# A RAM DISK FILESYSTEM

# Copyright Rich West

---

# Let's Party!

For this project you are required to implement a filesystem, preferably as an extension to FIFOS/MEMOS-2. The basic filesystem will be implemented in RAM and, if you get that far, you should work towards supporting a filesystem in persistent (secondary) storage (e.g., an IDE/SATA disk or USB flash drive).

# First Step: A RAM Disk Filesystem

To begin, you should start out with a RAM-only filesystem. You can choose one of two approaches (YOU ONLY NEED TO WORK ON ONE): (1) one approach is to implement your own filesystem using FIFOS/MEMOS-2 as the starting point, or (2) you can build a filesystem in the Linux kernel. If you choose the first approach, you should create a new system called DISCOS, as a play on words for a RAM DISK OS. For DISCOS, you will not be expected to implement system calls from user-space to kernel services, although that will be a bonus (see later). Instead, you can implement all your file operations as function calls, with all your code working in the kernel protection domain. If you choose the second approach, you will need to implement user-level code in Linux to test your kernel-based RAM disk, and you will have to use the `ioctl()` approach from the first primer assignment to pass control from user-space to kernel-level.

## Approach 1 (DISCOS) ** Worth more points **

In this approach, you will need to implement a RAM memory area in your DISCOS system that is initialized to 2MB using a constant defined in your code. Care must be taken to make sure the 2MB RAM disk "partition" is in contiguous RAM, so you will need to walk the memory map (from MEMOS-2) to find where free memory is available. You can assume your OS will run on a virtual machine with more than 2MB of RAM available. Alternatively, you can implement virtual memory support to treat your RAM disk as a logically contiguous region that maps to potentially non-contiguous frames of RAM.

## Approach 2 (Linux module)

Here, you will need to implement a Linux kernel loadable module for your RAM disk. However, you will **not** have to modify the core kernel.

- **Ramdisk initialization:**

The first step is to leverage the *ioctl()* code from the primer to implement your own routine that creates a RAM disk. Simply, you need to create an entry in the */proc* directory called */proc/ramdisk*. You will be able to see the "ramdisk" in */proc* when your module is loaded, but it should be removed from */proc* when your module is unloaded. To actually create a memory area for your ramdisk, you can *open /proc/ramdisk* and issue a corresponding *ioctl()* command. That is, you need to create an ioctl file operation on */proc/ramdisk* that invokes *vmalloc()* to establish a ramdisk memory area. Note that *kmalloc()* used to restrict memory allocations to a contiguous physical range up to 128KB (at least for Linux 2.4/2.6 kernels) while *vmalloc()* allows larger contiguous linear memory areas to be specified, although these areas may need paging support to map to non-contiguous page frames.

As with Approach 1, you can assume that your ramdisk memory area is initialized to 2MB using a constant defined in your code. That way, we can adjust this value if we recompile your code.

## Common to Both Approaches

- **Adding filesystem data structures:**

Once your ramdisk memory area is established, you can treat it like a raw partition, comprised of a sequence of blocks. You can assume that a ramdisk *block* is of size 256 bytes, and the first block of the partition is a *superblock* containing filesystem-specific information that we will describe later. The next  256 blocks contain an array of "index node" structures that contain attributes for a given file. Each *index node* is 64 bytes in size. After the index node array the partition will contain four blocks for a "block bitmap" used to keep track of free and allocated blocks in the rest of the partition. Each bit of the block bitmap will be set to one or zero, depending upon whether or not the corresponding block is free or allocated.

The rest of the partition contains the contents of two types of files: *directories* and *regular* files. A regular file can hold arbitrary data, while a directory file holds directory entries for its position in a *filesystem tree*. Each directory entry is a *(filename, index_node_number)* pair, where the *filename* is a string, padded as necessary, to be 14 bytes including the null terminator, and the *index_node_number* is a two-byte integer used to index into the index node array of your partition to find information about a given file. Observe that implicitly the directory structure is a tree of directory entries where each entry may be a regular or directory file. The tree is rooted with a directory file whose name is "/" and whose index_node_number is 0. That is, the root of the directory tree has an index node stored in the first entry of the index node array.

