Operating Systems

Course Project Report

RamFS

A Pure-Python In-Memory File System

Project Topic 12

Course	Operating Systems
Project Topic	Topic 12 — In-Memory File System (RamFS)
Institution	The Chinese University of Hong Kong, Shenzhen
Members	MA Guoqing (225040123) Yu Kuai (225040174)
Repository	https://github.com/YWh620/OS_Final
Date	April 2026

In-Memory File System · Pure Python · VFS-Style API · Snapshot Persistence · Quota Management

Group Members and Division of Work

1. Group Members

This project is jointly delivered by the following two members. Both members participated in the full development cycle covering implementation, testing, presentation, and report writing.

Student ID	Name	Primary Role
225040123	MA Guoqing	Core implementation and technical design
225040174	Yu Kuai	Testing, evaluation, CLI demonstration

2. Implementation Contribution

The implementation effort is divided so that one member focuses on the file-system kernel while the other member focuses on validation, demonstration, and the application layer that sits on top of the file system.

Student ID	Name	Implementation Contribution
225040123	MA Guoqing	Designed the overall architecture and implemented the central RamFS logic, including `VirtualFS`, inode and directory-tree operations, page-style data storage, superblock quota accounting, and snapshot serialization. Authored the technical explanation of the operation workflow.
225040174	Yu Kuai	Designed and executed functional tests, quota validation, snapshot validation, the CLI demonstration flow, the three application demos (cache, logger, session) plus the basic-operations walkthrough, microbenchmarks, experiment summaries, and the supporting evidence used in the project defense.

3. Report Writing Responsibility

The report writing is divided according to who is most familiar with each part of the system. Both members reviewed every section of the final report.

Student ID	Name	Sections Written
225040123	MA Guoqing	Sections 2 to 6 — Introduction, System Overview, RamFS Interface, Data Model, and Operation Semantics and Implementation Notes. Cross-checked technical descriptions against the repository implementation.
225040174	Yu Kuai	Section 7 and experiment-related content — Validation and Evaluation, Quota Experiment, Persistence Experiment, Application-Level Demonstrations, and Microbenchmark Summary.
Both	MA & Yu	Section 1 Abstract, Section 8 Limitations and Future Work, Section 9 Conclusion, References, and the final consistency review against the repository and presentation slides.

4. Declaration

We declare that the contents of this report and the associated source code are produced by the two members listed above. External material, where used as conceptual reference, is acknowledged in Section 10 References.

Contents

1. Abstract

RamFS is a pure-Python in-memory file system designed for an operating-systems course project. It models a compact subset of file-system concepts, including a virtual-file-system style interface, inode metadata, hierarchical directory entries, page-like memory blocks, superblock-level quota management, and JSON-based snapshot persistence. The project is not a kernel module and does not aim to provide a production-ready POSIX file system. Its purpose is to make file-system mechanisms explicit, runnable, and explainable.

The implementation provides a programmatic API through `VirtualFS`, an interactive command-line shell, automated functional tests, three application demos (in-memory cache, logger, and session manager), and a basic-operations walkthrough. The current test suite covers seven groups of behavior and validates basic operations, directory traversal, overwrite and deletion, quota failure, snapshot recovery, error handling, and stress creation of multiple files. Together they form a complete and self-contained system suitable for classroom demonstration.

2. Introduction

2.1 Scope

RamFS implements a user-space, memory-resident file system in Python. Its scope is intentionally educational: the system exposes important operating-system concepts without requiring kernel programming. It supports mounting, creating files and directories, reading, writing, appending, listing directories, querying metadata, enforcing memory quota, saving snapshots, and loading snapshots.

The project does not implement kernel integration, FUSE mounting, full POSIX permission checking, journaling, crash consistency, symbolic links, hard links, file descriptors, concurrent access control, or a real block-device layer. These omissions are deliberate so that the educational concepts remain visible without being obscured by engineering scaffolding.

2.2 Terminology

Term	Meaning in RamFS
VFS-style interface	High-level `VirtualFS` API hiding metadata and storage details.
SuperBlock	Global file-system state: mount status, quota, block size, root inode.
Inode	Metadata object representing either a regular file or a directory.
Page	A 4 KB memory block used to store a fragment of file content.
Snapshot	JSON representation of the superblock, inode tree, metadata, and pages.

