## *Lab 5: File system, Spawn and Shell*

### *Introduction*

*In this lab, you will implement spawn, a library call that loads and runs on-disk executables. You will then flesh out your kernel and library operating system enough to run a shell on the console. These features need a file system, and this lab introduces a simple read/write file system.*

### *File system preliminaries*

*The file system you will work with is much simpler than most "real" file systems including that of xv6 UNIX, but it is powerful enough to provide the basic features: creating, reading, writing, and deleting files organized in a hierarchical directory structure.*

*We are (for the moment anyway) developing only a single-user operating system, which provides protection sufficient to catch bugs but not to protect multiple mutually suspicious users from each other. Our file system therefore does not support the UNIX notions of file ownership or permissions. Our file system also currently does not support hard links, symbolic links, time stamps, or special device files like most UNIX file systems do.*

*Sectors and Blocks*

*Most disks cannot perform reads and writes at byte granularity and instead perform reads and writes in units of sectors. In JOS, sectors are 512 bytes each. File systems actually allocate and use disk storage in units of blocks. Be wary of the distinction between the two terms: sector size is a property of the disk hardware, whereas block size is an aspect of the operating system using the disk. A file system's block size must be a multiple of the sector size of the underlying disk.*

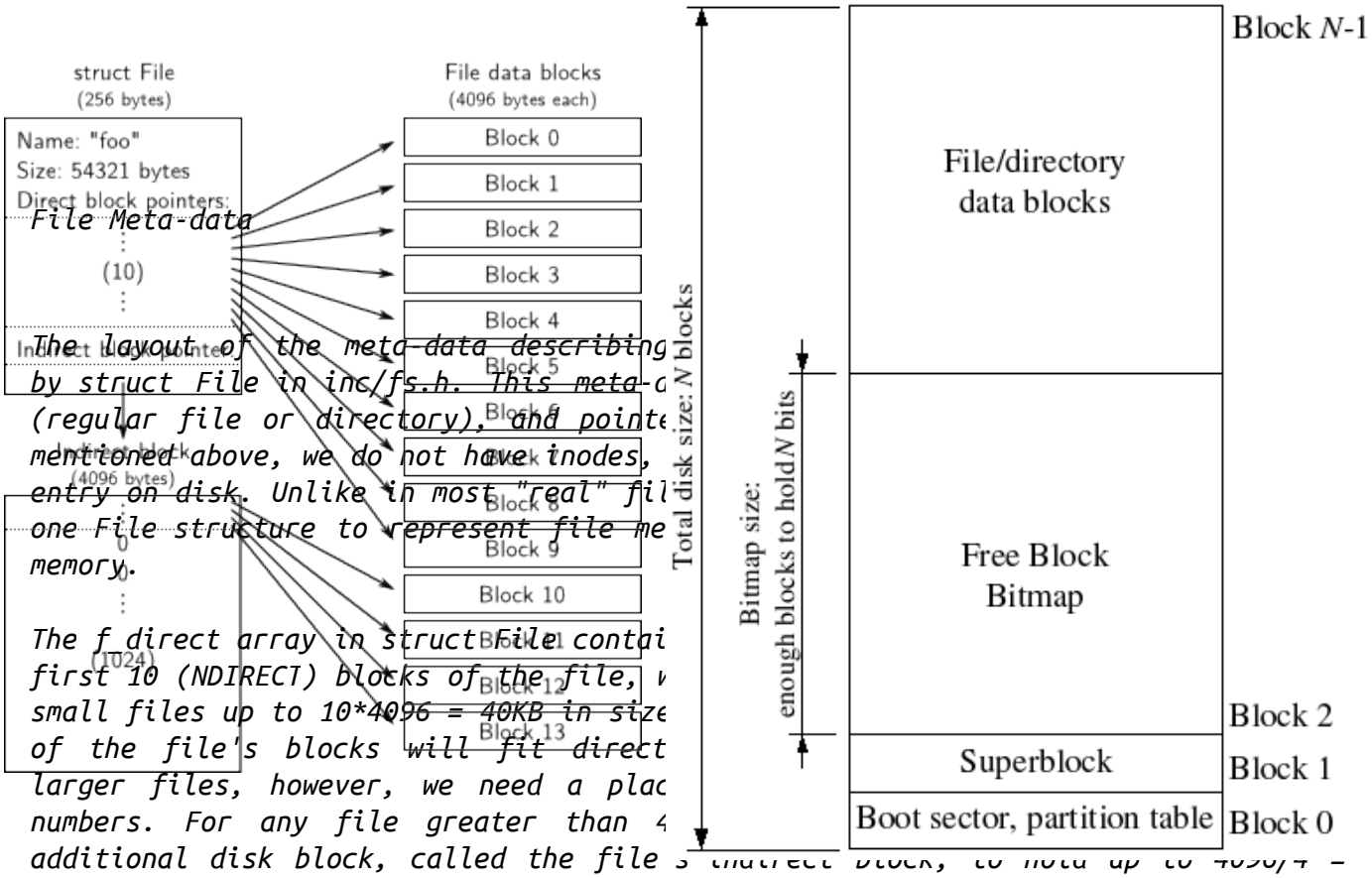
*The UNIX xv6 file system uses a block size of 512 bytes, the same as the sector size of the underlying disk. Most modern file systems use a larger block size, however, because storage space has gotten much cheaper and it is more efficient to manage storage at larger granularities. Our file system will use a block size of 4096 bytes, conveniently matching the processor's page size.*

### *Superblocks*

*File systems typically reserve certain disk blocks at "easy-to-find" locations on the disk (such as the very start or the very end) to hold meta-data describing properties of the file system as a whole, such as the block size, disk size, any meta-data required to find the root directory, the time the file system was last mounted, the time the file system was last checked for errors, and so on. These special blocks are called superblocks.*

*Our file system will have exactly one superblock, which will always be at block 1 on the disk. Its layout is defined by struct Super in inc/fs.h. Block 0 is*

*typically reserved to hold boot loaders and partition tables, so file systems generally do not use the very first disk block. Many "real" file systems maintain multiple superblocks, replicated throughout several widely-spaced regions of the disk, so that if one of them is corrupted or the disk develops a media error in that region, the other superblocks can still be found and used to access the file system.*




*File Meta-data*

*The layout of the meta-data describing by struct File in inc/fs.h. This meta-d (regular file or directory), and pointe mentioned above, we do not have inodes, entry on disk. Unlike in most "real" fil one File structure to represent file me memory.*

*The f direct array in struct File contai first 10 (NDIRECT) blocks of the file, small files up to 10*4096 = 40KB in size of the file's blocks will fit direct larger files, however, we need a plac numbers. For any file greater than 4 additional disk block, called the file's indirect block, to hold up to 4096/4 = 1024 additional block numbers. Our file system therefore allows files to be up to 1034 blocks, or just over four megabytes, in size. To support larger files, "real" file systems typically support double- and triple-indirect blocks as well.*