The following *attributes* are to be maintained for each index node (where there is one index node per file):

- *type* (4 bytes) -- either "dir" for directory or "reg" for regular files.
- *size* (4 bytes) -- the current size of the corresponding file in bytes. For directories, this will be the size of all entries in the corresponding directory file, where each entry's size is the number of bytes needed to record a *(filename, index_node_number)* pair.
- *location* (40 bytes) -- this attribute member identifies the *blocks* (potentially) scattered within the ramdisk that store the file's contents.

A file's storage space must occupy an integer number of ramdisk blocks. The location attribute of a file contains 10 block pointers of 4-bytes each: the first 8 pointers are *direct* block pointers, the 9th pointer is a *single-indrect* block pointer, and the 10th pointer is a *double-indirect* block pointer. This means that the maximum size of a single file will be: 256*8+64*256+64^2*256=1067008 bytes. Make sure you define constants for all these parameters!

To keep track of open files for each process using your filesystem, you need a table of open file descriptors on a per-process basis. You can think of this table as a per-process *ramdisk file descriptor table*. Each entry in this table will reference a *file object* that maintains two values: the current read/write *file position*, and a *pointer to the index node* in the ramdisk partition for the corresponding file.

**NOTE:** You can implement each process' ramdisk file descriptor table in memory outside the ramdisk partition.

- **The superblock:**

The superblock at the beginning of the filesystem partition contains information about:

- *the number of free blocks* that can be allocated for storing directory and regular file contents;
- *the number of free index nodes* that can be associated with newly created files. Note that each file in the system will have its own index node;

The additional space in the superblock can be used to record any additional information that you wish to use for convenience e.g., the first data block in your partition.

- **File operations:**

You need to create the following file operations (preferably in a library if implemented in Linux) that can be used by normal processes wishing to use your ramdisk filesystem:

- `int rd_creat(char *pathname)` -- create a regular file with absolute *pathname* from the root of the directory tree, where each directory filename is delimited by a "/" character. On success, you should return 0, else if the file corresponding to *pathname* already exists you should return -1, indicating an error. Note that you need to update the parent directory file, to include the new entry.
- `int rd_mkdir(char *pathname)` -- this behaves like `rd_creat()` but pathname refers to a *directory* file. If the file already exists, return -1 else return 0. Note that you need to update the parent directory file, to include the new entry.
- `int rd_open(char *pathname)` -- open an *existing* file corresponding to *pathname* (which can be a regular or directory file) or report an error if file does not exist. When opening a file, you should return a file descriptor value that will index into the process' *ramdisk file descriptor table*. As stated earlier, this table entry will contain a pointer to a file object. You can assume the file object has the *file position* set to 0. Return a value of -1 to indicate an error, if the file does not exist.
- `int rd_close(int fd)` -- close the corresponding file descriptor and release the file object matching the value returned from a previous `rd_open()`. Return 0 on success and -1 on error. An error occurs if *fd* refers to a non-existent file.

- `int rd_read(int fd, char *address, int num_bytes)` -- read up to *num_bytes* from a regular file identified by file descriptor, *fd*, into a process' location at *address*. You should return the number of bytes actually read, else -1 if there is an error. An error occurs if the value of *fd* refers either to a non-existent file or a directory file. If developing DISCOS, you may only have threads within a single shared address space, in which case you should identify a buffer region into which your data is read.
- `int rd_write(int fd, char *address, int num_bytes)` -- write up to *num_bytes* from the specified *address* in the calling process to a regular file identified by file descriptor, *fd*. You should return the actual number of bytes written, or -1 if there is an error. An error occurs if the value of *fd* refers either to a non-existent file or a directory file. If developing DISCOS, you may only have threads within a single shared address space, in which case you should identify a buffer region from which your data is written.
- `int rd_lseek(int fd, int offset)` -- set the file object's *file position* identified by file descriptor, *fd*, to *offset*, returning the new position, or the end of the file position if the offset is beyond the file's current size. This call should return -1 to indicate an error, if applied to directory files, else 0 to indicate success.
- `int rd_unlink(char *pathname)` -- remove the filename with absolute *pathname* from the filesystem, freeing its memory in the ramdisk. This function returns 0 if successful or -1 if there is an error. An error can occur if: (1) the *pathname* does not exist, (2) you attempt to unlink a non-empty directory file,

