Nexus: A Desktop File Organizer and Disk Visualizer Demonstrating Operating System Concepts

Shivansh Gupta 230054, Vivek 230119, Shashwat 230127
Department of Computer Science and AI
Rishihood University, Sonipat, India
Email: shivansh.g23csai@nst.rishihood.edu.in, vivek.23csai@nst.rishihood.edu.in,
shashwat.23csai@nst.rishihood.edu.in

***Abstract*—This paper presents Nexus, a cross-platform desktop file organizer and disk usage visualizer built using Electron, React, and Node.js. The application is designed as a practical demonstration of core Operating System concepts including multi-process architecture, inter-process communication (IPC), worker threads for concurrent execution, file system operations, memory management, and security through sandboxing. Nexus provides users with an interactive file explorer, a sunburst-based disk usage chart, a trash management system with journaling, and a guided walkthrough for onboarding. The architecture deliberately mirrors real OS design patterns such as kernel-user space separation, system call interfaces, and permission-based access control. This report explains the motivation, system design, implementation details, and how each module maps to specific OS theory.**

***Index Terms*—Operating Systems, File Management, Inter-Process Communication, Worker Threads, Electron, Disk Visualization**

I. Introduction

Modern operating systems perform four fundamental tasks: process management, memory management, file system management, and resource protection [1]. While these concepts are typically taught through theory and small C programs, we wanted to build something that uses all of them together in a real, usable application.

Nexus is a desktop file manager that lets users browse directories, visualize disk usage, manage a trash bin, and preview files. But underneath the user interface, it is structured around OS principles. The main process acts like a kernel, the renderer is a sandboxed user-space program, and a worker thread handles heavy I/O without blocking the UI.

The goal was not just to build a file manager, but to build one where every architectural decision can be traced back to an OS concept. This report explains how we did that.

The rest of the paper is organized as follows: Section II covers related work and alternative approaches. Section III describes the system architecture. Section IV goes into implementation details. Section V maps each feature to OS concepts. Section VI shows results, and Section VII concludes the paper.

II. Related Work

Several file managers exist across operating systems. macOS ships with Finder, Windows has File Explorer, and Linux distributions use Nautilus (GNOME) or Thunar (XFCE). These are all native applications written in C, C++, or Objective-C, tightly coupled to their respective OS kernels.

We considered multiple technology choices before settling on our stack:

A. Framework Selection

Electron vs. Tauri: Tauri is a newer framework that uses the system's native webview instead of bundling Chromium. It produces smaller binaries and uses less memory. However, Electron gave us a clear multi-process model (main + renderer + worker) that directly maps to OS concepts like kernel mode, user mode, and threads [3]. This made it a better fit for a project that needed to demonstrate OS architecture.

Native vs. Web-based: A native app in Swift or C++ would have direct access to OS APIs. But the whole point was to implement the separation ourselves – to show IPC, sandboxing, and threading rather than relying on the OS to handle them transparently.

B. Existing Tools

Tools like WinDirStat and GrandPerspective provide disk visualization, but they are read-only and do not offer file management. Nexus combines both browsing and visualization in one application, with a trash system that uses journaling for crash consistency.

III. System Architecture

The application follows a multi-process architecture with three execution contexts, each with its own memory space and responsibilities.

A. Process Model

- **Main Process (Node.js):** This is the privileged process. It has full access to the file system via `fs`, `path`, and `child_process` modules. It creates the application window, registers IPC handlers, manages the custom

media:// protocol, and spawns worker threads. This is analogous to the OS kernel.

- **Renderer Process (Chromium):** This runs the React-based user interface inside a Chromium sandbox. It has `nodeIntegration: false` and `sandbox: true`, meaning it cannot import `fs`, `path`, or any Node.js module. It can only communicate with the main process through a whitelisted preload bridge. This is analogous to a user-space application.
- **Worker Thread:** When the user navigates to a directory, the main process spawns a short-lived worker thread (`fileScanner.worker.ts`) that recursively scans the directory tree and computes folder sizes. The worker runs on a separate OS thread, has its own V8 heap, and communicates results back via `postMessage`. This is analogous to a kernel thread.

B. IPC Flow

All file operations follow a request-response pattern that mirrors system calls:

1) Renderer calls `window.electron.listDirectory(path)`
2) Preload script forwards via `ipcRenderer.invoke("app:listDirectory", path)`
3) Main process handles with `ipcMain.handle("app:listDirectory", ...)`
4) Main process runs `fs.promises.readdir()` and `fs.promises.stat()`
5) Result (`DirEntry[]`) is serialized and sent back to the renderer

The renderer never touches the file system directly, just like how a user program cannot directly access disk hardware without going through system calls.

