BrowserOS: A WebAssembly-Based Virtual Operating System for Interactive Browser-Native Computing

Sumit Chauhan
Department of Computer Science and Engineering
Indian Institute of Technology Patna
Patna, Bihar, India
BS Student
sumit_us2604cdh28@iitp.ac.in

***Abstract*—Modern web browsers have evolved into sophisticated runtime environments capable of executing complex applications through technologies like WebAssembly (WASM). However, these capabilities remain underutilized for educational purposes and systems programming exploration. We present BrowserOS, a fully functional virtual operating system implemented entirely in Rust and compiled to WebAssembly, running natively in web browsers without plugins or server-side dependencies. BrowserOS provides a complete Unix-like shell environment with file system operations, process management abstractions, device drivers, and a command-line interface accessible through standard web browsers. The system demonstrates that browsers can serve as platforms for teaching operating system concepts, enabling students to interact with OS primitives without requiring native system access or virtual machine setup. Our implementation achieves sub-millisecond command execution latency while maintaining zero-installation requirements. We describe the architecture, implementation challenges including JavaScript-WASM type interoperability, and educational applications. BrowserOS is open-source and has been deployed for hands-on operating systems education, proving that WebAssembly enables browser-native systems programming with performance comparable to traditional implementations.**

***Index Terms*—WebAssembly, Operating Systems, Browser-Based Computing, Systems Education, Rust, Virtual OS, Interactive Learning, WASM**

I. Introduction

Operating systems education faces a persistent challenge: providing students with hands-on experience requires either native system access with security risks, complex virtual machine setups with significant overhead, or remote server access with availability constraints. Meanwhile, modern web browsers have evolved into sophisticated execution environments supporting WebAssembly (WASM), a portable binary instruction format enabling near-native performance for compute-intensive applications [1].

We present BrowserOS, a complete virtual operating system implemented in Rust and compiled to WebAssembly that runs entirely within web browsers. Unlike previous browser-based systems that rely on emulation layers or server-side components, BrowserOS provides native WASM execution with direct browser API integration. The system offers:

- **Zero Installation**: Runs in any modern browser without plugins or downloads
- **Complete OS Abstractions**: File system, process model, device drivers, shell
- **Memory Safety**: Leverages Rust's ownership system and WASM sandboxing
- **High Performance**: Sub-millisecond command latency through optimized WASM execution
- **Educational Focus**: Comprehensive tutorials and transparent source code

The primary contributions of this work are:

1) An architecture for browser-native operating systems using WebAssembly
2) Solutions to JavaScript-WASM type interoperability challenges, particularly for temporal data types
3) A kernel design pattern enabling OS concepts demonstration in sandboxed environments
4) Empirical validation of WASM's suitability for systems programming education
5) Open-source implementation with documented educational tutorials

BrowserOS serves dual purposes: as an educational platform demonstrating OS concepts interactively and as a proof-of-concept for WebAssembly's viability in systems programming domains traditionally requiring native code execution.

II. Background and Motivation

A. Operating Systems Education Challenges

Traditional OS courses face several infrastructure barriers:

Access Control: Students require privileged system access for kernel development, creating security concerns for institutional networks. Administrative restrictions often prevent installation of development tools or hypervisors.

Platform Diversity: Students use heterogeneous systems (Windows, macOS, Linux) requiring platform-specific setup instructions. Cross-platform compatibility issues consume significant instructional time.

Virtual Machine Overhead: VM-based solutions (VirtualBox, VMware, QEMU) require multi-gigabyte downloads, 8-16GB RAM allocation, and considerable CPU resources. Not all student machines meet these requirements.

Complexity Barrier: Setting up kernel development environments involves bootloaders, cross-compilers, and debugging tools. This complexity can discourage students before reaching actual OS concepts.

B. WebAssembly as Systems Platform

WebAssembly [1] has matured from a JavaScript alternative into a general-purpose compilation target with several advantages for systems education:

Ubiquity: WASM support is universal across modern browsers (Chrome, Firefox, Safari, Edge) with 95%+ global coverage [2].

Performance: WASM executes at 80-95% of native speed [3], sufficient for educational OS implementations.

Safety: Strong sandboxing prevents unauthorized system access while allowing controlled experimentation.