*Directories versus Regular Files*

*A File structure in our file system can represent either a regular file or a directory; these two types of "files" are distinguished by the type field in the File structure. The file system manages regular files and directory-files in exactly the same way, except that it does not interpret the contents of the data blocks associated with regular files at all, whereas the file system interprets the contents of a directory-file as a series of File structures describing the files and subdirectories within the directory.*

*The superblock in our file system contains a File structure (the root field in struct Super) that holds the meta-data for the file system's root directory. The contents of this directory-file is a sequence of File structures describing the files and directories located within the root directory of the file system. Any subdirectories in the root directory may in turn contain more File structures representing sub-subdirectories, and so on.*

***The File System***

***Disk Access***

*The file system environment in our operating system needs to be able to access the disk, but we have not yet implemented any disk access functionality in our kernel. Instead of taking the conventional "monolithic" operating system strategy of adding an IDE disk driver to the kernel along with the necessary system calls to allow the file system to access it, we instead implement the IDE disk driver as part of the user-level file system environment. We will still need to modify the kernel slightly, in order to set things up so that the file system environment has the privileges it needs to implement disk access itself.*

*It is easy to implement disk access in user space this way as long as we rely on polling, "programmed I/O" (PIO)-based disk access and do not use disk interrupts. It is possible to implement interrupt-driven device drivers in user mode as well (the L3 and L4 kernels do this, for example), but it is more difficult since the kernel must field device interrupts and dispatch them to the correct user-mode environment.*

*The x86 processor uses the IOPL bits in the EFLAGS register to determine whether protected-mode code is allowed to perform special device I/O instructions such as the IN and OUT instructions. Since all of the IDE disk registers we need to access are located in the x86's I/O space rather than being memory-mapped, giving "I/O privilege" to the file system environment is the only thing we need to do in order to allow the file system to access these registers. In effect, the IOPL bits in the EFLAGS register provides the kernel with a simple "all-or-nothing" method of controlling whether user-mode code can access I/O space. In our case, we want the file system environment to be able to access I/O space, but we do not want any other environments to be able to access I/O space at all.*

*Exercise 1. i386_init identifies the file system environment by passing the type ENV_TYPE_FS to your environment creation function, env_create. Modify env_create in env.c, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment.*

*Make sure you can start the file environment without causing a General Protection fault. You should pass the "fs i/o" test in make grade.*

```
/*
      Check the type of the environment which is to be created, if the environment
is of the type ENV_TYPE_FS i.e. file system’s environment, we provide it with  IO
Privileges. The FL_IOPL_ 3 macro is used since file systems is actually a user
environment which has to be granted IO privileges so that it can access disk
registers in processoor’s IO space to perform file IO on behalf of other user-mode
environments.
*/
if (type == ENV_TYPE_FS)
```

```
{
        new_env->env_tf.tf_eflags |= FL_IOPL_3;
}
```

*Question*

1. *Do you have to do anything else to ensure that this I/O privilege setting is saved and restored properly when you subsequently switch from one environment to another? Why?*

*No, this is because the I/O privilege setting is part of the eflags register, which is saved separately for each environment. So if, for example, we switch from an i/o privileged environment to a non i/o privileged one, the eflags register will get reloaded and the new, non-privileged environment won't be able to access the i/o devices.*

*Note that the GNUmakefile file in this lab sets up QEMU to use the file obj/kern/kernel.img as the image for disk 0 (typically "Drive C" under DOS/Windows) as before, and to use the (new) file obj/fs/fs.img as the image for disk 1 ("Drive D"). In this lab our file system should only ever touch disk 1; disk 0 is used only to boot the kernel.*

*The Block Cache*

*In our file system, we will implement a simple "buffer cache" (really just a block cache) with the help of the processor's virtual memory system. The code for the block cache is in fs/bc.c.*

*Our file system will be limited to handling disks of size 3GB or less. We reserve a large, fixed 3GB region of the file system environment's address space, from 0x10000000 (DISKMAP) up to 0xD0000000 (DISKMAP+DISKMAX), as a "memory mapped" version of the disk. For example, disk block 0 is mapped at virtual address 0x10000000, disk block 1 is mapped at virtual address 0x10001000, and so on. The diskaddr function in fs/bc.c implements this translation from disk block numbers to virtual addresses (along with some sanity checking).*

*Since our file system environment has its own virtual address space independent of the virtual address spaces of all other environments in the system, and the only thing the file system environment needs to do is to implement file access, it is reasonable to reserve most of the file system environment's address space in this way. It would be awkward for a real file system implementation on a 32-bit machine to do this since modern disks are larger than 3GB. Such a buffer cache management approach may still be reasonable on a machine with a 64-bit address space.*

*Of course, it would take a long time to read the entire disk into memory, so instead we'll implement a form of demand paging, wherein we only allocate pages in the disk map region and read the corresponding block from the disk in response to*

*a page fault in this region. This way, we can pretend that the entire disk is in memory.*

*Exercise 2. Implement the bc_pgfault and flush_block functions in fs/bc.c. bc_pgfault is a page fault handler, just like the one your wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) addr may not be aligned to a block boundary and (2) ide_read operates in sectors, not blocks.*

*The flush_block function should write a block out to disk if necessary. flush_block shouldn't do anything if the block isn't even in the block cache (that is, the page isn't mapped) or if it's not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the PTE_D "dirty" bit is set in the uvpt entry. (The PTE_D bit is set by the processor in response to a write to that page; see 5.2.4.3 in chapter 5 of the 386 reference manual.) After writing the block to disk, flush_block should clear the PTE_D bit using sys_page_map.*

