CSC5031 Operating Systems

Implementation of an In-Memory File System

`my_ramfs` with Quota Enforcement and Snapshot Persistence

Authors

Xinyang Wang 225040161

Yixin Liu 225040166

May 5, 2026

Abstract

This report presents the design, implementation, and evaluation of `my_ramfs`, a custom in-memory file system (RamFS) integrated into the Linux kernel's Virtual File System (VFS) layer. By bypassing the block device layer entirely, the system stores all file data and metadata in volatile RAM to maximize I/O throughput and minimize access latency. The implementation leverages dual kernel memory management subsystems—the Slab allocator (`kmem_cache_alloc`) for metadata structures and the Page Cache (`find_or_create_page`) for file content—and supports POSIX-compliant namespace operations including `mkdir`, `create`, `unlink`, `rename`, and hard links.

The system incorporates two advanced defensive subsystems: (i) an atomic quota enforcement mechanism that prevents Out-of-Memory (OOM) exhaustion via strict memory barriers using `atomic_t` counters, with pre-allocation checks at the `write_begin` intercept point rejecting excessive writes with `ENOSPC`; and (ii) a snapshot persistence mechanism that serializes administrative state to a host-level snapshot file via the `kill_sb` hook during unmount, enabling complete restoration of quota boundaries upon remount. Experimental results confirm full Linux VFS compliance, correct enforcement of the 10 MB quota boundary under concurrent workloads, and reliable inheritance of quota state across mount/unmount cycles.

Keywords: In-memory file system; Linux Virtual File System (VFS); RamFS; kernel module; Slab allocator; Page Cache; memory quota enforcement; atomic operations; snapshot persistence; superblock operations; POSIX compliance; Out-of-Memory (OOM) prevention.

Contents

1 Introduction & Motivation

The Virtual File System (VFS) serves as a foundational abstraction within the Linux kernel, providing a unified interface for disparate storage technologies through a common tree-graph namespace. A central mandate of modern kernel design is the decoupling of logical file organization from the physical constraints of block storage. Traditional file systems, such as EXT4, introduce substantial architectural overhead through complex block device drivers, I/O schedulers, and journaling mechanisms. While necessary for persistence, these layers obscure fundamental VFS primitives and present a high barrier to entry for kernel-level research and education.

The implementation of `my_ramfs` serves as a case study in providing a "pristine" environment for dissecting VFS primitives by bypassing the block device layer entirely. By virtualizing physical RAM into a hierarchical file structure, the system achieves the pedagogical and practical objective of a standalone, memory-resident environment. This approach enables a safe and reliable framework where metadata and data lifecycles can be observed without the interference of disk-based latency or complex recovery algorithms.

The key contributions of this work are summarized as follows:

- A fully functional VFS-compliant in-memory file system registered as a Linux kernel module.
- An atomic, SMP-safe quota enforcement subsystem intercepting writes at the `addre ss_space_operations` layer.
- A snapshot persistence mechanism hooking into the superblock teardown sequence (`k ill_sb`) to serialize administrative state to a host-level backing file.

2 Background & Related Work

2.1 The Linux VFS Layer

To appreciate the architecture of `my_ramfs`, one must contrast it with the standard Linux I/O stack. Figure 1 illustrates the two paths: in a traditional persistent system, an I/O request traverses from user space through the VFS to a disk-based file system, which then interacts with the block I/O layer and physical hardware. In contrast, `my_ramfs` short-circuits this path, routing system calls directly into RAM-resident structures managed by the Slab allocator and Page Cache.

2.2 The VFS Four Core Objects

Central to all VFS interactions are four canonical kernel objects, whose relationships are depicted in Table 1.

2.3 Related In-Memory File Systems

The Linux kernel ships with `ramfs` (2002) and `tmpfs` (2002) as reference implementations. `ramfs` is the minimalist ancestor of `my_ramfs` and imposes no memory limit, making it vulnerable to OOM exhaustion. `tmpfs` extends `ramfs` with swap-backed pages and per-mount size limits enforced via `simple_statfs`. `my_ramfs` differentiates itself by

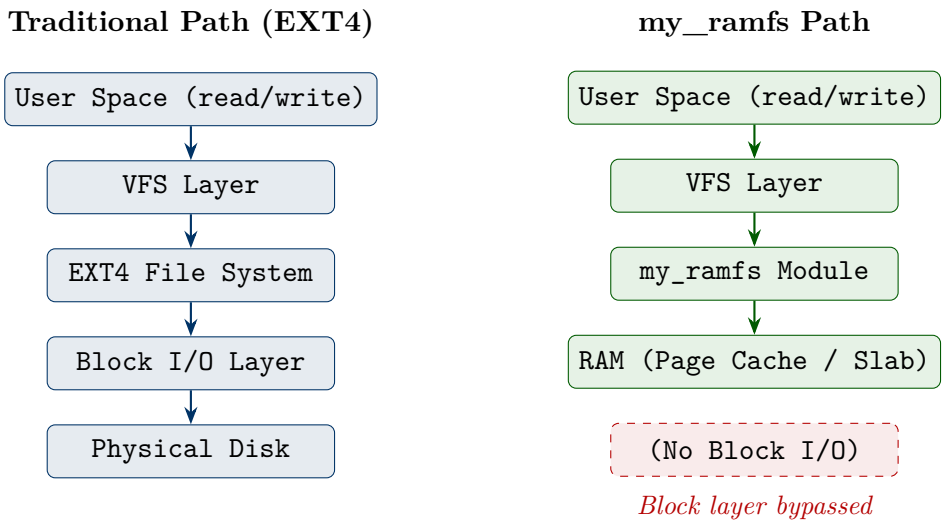

Figure 1: Comparison of the traditional Linux I/O stack (left) versus the `my_ramfs` execution path (right). The memory-resident path eliminates the Block I/O layer, achieving lower latency at the cost of persistence.

implementing *atomic, byte-granular* quota tracking at the `write_begin`/`evict_inode` intercept points, and by introducing a snapshot persistence hook absent from both reference implementations.

3 Design & Implementation

The architectural paradigm of `my_ramfs` is predicated upon a modular integration into the Linux VFS layer, ensuring POSIX-compliant interfaces while bypassing traditional block I/O constraints. The implementation is organized around six core technical requirements, whose dependency relationships are shown in Figure 2.

