Lab 7: File System & Shell

Lab 7: File System & Shell

1 File System
- 1.1 Buffer Cache layer
- 1.2 Logging Layer
- 1.3 Inode Layer
- 1.4 Directory Layer
- 1.5 Pathname Layer
- 1.6 File Descriptor Layer
- 1.7 File System Tests

2 Syscall

3 Shell
- 3.1 User-level memory
- 3.2 Syscall
- 3.3 Trapframe
- 3.4 Idle process
- 3.5 Entering Shell

1 File System

The xv6 file system is composed of seven layers: disk layer, buffer cache layer, logging layer, inode layer, directory layer, pathname layer and file descriptor layer. We have implemented the disk layer in Lab 6 (sd card device). Therefore, I will introduce the remaining layers in the following subsections.

1.1 Buffer Cache layer

The buffer cache layer maintains a copy of popular blocks to avoid frequent disk I/O. It also avoids data race in concurrent file operation using locks.

The buffer cache is a doubly-linked list of buffers. Some of the common operations on doubly-linked lists are implemented in `buf.h`, including `buf_add`, `buf_append`, `buf_del` and `buf_isempty`.

The interfaces of this layer are `bread`, `bwrite` and `brelse`. `bread` is called for getting a block buffer. All writes on this block are operated on the buffer cache until `bwrite` is called to write the buffer to the disk. Finishing all operations, users call `brelse` to release this buffer from buffer cache.

`bread` would call `bget` to get the buffer. `bget` involves two scans: the first one is to check whether the requested block is in the cache; if the cache does not hit, the second scan is to allocate a free buffer to cache this block.

The file system starts at block `0x20800` instead of `0x0`. Therefore, the superblock is located at block `0x20801`. This layer provides an abstraction to other layers as if the file system starts at block `0x0`. The `readsb` function would read block `0x1` instead of `0x20801` with this abstraction. Therefore, other layers can ignore the actual location of the file system on the disk but focus on the offset within the file system. The function `readsb` calls `bread` with parameter `blocknum = 1`.

1.2 Logging Layer

This layer performs two functions: transactionalizing and batching. To avoid inconsistency when crashing, the file system introduces transactions to keep the write operation atomic. According to locality principle, writes within one transaction are likely to be operated on the same blocks. Without logging, each write operation would touch the block layer and may cause I/O. Logging will group all writes upon a certain block and would call the interface of the block layer **only once** at committing.

`begin_op()` is called at the beginning of a file system call. It waits until the logging system is not committing and there is enough space for logging.

`log_write()` acts as a proxy for `bwrite()`. It records the block to be written and set the dirty flag. It also increments the reference count of this block to avoid the block to be released.

`end_op()` should be called when a file system call finishes. It tries committing the log if the `outstanding` count is zero. When committing, the log system scans the whole record to copy blocks to the log area and updates the log head. Then, it writes the blocks in log area to the proper place in the file system and finally clears the record.

1.3 Inode Layer

Inodes are maintained both on disk and in memory. On-disk inodes, `struct dinode`, are kept in a contiguous area, whose range is recorded by the superblock. In-memory inodes, `struct inode`, are copies of on-disk inodes and store more information of them.

`iget()` returns a pointer to an in-cache inode. Its ref is 1 so it won't be flushed out of the cache. However, it may contain nothing. To ensure that it holds the copy of the corresponding inode on disk, `ilock()` must be called.

`iput()` decrements the ref. If the ref becomes 0, this cache entry can be freed.

Inodes store the location of the data blocks of files. The mapping is translated by `bmap()`. This function has a parameter of the data block index which is calculated by $offset/BSIZE$. There are two thresholds: indexs below the first threshold are directly mapped and the address is stored in the inode; indexs between the two threshold are indirectly mapped and the address is stored in a data block; indexs over the second threshold are treated as invalid.

Data blocks are maintained by a bitmap. `balloc()` and `bfree()` are interfaces of this bitmap.

Data blocks owning by the inode can be written or read via `writei()` or `readi()`.

If the inode has no links, it should be freed from the disk by calling `itrunc()`.

1.4 Directory Layer

Directories are much like ordinary files. Their inode types are `T_DIR`.

They store the inums and names of the files under them by structure `dirent`. `dirlookup()` scans the data block owned by a directory to find the inode corresponding to the given file name.

`dirlink()` and `dirunlink()` adds and removes the link between a directory and file respectively.

1.5 Pathname Layer

`namex()` analyzes the path name and returns the inode if found. It has a bug: the parent of path name `/` should be `/` itself; however, `nameparent("/")` returns a zero pointer. Therefore, I add special check for file path `/` in `namex()`.

```
if (*path == '/') {
    ip = iget(ROOTDEV, ROOTINO);
    while (*path == '/') {
        path++;
    }
    if (*path == 0) {
        return ip;
    }
}
```

1.6 File Descriptor Layer

Most resources in the operating system are represented as files.

All open files are kept in a global file table `ftable`. `filealloc()` , `filedup()` and `fileclose()` can allocate, create and release a file reference respectively.

`filestat()` is the wrapper of `stati`.

`fileread()` and `filewrite()` implements the offset to allow I/O for large files. The offset will not reset until the process closes the file.

1.7 File System Tests

I add some tests in `fs_test.c`.

Before testing, one should execute `make clean` and `make all`.