*Use make grade to test your code. Your code should pass "check_bc", "check_super", and "check_bitmap".*

```
 // Allocate a page in the disk map region, read the contents
        // of the block from the disk into that page.
        // Hint: first round addr to page boundary. fs/ide.c has code to read
        // the disk.
/*
     Utilizing the hints we initially allocate a page through the system call of sys_page_alloc and map it to the faulting address. We then proceed to read the corresponding block from the disk into memory through ide_read function. The parameters of ide_read function are in accordance to the fact the ide_read operates in sectors, not bytes.
*/
// LAB 5: you code here:
        static_assert(PGSIZE == BLKSIZE);
        if ((r = sys_page_alloc(thisenv -> env_id, ROUNDDOWN (addr, PGSIZE), PTE_W | PTE_P | PTE_U)) < 0)
        {
                panic ("sys_page_alloc failed in file system. %e", r);
        }
```

```
         if ((r = ide_read((BLKSIZE/SECTSIZE) * blockno, ROUNDDOWN (addr, PGSIZE),
(BLKSIZE/SECTSIZE))) < 0)
        {
                panic ("IDE Read Failed. %e \n", r);
        }


// Flush the contents of the block containing VA out to disk if
// necessary, then clear the PTE_D bit using sys_page_map.
// If the block is not in the block cache or is not dirty, does
// nothing.
// Hint: Use va_is_mapped, va_is_dirty, and ide_write.
// Hint: Use the PTE_SYSCALL constant when calling sys_page_map.
// Hint: Don't forget to round addr down.
/*
Initially, we check if the dirty bit for the page containing the addr has been set
which means it has been modified since it was last written to the disk, along with
checking if the page is actually mapped. If either is the case, we simply return.
Else, we call the ide_write function to write the page onto the disk at its
appropriate place i.e. sector.
In the aftermath,  since we have to clear the PTE_D bit, we call the sys_page_map
function.
*/
        void
flush_block(void *addr)
{
        uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;

        if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
                panic("flush_block of bad va %08x", addr);

        // LAB 5: Your code here.
        static_assert(PGSIZE == BLKSIZE);
        int a  = 0;

        if (!va_is_mapped (addr) || !va_is_dirty (addr))
```

```
                return;


         if ((a = ide_write ((BLKSIZE/SECTSIZE)*blockno, ROUNDDOWN (addr, PGSIZE),
(BLKSIZE/SECTSIZE))) < 0)
        {
                panic ("IDE_WRITE FAILED. %e \n", a);
        }


          if ((a = sys_page_map(0, ROUNDDOWN (addr, PGSIZE), 0, ROUNDDOWN (addr,
PGSIZE), uvpt[PGNUM(addr)]&PTE_SYSCALL)) < 0)
        {
                   panic ("Failed to clear dirty bit in file system envirnment %e
\n", a);
        }


        //panic("flush_block not implemented");
}
```

*The fs_init function in fs/fs.c is a prime example of how to use the block cache. After initializing the block cache, it simply stores pointers into the disk map region in the super global variable. After this point, we can simply read from the super structure as if they were in memory and our page fault handler will read them from disk as necessary.*

*The Block Bitmap*

*After fs_init sets the bitmap pointer, we can treat bitmap as a packed array of bits, one for each block on the disk. See, for example, block_is_free, which simply checks whether a given block is marked free in the bitmap.*

*Exercise 3. Use free_block as a model to implement alloc_block in fs/fs.c, which should find a free disk block in the bitmap, mark it used, and return the number of that block. When you allocate a block, you should immediately flush the changed bitmap block to disk with flush_block, to help file system consistency.*

*Use make grade to test your code. Your code should now pass "alloc_block".*

```
// Search the bitmap for a free block and allocate it.  When you
```

*// allocate a block, immediately flush the changed bitmap block*

*// to disk.*

*//*

*// Return block number allocated on success,*

*// -E_NO_DISK if we are out of blocks.*

*//*

*// Hint: use free_block as an example for manipulating the bitmap.*

*/*

*bitmap has been defined in fs/fsformat.c to be an integer pointer. As such, after its initialization, it will be treated as an integer array each bit of which will indicate whether a block in disk is free or occupied. 1 means that a block is free and 0 means its occupied.*

*Assuming that the disk size is 3 GB which is equivalent to 3221225472 bytes and that each block is 4096 bytes in size, we get 786432 total blocks on disk. Now, a single block of bitmap of 4096 bytes will house information of 4096*8 = 32768 blocks. As such, a total of 786432 / 32768 blocks are required just for bitmaps i.e. 24 blocks.*

*Now, since bitmap is to treated as an array, each element of that array will house information of 32 blocks. As such, dividing total number of blocks on the disk by 32 will give the size of the bitmap array. This is exactly what is being done in step 1 of the function.*

*The DIVE_CEIL is a macro defined in inc/types.h which will return the ceiling of x being divided by y.*

*That length is then being used in the following for loop to determine a particular element which is not 0. An element of bitmap array will be zero if all the 32 blocks that the element represents are occupied. We will break from the for loop when first such element is encountered. Else, if all the elements of the bitmap array are traversed, then we return with not enough memory error.*

*Now to determine the exact bit in the particular element of the bitmap array which is 0, we use the second for loop, in which we test each bit of the element for having the value of 1, breaking from it the moment we encounter first such bit which has a value of 1.*

*Then we infer the block number that the particular bit represents by multiplying the array element containing the block by 32 and adding the bit number to the result.*

*We then, set the value of the particular bit to be 0 without disturbing the other bits.*

*Finally, we use the flush_block function to write that block on to the disk.*

*The condition is the for loop makes sure that we only write the blocks containing the bitmaps back to the disk. The*

*bmb = DIV_CEIL(super->s_nblocks, BLKBITSIZE)*

*will make sure that bmb has the value of the number of blocks occupied by the bitmaps which should be 24 for 3GB disk.*

*The starting address of each block containing the bitmaps needs to be passed to the flush_block function.*

*Since bitmap is actually an array, calculate the number of that array elements that can be stored in a single block by the expression (BLKSIZE / 32). Multiplying it by k will give us the number of elements of bitmap array being moved on to the disk.*

*For eg. BLKSIZE is 4096 bytes. Dividing it by 32 (since each element of bitmap array is 4 * 8 bits) we get 128 i.e. 128 elements of the bitmap array are moved to the disk during first iteration. The next 128 will be moved during the next iteration and so on until all blocks have been flushed on to the disk.*

**/*

```
        int
alloc_block(void)
{
        // The bitmap consists of one or more blocks.  A single bitmap block
        // contains the in-use bits for BLKBITSIZE blocks.  There are
        // super->s_nblocks blocks in the disk altogether.

        // LAB 5: Your code here.

        int bitmaplim = DIV_CEIL(super->s_nblocks, 32);

        int i;
        for (i = 0; i < bitmaplim; i++) {
                if (bitmap[i] != 0) {
                        break;
                }
        }
        if (i == bitmaplim) {
                return -E_NO_DISK;
        }
```

```
        // now find which bit is set
        uint32_t j;
        for (j = 0; j < 32; j++) {
                if (bitmap[i] & (1 << j)) {
                        break;
                }
        }
        assert(j < 32); // at least one bit must be 1
        int blockno = 32 * i + j;
        if (blockno >= super->s_nblocks) {
                return -E_NO_DISK;
        }

        // clear the chosen bit
        bitmap[i] &= ~(1 << j);

         for (int k = 0, bmb = DIV_CEIL(super->s_nblocks, BLKBITSIZE); k < bmb; k++)
        {
                uint32_t *bmaddr = &bitmap[k * (BLKSIZE / 32)];

                flush_block (bmaddr);

        }
        //     panic("alloc_block not implemented");
        return blockno;
}
```

*File Operations*

*We have provided a variety of functions in fs/fs.c to implement the basic facilities you will need to interpret and manage File structures, scan and manage the entries of directory-files, and walk the file system from the root to resolve an absolute pathname. Read through all of the code in fs/fs.c and make sure you understand what each function does before proceeding.*

*Exercise 4. Implement file_block_walk and file_get_block. file_block_walk maps from a block offset within a file to the pointer for that block in the struct*

File or the indirect block, very much like what pgdir_walk did for page tables. file_get_block goes one step further and maps to the actual disk block, allocating a new one if necessary.

