Ripgrep messages.rs: Thread-Safe Output Infrastructure

What This File Does

The messages.rs module provides thread-safe error and warning output for ripgrep. It's only 139 lines, but it solves a subtle problem: when multiple threads write to stderr simultaneously, their output can interleave at the character level, producing garbled messages.

The module also tracks global state: whether messages should be displayed at all, and whether any errors occurred during execution. This error flag ultimately affects ripgrep's exit status.

Section 1: The Problem Being Solved

When ripgrep runs in parallel mode, multiple threads search files simultaneously. Each thread might encounter errors — permission denied, file not found, encoding issues. If threads write error messages concurrently, the output becomes unreadable.

Consider two threads printing errors at the same time. Without synchronization, you might see: "rg: file1.trg:x t:f ilpee2r.mtixsts:i opne rdmenisesdi oenn ideed" — a jumbled mess of two messages interleaved character by character.

Standard library print macros don't prevent this. They lock stderr for each print call, but two writeln! calls from different threads can still interleave between calls. The solution requires holding a lock across the entire message.

See: Companion Code Section 1

Section 2: The Global State

Three AtomicBool values track global state. MESSAGES controls whether error messages display at all — the --no-messages flag disables them. IGNORE_MESSAGES controls whether ignore-rule parse errors display — useful for debugging gitignore issues. ERRORED tracks whether any error occurred during execution.

All three start as false and get configured once during argument parsing. After that, MESSAGES and IGNORE_MESSAGES are read-only. ERRORED starts false and gets flipped to true if any error occurs, never back to false.

The choice of AtomicBool over other synchronization primitives reflects the access pattern. These flags are set once, then read many times from many threads. Atomic operations are the lightest-weight synchronization that provides the necessary guarantees.

See: Companion Code Section 2

Section 3: Memory Ordering

All atomic operations use Ordering::Relaxed, the weakest memory ordering. This might seem surprising for shared state, but it's correct here.

Relaxed ordering guarantees atomicity — reads and writes won't tear — but provides no synchronization between threads. A write on one thread might not be immediately visible to reads on other threads.

This works for ripgrep because the flags are set during single-threaded initialization, before any parallel searching begins. By the time multiple threads read the flags, the writes have long since completed and propagated. There's no race between setting and reading.

For ERRORED, relaxed ordering means a late-arriving error might not be visible when exit status is determined. In practice, this doesn't matter — the error message itself was printed, and missing the exit status change is a minor inconsistency in an already-failing run.

See: Companion Code Section 3

Section 4: The eprintln_locked Macro

The eprintln_locked macro is the foundation of thread-safe output. Its implementation reveals a subtle trick: it locks stdout before writing to stderr.

This seems backwards until you consider how ripgrep's parallel search works. The search threads use termcolor to write colored output to stdout. Termcolor acquires stdout's lock when writing. Meanwhile, error messages go to stderr.

If both stdout and stderr point to the same terminal, interleaving can occur even between different streams. By acquiring stdout's lock first, error messages wait for any in-progress search output to complete. The output appears sequential even though it's generated concurrently.

The "abstraction violation" comment acknowledges this coupling. The macro knows implementation details of the search code — specifically that it uses termcolor and stdout's lock. If that changed, this macro would need updating.

See: Companion Code Section 4

Section 5: Broken Pipe Handling

Within eprintln_locked, broken pipe errors receive special treatment. If writing fails with BrokenPipe, the process exits with code zero. Any other write error exits with code two.

Broken pipe occurs when the receiving end of a pipe closes. If someone runs "rg pattern 2>&1 | head -5", the head command exits after five lines. The pipe to head's stdin breaks. Ripgrep's next write to stdout or stderr fails with BrokenPipe.

Treating this as success (exit code zero) matches Unix conventions. The user got what they asked for — head received its five lines. Ripgrep's inability to write more isn't an error in the meaningful sense.

The exit calls within the macro are somewhat unusual. Macros typically expand to expressions, not control flow. But here, a failed write is unrecoverable — there's nothing sensible to do except exit.

See: Companion Code Section 5

Section 6: The message Macro

The message macro conditionally prints messages based on the MESSAGES flag. It's a thin wrapper around eprintln_locked that checks the flag first.

When --no-messages is passed, MESSAGES is false, and this macro becomes a no-op. The formatting expressions still get evaluated (Rust macros don't short-circuit argument evaluation), but no I/O occurs.

This conditional display matters for scripting. When ripgrep runs as part of a pipeline, error messages might interfere with parsing downstream. Suppressing them keeps output clean while

still recording that errors occurred (via the ERRORED flag).

See: Companion Code Section 6

Section 7: The err_message Macro

The err_message macro extends message with error tracking. Before printing, it sets the ERRORED flag. This flag persists and affects the final exit status.

The separation between message and err_message matters for categorization. Some messages are informational — warnings about unusual conditions that don't indicate failure. Error messages indicate something went wrong that the user should know about.

Exit status conventions distinguish these cases. A search that found matches but encountered errors returns exit code two, not zero. The err_message macro ensures these errors get counted even if message display is suppressed.

See: Companion Code Section 7

Section 8: The ignore_message Macro

The ignore_message macro handles a specific category: parse errors in ignore files. These occur when .gitignore, .rgignore, or similar files contain invalid syntax.

The macro checks both MESSAGES and IGNORE_MESSAGES flags. Both must be true for output to occur. This two-level gating lets users suppress ignore warnings specifically while keeping other error messages visible.

Why separate ignore messages? Ignore file syntax errors are common and often benign. A .gitignore written for a different tool might use unsupported syntax. Flooding the terminal with these warnings during every search would be annoying.

See: Companion Code Section 8

Section 9: The Accessor Functions

Four pairs of functions provide controlled access to the global flags. The accessor functions (messages, ignore_messages, errored) read the current state. The mutator functions (set_messages, set_ignore_messages, set_errored) modify it.

The pub(crate) visibility restricts access to within the ripgrep binary. External code can't manipulate these flags — they're internal implementation details.

The set_errored function's doc comment notes that callers shouldn't use it directly. The err_message macro handles the common case. Direct calls might be needed for errors that don't warrant a message, but those are rare.

See: Companion Code Section 9

Section 10: Usage Patterns in main.rs

Looking at how main.rs uses these macros reveals the design's purpose.

In the search loop, file access errors use err_message. The search continues, but the error is recorded. At program exit, if ERRORED is true and the search otherwise succeeded, exit code becomes two instead of zero.

The broken pipe check appears both in the macro and in explicit checks throughout main.rs. The macro handles errors during error reporting — a meta-level concern. The explicit checks handle broken pipe during normal search output.

The initialization code in argument parsing calls set_messages and set_ignore_messages based on flag values. This happens once, before any parallelism, establishing the configuration that all threads will read.

See: Companion Code Section 10

Key Takeaways

First, thread-safe output requires holding locks across entire messages, not just individual writes.

Second, locking stdout before writing to stderr prevents interleaving when both streams go to the same terminal.

Third, global atomic flags work well for configuration set once and read many times.

Fourth, relaxed memory ordering suffices when writes complete before reads begin.

Fifth, exit status tracking separate from message display enables clean scripting while preserving error information.

What to Read Next

Understanding messages.rs completes the core module picture. The remaining questions involve the library crates:

How does termcolor manage colored output and interact with these locks? Read the termcolor crate.

How does the parallel walker coordinate with message output? Read the ignore crate's parallel module.

How do the actual search errors get generated? Read grep-searcher's error handling.