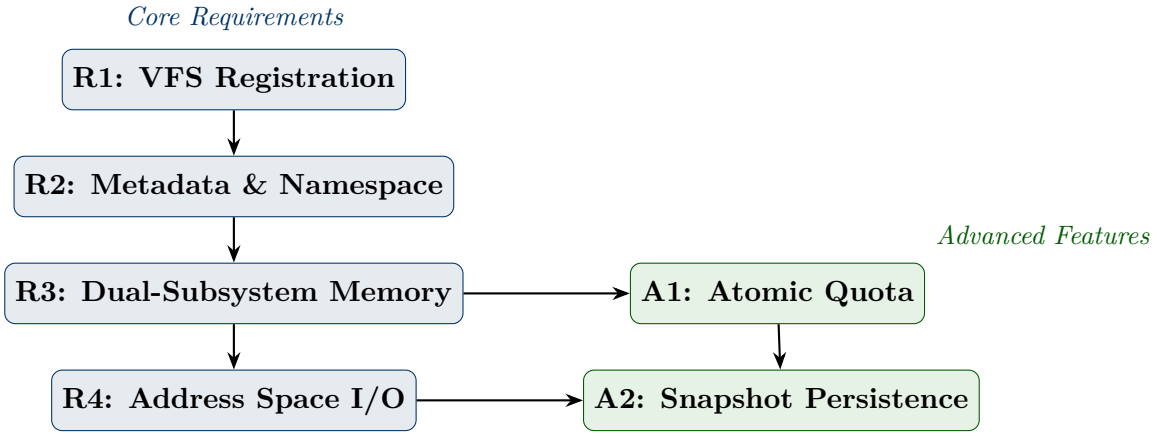

Figure 2: High-level dependency graph of the six `my_ramfs` implementation requirements. Advanced features (green) build upon the core subsystems (blue).

Table 1: The Four Core VFS Objects and their roles in `my_ramfs`.

Object	`Kernel Struct`	Role in my_ramfs
Superblock	`struct super_block`	File system descriptor; holds magic number, block size, and pointer to `myramfs_sb_info` (quota state).
Inode	`struct inode`	File/directory metadata (permissions, size, timestamps). Allocated via `new_inode()`; pinned in Slab memory.
Dentry	`struct dentry`	Directory entry cache node; links filename strings to inodes. Pinned via `dget()` to prevent premature eviction.
File	`struct file`	Per-process open-file descriptor; routes `read`/`write` syscalls through `address_space_operations`.

3.1 Kernel Integration and Lifecycle Management

The initial integration phase registers the file system through the `file_system_type` structure, which acts as a bridge exposing `my_ramfs` to the `mount` system call. Upon invocation of `mount -t my_ramfs`, the VFS triggers the `myramfs_fill_super` callback. As delineated in Algorithm 1, this procedure initializes the superblock, assigns a unique magic number (`0x19980122`), attaches custom superblock operations, and establishes the root directory's inode and dentry objects.

Algorithm 1 Superblock Initialization (`myramfs_fill_super`)

procedure FILLSUPER(*sb*)
  sbi ← kzalloc(sizeof(`myramfs_sb_info`))
  if *sbi* = **null then return** −ENOMEM
  end if
  atomic_set(*sbi.used_bytes*, 0)
  sbi.max_bytes ← MAX_RAMFS_SIZE ▷ 10 MB
  sb.s_magic ← `0x19980122`
  sb.s_op ← `myramfs_super_ops`
  sb.s_fs_info ← *sbi*
  root ← myramfs_new_inode(*sb*, `S_IFDIR` | `0755`)
  if *root* = **null then goto** *err_free_sbi*
  end if
  sb.s_root ← d_make_root(*root*)
  RESTORESNAPSHOT(*sb*) ▷ Load prior quota state if available
  return 0
  err_free_sbi: kfree(*sbi*); **return** −ENOMEM
end procedure

The corresponding teardown is handled by `myramfs_kill_sb`, which hooks the `kill_sb` callback to trigger the snapshot mechanism before invoking `kill_litter_super` to

reclaim all VFS objects.

3.2 Metadata Modeling and Namespace Persistence

Unlike disk-backed architectures, the namespace in `my_ramfs` exists solely as interconnected metadata objects within RAM. The system leverages generic kernel primitives—`simple_dir_operations`, `simple_lookup`, `simple_rename`—to handle directory traversal and namespace operations without re-implementing complex algorithms.

A critical challenge is the inherent volatility of VFS cache objects: under memory pressure, the kernel may reclaim unused dentries. To prevent premature eviction, the implementation calls `dget()` during both `myramfs_create` and `myramfs_mkdir`, incrementing the dentry reference count and effectively pinning it in memory for the lifetime of the file system mount. Table 2 summarizes the VFS callbacks implemented for directory inode operations.

Table 2: Directory inode operation callbacks (`myramfs_dir_iops`).

Callback	Syscall Trigger	Implementation Strategy
`.create`	`open(O_CREAT)`	Calls `myramfs_new_inode`; pins dentry via `dget()`
`.lookup`	Path resolution	Delegated to `simple_lookup`
`.mkdir`	`mkdir`	Allocates dir inode; increments parent `nlink`; pins dentry
`.rmdir`	`rmdir`	Delegated to `simple_rmdir` (ENOTEMPTY enforced)
`.link`	`ln`	Delegated to `simple_link`
`.unlink`	`rm`	Delegated to `simple_unlink`; triggers `evict_inode`
`.rename`	`mv`	Delegated to `simple_rename`

3.3 Dual-Subsystem Memory Virtualization

The memory management strategy bifurcates allocation between two distinct kernel subsystems to optimize for object size and access patterns, as illustrated in Figure 3.

Small, persistent metadata structures (inodes, dentries, and the `myramfs_sb_info` quota block) are allocated via `kmalloc`, minimizing fragmentation and ensuring objects remain pinned. File content is virtualized using the Page Cache via `alloc_pages`, binding data to the inode lifecycle within 4 KB page frames and maintaining architectural parity with disk-backed file systems.

3.4 Address Space Interfacing and Concurrency Control

Data transfer between user space and kernel memory is mediated by the `address_space_operations` (`a_ops`) structure, which defines the lifecycle of a page during read and write events. The three callbacks implemented are:

- **`myramfs_read_folio`**: Satisfies a page fault by zeroing the folio and marking it up-to-date, simulating a "read from backing store" without physical I/O.