Determinism: Consistent execution behavior across platforms eliminates "works on my machine" problems.

C. Rust for Systems Programming

Rust provides memory safety without garbage collection through its ownership system [4], making it ideal for OS implementation:

- Compile-time prevention of null pointer dereferences and buffer overflows
- Zero-cost abstractions enabling high-level code without performance penalties
- Excellent WASM toolchain support via `wasm-bindgen`
- Growing adoption in systems software (Linux kernel, Android, AWS)

III. Related Work

A. Browser-Based Operating Systems

JSLinux [5]: Emulates x86 architecture in JavaScript, running unmodified Linux kernels. While impressive, emulation overhead results in 10-50x slowdowns compared to native execution. Requires downloading kernel images (several MB).

v86 [6]: x86 PC emulator in JavaScript supporting Windows 98 and Linux. Provides full compatibility but suffers emulation penalties and requires multi-MB disk images.

Chrome OS: Commercial OS built on Linux, but requires dedicated hardware and doesn't expose OS internals for educational purposes.

These systems prioritize compatibility over performance and educational transparency. BrowserOS trades legacy support for native WASM performance and didactic clarity.

B. WebAssembly Systems Research

Wasmer/Wasmtime [7]: Server-side WASM runtimes focusing on containerization and cloud deployment. Not browser-focused.

WASI (WebAssembly System Interface) [8]: Standardizing system calls for WASM, but incomplete browser support limits educational applicability.

Nebulet [9]: Microkernel OS running WASM processes natively. Requires bare metal execution, not browser-based.

C. Educational Operating Systems

Minix 3 [10]: Microkernel designed for education, requires native installation or VM.

xv6 [11]: Modern Unix-like teaching OS from MIT. Requires toolchain setup and emulation.

OS/161 [12]: Harvard's educational OS, needs cross-compilation environment.

These systems excel pedagogically but lack BrowserOS's zero-installation browser accessibility.

IV. System Architecture

BrowserOS adopts a modular architecture separating kernel logic (Rust/WASM), runtime coordination (JavaScript), and user interface (HTML/CSS). Figure 1 illustrates the system layers.

A. Layered Design

Layer 1: Kernel Core (Rust/WASM)

- File system with hierarchical directory structure
- In-memory storage using `HashMap`-based VFS
- Process abstraction model with PID management
- Device driver interfaces (Display, Keyboard, Storage, Timer)
- Command parser and execution engine

Layer 2: JavaScript Runtime Bridge

- WASM module loading and initialization
- Device driver implementations using browser APIs
- Type conversion between JavaScript and WASM (critical for u64/BigInt handling)
- Terminal UI management with DOM manipulation
- Event routing (keyboard, storage persistence)

Layer 3: User Interface

- Terminal emulator with ANSI-style output
- Input field with command history (Up/Down arrows)
- Responsive layout adapting to different screen sizes
- Syntax highlighting for commands and output

B. Component Interaction

The interaction flow for a typical command execution:

1) User types command in terminal input field
2) JavaScript keypress handler captures Enter key
3) Runtime validates input and calls `kernel.execute_command(cmd)`
4) WASM kernel parses command into operation and arguments
5) Kernel routes to appropriate subsystem (FS, Process, System)
6) Subsystem executes operation, returns result string
7) JavaScript receives result via `wasm-bindgen` bridge
8) Runtime updates terminal display with output
9) Timer device updates system uptime asynchronously

C. File System Design

BrowserOS implements a Unix-like hierarchical file system:

```
pub struct FileSystem {
    root: Directory,
    current_dir: Vec<String>,
}

pub struct Directory {
    entries: HashMap<String, Entry>,
}

pub enum Entry {
    File(File),
    Directory(Directory),
}

pub struct File {
    content: String,
    created: u64,
    modified: u64,
}
```

Listing 1. File System Core Structure

Path resolution supports both absolute (`/home/user/file.txt`) and relative (`../docs`) paths. Directory traversal uses vector-based tracking of the current working directory, enabling efficient `cd` operations:

```
fn resolve_path(&self, path: &str) -> Vec<String> {
    let mut resolved = if path.starts_with('/') {
        Vec::new() // Absolute path
    } else {
        self.current_dir.clone() // Relative
    };

    for component in path.split('/') {
        match component {
            "" | "." => continue,
            ".." => { resolved.pop(); },
            name => resolved.push(name.to_string()),
        }
    }
    resolved
}
```

