Operating Systems Project: Topic 12

Implement a Simple In-Memory File System (RamFS)

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.29

Outline

1. Project Goals & Requirements
2. Theory: Architecture
3. Implementation Logic
4. Evaluation Strategy
5. Action Plan

Outline

1. Project Goals & Requirements
2. Theory: Architecture
3. Implementation Logic
4. Evaluation Strategy
5. Action Plan

The Mission: Topic 12 Requirements

Objective: Build a file system that lives entirely in RAM (like `tmpfs`).

Requirement 1: System Integration

- **No Block Device:** Must mount with `mount -t myramfs none /mnt`.
- **VFS Compliance:** Must integrate with Linux Virtual File System.

Requirement 2: Core Operations

- **Metadata:** Create files (`touch`) and directories (`mkdir`).
- **Data:** Read and write file content using the Page Cache.

Advanced Options

Option A: Memory Quotas (Safety)

- **Risk:** A user writing ‘yes > /mnt/file‘ can crash the OS (OOM).
- **Solution:** Limit max pages per superblock. Return `-ENOSPC`.

Option B: Snapshot Persistence

- **Risk:** RAM is volatile. Reboot = Data Loss.
- **Solution:** Serialize the memory tree to a file on `umount` and restore on `mount`.

Outline

1 Project Goals & Requirements

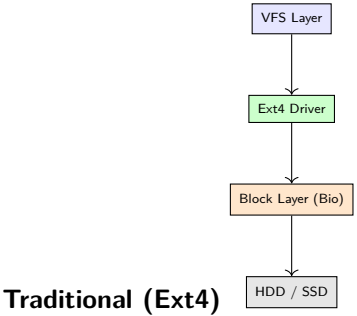
2 Theory: Architecture

3 Implementation Logic

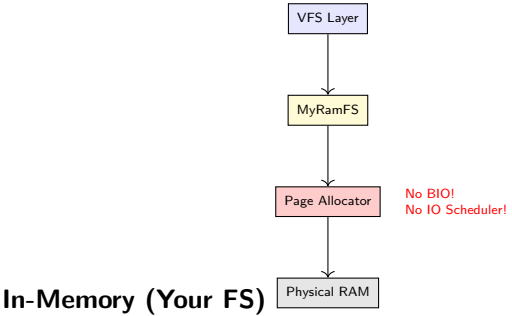
4 Evaluation Strategy

5 Action Plan

Theory 1: Architecture Comparison



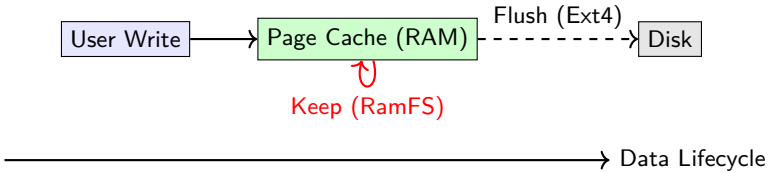
Traditional (Ext4)



In-Memory (Your FS)

Theory 2: The "Backing Store" Secret

Where is the file data actually stored? **The Page Cache**.



RamFS Strategy:
Mark pages as "Dirty" but **never flush** them.
The cache *is* the storage.

Theory 3: Memory Objects

Since we don't have a disk to store metadata, we use kernel structures.

Struct Inode (The File)

- Permissions (0755)
- Timestamps
- **Mapping:** Points to Pages

Struct Dentry (The Name)

- Name ("home", "bin")
- Parent/Child pointers
- Points to Inode

Crucial Rule: RamFS must **PIN** these in memory. Normal VFS deletes unused dentries. If RamFS lets a dentry die, the file is gone forever!

Outline

1 Project Goals & Requirements

2 Theory: Architecture

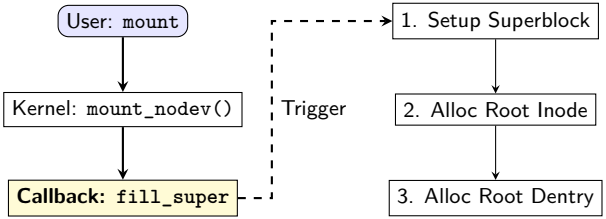
3 Implementation Logic

4 Evaluation Strategy

5 Action Plan

Logic 1: The Mount Process

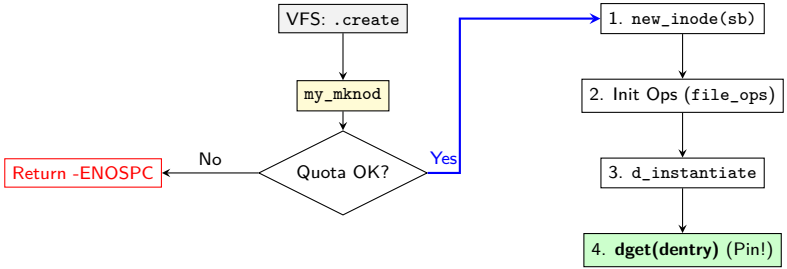
What happens when you type `mount -t myramfs ...`?