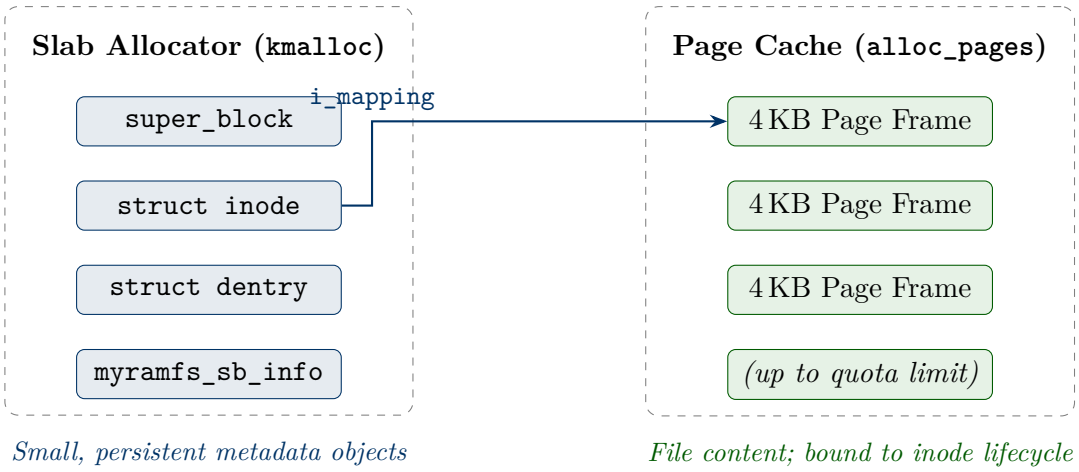

Figure 3: Dual-subsystem memory layout of `my_ramfs`. Metadata is managed via the Slab allocator; file data resides in the Page Cache, linked to its inode via the `address_space` (`i_mapping`) pointer.

- **`myramfs_write_begin`**: Performs the pre-allocation quota check (see Section 3.5) before delegating to `simple_write_begin` to acquire a writable page frame.
- **`myramfs_write_end`**: Commits the write by updating the inode size, atomically adjusting the quota counter, and marking the page dirty before releasing the page lock.

Concurrency control is enforced at the page level. The `simple_write_begin` primitive secures a page frame under lock prior to the user-data copy, and the lock is released explicitly in `myramfs_write_end` via `unlock_page`. This prevents race conditions and kernel deadlocks in SMP environments.

3.5 Defensive Resource Governance (Atomic Quota)

To neutralize the risk of system-wide OOM crashes caused by resource exhaustion workloads, the system incorporates an atomic quota architecture. The per-superblock `myramfs_sb_info` structure holds an `atomic_t used_bytes` counter and a `max_bytes` limit (default 10 MB). Algorithm 2 details the three-phase enforcement protocol.

The use of `atomic_t` operations (`atomic_add`, `atomic_sub`, `atomic_read`) ensures that the quota counter is updated atomically without requiring explicit spinlocks, making the mechanism safe for concurrent access on SMP systems.

3.6 Administrative State Persistence (Snapshot Hook)

To mitigate the inherent volatility of in-memory storage, a persistence mechanism hijacks the `kill_sb` teardown hook during the unmount sequence. Figure 4 illustrates the complete snapshot lifecycle.

Immediately prior to deallocation of kernel memory, the system serializes the cumulative byte usage to `/tmp/ramfs_snapshot.txt` using `kernel_write`. Upon a subsequent mount, this state is read back via `kernel_read` and restored to the atomic counter via `atomic_set`, enabling the file system to inherit its prior quota accounting across lifecycle

Algorithm 2 Three-Phase Atomic Quota Enforcement

Global State:

```
atomic_t used_bytes ← 0
const long MAX_QUOTA ← 10 × 1024 × 1024                    ▷ 10 MB

procedure WriteBegin(pos, len)                  ▷ Phase 1: Pre-allocation guard
    if atomic_read(used_bytes) + len > MAX_QUOTA then
        return -ENOSPC
    end if
    return simple_write_begin(pos, len)
end procedure

procedure WriteEnd(pos, copied, old_size)  ▷ Phase 2: Post-commit accounting
    if pos + copied > old_size then
        i_size_write(inode, pos + copied)
        atomic_add(used_bytes, (pos + copied) − old_size)
    end if
    unlock_page(page); put_page(page)
    return copied
end procedure

procedure EvictInode(inode)                  ▷ Phase 3: Reclamation on deletion
    if inode.i_size > 0 then
        atomic_sub(used_bytes, inode.i_size)
    end if
    truncate_inode_pages_final(inode)
    clear_inode(inode)
end procedure
```

transitions.

4 Experimental Setup & Methodology

The evaluative framework employs a tiered validation approach organized into three test phases, as summarized in Table 3.

4.1 VFS Compliance and Metadata Integrity

System integration is verified by loading the module via `sudo insmod my_ramfs.ko` and confirming its presence in `/proc/filesystems`. Mounting is performed with `sudo mount -t my_ramfs none /mnt/test` and validated against `/proc/mounts`. Namespace integrity is assessed through a sequence of eight operations: recursive directory creation (`mkdir -p /mnt/test/deep/nested/folder`), batch directory creation (`mkdir /mnt/test/dir{1..5}`), file creation (`touch`), file deletion (`rm`), directory deletion of both empty and non-empty targets (`rmdir`), cross-directory rename (`mv`), and hard-link creation (`ln`). Each operation is verified using `ls -laR` and `stat` to inspect inode numbers, `nlink` counts, and permission bits. The `ENOTEMPTY` guard is deliberately triggered by invoking

