        int valid; // inode has been read from disk?

        short type; // copy of disk inode
        short major;
        short minor;
        short nlink;
        uint size;
        uint addrs[NDIRECT+1];
};
```

A directory is implemented internally much like a file. Its inode has type T_DIR and its data is a sequence of directory entries. Each entry is a struct dirent (*fs.h*), which contains a name and an inode number. The name is at most DIRSIZ characters; if shorter, it is terminated by a NULL byte. Directory entries with inode number zero are free. The function dirlookup (*fs.c*) searches a directory for an entry with the given name. If it finds one, it returns a pointer to the corresponding inode.

```
struct dirent {
        ushort inum;
        char name[DIRSIZ];
};
```

In this assignment you will extend xv6's file system to support a simplified *pseudo file-system* - procfs (http://en.wikipedia.org/wiki/Procfs). A nice aspect of the Unix interface is that most resources in Unix are represented as special files, including devices such as the console, pipes, and of course, real files. Procfs is a pseudo file-system that the kernel creates in memory (i.e., it does not actually exist on the disk) and usually mounts it to the directory /proc. Procfs lets the kernel report information about processes and other system information to processes in a hierarchical structure (using file management system calls).

- Before implementing this assignment you must read chapter 6 of the xv6 documentation (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev10.pdf).

A possible way to implement procfs is as a (virtual) device (like console). Originally, each device in xv6 supports only read/write operations. Since we want to implement a pseudo file-system (hierarchical structure), we extended the current implementation (in the assignment's repository) with all needed interfaces (`struct devsw, file.h`).

We mounted the procfs to "/proc" directory by creating an on-disk inode. The type of this inode is "device" (`T_DEV, stat.h`) and major (device number) = PROCFS (`file.h`), while the minor value can be used internally by the driver. This mounting is taking place at the first user process (`init.c`). The result of this mounting is that every time the kernel tries to read/write from "/proc" file it uses a function provided by the device (`procfs.c`) instead of performing operations on the disk.

Your assignment is to implement an interface provided by struct devsw (`file.h`) in order to create procfs. In order to make your life easier, we provided a procfs.c source file that describes the PROCFS device.

- ***int procfsisdir(struct inode *ip)***
  The function returns zero if the file represented by ip is not a directory and a non-zero value otherwise
- ***void procfsiread(struct inode* dp, struct inode *ip)***
  The function receives ip (with initialized ip->inum) and initialized dp that represents ip's parent directory. This function can update all ip fields. Note that if ip->valid is not set, the inode will be read from the disk (since all files in procfs are "virtual", they will not reside on the disk).
- ***int procfsread(struct inode *ip, char *dst, int off, int n)***
- ***int procfswrite(struct inode *ip, char *buf, int n)***
  These functions must implement read/write operations from the file represented by ip.

The function ***procfsinit*** is called from main.c by the first CPU. It is responsible for adding the functions into the "driver" table on device number 2. The first process to run (init) is responsible for creating the inode for the device file *proc* with major = 2.

Under the /proc directory, there should be a numerical subdirectory for each running process. The subdirectory's name is the corresponding process ID. Each such subdirectory should contain the following pseudo-files and directories:

/proc/ideinfo

A file containing information about currently waiting ide operations such as:

Waiting operations: <Number of waiting operations starting from idequeue>
Read waiting operations: <Number of read operations>
Write waiting operations: <Number of write operations>
Working blocks: <List (#device,#block) that are currently in the queue separated by the ';' symbol>

/proc/filestat

The content of the file should be the following information on the file descriptor table:

Free fds: <free fd number (ref = 0)>
Unique inode fds: <Number of different inodes open by all the fds>
Writeable fds: <Writable fd number>
Readable fds: <Readable fd number>
Refs per fds: <ratio of total number of refs / number of used fds>

/proc/inodeinfo

A directory containing a list of inodes in use. The file names should be the index in the open inode table and should only show indexes that are currently in use. Each file should contain the following information:

Device: <device the inode belongs to>
Inode number: <inode number in the device>
is valid: <0 for no, 1 for yes>
type: <DIR, FILE or DEV>
major minor: <(major number, minor number)>
hard links: <number of hardlinks>
blocks used: <number of blocks used in the file, 0 for DEV files>

/proc/PID/name

This file should be a link to the name of the process.

/proc/PID/status

The content of this file must include the process' run state and its memory usage (proc->sz).

Sanity check:

Write a user program called "*lsnd*" that will pull the following information for each inode in use:
<#device> <#inode> <is valid> <type> <(major,minor)> <hard links> <blocks used>

- You must implement the procfs as a read-only file system.
- Since the OS has a dynamic nature (processes and files may be "created" and "destroyed"), procfs may present partially incorrect information. In the current assignment you are not required to deal with these situations (and they won't be checked).
- As usual, for this assignment you should clone the xv6 code from: https://www.cs.bgu.ac.il/~os192/git/Assignment4 **solutions not using code from this repository will be considered invalid.**

Submission Guidelines

Make sure that your Makefile is properly updated and that your code compiles with no warnings whatsoever. We strongly recommend documenting your code changes with comments – these are often handy when discussing your code with the graders.

Due to our constrained resources, assignments are only allowed in pairs. Please note this important point and try to match up with a partner as soon as possible.

Before creating the patch review the change list and make sure it contains all the changes that you applied and noting more. Modified files are automatically detected by git but new files must be added explicitly with the 'git add' command:

```
> git add . –Av; git commit –m "commit message"
```

At this point you may examine the differences (the patch):

```
> git diff origin
```

Once you are ready to create a patch simply make sure the output is redirected to the patch file:

```
> git diff origin > ID1_ID2.patch
```

- Tip: Although grades will only apply your latest patch, the submission system supports multiple uploads. Use this feature often and make sure you upload patches of your current work even if you haven't completed the assignment.

Finally, you should note that the graders are instructed to examine your code on lab computers only!

We advise you to test your code on lab computers prior to submission, and in addition after submission to download your assignment, apply the patch, compile it, and make sure everything runs and works.