2.3 Organization of This Report

Section 3 describes the system overview. Section 4 specifies the public interface. Section 5 defines the data model. Section 6 explains operation semantics and implementation notes. Section 7 validates correctness through tests and experiments. Sections 8 to 10 summarize limitations, conclusions, and references.

3. System Overview

3.1 Design Goals

RamFS aims to provide a compact file-system model with familiar operations such as `mount`, `mkdir`, `touch`, `read`, `write`, `ls`, `stat`, `rm`, `save_snapshot`, and `load_snapshot`. The design also makes operating-system concepts visible in code through explicit classes such as `VirtualFS`, `SuperBlock`, and `Inode`. The implementation is intentionally easy to run and demonstrate with no required external dependency, while still providing enough tests and examples to support a project defense.

The high-level goals can be summarized as follows:

- **Educational clarity.** Each layer of a file system maps to a Python class so the mechanism is observable.
- **Self-containment.** No external dependency is required; the system runs with the standard Python interpreter.
- **Testability.** Functional, quota, persistence, error, and stress behavior are covered by an automated suite.
- **Demonstrability.** An interactive CLI and three application demos make the system easy to present.

3.2 Architectural Layers

All user requests enter through the API layer. `VirtualFS` resolves paths and dispatches operations. Inodes hold metadata and directory hierarchy. The superblock owns global allocation state. Actual file payload remains in memory pages, and snapshots provide a portable recovery mechanism.

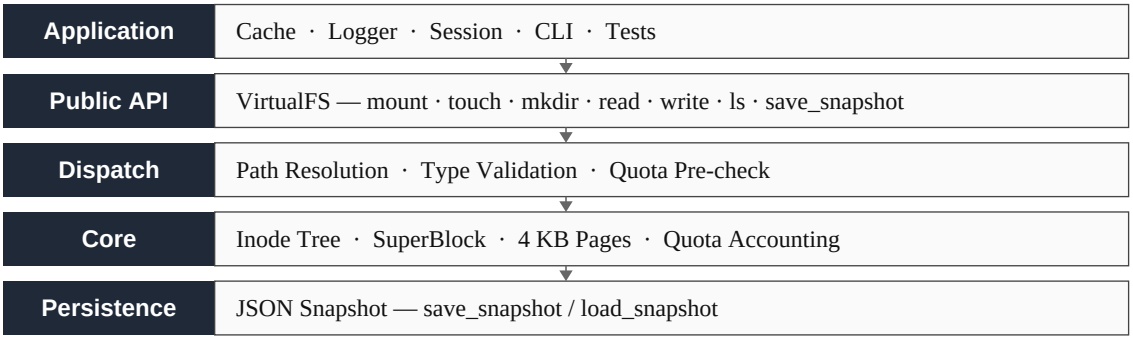

Figure 1. Layered architecture of RamFS.

3.3 Module Mapping

Module	Main Responsibility
`ramfs/fs.py`	Main `VirtualFS` implementation and public file-system operations.
`ramfs/core/superblock.py`	Global state, inode allocation, block allocation, and quota tracking.
`ramfs/core/inode.py`	File and directory metadata, page storage, and child mapping.
`ramfs/errors.py`	Domain-specific exceptions raised by the interface layer.
`cli.py`	Interactive shell interface used during demonstrations.
`tests.py`	Functional validation suite covering seven groups of behavior.
`ramfs/examples/`	Three application demos (cache, logger, session) and a basic-operations walkthrough built on the API.

4. RamFS Interface

The public API is concentrated in `VirtualFS` and grouped into four categories. The same operations are exposed in the CLI, in the test suite, and in the example applications, ensuring that there is exactly one source of behavior across the project.

4.1 Lifecycle and File Operations

Category	Operations	Purpose
Lifecycle	`mount`, `umount`	Initialize and deactivate the file system. Reject operations on an inactive instance.
File	`touch`, `write`, `append`, `read`, `rm`	Create, update, read, and delete regular files. Encoded as UTF-8 byte content.