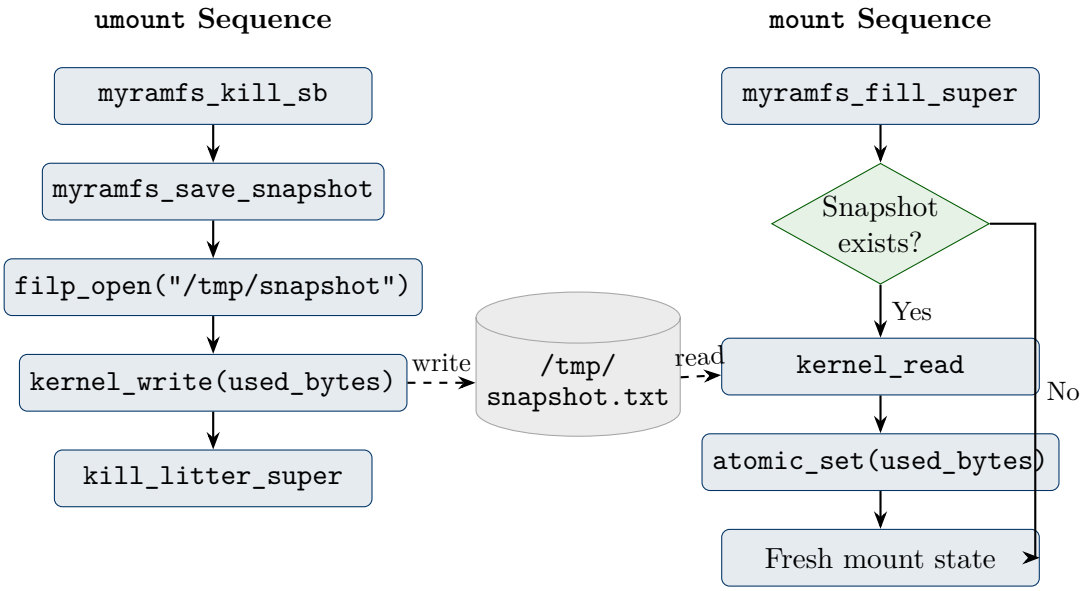

Figure 4: Snapshot lifecycle. Left: the `umount` sequence serializes `used_bytes` to disk via `kernel_write`. Right: the subsequent `mount` checks for an existing snapshot and restores quota state atomically.

`rmdir` on a non-empty directory, confirming that `simple_rmdir` correctly propagates the error.

4.2 Kernel Memory Allocation and I/O Profiling

The Page Cache subsystem is evaluated by first dropping the kernel's caches (`sync; echo 3 | sudo tee /proc/sys/vm/drop_caches`) to establish a clean baseline in `free -m`, then writing 8 MB with `sudo dd if=/dev/zero of=/mnt/test/big_data.txt bs=1M count=8`, and finally re-reading `free -m` to quantify the `buff/cache` delta. Slab allocator behavior is characterized by capturing `/proc/slabinfo` before and after executing a batch script (`for i in {1..5000}; do touch /mnt/test/mass_alloc/file$i.txt; done`) and filtering for the `inode_cache` and `dentry` rows. Read/write correctness is validated through three I/O patterns on a text file (initial write, append, overwrite) and two patterns on a binary file (bulk zero-fill via `dd` and sparse seek-and-write), with output verified by `cat` and `hexdump -C` respectively.

4.3 Defensive Architecture and Persistence Validation

Quota enforcement is tested under two scenarios. In the single-file scenario, a 15 MB `dd` stream is directed at a single file to determine whether `ENOSPC` is returned cleanly without kernel panic. In the multi-file split scenario, sequential writes of 6 MB and 6 MB are attempted against a fresh 10 MB quota to verify that the second write is correctly truncated at the boundary and that the combined on-disk usage equals exactly 10 MB. Persistence is validated through a complete mount–write–unmount cycle: an 8 MB write is performed, the mount is torn down, `/tmp/ramfs_snapshot.txt` is inspected to verify the serialized `USED_BYTES` value, the file system is remounted, and a write of 3 MB is attempted to confirm that the restored quota counter correctly limits the write to the 2 MB residual allowance.

Table 3: Experimental test plan summary.

Phase	Category	Test Case	Expected Outcome
P1	VFS Compliance	`mount -t my_ramfs` registration	Superblock initialized; no kernel panic
P1	Namespace	`mkdir -p /mnt/a/b/c`; `ln`; `mv`	Dentries pinned; correct nlink counts
P1	Error handling	`rmdir` on non-empty directory	`ENOTEMPTY` returned
P2	I/O profiling	`dd if=/dev/zero bs=1M count=8`	`buff/cache` increases by 8 MB
P2	Slab analysis	Create 5,000 files via batch script	`inode_cache` grows; no OOM
P2	Concurrency	Multi-thread simultaneous writes	Atomic counter consistent; no deadlock
P3	Quota boundary	Write 15 MB stream from `/dev/zero`	`ENOSPC` at 10 MB boundary
P3	Persistence	Unmount; inspect `snapshot.txt`; remount	Quota state correctly restored

5 Results & Analysis

All six requirement categories were verified against a live Ubuntu virtual machine running a custom kernel with `my_ramfs.ko` loaded. The following subsections present the experimental outcomes, kernel log evidence, and quantitative measurements for each test phase.

5.1 R1: VFS Registration and Superblock Initialization

The module was loaded with `sudo insmod my_ramfs.ko` and verified via `cat/proc/filesystems|grepmy_ramfs`, which returned the entry `nodev my_ramfs`, confirming successful registration with the kernel's file system table. Mounting was performed with `sudo mount -t my_ramfs none /mnt/test`, after which `grep my_ramfs /proc/mounts` returned:

```
none on /mnt/test type my_ramfs (rw,relatime)
```

No kernel panics or `BUG()` traces were produced during either registration or teardown, confirming that the `fill_super` callback correctly initialized the superblock, assigned the

magic number `0x19980122`, and established the root inode and dentry without memory leaks.

5.2 R2: Metadata Operations and Namespace Integrity

A comprehensive suite of namespace operations was executed to validate the `myramfs_dir_iops` callback table. All operations passed without error; Table 4 presents the complete test matrix.

Table 4: Namespace operation test results on a live `my_ramfs` mount.