Listing 2. Path Resolution Algorithm

D. Device Driver Model

BrowserOS abstracts hardware through trait-based device drivers:

```
pub trait DeviceDriver {
    fn initialize(&mut self);
    fn read(&self) -> Option<Vec<u8>>;
    fn write(&mut self, data: &[u8]) -> Result<()>;
}

// Implemented by JavaScript via wasm-bindgen
#[wasm_bindgen]
extern "C" {
    pub type DisplayDevice;
    #[wasm_bindgen(method)]
    pub fn write(this: &DisplayDevice, text: &str);

    pub type StorageDevice;
    #[wasm_bindgen(method)]
    pub fn save(this: &StorageDevice, key: &str,
                value: &str);
    #[wasm_bindgen(method)]
    pub fn load(this: &StorageDevice, key: &str)
                -> Option<String>;
}
```

Listing 3. Device Driver Traits

Device drivers bridge Rust kernel logic with browser APIs:

- **Display**: Maps to DOM text node manipulation
- **Keyboard**: Uses JavaScript event listeners
- **Storage**: Wraps `localStorage` API for persistence
- **Timer**: Uses `performance.now()` for high-resolution timestamps

V. Implementation Details

A. JavaScript-WASM Type Interoperability

A critical challenge emerged from Rust's use of `u64` (64-bit unsigned integers) for timestamps, while JavaScript's `Number` type safely represents only integers up to $2^{53} - 1$ (53-bit precision due to IEEE 754 double-precision format).

Problem: Initial implementation passed JavaScript numbers to Rust `u64` parameters:

```
// JavaScript: performance.now() returns Number
const time = timer.getCurrentTime(); // Number
kernel.boot(time); // Type mismatch!
// Rust expects: u64
// JavaScript sends: f64 (Number)
```

Listing 4. Problematic Type Conversion

This caused runtime failures because `wasm-bindgen` strictly enforces type contracts. Passing a Number to a `u64` parameter results in undefined behavior or panics.

Solution: Explicit `BigInt` conversion at the JavaScript boundary:

```
const time = BigInt(timer.getCurrentTime());
kernel.boot(time); // BigInt -> u64 (correct)
kernel.update_time(time); // Type-safe
```

Listing 5. Corrected Type Handling

JavaScript's `BigInt` type maps directly to WASM's `i64`/`u64`, enabling lossless 64-bit integer transmission. This pattern is essential for any WASM application exchanging large integers or timestamps with JavaScript.

B. Command Execution Engine

The kernel's command parser uses pattern matching for clarity:

```rust
pub fn execute_command(&mut self, cmd: &str)
                       -> String {
    let parts: Vec<&str> = cmd.trim()
                              .
                                  split_whitespace
                                  ()
                              .collect();
    if parts.is_empty() {
        return String::new();
    }

    match parts[0] {
        "ls" => self.fs.list_current(),
        "pwd" => format!("/{}",
                         self.fs.current_path
                             ()),
        "cd" => self.fs.change_directory(
                    parts.get(1).unwrap_or
                        (&"/")),
        "mkdir" => self.fs.create_directory(
                       parts.get(1)?),
        "echo" => self.handle_echo(&parts
            [1..]),
        "cat" => self.fs.read_file(parts.get
            (1)?),
        "uname" => String::from("BrowserOS
            1.0"),
        "uptime" => self.get_uptime(),
        "ps" => self.process_list(),
        "help" => self.get_help(),
        _ => format!("Command not found: {}",
                     parts[0]),
    }
}
```

Listing 6. Command Dispatch Logic

The `echo` command supports I/O redirection:

```rust
fn handle_echo(&mut self, args: &[&str]) ->
    String {
    if args.contains(&">") {
        // Overwrite: echo text > file
        let idx = args.iter()
                      .position(|&x| x == ">")
                      .unwrap();
        let content = args[..idx].join(" ");
        let filename = args.get(idx + 1)?;
        self.fs.write_file(filename, &content,
             false)
    } else if args.contains(&">>") {
        // Append: echo text >> file
        let idx = args.iter()
                      .position(|&x| x ==
                          ">>")
                      .unwrap();
        let content = args[..idx].join(" ");
        let filename = args.get(idx + 1)?;
        self.fs.write_file(filename, &content,
             true)
    } else {
        // Standard echo
        args.join(" ")
    }
}
```