Use make grade to test your code. Your code should pass "file_open", "file_get_block", and "file_flush/file_truncated/file rewrite", and "testfile".

```
// Find the disk block number slot for the 'filebno'th block in file 'f'.

// Set '*ppdiskbno' to point to that slot.

// The slot will be one of the f->f_direct[] entries,

// or an entry in the indirect block.

// When 'alloc' is set, this function will allocate an indirect block

// if necessary.

//

// Returns:

//   0 on success (but note that *ppdiskbno might equal 0).

//   -E_NOT_FOUND if the function needed to allocate an indirect block, but

//        alloc was 0.

//   -E_NO_DISK if there's no space on the disk for an indirect block.

//   -E_INVAL if filebno is out of range (it's >= NDIRECT + NINDIRECT).

//

// Analogy: This is like pgdir_walk for files.

// Hint: Don't forget to clear any block you allocate.
```

/*

As asked initially, we check if the filebno i.e. file’s block no number provided is not greater than NDIRECT + NINDIRECT i.e. 1034. If it is, we return with error code. If it is not greater, we check if  filebno is less than NDIRECT i.e. less than 10. If it is, we straight away set the point ppdiskbno to point towards the appropriate block and return.

AN IMPORTANT THING TI NOTE HERE IS THAT IN THE STRUCT FILE, THE F_DIRECT ARRAY WILL HOUSE UPTO 10 POINTERS WHICH POINT TOWARDS THE BLOCKS CONTAINING THE FILE DATA. THE F_INDIRECT BLOCK HOWEVER, WILL CONTAIN THE BLOCK NUMBER OF THE DISK WHICH CONTAINS THE BLOCK NUMBER OF THE DISK WHICH IS THE INDIRECT BLOCK OF THE FILE. THAT BLOCK THEN WILL HOLD POINTERS TO THE ADDITIONAL BLOCKS OF FILE. THE ADDRESS OF F_INDIRECT BLOCK, THEREFORE HAS TO BE OBTAINED THROUGH THE DISKADDR METHOD IN FS/BC.C.

In the event of f_indirect fo the struct File* f being 0, and alloc being false, we return with error. In cacse of alloc being set, we allocate an additional block on disk for the f_indirect for the struct File* f and set it to 0. Now since the alloc_block method will return a dsik block number, we calculate it is address

*through the diskaddr function while providing it to memset.*

*Finally, we get a pointer to the address of f_indirect block through diskaddr method and then set ppdiskbno to the appropriate block with the f_indirect block of the file. We have subtracted NDIRECT from filebno before returning since filenbno will be the block number within the file considering the NDIRECT blocks of the file.*

**/*

```
        static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
{
        // LAB 5: Your code here.

        if (filebno >= NDIRECT+NINDIRECT)
                return -E_INVAL;

        if (filebno < NDIRECT)
        {
                *ppdiskbno = f->f_direct + filebno;
                return 0;
        }

        if (f->f_indirect == 0 && !alloc)
        {
                return -E_NOT_FOUND;
        } else if (f->f_indirect == 0 && alloc)
        {
                int blocknum = alloc_block ();
                if (blocknum < 0)
                        return -E_NO_DISK;
                f->f_indirect = blocknum;
                memset (diskaddr(f->f_indirect), 0, BLKSIZE);
                flush_block (diskaddr(f->f_indirect));
        }
```

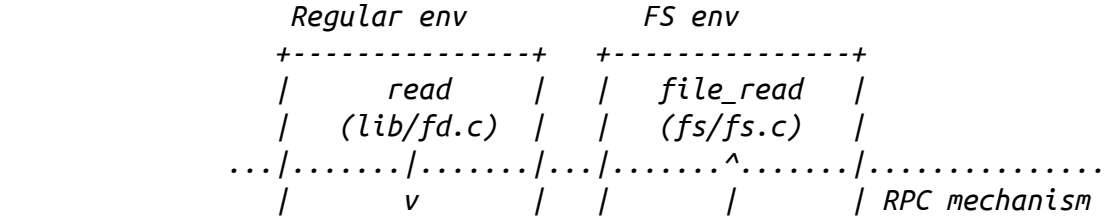
```
        uint32_t *blk = (uint32_t *) diskaddr(f->f_indirect);
        *ppdiskbno = &blk[filebno - NDIRECT];
        return 0;
        //        panic("file_block_walk not implemented");
}


// Set *blk to the address in memory where the filebno'th
// block of file 'f' would be mapped.
//
// Returns 0 on success, < 0 on error.  Errors are:
//   -E_NO_DISK if a block needed to be allocated but the disk is full.
//   -E_INVAL if filebno is out of range.
//
// Hint: Use file_block_walk and alloc_block.
/*
As asked initially, we check if the filebno i.e. file’s block no number provided
is not greater than NDIRECT + NINDIRECT i.e. 1034. If it is, we return with error
code. We then call the file_block_walk method to obtain a pointer to the filebno
block within the file. If the pointer is still 0 which is quite possible, we then
allocate an extra block for the file through alloc_block method and make *ptr hold
the value of the block. FINally, we obtain the virtial address of the block number
held by pointer ptr and set *blk to point towards it.
*/
        int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
        // LAB 5: Your code here.
        if (filebno >= NDIRECT + NINDIRECT)
                return -E_INVAL;
        uint32_t* ptr = NULL;
        int a;

        if ((a = file_block_walk (f, filebno, &ptr, true)) < 0)
                return a;
```

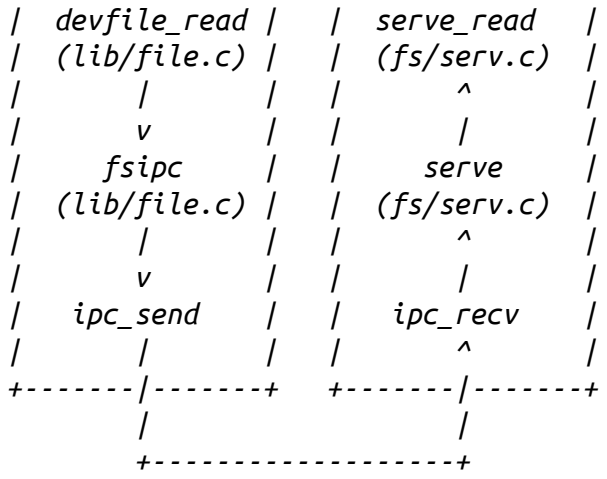
```
        if (*ptr == 0)
        {
                int blocknum = alloc_block ();
                if (blocknum < 0)
                        return -E_NO_DISK;

                *ptr = blocknum;
        }

        *blk = (char*) diskaddr (*ptr);
        return 0;

        //panic("file_get_block not implemented");
}
```