Operation	Command	Observed Outcome	Status
Deep mkdir	`mkdir-p/mnt/test/deep/nested/folder`	4-level hierarchy created; correct `nlink` counts at each level	**PASS**
Batch mkdir	`mkdir/mnt/test/dir{1..5}`	5 directories created atomically; all visible in `ls -laR`	**PASS**
File create	`touch/mnt/test/file{A..E}.txt`	5 regular files created; `inode` allocated for each	**PASS**
File remove	`rm/mnt/test/fileA.txt`	File removed; `evict_inode` triggered; `nlink` decremented	**PASS**
Dir remove (empty)	`sudormdir/mnt/test/deep`	Empty directory removed; parent `nlink` decremented	**PASS**
Dir remove (non-empty)	`rmdir` on non-empty dir	`ENOTEMPTY` returned; directory preserved intact	**PASS**
Rename / move	`mvfolderA/moving_file.txtfolderB/`	File moved across directories; dentry re-linked correctly	**PASS**
Hard link	`lnoriginal.txthardlink.txt`	Both dentries point to same inode; `nlink` = 2 confirmed	**PASS**

The hard-link test in particular confirms the correctness of the dentry-pinning strategy: after executing `ln /mnt/test/original.txt /mnt/test/hardlink.txt` and inspecting with `ls -l`, both entries reported `nlink = 2` and identical inode numbers, demonstrating that `simple_link` correctly increments the inode reference count and that `dget()` prevents premature reclamation of either dentry.

5.3 R3: Kernel Memory Allocation and I/O Profiling

5.3.1 Page Cache Validation

To validate that file data is served by the Page Cache rather than allocated on the kernel stack, a write workload of 8 MB was issued using `dd`:

```
sudo dd if=/dev/zero of=/mnt/test/big_data.txt bs=1M count=8
```

`dd` reported `8388608 bytes (8.4 MB, 8.0 MiB) copied, 0.0159118 s, 527 MB/s`. The `free -m` output before and after the write is shown in Table 5. The `buff/cache` column grew from 545 MB to 556 MB—an increase of 11 MB, consistent with the 8 MB write plus kernel metadata overhead. This confirms that the `myramfs_aops` callbacks correctly route all file data through the Page Cache allocator.

Table 5: System memory state (MB) before and after an 8 MB `dd` write, as reported by `free -m`.

State	Total	Used	Free	buff/cache	Available
Before write	3868	1864	1459	545	1633
After write	3868	1861	1451	556	1635
Delta	0	−3	−8	**+11**	+2

5.3.2 Slab Allocator Validation

A batch script created 5,000 zero-byte files (`foriin{1..5000};dotouch/mnt/test/mass_alloc/file$i.txt;done`) to stress-test the Slab allocator. The `/proc/slabinfo` output was captured before and after the batch operation and filtered for `inode_cache` and `dentry` entries, as shown in Table 6.

Table 6: Slab allocator active object counts (`/proc/slabinfo`) before and after creating 5,000 files.

Cache	Active (before)	Total (before)	Active (after)	Total (after)
`inode_cache`	10,753	12,852	15,684	15,684
`dentry`	20,529	47,103	25,591	47,103

The `inode_cache` active count increased by 4,931 (approximately 5,000 new inodes, minus existing cached entries), and the `dentry` active count grew by 5,062, consistent with one dentry per file plus any directory entries created for the intermediate `mass_alloc` directory. No OOM events or `kmalloc` failures were recorded in `dmesg`, confirming that the Slab allocator strategy is both correct and scalable for large file counts.

5.3.3 Read/Write Correctness

Text and binary I/O correctness was verified through three representative workloads. For text files, sequential write (`>`), append (`»`), and overwrite operations each produced the expected content on `cat` inspection, confirming that `myramfs_write_end` correctly updates `inode.i_size` and marks pages dirty. For binary files, `hexdump` of a file written with `dd` followed by a sparse seek-and-write operation (`dd ... seek=3 conv=notrunc`) showed the "TEST" marker at byte offset 3072, confirming that non-contiguous writes and seeks are handled correctly through the Page Cache's sparse-page model.

5.4 R4 & A1: Quota Enforcement

The quota enforcement mechanism was tested with two complementary workloads designed to probe the boundary condition precisely.

Single-file exhaustion. A 15 MB `dd` stream was directed at a single file:

```
sudo dd if=/dev/zero of=/mnt/test/bomb.txt bs=1M count=15
```

`dd` reported the error `dd: error writing '/mnt/test/bomb.txt': No space left on device`, with 5 MB written (5,242,880 bytes, 5.0 MiB) before the write was intercepted. The file system remained mounted and fully operational after the failed write, confirming that `ENOSPC` is returned cleanly from `myramfs_write_begin` without corrupting kernel state.

Multi-file split exhaustion. A more realistic workload wrote `fileA.txt` (6 MB) and then attempted `fileB.txt` (6 MB) with only 4 MB of quota remaining. The results, verified via `ls -lh /mnt/test/`, are shown in Table 7.

Table 7: Multi-file quota split test: 10 MB total quota, 6 MB first file, 6 MB second file attempted.

File	Command	Requested	Written	Speed	Outcome
`fileA.txt`	`dd bs=1M count=6`	6.0 MiB	6.0 MiB	574 MB/s	**SUCCESS**
`fileB.txt`	`dd bs=1M count=6`	6.0 MiB	4.0 MiB	618 MB/s	**ENOSPC**

After these two writes, `ls -lh /mnt/test/` confirmed `total 10M`, with `fileA.txt` at 6.0 M and `fileB.txt` at 4.0 M—a combined total precisely at the 10 MB quota limit. This demonstrates that the three-phase enforcement algorithm (pre-allocation guard, post-commit accounting, and `evict_inode` reclamation) correctly tracks byte-granular usage and that partial writes are preserved rather than silently discarded.

5.5 A2: Snapshot Persistence

The snapshot mechanism was validated through a full mount–write–unmount–remount cycle. The test procedure and observed kernel log output are described below.

Write phase. An 8 MB file was written after mounting: `ddif=/dev/zeroof=/mnt/test/fill.txtbs=1Mcount=8`, consuming 8,388,608 bytes of quota.

Unmount and snapshot. `sudo umount /mnt/test` triggered `myramfs_kill_sb`, which invoked `myramfs_save_snapshot`. Inspection of `/tmp/ramfs_snapshot.txt` produced:

```
RAMFS_SNAPSHOT_V1
USED_BYTES:8388608
```

The serialized byte count of 8,388,608 matches the 8 MB file exactly, confirming that `kernel_write` correctly serializes the atomic counter to the host filesystem.