Tests are added in `forkret()`. Only the process with pid 1 would execute the tests. A macro `TEST_FILE_SYSTEM` is added to provide non-invasive tests.

```
if (thiscpu->proc->pid == 1) {
#ifdef TEST_FILE_SYSTEM
    test_file_system();
#endif
}
```

It contains five tests:

`test_initial_scan()` is like command `ls` only showing the filename. Initially, files are `init`, `ls`, `mkfs`, `sh` and `cat`. If the output contains the five files, the test passes.

`test_initial_read()` reads the ELF magic number of the five files.

`test_file_write()` writes two files: `hello.cpp` and `readme.md`. It first writes the pre-defined words to the file and then reads from the file to check if the words read from the file are the same as pre-defined ones.

`test_mkdir()` tests `dirlink()` and `test_rmdir()` tests `dirunlink()`. After `test_mkdir()`, I add a `test_initial_scan()` to see if the two files and one directory are added to the root directory.

The output would be:

```
test_initial_scan start.
.
..
init
ls
sh
mkfs
mkdir
cat
test_initial_scan pass!
test_initial_read start.
test_initial_read pass!
test_file_write start.
test_file_write pass!
test_mkdir start.
test_mkdir pass!
test_initial_scan start.
.
..
init
ls
sh
mkfs
mkdir
cat
hello.cpp
readme.md
dir
test_initial_scan pass!
test_rmdir start.
test_rmdir pass!
```

2 Syscall

`musl` stores the syscall number in `r8` and the parameters in `r0` to `r7`. Therefore, the `initcode.S` and `argint()` should be modified.

Functions in `sysfile.c` are syscall wrappers for calling the interfaces of the file system.

`create()` first gets the inode of the parent directory. Then it check whether the file to be created has existed. If it does not exist, the function creates an inode for this file and links it with its parent directory. If the type of the new file is directory, `.` and `..` are linked to it.

3 Shell

To run shell, some support has to be added.

3.1 User-level memory

`copyuvm()` is called when a parent process spawns child processes. It copies the memory of virtual address `[0, sz)` to the child process and adds mapping. The address mapping of a parent process is different from child processes.

`allocuvm()` and `deallocuvm()` allocates and frees user-level memory respectively.

`loaduvm()` loads data from files to user-level memory. The read process can be divided into two parts: reading the first page the reading the remaining. Data to be read from the first page may not start from the beginning of the page.

`copyout()` loads data from kernel memory to user-level memory. The procedure is much like `loaduvm()`.

3.2 Syscall

`fork()` copies user-level memory, trapframe, file descriptors and cwd.

`sys_brk()` increases or decreases the user-level memory.

`wait()` waits for a child process to exit.

`exit()` wakes up its parent process. The parent of its child processes would change to the init process.

`exec()` loads initial memory from disk and allocates the initial stack. The initial stack would be:

Initial Stack (each row means 8B)
argv points here (the address of this row is 16B aligned)
Maybe empty
auvx[3] = 0
auvx[2] = AT_NULL
auvx[1] = PGSIZE
auvx[0] = AT_PAGESZ
envp[0] = 0
argv[argc] = 0
...
argv[0]
argc

At first, argument strings would be pushed. Then, to make the stack pointer 16B aligned at last, I will check if the stack pointer should be minus 8.

```
The following size is:
8 * (  1   + (argc + 1) +   1   +   4   )
     argc      argv       envp    auxv
Therefore, if argc is an even number, sp should become (sp - 8).
```

3.3 Trapframe

Add `q0` and `tpidr_el0` in trapframe. The stack pointer is kept to be 16B aligned.

3.4 Idle process

Idle process should not execute the code of `/init`. I make use of `initcode` to handle it. I add an infinite loop in `initcode.S`, and let it be the first instruction of the idle process.

```
# initcode.S
start:
    ldr     x0, =init
    ldr     x1, =argv
    mov     x8, #SYS_execve
    svc     0x00

exit:
    mov     x8, #SYS_exit
    svc     0x00
    b       exit

exit1:
    b exit1 # The 8th instruction
```

```
void user_idle_init() {

    // points to the infinite loop
    uvm_init(p->pgdir, \
             _binary_obj_user_initcode_start + (7 << 2), \
             (long)(_binary_obj_user_initcode_size - (7 << 2)));

    // other codes are the same as user_init()
}
```

3.5 Entering Shell

`ls`: After tests, two new files are listed:

```
$ /ls
.              4000 1 512
..             4000 1 512
init           8000 2 25336
ls             8000 3 45760
sh             8000 4 63072
mkfs           8000 5 47568
mkdir          8000 6 39632
cat            8000 7 40840
hello.cpp      8000 8 74
readme.md      8000 9 22
console        0 11 0
```

I implement `cat` and `mkdir`.

`cat` :

```
$ cat hello.cpp
#include <cstdio>
 int main() {
        printf("Hello world!\n");
        return 0;
}
$ cat hello.cpp readme.md
#include <cstdio>
 int main() {
        printf("Hello world!\n");
        return 0;
}
This is a readme file
```

`mkdir` :

```
$ mkdir usr
$ ls
.              4000 1 512
..             4000 1 512
init           8000 2 25336
ls             8000 3 45760
sh             8000 4 63072
mkfs           8000 5 47568
mkdir          8000 6 39632
cat            8000 7 40840
hello.cpp      8000 8 74
readme.md      8000 9 22
console        0 11 0
usr            4000 12 32
$ cd usr
$ /ls
.              4000 12 32
..             4000 1 512
```

To add new text files, use `cat console` and redirect `stdout` to a file. Using `Ctrl + D` in the end.

```
$ cat console > usr/README
This is another readme file.
$ cat usr/README
This is another readme file.
```