4.2 Directory and Metadata Operations

Category	Operations	Purpose
Directory	`mkdir`, `ls`, `rmdir`	Maintain the namespace and produce directory listings. Reject removal of non-empty directories.
Metadata & persistence	`stat`, `get_usage`, `save_snapshot`, `load_snapshot`	Inspect metadata, observe quota usage, and persist or restore the entire file-system state.

5. Data Model

5.1 SuperBlock

The `SuperBlock` stores file-system-wide state: block size, maximum block count, allocated block count, inode counter, root inode, mount status, mount path, and creation time. Its allocation methods enforce the global quota before new page blocks are committed.

5.2 Inode

An `Inode` represents either a regular file or a directory and stores inode number, file type, permission bits, size, timestamps, file pages, and child directory entries. Distinguishing files from directories is done through the file-type field, which is checked at every operation entry.

5.3 Directory Entries

Directory entries are represented through the `children` dictionary of a directory inode. Each key is a name, and each value is a child inode, giving expected O(1) lookup per path component. The dictionary is also the natural unit of serialization in the snapshot format.

5.4 Page-Based File Content

File content is encoded as UTF-8 bytes and stored in page-like chunks of 4 KB. Each file inode maps page indexes to byte strings, making file size, block allocation, and the overlap between user data and quota directly visible during demonstration.

6. Operation Semantics and Implementation Notes

6.1 Path Resolution

Path resolution starts at the root inode and walks each path component through directory `children` maps. Intermediate components must exist and must be directories. The resolver returns the parent inode and final target name. Total cost is O(path depth) because each component lookup is dictionary-based, which keeps the hot path predictable even under stress tests.

6.2 Creation, I/O, Removal, and Snapshot

Area	Semantics
Creation	`touch` and `mkdir` resolve the parent, reject name conflicts, check quota, allocate an inode, and link it into the parent directory.
Read / write / append	Writes encode strings into bytes, allocate page blocks, store chunks in the inode page map, update size, and refresh modification time. Reads concatenate pages in page-index order.
Removal	`rm` frees file pages and releases superblock quota; `rmdir` removes only empty directories.
Snapshot	`save_snapshot` serializes metadata and page bytes into JSON. `load_snapshot` reconstructs the superblock and inode tree.

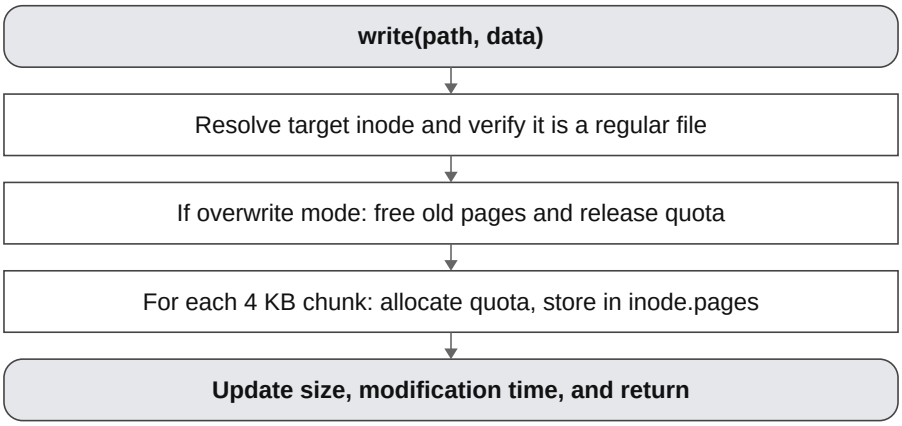

Figure 2. Control-flow of the write() operation.

6.3 Error Handling and Complexity

Aspect	Summary
Error classes	`FileNotFound`, `FileExists`, `IsADirectory`, `NotADirectory`, `DirectoryNotEmpty`, and `NoSpaceLeft`.
Complexity	`touch/mkdir/stat`: O(depth); `read/write/append`: O(depth + pages); `ls`: O(children); snapshots: O(inodes + pages).
Failure model	Path, type, and quota are checked before any mutation. `write` in overwrite mode is best-effort: it releases the existing pages first, so a midstream `NoSpaceLeft` can leave the file empty.