*file_block_walk and file_get_block are the workhorses of the file system. For example, file_read and file_write are little more than the bookkeeping atop file_get_block necessary to copy bytes between scattered blocks and a sequential buffer.*

*The file system interface*

*Now that we have the necessary functionality within the file system environment itself, we must make it accessible to other environments that wish to use the file system. Since other environments can't directly call functions in the file system environment, we'll expose access to the file system environment via a remote procedure call, or RPC, abstraction, built atop JOS's IPC mechanism. Graphically, here's what a call to the file system server (say, read) looks like*






*Everything below the dotted line is simply the mechanics of getting a read request from the regular environment to the file system environment. Starting at the beginning, read (which we provide) works on any file descriptor and simply dispatches to the appropriate device read function, in this case devfile_read (we can have more device types, like pipes). devfile_read implements read specifically for on-disk files. This and the other devfile_* functions in lib/file.c implement the client side of the FS operations and all work in roughly the same way, bundling up arguments in a request structure, calling fsipc to send the IPC request, and unpacking and returning the results. The fsipc function simply handles the common details of sending a request to the server and receiving the reply.*

*The file system server code can be found in fs/serv.c. It loops in the serve function, endlessly receiving a request over IPC, dispatching that request to the appropriate handler function, and sending the result back via IPC. In the read example, serve will dispatch to serve_read, which will take care of the IPC details specific to read requests such as unpacking the request structure and finally call file_read to actually perform the file read.*

*Recall that JOS's IPC mechanism lets an environment send a single 32-bit number and, optionally, share a page. To send a request from the client to the server, we use the 32-bit number for the request type (the file system server RPCs are numbered, just like how*

*syscalls were numbered) and store the arguments to the request in a union Fsipc on the page shared via the IPC. On the client side, we always share the page at fsipcbuf; on the server side, we map the incoming request page at fsreq (0x0ffff000).*

*The server also sends the response back via IPC. We use the 32-bit number for the function's return code. For most RPCs, this is all they return. FSREQ_READ and FSREQ_STAT also return data, which they simply write to the page that the client sent its request on. There's no need to send this page in the response IPC, since the client shared it with the file system server in the first*

*place. Also, in its response, FSREQ_OPEN shares with the client a new "Fd page". We'll return to the file descriptor page shortly.*

*Exercise 5. Implement serve_read in fs/serv.c.*

*serve_read's heavy lifting will be done by the already-implemented file_read in fs/fs.c (which, in turn, is just a bunch of calls to file_get_block). serve_read just has to provide the RPC interface for file reading. Look at the comments and code in serve_set_size to get a general idea of how the server functions should be structured.*

*Use make grade to test your code. Your code should pass "serve_open/file_stat/file_close" and "file_read" for a score of 70/150.*

*// Read at most ipc->read.req_n bytes from the current seek position*

*// in ipc->read.req_fileid. Return the bytes read from the file to*

*// the caller in ipc->readRet, then update the seek position. Returns*

*// the number of bytes successfully read, or < 0 on error.*

*/*

*As is stated in the serve_set size function, we initially use the openfile_lookup function to locate the file from which read operation is to be performed. openfile_lookup function will search for the relavant file whose file id has been passed to it as a parameter and return the corresponding file's struct Openfile which basically integrated struct File and struct Fd for the file. The struct Openfile is kept private to file system environment and houses data of all open files. Details of how file operations are actually perfromed follow shortly. Getting the file's vital information i.e. struct Openfile, we then call the file_read function. The file_read function will read the number of bytes, asked by the user environment, from the file which are passed to the File System through struct Fsreq_read in union Fsipc which is written to a page and shared with the FS through IPC along with RPC numbers defined in inc/fs.h. The data will be read from the file from the offset provided as the fourth parameter which is contained in struct Fd i.e. File Descriptor struct of the file. The bytes read will then be written to the buffer whose address is passed to the file_read function as second argument i.e. the starting address of buffer in struct Fsret_read which will be returned to the user environment in the IPC. The bytes are read from the file through the file_get_block method, transfered to the buffer and the offset of the file moved ahead by bytes read. With file_reaad methdo returning the number of bytes actually read from the file, we use the very return value to update the file offset in struct Fd and return the number of bytes read from the file.*

**/*

```
        int
serve_read(envid_t envid, union Fsipc *ipc)
{
```

```
        struct Fsreq_read *req = &ipc->read;
        struct Fsret_read *ret = &ipc->readRet;


        if (debug)
                cprintf("serve_read %08x %08x %08x\n", envid, req->req_fileid,
req->req_n);
        // Lab 5: Your code here:
        struct OpenFile *o;


        if ((int sa = openfile_lookup (envid, req -> req_fileid, &o)) < 0)
                return a;
        int bytes_read = 0;
           if (( bytes_read = file_read (o->o_file, ret->ret_buf, req->req_n, o-
>o_fd->fd_offset)) < 0)
                return bytes_read;
        o->o_fd->fd_offset += bytes_read;
        return bytes_read;
}


Exercise 6. Implement serve_write in fs/serv.c and devfile_write in lib/file.c.
Use make grade to test your code. Your code should pass "file_write", "file_read
after file_write", "open", and "large file" for a score of 90/150.
In file fs/serv.c
// Write req->req_n bytes from req->req_buf to req_fileid, starting at
// the current seek position, and update the seek position
// accordingly.  Extend the file if necessary.  Returns the number of
// bytes written, or < 0 on error.
/*
Similar to the serve_read function only the responsibilities of functions change.
*/
        int
serve_write(envid_t envid, struct Fsreq_write *req)
{
```

```
        if (debug)
                cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid,
req->req_n);
        // LAB 5: Your code here.
        struct OpenFile* o;
        int a ;
        if ((a = openfile_lookup(envid, req->req_fileid, &o)) < 0)
                return a;
        int bytes_written = 0;
          if ((bytes_written = file_write (o->o_file, req->req_buf, req->req_n, o-
>o_fd->fd_offset)) < 0)
                return bytes_written;

        o->o_fd->fd_offset += bytes_written;
        return bytes_written;
        //        panic("serve_write not implemented");
}
```

In file lib/file.c

```
// Write at most 'n' bytes from 'buf' to 'fd' at the current seek position.
//
// Returns:
//    The number of bytes successfully written.
//    < 0 on error.
/*
```

The function is used to bundle together the arguments in order for fsipc function to send an IPC request to the FS for writing a specific file.

Initially, we set parameters of the struct Fsreq_write of union fsipc the page of which will be shared with the FS by the fsipc function through ipc_send. In the event the number of bytes to be written, provided as a afuntion parameters is greater than the max size of the buffer contained in the struct Fsreq_write, we do a sanity check and only write as many number of bytes as are permissible by buffer in the structure.