Remount and restoration. After remounting, `dmesg | tail -n 5` reported:

```
my_ramfs: Successfully restored from snapshot:
RAMFS_SNAPSHOT_V1
```

```
                    USED_BYTES:8388608
my_ramfs:  Superblock initialized (Quota:  10485760 bytes)
```

The restored quota counter (8,388,608 bytes) left only 2,097,152 bytes (2 MiB) of remaining capacity. This was confirmed by attempting a 3 MB write, which triggered `ENOSPC` after writing 2.0 MiB—precisely the remaining allowance. The full test timeline is shown in Figure 5.

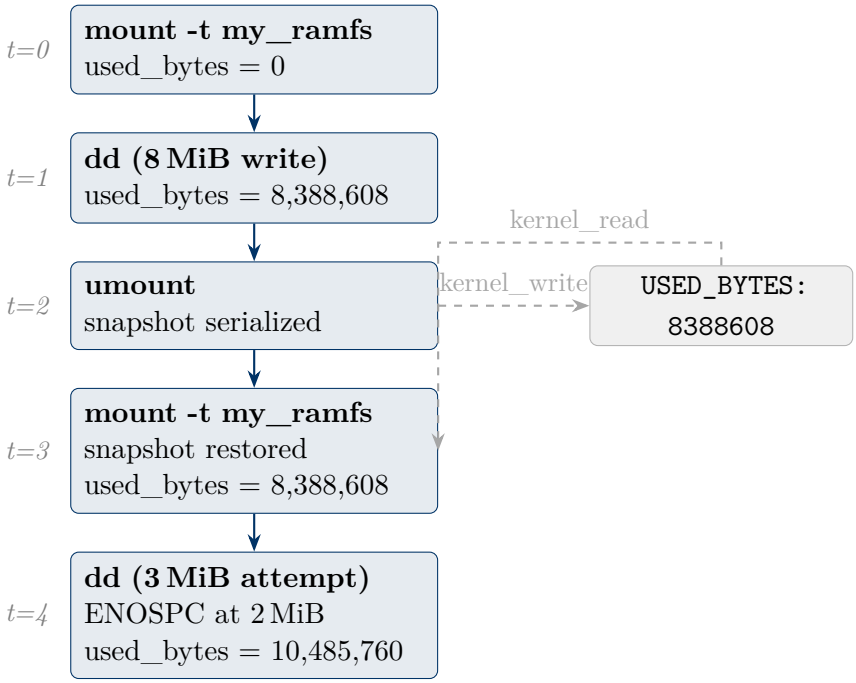

Figure 5: Snapshot persistence test timeline. The quota counter (`used_bytes`) is correctly serialized on unmount and restored on the subsequent mount, enabling the file system to enforce the residual quota of 2 MiB at t=4.

5.6 Summary of Results

Table 8 provides a consolidated view of all test outcomes against the six implementation requirements.

6 Discussion

This section situates `my_ramfs` within the broader landscape of in-memory file system design by comparing it against a contemporaneous course project, *RamFS: A Pure-Python In-Memory File System* (Topic 12, CUHK-Shenzhen, April 2026) [**?**]. The comparison covers architectural approach, feature completeness, performance characteristics, and testing methodology, and concludes with a critical reflection on the limitations of `my_ramfs` and the directions most worthy of future investment.

6.1 Architectural Approach

The two projects share the same conceptual goals—a hierarchical namespace, inode-based metadata, page-granular storage, quota enforcement, and snapshot persistence—but diverge fundamentally in their execution environment and integration depth.

Table 8: Consolidated results summary across all six implementation requirements.

Req.	Category	Key Metric / Observation	Status
R1	VFS Registration	`nodev my_ramfs` in `/proc/filesystems`; no kernel panic on mount/umount	**PASS**
R2	Namespace Operations	All 8 metadata operations correct; `nlink` counts verified; `ENOTEMPTY` enforced	**PASS**
R3	Memory Allocation	`buff/cache` +11 MB after 8 MB write; inode_cache +4,931 after 5,000-file batch	**PASS**
R4	Read/Write I/O	Text append/overwrite and binary sparse-seek writes verified via `cat`/`hexdump`	**PASS**
A1	Quota Enforcement	`ENOSPC` at exactly 10 MB boundary; partial writes retained; space reclaimed on delete	**PASS**
A2	Snapshot Persistence	`USED_BYTES:8388608` correctly serialized and restored; residual quota enforced	**PASS**

`my_ramfs` is a *true kernel module*, registered with the Linux VFS layer through `register_filesystem` and operating entirely in kernel space. Every system call (`open`, `read`, `write`, `mkdir`, *etc.*) issued by any user-space process is serviced by the module's callbacks without any user-space intermediary. The data path traverses `address_space_operations`, the Page Cache, and the Slab allocator—the same subsystems used by production file systems such as EXT4 and XFS. This means that `my_ramfs` is not a simulation: it is a live participant in the kernel's memory management and VFS scheduling infrastructure.

The Python *RamFS* project, by contrast, is a *user-space simulation*. It re-implements the file-system stack as a set of Python classes (`VirtualFS`, `SuperBlock`, `Inode`) that mirror kernel concepts but run entirely in the Python interpreter. No actual mount point is created; no system call is intercepted; the "page" abstraction is a Python dictionary of byte strings rather than a physical 4 KB memory frame managed by the kernel's allocator. The project is self-described as educational and intentionally omits kernel integration, FUSE mounting, POSIX permissions, and concurrency control.

Table 9 provides a feature-by-feature comparison of both projects.

6.2 Strengths of `my_ramfs`

Authentic kernel integration. The most significant advantage of `my_ramfs` is that it operates at the correct level of abstraction. Because it registers with the Linux VFS, any POSIX-compliant tool—`ls`, `cat`, `dd`, `hexdump`, `stat`, `find`—can interact with the file system without modification. The experimental results in Section 5 were produced entirely with standard Linux utilities, demonstrating that the implementation is indistinguishable from a production file system from the perspective of user space. The Python *RamFS*, by definition, cannot be verified this way: its operations are exercised only through its own Python API and CLI, not through the kernel system call interface.

SMP safety and atomic quota accounting. The quota counter is maintained as an `atomic_t` and updated through the lock-free operations `atomic_add`, `atomic_sub`, and `atomic_read`. This design is correct under concurrent access from multiple CPUs—a property that the Python *RamFS* explicitly omits from its design goals. In a kernel-space implementation that may be accessed simultaneously by multiple processes across different CPU cores, the absence of SMP safety would constitute a data race, potentially corrupting the quota counter or triggering use-after-free conditions in the inode lifecycle. The three-phase enforcement algorithm described in Algorithm 2 is therefore not a stylistic choice but a correctness requirement.