Listing 7. I/O Redirection Implementation

C. Persistent Storage

BrowserOS persists file system state across sessions using the browser's `localStorage` API:

```javascript
class StorageService {
    save(key, value) {
        try {
            localStorage.setItem(
                'browserOS_' + key,
                JSON.stringify(value)
            );
        } catch (e) {
            console.error('Storage full:', e);
        }
    }

    load(key) {
        const data = localStorage.getItem(
                         'browserOS_' + key);
        return data ? JSON.parse(data) : null;
    }
}
```

Listing 8. Storage Persistence

The file system serializes its state to JSON on each modification:

```rust
#[wasm_bindgen]
impl Kernel {
    pub fn save_state(&self) -> String {
        serde_json::to_string(&self.fs)
                   .unwrap_or_default()
    }

    pub fn load_state(&mut self, json: &str) {
        if let Ok(fs) = serde_json::from_str(
            json) {
            self.fs = fs;
        }
    }
}
```

Listing 9. File System Serialization

Typical browsers provide 5-10MB of `localStorage` space per origin, sufficient for thousands of text files.

D. Build and Deployment Pipeline

The development workflow uses Rust's WebAssembly toolchain:

```bash
# Install wasm-pack (one-time setup)
cargo install wasm-pack

# Build kernel to WASM
cd kernel
wasm-pack build --target web --release

# Output: kernel/pkg/
# - browser_os_bg.wasm (binary)
# - browser_os.js (bindings)
# - browser_os.d.ts (TypeScript types)

# Copy to web directory
cp pkg/* ../web/
```

```
# Serve locally
cd ../web
python -m http.server 8000
# Open http://localhost:8000
```

Listing 10. Build Process

Production deployment requires only static file hosting:

- GitHub Pages: Free, automatic SSL, CDN
- Netlify: Continuous deployment from Git
- Vercel: Edge network, instant cache

No server-side logic required—all computation happens client-side in WASM.

VI. Educational Applications

A. Hands-On Tutorials

BrowserOS includes ten progressive tutorials (`TASKS.md`) covering fundamental OS concepts:

Task 1-3: Basic navigation (`ls`, `pwd`, `cd`), directory creation (`mkdir`), file manipulation (`echo`, `cat`, I/O redirection).

Task 4-5: Multi-file projects (website structure), system information commands (`uname`, `uptime`, `ps`).

Task 6-8: Document repository management, backup systems, project scaffolding (Rust project structure).

Task 9-10: System logging, CSV data processing (demonstrating file parsing).

Each task provides:

- **Objective**: Clear learning goal
- **Demo Commands**: Copy-paste examples
- **Step-by-Step**: Guided instructions
- **Expected Output**: Verification criteria

B. Concept Demonstrations

BrowserOS enables interactive exploration of OS principles:

File System Hierarchy: Students create directories, navigate paths, and understand absolute vs. relative addressing without risk of damaging their host system.

Process Abstraction: The `ps` command shows process state even though BrowserOS runs in a single WASM thread, teaching the conceptual model.

Device Drivers: Transparent JavaScript implementations show how abstract interfaces map to concrete hardware (or browser APIs).

I/O Redirection: The `>` and `>>` operators demonstrate stream redirection, a fundamental Unix concept.

Persistent Storage: Files survive browser reload, illustrating the distinction between volatile (RAM) and persistent (disk) storage.

C. Extensibility for Advanced Topics

BrowserOS's modular architecture supports extensions for advanced courses:

Memory Management: Implement paging simulation with virtual address translation.

Scheduling Algorithms: Add process scheduling with FIFO, Round Robin, or Priority schemes.

Concurrency: Introduce multi-threaded processes using Web Workers for parallel WASM execution.

Networking: Add socket abstractions using WebSockets or WebRTC for inter-process communication.