The bytes to written to the file which are held in a buffer by the user environment are then moved into the buffer of the struct Fsreq_write through memmove and fsipc method is called with its type parameter as FSREQ_WRITE.

```
*/
        static ssize_t
devfile_write(struct Fd *fd, const void *buf, size_t n)
{
        // Make an FSREQ_WRITE request to the file system server.  Be
        // careful: fsipcbuf.write.req_buf is only so large, but
        // remember that write is always allowed to write *fewer*
        // bytes than requested.
        // LAB 5: Your code here
        fsipcbuf.write.req_fileid = fd->fd_file.id;
        fsipcbuf.write.req_n = n;
        int bytes_written = sizeof(fsipcbuf.write.req_buf);
        memmove (fsipcbuf.write.req_buf, buf, MIN(bytes_written, n));
        int r;
        if ((r = fsipc (FSREQ_WRITE, NULL)) < 0)
                return r;
        assert (r <= n);
        assert (r <= bytes_written);
        return r;
        //panic("devfile_write not implemented");
}
```

*Spawning Processes*

*We have given you the code for spawn (see lib/spawn.c) which creates a new environment, loads a program image from the file system into it, and then starts the child environment running this program. The parent process then continues running independently of the child. The spawn function effectively acts like a fork in UNIX followed by an immediate exec in the child process.*

*We implemented spawn rather than a UNIX-style exec because spawn is easier to implement from user space in "exokernel fashion", without special help from the kernel.*

*Exercise 7. spawn relies on the new syscall sys_env_set_trapframe to initialize the state of the newly created environment. Implement sys_env_set_trapframe in kern/syscall.c (don't forget to dispatch the new system call in syscall()).*

*Test your code by running the user/spawnhello program from kern/init.c, which will attempt to spawn /hello from the file system.*

*Use make grade to test your code.*

```
// Set envid's trap frame to 'tf'.
// tf is modified to make sure that user environments always run at code
// protection level 3 (CPL 3), interrupts enabled, and IOPL of 0.
//
// Returns 0 on success, < 0 on error.  Errors are:
//   -E_BAD_ENV if environment envid doesn't currently exist,
//        or the caller doesn't have permission to change envid.
/*
```

*Simple Function. We get the struct Env corresponding to the envid through the envidtoenv method with permissions set for sanity chehcking. We then check if the environment has the right to modify the memory pointed to by pointer  tf. If it has the right, we assignment envid's trap frame to tf and later modify it as has been asked by setting the RPL for the trapframe to be 3.*

```
*/
        static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
        // LAB 5: Your code here.
        // Remember to check whether the user has supplied us with a good
        // address!
        int err;
        struct Env *env;
        if ((err = envid2env(envid, &env, true)) < 0) {
                return err;
        }
         if ((err = user_mem_check(env, tf, sizeof(struct Trapframe), PTE_U)) < 0)
{
                return err;
        }
        env->env_tf = *tf;

        // set the IOPL to 0
```

```
        env->env_tf.tf_eflags &= ~FL_IOPL_MASK;

        // enable interrupts
        env->env_tf.tf_eflags |= FL_IF;

        // set user privilage level
        env->env_tf.tf_ds = GD_UD | 3;
        env->env_tf.tf_es = GD_UD | 3;
        env->env_tf.tf_ss = GD_UD | 3;
        env->env_tf.tf_cs = GD_UT | 3;
        return 0;
        //panic("sys_env_set_trapframe not implemented");
}
```

*Sharing library state across fork and spawn*

*The UNIX file descriptors are a general notion that also encompasses pipes, console I/O, etc. In JOS, each of these device types has a corresponding struct Dev, with pointers to the functions that implement read/write/etc. for that device type. lib/fd.c implements the general UNIX-like file descriptor interface on top of this. Each struct Fd indicates its device type, and most of the functions in lib/fd.c simply dispatch operations to functions in the appropriate struct Dev.*

*lib/fd.c also maintains the file descriptor table region in each application environment's address space, starting at FDTABLE. This area reserves a page's worth (4KB) of address space for each of the up to MAXFD (currently 32) file descriptors the application can have open at once. At any given time, a particular file descriptor table page is mapped if and only if the corresponding file descriptor is in use. Each file descriptor also has an optional "data page" in the region starting at FILEDATA, which devices can use if they choose.*

*We would like to share file descriptor state across fork and spawn, but file descriptor state is kept in user-space memory. Right now, on fork, the memory will be marked copy-on-write, so the state will be duplicated rather than shared. (This means environments won't be able to seek in files they didn't open themselves and that pipes won't work across a fork.) On spawn, the memory will be left behind, not copied at all. (Effectively, the spawned environment starts with no open file descriptors.)*

*We will change fork to know that certain regions of memory are used by the "library operating system" and should always be shared. Rather than hard-code a list of regions somewhere, we will set an otherwise-unused bit in the page table entries (just like we did with the PTE_COW bit in fork).*

*We have defined a new PTE_SHARE bit in inc/lib.h. This bit is one of the three PTE bits that are marked "available for software use" in the Intel and AMD manuals. We will establish the convention that if a page table entry has this bit set, the PTE should be copied directly from parent to child in both fork and spawn. Note that this is different from marking it copy-on-write: as described in the first paragraph, we want to make sure to share updates to the page.*

*Exercise 8. Change duppage in lib/fork.c to follow the new convention. If the page table entry has the PTE_SHARE bit set, just copy the mapping directly. (You should use PTE_SYSCALL, not 0xfff, to mask out the relevant bits from the page table entry. 0xfff picks up the accessed and dirty bits as well.)*

*Likewise, implement copy_shared_pages in lib/spawn.c. It should loop through all page table entries in the current process (just like fork did), copying any page mappings that have the PTE_SHARE bit set into the child process.*

```
// Map our virtual page pn (address pn*PGSIZE) into the target envid
// at the same virtual address.  If the page is writable or copy-on-write,
// the new mapping must be created copy-on-write, and then our mapping must be
// marked copy-on-write as well.  (Exercise: Why do we need to mark ours
// copy-on-write again if it was already copy-on-write at the beginning of
// this function?)
//
// Returns: 0 on success, < 0 on error.
// It is also OK to panic on error.
//
/*
Changes to the function are minor. While we were earlier copying pages which
didn't have PTE_W and PTE_COW set in the page table entries directly into the
child, we add an aditional condition in the first if loop to also copy thse [ages
directly into the child which PTE_SHARE set. The rest of the program remains the
same. For those pages which have PTE_W set, we change it to PTE_COW and copy only
the mappings from parent to child.
*/
        static int
duppage(envid_t envid, unsigned pn)
{
        // LAB 4: Your code here.
```

```
        int r;
        pte_t pte = uvpt[pn];
        if ((!(pte & PTE_W) && !(pte & PTE_COW)) || (pte & PTE_SHARE))
        {
                    if ((r = sys_page_map(thisenv->env_id, (void *)(pn * PGSIZE),
envid, (void *)(pn * PGSIZE), pte & PTE_SYSCALL)) < 0) {
                        panic("sys_page_map: %e", r);
                }
                return 0;
        }
        // remove write bit and set copy on write
        pte &= ~PTE_W;
        pte |= PTE_COW;
         if ((r = sys_page_map(thisenv->env_id, (void *)(pn * PGSIZE), envid, (void
*)(pn * PGSIZE),pte & PTE_SYSCALL)) < 0)
        {
                panic("sys_page_map: %e", r);
        }
        // remap our page to have copy on write
           if ((r = sys_page_map(thisenv->env_id, (void *)(pn * PGSIZE), thisenv-
>env_id, (void *)(pn * PGSIZE), pte & PTE_SYSCALL)) < 0)
        {
                panic("sys_page_map: %e", r);
        }


        return 0;
}
In file lib/spawn.c
// Copy the mappings for shared pages into the child address space.
/*
spawn is basically a functionality like UNIX ‘exec’ but implemented entirely in
user space library. Spawn will basically load an executable file from disk using
the FS, then make the user environment fork a child, load the executable into the
child’s address space and get the child running the executable. In this scenario
however, we want the library part of the parent to be same for child i.e. we want
```