(3) you attempt to unlink an open file, or (4) you attempt to unlink the root directory file.
- `int rd_readdir(int fd, char *address)` -- read one entry from a directory file identified by *fd* and store the result in memory at the specified value of *address*. When reading a given entry, you should return a 14 byte filename, followed by a two byte index node number for the file. 1 is returned on success (to indicate a successfully read directory entry) and -1 on error. An error occurs if either of the arguments to rd_readdir() are invalid. Each time a process calls `rd_readdir()` you should increment the *file position* in the corresponding file object to refer to the next entry. If either the directory is empty or the last entry has been read by a previous call to `rd_readdir()` you should return 0, to indicate end-of-file.

- **Additional error checking:**

Observe that all the above operations should return -1 to indicate additional error conditions not stated. In the cases of `rd_creat()`, `rd_mkdir()` and `rd_write()` you need to return -1 if the file system has insufficient space and/or free index nodes to satisfy the requests. For `rd_creat()` and `rd_mkdir()` you should also report an error if you attempt to create a file whose pathname prefix refers to a non-existent directory. For example, if "home" was the name of a valid directory file with complete pathname "/home" but "/home/cs552" was non-existent, it would be an error to create "/home/cs552/project" but you could create "/home/cs552" first and then "/home/cs552/project".

## Testing

You should test your ramdisk filesystem using at least two processes (in Linux) or two threads (in DISCOS), where each process/thread creates and manipulates several ramdisk files. In your test routine(s) you need to non-trivially exercise each of the above file operations to show their correct functionality. Irrespective of Linux or DISCOS, you should use your file operations to create a file system tree of files, as well as special cases such as the largest single file or the maximum number of files your RAM disk will hold. Each file should then be written, as necessary, to store valid data that can be read correctly.

We will provide a test script for testing a Linux-based implementation of your filesystem. If you implement DISCOS (standalone), you will need to implement filesystem calls in one or more threads of your own creation. For DISCOS, you should show how you create your threads to make the filesystem calls in the provided test file.

The test file will be provided after the Thanksgiving break, with sufficient time in advance of the final demos.

# Second Step (Bonuses, bonuses, bonuses): Only If You Get This Far...

Extra credit will be given to filesystem implementations that support persistence in secondary storage. This means either providing the option to replace the RAM disk with a partition on a disk device (which can be a virtual disk for testing purposes), or supporting the ability to take snapshots of a RAM disk and store them back to a file. Clearly, this is more difficult in DISCOS as code is needed to read and write disk blocks and to interact with a disk controller. In this case, you will need to implement an IDE or SATA disk controller driver depending on the Qemu/PC emulated environment you use to test your system. If you do this in Linux, you can use low-level kernel routines to take snapshots of your RAM disk and store them in a swapfile of your own creation. Using the Linux approach, you can enforce snapshots when your filesystem module is removed from the kernel. Similarly, when it is loaded, you can check to see if there is already a snapshot in your swapfile and you can load that to re-initialize the state of your RAM disk. Since there is no notion of kernel modules in DISCOS, you will need to implement kernel functions to save and restore snapshots.