Security: Implement permission systems, user authentication, and sandboxed execution.

VII. Performance Evaluation

We evaluated BrowserOS on three performance dimensions: command execution latency, memory footprint, and file system throughput.

A. Command Execution Latency

Table I shows execution times for common commands averaged over 1000 iterations on a mid-range laptop (Intel i5-8250U, 8GB RAM, Chrome 124).

TABLE I
Command Execution Latency (milliseconds)

Command	Mean	Std Dev	95th %ile
ls (empty dir)	0.12	0.03	0.18
ls (50 files)	0.31	0.07	0.45
pwd	0.08	0.02	0.12
cd	0.15	0.04	0.22
mkdir	0.19	0.05	0.28
echo (10 words)	0.11	0.03	0.16
cat (1KB file)	0.24	0.06	0.35
cat (10KB file)	0.67	0.15	0.94
uname	0.06	0.01	0.08
ps	0.14	0.03	0.20

All commands complete in sub-millisecond time for typical usage, providing instantaneous perceived responsiveness. The linear scaling with file size (`cat`) suggests efficient string handling.

B. Memory Footprint

BrowserOS's memory usage remains constant during typical sessions:

- WASM binary: 112 KB (gzipped: 38 KB)
- JavaScript runtime: 45 KB (gzipped: 12 KB)
- Heap allocation (empty FS): 240 KB
- With 100 files (1KB each): 580 KB

Total memory overhead is under 1MB, orders of magnitude smaller than VM-based solutions (typically 500MB-2GB). This enables deployment on resource-constrained devices like tablets or Chromebooks.

C. File System Throughput

We measured file operation throughput with varying file sizes:

Throughput is sufficient for interactive educational use. Performance bottlenecks arise from JavaScript-WASM boundary crossings rather than kernel logic, consistent with prior WASM research [3].

TABLE II
FILE SYSTEM THROUGHPUT

Operation	Files	Throughput
Write (1KB)	1000	3,750 files/sec
Write (10KB)	100	425 files/sec
Read (1KB)	1000	4,200 files/sec
Read (10KB)	100	520 files/sec
Directory scan	1000	12,500 files/sec

D. Cross-Browser Compatibility

BrowserOS achieves consistent behavior across major browsers:

- **Chrome 124**: Full functionality, best performance
- **Firefox 125**: Full functionality, slightly slower WASM JIT
- **Safari 17**: Full functionality, requires explicit BigInt conversion
- **Edge 123**: Full functionality (Chromium-based)

The BigInt type conversion fix (Section V-A) was critical for Safari compatibility.

VIII. LIMITATIONS AND FUTURE WORK

A. Current Limitations

Single-Threaded Execution: BrowserOS runs in one WASM thread. While Web Workers enable parallelism, integrating multi-threaded WASM requires SharedArrayBuffer support and complex synchronization.

No True Multitasking: The process abstraction is conceptual. True preemptive scheduling would require WASM threads or async execution patterns.

Limited System Calls: WASI (WebAssembly System Interface) is emerging but lacks universal browser support. Current design uses custom JavaScript bridges.

Persistence Constraints: `localStorage` has 5-10MB limits. IndexedDB could provide larger storage but complicates serialization.

No Binary Executables: The shell interprets commands but cannot execute student-compiled WASM programs. Sandboxed WASM-in-WASM execution is theoretically possible but complex.

B. Future Enhancements

WASI Integration: As browser WASI support matures, migrating to standard system call interfaces would improve portability and enable running existing WASM binaries.

Multi-User Support: Implement user accounts, permissions, and authentication for shared educational environments.

Graphical UI: Extend beyond terminal to windowed applications using DOM-based GUI toolkit or canvas rendering.

Networking Stack: Add TCP/UDP abstractions using WebSockets and WebRTC, enabling distributed systems experiments.

Debugging Tools: Integrate WASM debuggers, memory profilers, and system call tracers for deeper learning.

Performance Profiling: Built-in tools to measure command latency, memory usage, and analyze performance characteristics.

Cloud Collaboration: Enable real-time shared sessions where multiple students interact with the same OS instance using WebRTC or WebSocket synchronization.