*all the files that were open and edited by the parent to be available to child as well, to edit if it wishes to, the result of which will be displayed in the File descriptor section of memory i.e. FDTABLE. As such, we set PTE_SHARE bit for those corresponding pages in their page table entries which means those entries will be directly copied into the child by the below function instead of being duplicated on attempt being made to write to them.*

*Traversing through the page directory of the parent, we head into the page table for each directory entry. If the present bit for the directory entry is not set we move to the next entry. In case it is set, we head into the corresponding page table and obtain the page table entry. We then check if the page table entry is above UTOP which is the kernel section. In case it is, we exit the neted for loops. If the address is not, we check if the pte has PTE_SHARE set. In case it has, we plainly copy the mapping to set into the child environment using the sys_page_map system call.*

*/*

```
        static int
copy_shared_pages(envid_t child)
{
        // LAB 5: Your code here.
        int r;
        bool is_below_ulim = true;
        for (int i = 0; is_below_ulim && i < NPDENTRIES ; i++)
        {
                if (!(uvpd[i] & PTE_P))
                        continue;
                for (int j = 0; is_below_ulim && j < NPTENTRIES; j++)
                {
                        unsigned pn = i * NPTENTRIES + j;
                        pte_t pte = uvpt[pn];
                        if (pn >= (UTOP >> PGSHIFT))
                        {
                                is_below_ulim = false;
                        } else if (pte & PTE_SHARE)
                        {
                                        if ((r = sys_page_map(0, (void *)(pn * PGSIZE),
child, (void *)(pn * PGSIZE), pte & PTE_SYSCALL)) < 0) {
                                                return r;
```

```
                                }}}}
        return 0;
}
```

*The keyboard interface*

*For the shell to work, we need a way to type at it. QEMU has been displaying output we write to the CGA display and the serial port, but so far we've only taken input while in the kernel monitor. In QEMU, input typed in the graphical window appear as input from the keyboard to JOS, while input typed to the console appear as characters on the serial port. kern/console.c already contains the keyboard and serial drivers that have been used by the kernel monitor since lab 1, but now you need to attach these to the rest of the system.*

*Exercise 9. In your kern/trap.c, call kbd_intr to handle trap IRQ_OFFSET +IRQ_KBD and serial_intr to handle trap IRQ_OFFSET +IRQ_SERIAL.*

```
/*
Simple Exercise. No explanation needed.
*/
```

*In file kern/trap.c*

```
 SETGATE (idt[IRQ_OFFSET + IRQ_KBD], false, GD_KT, IRQ_KeyboardINT, 0);
        SETGATE (idt[IRQ_OFFSET + IRQ_SERIAL], false, GD_KT, IRQ_SerialINT, 0);
```

*In function trap_dispatch*

```
// Handle keyboard and serial interrupts.
        // LAB 5: Your code here.
        if (tf->tf_trapno == IRQ_OFFSET + IRQ_KBD) {
                kbd_intr();
                return;
        }
        if (tf->tf_trapno == IRQ_OFFSET + IRQ_SERIAL) {
                serial_intr();
                return;
        }
```

*Make sure to make appropriate entries for handlers in kern/trapentry.S and kern/trap.h*

*The Shell*

*Run make run-icode or make run-icode-nox. This will run your kernel and start user/icode. icode execs init, which will set up the console as file descriptors 0 and 1 (standard input and standard output). It will then spawn sh, the shell. You should be able to run the following commands:*

```
echo hello world | cat
cat lorem |cat
cat lorem |num
cat lorem |num |num |num |num |num
lsfd
```

*Note that the user library routine cprintf prints straight to the console, without using the file descriptor code. This is great for debugging but not great for piping into other programs. To print output to a particular file descriptor (for example, 1, standard output), use fprintf(1, "...", ...). printf("...", ...) is a short-cut for printing to FD 1. See user/lsfd.c for examples.*

*Exercise 10.*

*The shell doesn't support I/O redirection. It would be nice to run sh <script instead of having to type in all the commands in the script by hand, as you did above. Add I/O redirection for < to user/sh.c.*

*Test your implementation by typing sh <script into your shell*

*Run make run-testshell to test your shell. testshell simply feeds the above commands (also found in fs/testshell.sh) into the shell and then checks that the output matches fs/testshell.key.*

```
// Open 't' for reading as file descriptor 0
                                // (which environments use as standard input).
                        // We can't open a file onto a particular descriptor,
                                // so open the file as 'fd',
                                // then check whether 'fd' is 0.
                                // If not, dup 'fd' onto file descriptor 0,
                                // then close the original 'fd'.

                                // LAB 5: Your code here.
                                //panic("< redirection not implemented");
/*
Simple Code. Just follow the instructions.
*/
```

```
if ((fd = open(t, O_RDONLY)) < 0) {
        cprintf("open %s for read: %e\n", t, fd);
        exit();
}
if (fd != 0) {
        if ((r = dup(fd, 0)) < 0) {
                cprintf("failed dup: %x\n", r);
                exit();
        }
        close(fd);
}
```

***OVERVIEW:***

*The File System Server (FS) in JOS is actually an user environment of type ENV_TYPE_FS. With the FS needing to access the disk which in our case is assumed to be a hard disk of size 3 GB or less, the FS has been given IO privileges for it to implement to disk access at the time of its creation in env_alloc function in kern/env.c. This has been implemented as an alternative to implementing IDE disk drivers in the kernel for the FS to access and modify contents of the disk.*

*The FS handles all the file system operations, including writing directly to disk. The disk itself writes and reads in units of 512 bytes called Sectors. The File System, however, uses a bigger unit which is called a Block (the size of the block must be divisible by the size of the sector). The block size was chosen to be the same size as a page i.e. 4096 bytes. The reason why the block size is the same as the page size is because we'll be using x86's paging hardware to implement a block cache.*