6.4 Snapshot Format

A snapshot is a single JSON document with a content-addressed layout. A top-level `blobs` map holds every distinct page exactly once, keyed by a 16-character SHA-256 prefix and stored as base64. The inode tree is written recursively from the `root`, each entry carrying its metadata, a `pages` map of page-index to blob hash, and a `children` map of nested child inodes. `load_snapshot` consumes the same shape and also accepts the legacy hex layout. Deduplication plus base64 (three bytes per four characters, vs hex two-per-byte) cuts a typical snapshot from about 2.3× the raw payload down to 1.5–1.7× while keeping it fully restorable and human-readable.

7. Validation and Evaluation

7.1 Test Environment

The implementation is written in Python and requires no external dependency for normal execution. The test suite is located in `tests.py` and is executed with `python3 tests.py`. Application examples under `ramfs/examples/` can be invoked individually for live demonstration.

7.2 Functional Validation

Test Group	Coverage	Result
Basic operations	Mount, directory creation, file creation, write, read, append.	Passed
Directory listing	Nested directories, `ls`, and `stat`.	Passed
File operations	Overwrite and deletion behaviors.	Passed
Memory quota	1 MB quota boundary and ENOSPC handling.	Passed
Persistence	Snapshot save, load, and content verification.	Passed
Error handling	Missing path, missing parent, wrong type, invalid removal.	Passed
Stress test	Thirty files across multiple directories.	Passed

7.3 Quota Experiment

A 1 MB file system accepts a 512 KB write and reports about 0.50 MB usage. A subsequent 600 KB write exceeds remaining capacity and is rejected with an ENOSPC-style error, confirming that quota is checked before new pages are committed.

7.4 Persistence Experiment

After creating a directory and a file, writing content, exporting a snapshot, and loading it into a new `VirtualFS` instance, the recovered content matches the original data. This confirms both metadata reconstruction and page-data restoration.

7.5 Application-Level Demonstrations

Demo	Reported Result
Cache	Two cached objects and a cache hit for user Alice.
Logger	Two log files with INFO, DEBUG, WARNING, and ERROR entries.
Sessions	Two sessions created, one removed, one remains active.

7.6 Microbenchmark Summary

Latencies measured during defense preparation with `time.perf_counter` on stock Python 3, matching the figures reported on the corresponding presentation slide.

Operation	Reported Latency
`write`	0.178 ms
`read`	1.279 ms
`save_snapshot`	1.030 ms

8. Limitations and Future Work

RamFS is a user-space simulation, not a mountable operating-system file system. It does not enforce POSIX permissions, does not support journaling or crash consistency, and does not protect shared state with locks. The page format is a Python dictionary of byte strings, which is convenient for an educational project but is not aligned with real block-device layouts.

Future work should extend the system in several directions:

- **FUSE integration** so the file system becomes mountable and accessible to standard tools.
- **Concurrency control** with fine-grained locks or transactional snapshots.
- **Richer metadata** covering UID, GID, permission masks, and access control entries.
- **Binary-safe APIs** so non-text payloads can be stored without UTF-8 assumptions.
- **Memory reclamation policies** such as LRU page eviction under tight quota.
- **Randomized stress tests** to surface corner cases beyond the current functional suite.

9. Conclusion

RamFS implements a compact and demonstrable in-memory file system in pure Python. It captures the essential structure of a file-system stack: a VFS-style public interface, hierarchical namespace resolution, inode metadata, page-like storage, superblock quota management, error handling, and snapshot persistence. The project is suitable for an operating-systems course because it turns kernel-level ideas into readable and testable user-space code, while still preserving the key vocabulary used in real file systems.

From the engineering perspective, the system is delivered as a single Python package with no external dependency. From the academic perspective, every operation is supported by observable evidence: a public API entry, a unit test, a CLI command, and an example application. This four-layer evidence chain makes the report claims directly verifiable against the repository and against the project presentation. The combination of the implementation, the test suite, and the application demos is therefore self-contained and reproducible by any reader who has Python 3 installed on a standard development machine.

10. References

[1] Project repository. *RamFS source code, tests, and examples.* https://github.com/YWh620/OS_Final.

[2] Linux Kernel Documentation. *Virtual File System concepts.*

[3] Linux kernel source tree, `fs/ramfs/`. Conceptual background on RAM-backed file systems.

[4] Python Standard Library Documentation: `json`, `dataclasses`, `enum`, and file I/O modules.