C. Security Model

The security model follows the principle of least privilege:

- **Context Isolation:** The preload script and the page JavaScript run in separate V8 contexts. The page cannot access preload variables.
- **Whitelisted API:** Only 20 specific functions are exposed via `contextBridge.exposeInMainWorld()`. The renderer gets a minimal interface, not raw Node.js access.
- **Sandbox:** The renderer process runs with `sandbox: true`, preventing any direct system access.

This is similar to how x86 CPUs use protection rings – Ring 0 (kernel) has full access, Ring 3 (user) is restricted, and the system call table controls what operations are available.

IV. Implementation

The project is organized into two main directories: `electron/` for the main process and workers, and `src/` for the renderer UI.

A. File Scanner (*electron/fileScanner.ts*)

This module performs a depth-limited recursive scan of a directory. It uses `fs.promises.readdir()` with `withFileTypes: true` to get both the name and type of each entry in one call. Each file is categorized by extension into groups: Image, Document, Code, Audio, Video, Archive, and Other. The output is a flat file list and a `byCategory` map used by the dashboard.

Depth limiting is important because without it, a scan of `/Users/` could open tens of thousands of file descriptors and exhaust the per-process limit (`ulimit -n`). We default to depth 2 for overview and depth 6 for targeted deep scans.

B. Worker Thread (*electron/fileScanner.worker.ts*)

The disk visualizer requires building a complete tree of directories with computed sizes. This involves calling `stat()` on every file, which is both CPU and I/O intensive. Doing this on the main thread would block the event loop and freeze the UI.

We solve this by spawning a `Worker` thread:

```
const worker = new Worker(
  path.join(__dirname,
    "fileScanner.worker.js"),
  { workerData: { dirPath, depth } }
);
worker.once("message", (tree) => {
  resolve(tree);
  worker.terminate();
});
```

Key design decisions:

- `workerData` is *copied* into the worker's heap using structured cloning. There is no shared memory, which avoids race conditions.
- The worker uses `parentPort.postMessage(tree)` to send the result back. This is message-passing IPC, the same model used by microkernels like Mach and QNX [2].
- The worker is created per scan request and terminated after sending the result. This is a short-lived thread pattern, similar to a thread pool task.

C. Trash System (*electron/main.ts*)

Instead of permanently deleting files with `fs.rm()`, Nexus implements a two-phase soft-delete with journaling:

1) **Move to trash:** Generate a UUID, record the original path, size, and timestamp in a `manifest.json` file, then `fs.rename()` the file into `~/.nexus-trash/files/`.
2) **Restore:** Read the manifest, find the entry by ID, `fs.rename()` back to the original path, and remove the manifest entry.
3) **Permanent delete:** `fs.rm()` the stored copy and remove the manifest entry.

The manifest acts as a journal – it records metadata *before* the actual file move. If the app crashes after writing the manifest but before moving the file, no data is lost. This is

the same concept as journaling in ext4 and NTFS file systems [2].

Additionally, Nexus reads the macOS system trash (`~/.Trash`) to show a unified view of both app-managed and system-managed trash items.

D. Preload Bridge (`electron/preload.ts`)

The preload script is the system call table of our application. It maps renderer function calls to IPC channels:

```
const electronAPI = {
  listDirectory: (dirPath) =>
    ipcRenderer.invoke(
      "app:listDirectory", dirPath),
  checkAccess: (path) =>
    ipcRenderer.invoke(
      "app:checkAccess", path),
  // ... 18 more functions
};
contextBridge.exposeInMainWorld(
  "electron", electronAPI);
```

The renderer sees `window.electron.listDirectory()` but has no idea it is making an IPC call. This abstraction is identical to how `libc` wraps system calls – the user program calls `read()`, which internally triggers a trap into kernel mode.

E. Permission Handling

On macOS, applications need Full Disk Access to read certain directories. Nexus checks this on every launch:

```
await fs.promises.access(
  path.resolve(targetPath),
  fs.constants.R_OK
);
```

`fs.constants.R_OK` checks the Unix permission bits (read bit) for the current process. If this fails with `EACCES`, the app shows a centered permission screen that links directly to macOS System Settings using deep-linking:

```
shell.openExternal(
  "x-apple.systempreferences:" +
  "com.apple.preference.security" +
  "?Privacy_AllFiles"
);
```

This demonstrates the OS access control model – the kernel checks UID/GID against the file's inode permission bits before allowing operations.