As DISCOS is fundamentally more challenging (and, hopefully, more rewarding) than building your own filesystem in Linux, it will be given extra credit regardless of supporting persistent snapshots. Another opportunity for extra credit will be given for a DISCOS implementation that supports the equivalent of an `ioctl` system call, with the ability to call down from a test application in user space down into the kernel. That way, you can write more realistic user-level test applications that mimic those found on POSIX systems. To support filesystem requests from user-level, your `ioctl` implementation will need to work like a system call. This means the GDT that you set up for FIFOS will need to be extended with two extra segments: a user data segment, and a user code segment. You will also need to add a system call entry for your `ioctl` into your IDT. One way to do this is to make your ioctl code be a wrapper around a software interrupt, such as:

```
int ioctl(int fd, unsigned long request, .../* arg(s) */)
{

    // request is either RD_CREAT, RD_MKDIR, RD_OPEN, etc
based on the file operations described above
    // arg(s) is replaced by the corresponding arguments
for the file operation, and these are loaded
    // into machine registers.

    // Let eax store the file operation (e.g., RD_CREAT),
ebx stores the first argument,
    // ecx the second argument, and edx the third argument


    // Here, 0x80 is the system call entry in your IDT to
handle the ioctl
```

```
    asm volatile ("int 0x80\n":: "a" (request), "b"
(arg1), "c" (arg2), "d" (arg3));

    // You can implement a trap or interrupt gate for your
ioctl,
    // but be careful that the trap gate keeps interrupts
enabled.

    // Once in the kernel, you system call handler can
implement
    // a switch-case statement, to call the appropriate
file operation
    // based on the value of the request.

}
```

- If working on DISCOS, the opportunities for bonuses include: (1) support for persistent storage of RAM disk contents, and (2) support for user-level threads that can call down into your kernel to make filesystem requests. For a Linux-based filesystem, you can still get extra credit for implementing persistent filesystem support by calling low-level kernel routines that take snapshots of your filesystem e.g., when you remove your kernel module you take a persistent snapshot, and when you load your kernel module you look to see if there is a snapshot that should be restored.

## Grading

### First Step:

| | |
|---|---|
| - Ramdisk initialization<br>- Correct file system data structures, including:<br>  ○ superblock, index node array and block bitmap<br>  ○ per-process *ramdisk file descriptor tables*<br>  ○ file objects | 25% |
| - Correct and complete implementation of:<br>  ○ file operations<br>  ○ block management<br>  ○ WE WILL PROVIDE A TEST SCRIPT FOR THOSE WORKING WITH LINUX. YOU MUST PORT THIS SCRIPT TO YOUR OS IF WORKING ON DISCOS. WE WILL TEST THAT THE SCRIPT WORKS FOR ALL SPECIFIED CASES. IN DISCOS, YOU CAN REPLACE PROCESSES WITH YOUR THREADS.<br>- Correct program output and valid test scenarios showing expected behavior of all file operations | 70% |

| | |
|---|---|
| ○ You should consider boundary cases (e.g., maximum # files, and single largest file) and what happens when error conditions arise<br>○ Include demonstration of single and double-indirect block pointer usage for large files | |
| • Program style and comments | 5% |
| • **DISCOS Implementation** | +20% |

### Second Step (Bonuses):

| | |
|---|---|
| • Persistent storage support (Linux) | 15% |
| • Persistent storage support (DISCOS) including disk driver implementation | 20% (IDE driver support), or<br>30% (SATA driver support) |
| • Dual mode (User-Kernel) support in DISCOS | 20% |

### Team Effort and Submission:

- You can work in groups of up to TWO people. Do not copy other work, or share this assignment outside the classroom. Please do NOT upload this assignment to any publicly accessible repository, online site, or anywhere that allows people outside the class to see it. Chegg, UpWorks, and other sites are strictly forbidden.
- If you find sources of information on the Internet, in books, or elsewhere, you must provide references to those sources.
- We require you to show evidence that you understand the code for your solution when it comes to assignment demonstrations.
- Please upload all your files to Gradescope, with the most complete solution to your assignment.
- Only one member of a group needs to upload their solution but you should include the name of the other team member, if there is one in a README file.
- Everyone should be present at the demo to get a grade, and you will be expected to know how your solution works.

Happy Programming!