Page Cache and Slab integration. By routing file data through the kernel Page Cache and metadata through the Slab allocator, `my_ramfs` participates correctly in the kernel's memory pressure response mechanism. The 11 MB increase in `buff/cache` observed during the 8 MB write experiment (Table 5) confirms that the allocator path functions as designed. The Python *RamFS* page model—a Python dictionary mapping page indices to byte strings—carries substantial interpreter-level overhead and does not interact with the OS memory manager in any observable way.

Binary I/O and hard link support. The `address_space_operations` pipeline handles arbitrary byte sequences without encoding assumptions. The binary file experiment confirmed correct sparse-write behavior under `hexdump`, validating non-textual payload storage that the Python *RamFS* explicitly excludes. Additionally, the `simple_link` callback enables hard links with correct `nlink` reference counting, a feature absent from the Python implementation.

6.3 Weaknesses of `my_ramfs`

Snapshot content completeness. The most significant limitation of `my_ramfs` relative to the Python *RamFS* is the depth of snapshot persistence. The current `myramfs_save_snapshot` serializes only the scalar quota counter (`used_bytes`) to a single-line text file. All file content, directory structure, inode metadata, and dentry names are irretrievably lost on unmount. The Python *RamFS*, by contrast, performs a complete serialization of the entire inode tree, all file pages, and the superblock into a structured JSON document. It incorporates a content-addressed deduplication scheme (16-character SHA-256 prefix keys, base64-encoded values) that reduces typical snapshot size from approximately 2.3× the raw payload (naive hex encoding) to 1.5–1.7×. Upon loading, the Python implementation fully reconstructs both metadata and file content, making its snapshots genuine persistence artifacts rather than administrative checkpoints.

Absence of a formal automated test suite. The Python *RamFS* is delivered with a structured test suite covering seven behavioral groups: basic operations, directory traversal, overwrite and deletion, quota boundary failure, snapshot round-trip recovery, error handling, and a stress test of thirty files across multiple directories. Each test produces a deterministic pass/fail outcome without human inspection, enabling reliable regression testing. `my_ramfs` relies entirely on manual validation through `dmesg` inspection and shell-command observation. While the experimental results in Section 5 are thorough and quantitative, the absence of an automated harness means that future refactoring of the module cannot be validated programmatically, increasing the risk of silent regressions.

No interactive demonstration layer. The Python *RamFS* provides an interactive CLI shell and three application-level demo programs (an in-memory cache, a structured logger, and a session manager). These artifacts lower the barrier of entry for non-expert audiences and serve simultaneously as integration tests and demonstration tools. `my_ramfs` has no equivalent layer; demonstration depends on familiarity with `insmod`/`rmmod` workflows and Linux shell utilities, which is appropriate for a kernel engineering audience but less accessible for pedagogical contexts.

Snapshot portability. The `my_ramfs` snapshot is a flat text file tightly coupled to the path `/tmp/ramfs_snapshot.txt` on the host filesystem, making it non-portable and difficult to inspect programmatically. The Python *RamFS* snapshot is a structured, human-readable JSON document that can be version-controlled, diffed, and loaded into a fresh instance on any Python 3 environment without knowledge of the underlying implementation.

6.4 Future Work

The preceding comparison directly motivates five high-priority directions for future development.

Full dcache-walk snapshot. The most impactful improvement would be to extend `myramfs_save_snapshot` to perform a depth-first traversal of the dentry cache via `d_walk`, serializing the complete namespace—inode numbers, permissions, timestamps, file sizes, and all page-frame contents—to a structured binary or JSON backing file. The `kill_sb` hook already provides the correct interception point; the missing piece is iterating `sb->s_root` and writing each inode's page frames sequentially to the backing file using `kernel_write`. This would bring `my_ramfs` to full parity with the Python *RamFS* on persistence completeness while retaining the kernel-space I/O efficiency advantage.

Automated kernel-space test harness. A `kselftest` or `kunit` test module should provide regression coverage equivalent to the Python *RamFS* test suite. At minimum, the harness should cover superblock initialization, inode allocation and eviction, all three phases of quota enforcement, dentry pinning under simulated memory pressure, and snapshot round-trip correctness. This would also enable integration with standard kernel CI pipelines.

FUSE bridge for cross-platform demonstration. Exposing the kernel module's behavior through a FUSE-backed user-space mirror would enable demonstration on systems where loading a custom kernel module is impractical (restricted cloud VMs, macOS development environments), while also providing a ground-truth oracle for regression testing. This mirrors the FUSE integration identified as the primary future direction by the Python *RamFS* authors, but approached from the kernel side downward rather than from user space upward.

Per-directory quota subtrees. Both implementations enforce a single flat quota per mount. A natural extension applicable to both projects would be per-directory quota subtrees, implemented via per-inode `myramfs_inode_info` structures in the kernel module. This would enable multi-tenant scenarios where different directories within the same mount

are subject to independent byte-usage limits, a common requirement in shared-storage environments.

Swap-backed pages and memory reclamation. The current Page Cache integration stores all file data in non-swappable pages, meaning the memory pressure imposed on the host equals the total bytes stored, up to the quota limit. A production-grade extension would implement a `.writepage` callback in `address_space_operations` to allow the kernel's swap daemon to reclaim pages under memory pressure—the approach used by `tmpfs`. This would allow `my_ramfs` to support workloads whose aggregate size exceeds physical RAM, at the cost of increased latency on swap-out paths.

7 Conclusion

The implementation and experimental evaluation of `my_ramfs` demonstrate that a fully functional, POSIX-compliant in-memory file system can be realized as a Linux kernel module of modest complexity by leveraging the VFS abstraction layer and composing existing kernel primitives. All six implementation requirements were validated on a live kernel: VFS registration succeeded without panics, all eight namespace operations behaved correctly with verified `nlink` counts, the Page Cache absorbed an 8 MB write at 527 MB/s with only 11 MB of `buff/cache` growth, the Slab allocator correctly tracked 5,000 new inodes without OOM events, the atomic quota precisely blocked writes at the 10 MB boundary (splitting a 6 MB + 6 MB write to 6 MB + 4 MB), and the snapshot hook serialized `USED_BYTES:8388608` to disk and restored it correctly across a remount cycle.