F. File Watching

Nexus uses `fs.watch()` to monitor the current directory for changes:

```
fs.watch(dirPath, () => {
  event.sender.send(
    "directory-changed", dirPath);
});
```

This uses the kernel's native file event system: `kqueue` on macOS and `inotify` on Linux [4]. The kernel maintains a watch list and notifies our process when any other process modifies the watched directory. This is interrupt-driven, not polling – zero CPU usage while waiting.

G. Custom Protocol (`media://`)

The renderer cannot use `file://` URLs due to security restrictions. We registered a custom `media://` protocol that the main process intercepts and resolves:

```
protocol.handle("media", (request) => {
  const decoded = decodeURIComponent(
    parsed.pathname.slice(1));
  return net.fetch(
    pathToFileURL(decoded).toString());
});
```

This is similar to a Virtual File System (VFS) layer in Linux – the renderer sees a `media://` namespace, but the main process translates it to real file paths, just like how `/proc` maps virtual paths to kernel data structures.

V. OS Concepts Demonstrated

Table I maps each OS concept taught in theory to its implementation in Nexus.

TABLE I
MAPPING OF OS CONCEPTS TO IMPLEMENTATION

OS Concept	Implementation in Nexus
Process Isolation	Main process (kernel) and renderer (user space) have separate memory spaces and capabilities.
IPC	All file operations use `invoke/handle` message passing. No shared memory between processes.
Threads	Worker thread for disk scanning runs on a separate OS thread with its own V8 heap.
File System APIs	`readdir`, `stat`, `access`, `rename`, `rm`, `watch` – direct use of POSIX-equivalent calls.
Scheduling	Worker thread offloads CPU work so the main event loop handles UI and IPC without blocking.
Memory Mgmt	Bounded thumbnail cache (3000 entries max). Separate heaps for main, renderer, and worker.
Security	Sandbox, context isolation, whitelisted API bridge, `R_OK` permission checks.
Journaling	Trash manifest records operations before file moves for crash consistency.
Virtual FS	Custom `media://` protocol translates virtual URLs to real file paths.
Kernel Events	`fs.watch` uses macOS `kqueue` / Linux `inotify` for interrupt-driven file notifications.

VI. Results

Nexus was successfully built and tested on macOS (Apple Silicon). The application provides the following features:

- File explorer with grid and list views, sorting by name, date, and size.
- Native thumbnails for images, code files, and documents.
- Interactive sunburst disk usage chart with click-to-drill-down.
- Auto-organize dashboard categorizing files by type.
- Trash bin with restore, permanent delete, and system trash integration.
- Guided walkthrough tour highlighting every feature.
- Permission screen for macOS Full Disk Access.
- Glassmorphism UI with frosted glass effects and smooth transitions.

The production build is packaged as a DMG installer using `electron-builder` with ad-hoc code signing.

VII. Conclusion

We built Nexus as a practical demonstration that OS concepts are not just theory – they are the foundation of every real application. The multi-process architecture mirrors kernel-user separation. IPC channels act as system calls. Worker threads provide concurrent execution without shared memory hazards. The trash system uses journaling for crash safety. And the security model enforces least privilege through sandboxing and a whitelisted API.

A. Limitations

The application currently only supports macOS. The ad-hoc code signing means users get a one-time Gatekeeper warning. Deep scans on very large directories (100,000+ files) can still take several seconds.

B. Future Work

- Cross-platform support for Linux and Windows.
- Apple Developer certificate for proper code signing and notarization.
- Duplicate file detection using content hashing.
- Batch file operations (multi-select rename, move, delete).
- Integration with cloud storage providers.

Acknowledgment

We would like to thank our faculty advisor for guidance on Operating Systems concepts and the Electron open-source community for their comprehensive documentation.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ: Wiley, 2018.

[2] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ: Pearson, 2015.

[3] (2024) Electron Documentation. [Online]. Available: https://www.electronjs.org/docs

[4] (2024) Node.js Documentation – Worker Threads. [Online]. Available: https://nodejs.org/api/worker_threads.html

[5] (2024) Node.js Documentation – File System. [Online]. Available: https://nodejs.org/api/fs.html

[6] (2024) Chromium Sandbox Design. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md

[7] IEEE Std 1003.1-2017, *IEEE Standard for Information Technology – Portable Operating System Interface (POSIX)*, IEEE, 2017.

[8] (2024) Vite Documentation. [Online]. Available: https://vitejs.dev/guide/