Success: Mounted at /mnt

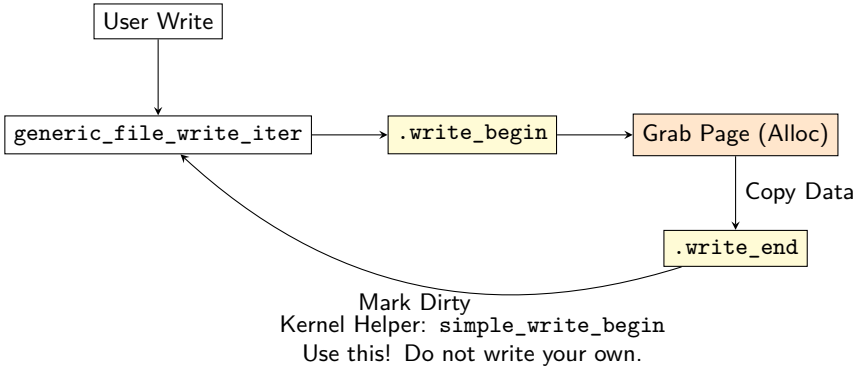
Logic 2: Creating a File (touch)

This flow splits into two phases: Validation and Allocation.



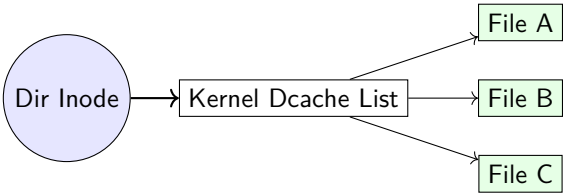
Logic 3: Writing Data (`echo "hi"`)

You don't write "write" logic. You leverage the page cache helpers.



Logic 4: Directory Lookup (`ls`)

How does the kernel find files in your memory?



Magic: `simple_dir_inode_operations`
The kernel maintains the list of children for you.
You just need to allocate the Inodes.

Outline

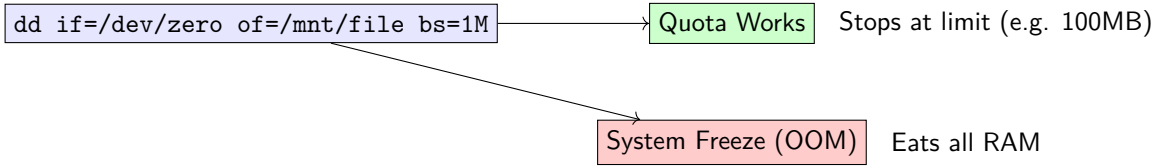
1. Project Goals & Requirements
2. Theory: Architecture
3. Implementation Logic
4. Evaluation Strategy
5. Action Plan

Evaluation 1: Functional Verification

1. **Load Module:** `insmod myramfs.ko`
2. **Mount:** `mount -t myramfs none /mnt/test`
3. **Basic Ops:**
   - `touch /mnt/test/a.txt` (Test Create)
   - `echo "Hello" > /mnt/test/a.txt` (Test Write)
   - `cat /mnt/test/a.txt` (Test Read)
   - `rm /mnt/test/a.txt` (Test Unlink/Free)
4. **Unmount:** `umount /mnt/test`

Evaluation 2: Stress & Quota Test

The "Bomb" Test:



Outline

1 Project Goals & Requirements

2 Theory: Architecture

3 Implementation Logic

4 Evaluation Strategy

5 Action Plan

Roadmap & Resources

Action Plan:

1. **Setup:** Write the module skeleton. Register `file_system_type`.
2. **Mount:** Implement `fill_super`. Create the Root Inode and Dentry.
3. **Logic:** Implement `create` and `mkdir` using `simple_*` helpers.
4. **Safety:** Add memory quotas or snapshotting logic to prevent data loss.

Resources:

- `fs/ramfs/`: The official reference implementation (Gold Standard).
- `fs/libfs.c`: Contains `simple_read_folio` and other VFS helpers.
- `include/linux/fs.h`: Definitions for `inode`, `dentry`, and `super_block`.