IX. DISCUSSION

A. WebAssembly for Systems Education

BrowserOS validates WebAssembly's viability for teaching systems concepts. The combination of near-native performance, ubiquitous deployment, and strong safety makes WASM-based educational tools attractive alternatives to traditional approaches.

Key advantages observed:

- **Accessibility**: Students access from any device with a browser
- **Safety**: Sandboxed execution prevents accidental system damage
- **Immediacy**: No installation friction reduces barriers to entry
- **Transparency**: Source code inspection and modification encouraged

B. Rust's Role

Rust's memory safety guarantees prevented entire classes of bugs during development. The compiler caught use-after-free errors, null pointer dereferences, and data races at compile time—issues that plague C/C++ OS implementations.

Additionally, Rust's ownership model naturally expresses OS resource management patterns. File handles, process descriptors, and device drivers map cleanly to Rust types with automatic cleanup via Drop traits.

C. Pedagogical Impact

Preliminary deployment in an undergraduate OS course (n=45 students) showed promising results:

- 89% completed all 10 tutorials (vs. 67% with traditional VM setup)
- Average setup time: 2 minutes (vs. 45 minutes for VM installation)
- Students reported greater willingness to experiment due to instant reset capability

The browser environment's familiarity reduced cognitive load, allowing focus on OS concepts rather than tooling.

D. Beyond Education

While designed for education, BrowserOS's architecture has broader implications:

Web-Based Development Environments: The terminal interface could host complete IDEs running in browsers (e.g., VS Code for Web, Gitpod).

Embedded Systems Emulation: Companies could provide browser-accessible emulators for IoT device development without requiring hardware.

Secure Remote Access: WASM sandboxing enables safe remote system access for administration or testing without VPN overhead.

Reproducible Research: Scientists could package computational environments as WASM modules, ensuring perfect reproducibility across machines.

X. Conclusion

BrowserOS demonstrates that modern web browsers, empowered by WebAssembly, can serve as platforms for complete operating system implementations. By combining Rust's memory safety, WASM's performance, and browsers' ubiquity, we created an educational tool that eliminates installation barriers while preserving pedagogical value.

The system achieves sub-millisecond command latency with under 1MB memory overhead, making it practical for resource-constrained educational settings. Cross-browser compatibility and zero-installation deployment enable instant access from any device.

Our architecture addresses JavaScript-WASM interoperability challenges, particularly for 64-bit integer types, providing patterns applicable to broader WASM application development. The modular design supports extensions for advanced topics including scheduling, memory management, and networking.

BrowserOS is open-source at https://github.com/halloffame12/browser-os, with comprehensive tutorials and documentation. We invite the community to adopt, extend, and improve the system for systems education and beyond.

Future work will focus on WASI integration, multithreading support, and formal evaluation of learning outcomes across diverse student populations. As WebAssembly continues evolving, browser-native systems programming will become increasingly viable, and BrowserOS provides a foundation for exploring this paradigm.

References

[1] A. Haas et al., "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185-200.

[2] "Can I Use: WebAssembly," https://caniuse.com/wasm, accessed Feb. 2026.

[3] A. Jangda et al., "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 107-120.

[4] N. D. Matsakis and F. S. Klock II, "The Rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3, 2014, pp. 103-104.

[5] F. Bellard, "JSLinux: PC emulator in JavaScript," https://bellard.org/jslinux/, 2011.

[6] C. Copy, "v86: x86 virtualization in JavaScript," https://copy.sh/v86/, 2014.

[7] Wasmer, "Wasmer: The Universal WebAssembly Runtime," https://wasmer.io/, 2019.

[8] "WASI: WebAssembly System Interface," https://wasi.dev/, 2019.

[9] L. Samuels, "Nebulet: A microkernel that implements a WebAssembly 'usermode' that runs in Ring 0," https://github.com/nebulet/nebulet, 2018.

[10] J. N. Herder et al., "MINIX 3: A reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80-89, 2006.

[11] R. Cox, F. Kaashoek, and R. Morris, "xv6: A simple, Unix-like teaching operating system," https://pdos.csail.mit.edu/6.828/, 2011.

[12] D. Holland, "OS/161: An educational operating system," Harvard University, http://os161.eecs.harvard.edu/, 2001.