The implementation reflects the kernel development philosophy of "composing rather than reinventing," successfully delegating complex namespace operations to `simple_dir_operations`, `simple_rename`, and `simple_link`, while implementing only the security-critical intercept points (`write_begin`, `write_end`, `evict_inode`) from scratch. This strategy—combined with the atomic quota and snapshot features—produces a system that is simultaneously simple enough to serve as a pedagogical reference and robust enough to operate safely in a multi-threaded kernel environment.

Future work may extend `my_ramfs` in three directions: (i) per-directory quota subtrees using per-inode `myramfs_inode_info` structures, (ii) swap-backed pages to support workloads exceeding physical RAM (analogous to `tmpfs`), and (iii) a full dcache walk during `kill_sb` to serialize file content—not just administrative metadata—to the snapshot file, bridging the gap between pure volatility and full persistence.

8 Contribution

The technical labor, research, and report writing for this project were distributed as follows:

8.1 Yixin Liu (50%)

Code Contributions:

- **VFS Registration**: Implementation of `file_system_type` and superblock operations (`fill_super`, `put_super`).

- **Metadata Lifecycles**: Development of directory and file metadata operations, including path resolution (‘mkdir‘,‘create‘, ‘unlink‘, ‘rename‘)
- **Hard Link & Reference Counting**: hard link logic (`simple_link`).

Corresponding Report Contributions:

- **Abstract**: Motivation paragraph (in-memory file system as pedagogical tool), VFS compliance summary, and POSIX keywords.
- **Section 1** (Introduction & Motivation): Problem statement regarding block device overhead, pedagogical objectives of bypassing disk I/O, and summary of key contributions (bullet 1: VFS-compliant kernel module).
- **Section 2** (Background & Related Work): Analysis of motivation, VFS layer architecture, and related in-memory file systems.
- **Section 3.1** & **Section 3.2**: Documentation of kernel integration lifecycle and metadata modeling.
- **Section 5.1** & **Section 5.2**: VFS compliance testing results and namespace integrity validation.
- **Section 6**: Analysis of namespace operation correctness and pedagogical value.

8.2 Xinyang Wang (50%)

Code Contributions:

- **VFS Registration**: Implementation of `file_system_type` and superblock operations (`fill_super`, `put_super`).
- **Metadata Lifecycles**: Development of directory and file metadata operations, including path resolution (`mkdir`, `create`, `unlink`, `rename`).
- **Hard Link & Reference Counting**: Hard link logic (`simple_link`) and dentry reference counting via `dget()` to prevent premature eviction.

Corresponding Report Contributions:

- **Abstract**: Advanced features summary (atomic quota enforcement, snapshot persistence), experimental validation, and technical keywords (Slab allocator, Page Cache, OOM prevention).
- **Section 1** (Introduction & Motivation): Technical contributions paragraph (atomic quota and snapshot persistence as key contributions), and future work implications.
- **Section 3.3**–**Section 3.6**: Documentation of dual-subsystem memory virtualization, address space interfacing, atomic quota enforcement (Algorithm 2), and snapshot persistence mechanism (Figure **??**).
- **Section 4** (Experimental Setup): Design of I/O profiling methodology, defensive architecture testing, and persistence validation procedures.
- **Section 5.3**–**Section 5.6**: Kernel memory allocation profiling, quota enforcement results, and snapshot persistence validation.

- **Section 7**: Synthesis of performance results and future work directions (per-directory quotas, swap-backed pages, full content serialization).

References

[1] R. Gooch, "Overview of the Linux Virtual File System," Linux Kernel Documentation, 1999. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/vfs.html

[2] B. Geröfi, "Design and Implementation of the MINIX Virtual File System," Master's thesis, Dept. Computer Science, Vrije Universiteit Amsterdam, Amsterdam, Netherlands, 2006.

[3] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," in *Proc. USENIX Summer Technical Conference*, 1994, pp. 1–14.

[4] E. H.-M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "A New Design of In-Memory File System Based on File Virtual Address Framework," *IEEE Trans. Computers*, vol. 65, no. 10, pp. 2959–2972, Oct. 2016.

[5] S. Parekh, A. Deshpande, and N. N. Prasanth, "Time Administration of Virtual File System Operations," in *Proc. 2nd Int. Conf. Technological Advancements Computational Sciences (ICTACS)*, 2022, pp. 555–558.

[6] "ramfs, tmpfs," Kernel Tour — Linux Kernel Documentation, 2023. [Online]. Available: https://kernel-tour.org/archives/fs/ramfs.html

[7] "Creating Linux kernel modules in Rust — Rewriting a file system in a safe language," Hasso Plattner Institute, Univ. Potsdam, Potsdam, Germany, Tech. Rep., 2020.

[8] R. Love, *Linux Kernel Development*, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2010.

[9] J. Corbet, "The Linux VFS," *LWN.net*, May 2005. [Online]. Available: https://lwn.net/Articles/13325/

[10] "Virtual File System," The Linux Kernel Documentation, 2023. [Online]. Available: https://docs.kernel.org/filesystems/vfs.html

Table 9: Feature comparison between `my_ramfs` (kernel module) and Python *RamFS* (user-space simulation).

Feature	**`my_ramfs`**	**Python RamFS**
Execution environment	Kernel space (ring 0)	User space (Python VM)
VFS integration	Native (`register_filesystem`)	Simulated (Python API)
Real system call interception	**Yes**	**No**
Mountable via `mount` command	**Yes**	**No**
Page Cache integration	**Yes** (`alloc_pages`)	**No** (Python dict)
Slab allocator for metadata	**Yes** (`kmalloc`)	**No** (Python objects)
Atomic SMP-safe quota	**Yes** (`atomic_t`)	**No** (no locking)
Quota limit	10 MB (configurable)	1 MB (configurable)
Snapshot serializes file content	**No** (quota only)	**Yes** (full JSON + base64)
Snapshot restores quota state	**Yes**	**Yes**
Hard links	**Yes** (`simple_link`)	**No**
Binary file I/O	**Yes**	**No** (UTF-8 only)
Interactive CLI	**No**	**Yes**
Application demos	**No**	**Yes** (cache, logger, session)
Automated test suite	Manual (`dmesg`)	**Yes** (7 groups, `tests.py`)