*The entire disk (assuming the disk is less than 3GB in size) is lazily mapped into memory starting at address DISKMAP in the FS server. Lazily mapped means that the actual blocks will be loaded into memory from disk only when we attempt to read them (which is done using the void* diskaddr(uint32_t blockno) funtion). If, for example, we want to access block 3, then we'd read the page at address DISKMAP + 3*PGSIZE (this works because the size of each page is exactly the size of each block). If our read results in a page fault (which occurs when we read a block for the first time), FS' page fault handler will load 4096 bytes from the disk into a page frame (physical page) and map it into the corresponding page in the FS server's address space. Disk block n, when in memory, is mapped into the file system server's address space at DISKMAP + (n*BLKSIZE).*

*Subsequent block reads will therefore be read from memory directly, thus functioning like a block cache. Block writes will be handled in a similar way: we'll be writing directly to the memory mapped above DISKMAP, and then using the*

*void flush_block(void *addr) function the contents will be written back to the disk.*

***File system format on disk***

*sector 0 is reserved and is not touched by the FS server. Sector 1 stores the super block that stores the meta data about the file system including: 1) magic number (so we can identify the type of the file system) 2) the number of blocks on the disk 3) the meta data of the root directory-file. Sector 2 and up stores bitmap of free blocks (where 1 means free block and 0 means occupied block). Operation on the bitmap as the name suggests will be undertaken at the bit level. Each bit of the bitmap, which is actually an array, will store if the corresponding block on disk is free or occupied. The rest of the sectors contain the other file/directory data.*

***How is file/directory data represented***

*File meta data is represted by struct File and is exactly 256 bytes (half a sector or 1/16 of a block). File meta data contains the file name, size, type (directory or regular file), array of 10 block numbers where the actual data is stored, and another block number of a block that stores another 4096/4 = 1024 block numbers.*
*An important thing to note here is that while the direct block will hold pointers to the blocks containing the file data, the indirect field of the struct File will just hold the disk block number which contains the pointers to additional blocks retaining the file data.*
*Thus, the maximum file size is 10*4096 + 1024*4096 = 4,235,264 bytes and the maximum overhead of storing the metadata is 256+4096 = 4,352 bytes.*
*The contents of each directory file is just the file metadata (struct File) of the files that this directory stores (and also a directory file must be a multiple of block size). This means that a directory can store up to 4,235,264/256 = 16,544 files.*

***Opening a File***

*In order to start reading from or writing to a file, we first need to "open" the file. Opening the file means that we obtain a file descriptor, and then later operations (such as reading and writing) are performed on that file descriptor.*
*Each File will have a struct Dev assocaited with it as follows:*

```
struct Dev {
     int dev_id;
     const char *dev_name;
     ssize_t (*dev_read)(struct Fd *fd, void *buf, size_t len);
     ssize_t (*dev_write)(struct Fd *fd, const void *buf, size_t len);
     int (*dev_close)(struct Fd *fd);
     int (*dev_stat)(struct Fd *fd, struct Stat *stat);
     int (*dev_trunc)(struct Fd *fd, off_t length);
};
```

*dev_id field identifies the type of file. In JOS we have three types of files, on disk files, having dev_id of ‘f’, pipes having dev_id of ‘p’ and console I/O into and from file having dev_id of ‘c’.*
*the dev_name element is name of the file. The rest of the elements of struct Dev are pointers to functions that will perform different operations on the files from*

*the user library side. Functions to actually perform those operations on fies exist on File System side as well.*
*The user library side of functions basically will just send IPC to the file system environment which depending on the value sen  over the IPC will perform the operation on the file and return some value, again through IPC. The operation of Open however, will also return a page through IPC for on disk files.*

*Now, File descriptor of a each file that is open in a user environment will be stored at location starting from FDTABLE i.e. 0xD0000000. Each descriptor wil lhave a page in this memory region and a user environment can at most have only 32 file descriptors i.e. 32 files open (MAXFD). Additionally, a data page is also allocated for each descriptor starting from memory address FDTABLE + MAXFD*PGSIZE.*

*Now, when a user environment want to open an on-disk file, it will use the user library function of open in lib/file.c. The function signature is:*
*int open(const char *path, int mode)*
*The function will initially call the fd_alloc function defined in lib/fd.c which will allocate an empty page in the FDTABLE region of memory of the environment i.e. basically it will return a pointer to the start of the page. It will then call the function of fsipc:*
*static int fsipc(unsigned type, void *dstva)*
*type represents the type of functions to be performed on the file and as such can be found in inc/fs.h. In the same file, we have the union Fsipc which contains various structures. Depending on the type of operation to be performed, the user environment and FS will exchange data over the IPC in format of one of these structures.*

*The fsipc function will send and IPC request to the FS using the page of union Fsipc's instance mapped at fsipcbuf as arguments to the FS to perform the Operation. The function will return when it receives a reply from the FS.*
*In case of open, the IPC request, received by the serve function in fs/serv.c will be redirected to the serve_open function by checking the value received through IPC. The value received will be returned by the ipc_recv method. The serve_open function, likewise, will open the file, creating it if that respective flag has been set or plainly opening it, or even truncating it if that flag has been set. It will then fill out the File Descriptor structure struct Fd for that on disk file and store the location in a pointer variable and return. That very page coontaining the location will then be returned by the serve function througb ipc_send.*
*The user environment, now, will map the page received through IPC from FS at the memory location returned by the fd_alloc method.*
*On successfully opening file, FS will return the very 32-bit value it received from user environment denoting operation to be performed on the file.*
*For pipes and console I/O, the struct FD will be filled in by the respective programs only and aas such no IPC will be sent.*

*Similarly, for read, write, flush, remove operations on disk files, the struct Dev for the file will be initially refereed by the user environment to head to the appropriate function. The functions will then use their corresponding structures*

*from Union Fsipc as arguments for the FS to perform the operation and call function fsipc to send the IPC. The instance of union Fsipc which will be used to provide these very arguments is again fsipcbuf.*
*On the FS server side, the serv function will reference the operation to be performed as a 32-bit number against the array fshandler handlers[] which basically contains function pointers for functions to be executed for different operations. Upon successful lexecution of fucntion which will basically be undertaken through calling additional functions in fs/fs.c and fs/bc.c (for example, read will call file_get_block() repeatedly andnwrite will call flush bock every time a complete block is written), serve function will send IPC back to user environment using the 32-bit number representing operation tot be performed as a sign of successful comopletion of operation.*

*In case of read and stat, the FS server, will write the data read from the file , which is something that will be provided by the user environment, or the stats of the file upon the very page through which the user environment provided arguments for FS to perform file operations i.e. the page of fsipcbuf. No additional page will be shared through IPC by the FS.*