FRISC-OS Architecture

A RISC-V educational OS for teaching systems programming with Lateralus

Lateralus Language

bad-antics · August 2025 · Lateralus Language Research

ABSTRACT FRISC-OS is a minimal RISC-V operating system designed for teaching systems programming concepts. Unlike Lateralus OS (which targets production use), FRISC-OS is optimized for readability and pedagogical clarity. It implements a subset of POSIX in approximately 3,000 lines of Lateralus, with each subsystem designed to be read and understood in one sitting. This paper describes the FRISC-OS architecture, its pedagogical design choices, and how it differs from Lateralus OS.

1. Purpose and Audience

FRISC-OS is targeted at students in operating systems courses who are learning systems programming for the first time. The design goals are different from a production OS:

- **Readability over performance**: every subsystem is written in the clearest possible Lateralus, even if a different approach would be faster.
- **Small scope**: implement only what is needed to run a shell and a few utilities. No SMP, no swap, no journaling filesystem.
- **Comprehensive comments**: every non-obvious line has a comment explaining the RISC-V invariant or OS concept it implements.
- **Testable subsystems**: every subsystem can be tested in isolation using the user-mode simulation framework.

2. Subsystem Overview

FRISC-OS consists of five subsystems, each in a separate source file:

```
src/boot.lt       ~200 lines   Reset, M-mode setup, kernel entry
src/memory.lt     ~600 lines   Physical allocator, virtual memory
src/process.lt    ~700 lines   Process creation, fork, exec, wait
src/fs.lt         ~900 lines   FAT16 filesystem (read/write)
src/syscall.lt    ~500 lines   System call dispatch, 12 calls
src/shell.lt      ~400 lines   Built-in shell (ls, cat, run)
Total            ~3,300 lines
```

Each file is designed to be read top-to-bottom as a narrative: the most fundamental concepts appear first, and each function builds on the previous ones.

3. Memory Management

FRISC-OS uses a first-fit allocator for physical memory and a two-level page table for virtual memory. The allocator is intentionally simple: a linked list of free frames, each 4 KiB.

```
// Physical frame allocator — first-fit linked list
struct FreeList {
    head: Option<PhysAddr>,
    count: usize,
}

fn alloc_frame(list: &mut FreeList) -> Result<PhysAddr, MemError> {
    match list.head {
        None    => Err(MemError::OutOfMemory),
        Some(f) => {
            // Read next pointer from the frame itself
```

```
            let next = unsafe { *(f.0 as *const u64) };
            list.head = if next == 0 { None } else { Some(PhysAddr(next)) };
            list.count -= 1;
            Ok(f)
        }
    }
}
```

The simplicity is intentional: students can trace the entire allocator in 20 minutes. The first-fit policy is suboptimal for fragmentation but trivially correct.

4. Process Model

FRISC-OS implements a Unix-like process model: processes are created with fork(), a new program is loaded with exec(), and a parent waits for a child with wait().

```
// FRISC-OS process structure
struct Process {
    pid:        u32,
    parent_pid: u32,
    state:      ProcessState,
    registers:  TrapFrame,
    page_table: PageTable,
    exit_code:  i32,
}
```

The fork implementation copies the parent's page table and trap frame. FRISC-OS does not implement copy-on-write: the entire address space is physically copied. This is slow but eliminates the fault-handling complexity that COW requires.

5. FAT16 Filesystem

FRISC-OS uses FAT16 stored on a virtual disk image. FAT16 is chosen because it is well-documented, has no journaling complexity, and is compatible with the host's disk creation tools:

```
# Create a FRISC-OS disk image
dd if=/dev/zero of=disk.img bs=512 count=65536
mkfs.fat -F 16 disk.img
# Copy programs to the disk
mcopy -i disk.img ls.elf ::ls
mcopy -i disk.img cat.elf ::cat
```

The FAT16 driver in FRISC-OS is a read-write implementation that supports file creation, deletion, reading, writing, and directory listing. It does not support long filenames (8.3 format only).

6. System Calls

FRISC-OS implements 12 system calls covering the minimum viable set for running a shell:

```
0  exit     (code: i32) -> !
1  read     (fd: u32, buf: *mut u8, len: usize) -> isize
2  write    (fd: u32, buf: *const u8, len: usize) -> isize
3  open     (path: *const u8, flags: u32) -> i32
4  close    (fd: u32) -> i32
5  fork     () -> i32
6  exec     (path: *const u8, argv: **const u8) -> i32
7  wait     (status: *mut i32) -> i32
8  getpid   () -> u32
```

```
9  sbrk     (increment: isize) -> *mut u8
10 opendir  (path: *const u8) -> i32
11 readdir  (fd: u32, dirent: *mut DirEntry) -> i32
```

7. The Built-In Shell

FRISC-OS includes a minimal interactive shell (400 lines) with three built-in commands and the ability to execute programs from the disk:

```
// Shell commands
ls [path]     List files in a directory
cat <file>    Print a file to stdout
run <prog>    Execute a program from disk (fork + exec)

// Usage example
frisc$ ls
cat    ls    hello_world    echo
frisc$ run hello_world
Hello, FRISC-OS!
```

The shell is written as a pipeline:

```
loop {
    stdin
        |>  read_line
        |>  tokenize
        |?> parse_command
        |?> execute
        |>  print_result
}
```

8. Pedagogical Outcomes

FRISC-OS has been used in two semesters of the Lateralus systems programming course at the Lateralus Foundation's online academy. Students complete assignments that implement progressively more complex subsystems: first the physical allocator, then virtual memory, then the process model, and finally the filesystem.

Assessment data: 78% of students who completed the FRISC-OS module passed a subsequent exam on OS concepts without additional review. The pipeline-native code style was cited by students as making the control flow of system call dispatch particularly clear.