ripgrep crates/searcher/src/searcher/core.rs: Code Companion

Reference code for the Search Algorithm lecture. Sections correspond to the lecture document.

Section 1: The Core State Machine

```rust
#[derive(Debug)]
pub(crate) struct Core<'s, M: 's, S> {
    // Configuration and dependencies (borrowed)
    config: &'s Config,
    matcher: M,
    searcher: &'s Searcher,
    sink: S,

    // Binary detection state
    binary: bool,
    binary_byte_offset: Option<usize>,

    // Position tracking - dual granularity
    pos: usize,                     // Position within current buffer
    absolute_byte_offset: u64,      // Position in the overall file

    // Line number optimization
    line_number: Option<u64>,       // Current line number (if tracking)
    last_line_counted: usize,       // Checkpoint for incremental counting
    last_line_visited: usize,       // Last line we processed

    // Context and match state
    after_context_left: usize,      // Remaining after-context lines to emit
    has_sunk: bool,                 // Have we output anything?
    has_matched: bool,              // Have we found any match?
    count: u64,                     // Total matches found
}
```

The struct is generic over `M: Matcher` and `S: Sink`, allowing different regex engines and output destinations. The `'s` lifetime ties borrowed references to the searcher's lifetime.

```rust
impl<'s, M: Matcher, S: Sink> Core<'s, M, S> {
    pub(crate) fn new(
        searcher: &'s Searcher,
        matcher: M,
        sink: S,
        binary: bool,
    ) -> Core<'s, M, S> {
        let line_number =
            if searcher.config.line_number { Some(1) } else { None };
        let core = Core {
            config: &searcher.config,
            matcher,
            searcher,
            sink,
            binary,
            pos: 0,
            absolute_byte_offset: 0,
            binary_byte_offset: None,
            line_number,
            last_line_counted: 0,
            last_line_visited: 0,
            after_context_left: 0,
            has_sunk: false,
            has_matched: false,
            count: 0,
        };
        // Log which search path will be used
        if !core.searcher.multi_line_with_matcher(&core.matcher) {
            if core.is_line_by_line_fast() {
                log::trace!("searcher core: will use fast line searcher");
            } else {
                log::trace!("searcher core: will use slow line searcher");
            }
        }
        core
    }
}
```

Line number tracking is optional (`Option<u64>`) to avoid counting overhead when not needed.

Section 2: The Fast Path vs. Slow Path Decision

```
fn is_line_by_line_fast(&self) -> bool {
    debug_assert!(!self.searcher.multi_line_with_matcher(&self.matcher));

    // Passthrough mode requires slow path (must see every line)
    if self.config.passthru {
        return false;
    }
    // Stop-on-nonmatch after a match needs slow path
    if self.config.stop_on_nonmatch && self.has_matched {
        return false;
    }

    // Check if matcher understands line terminators
    if let Some(line_term) = self.matcher.line_terminator() {
        // FIXME: Workaround for grep-regex bug with NUL terminators
        // Line anchors like (?m:^) and (?m:$) don't work with \x00
        if line_term.as_byte() == b'\x00' {
            return false;
        }
        // Matcher's line terminator matches our config
        if line_term == self.config.line_term {
            return true;
        }
    }

    // Alternative: check non_matching_bytes
    if let Some(non_matching) = self.matcher.non_matching_bytes() {
        // For CRLF, we only care about \n (the required part)
        if non_matching.contains(self.config.line_term.as_byte()) {
            return true;
        }
    }
    false
}
```

The method returns `true` only when the matcher can correctly handle line boundaries during its search, enabling the optimized fast path.

```rust
pub(crate) fn match_by_line(
    &mut self,
    buf: &[u8],
) -> Result<bool, S::Error> {
    if self.is_line_by_line_fast() {
        match self.match_by_line_fast(buf)? {
            FastMatchResult::SwitchToSlow => self.match_by_line_slow(buf),
            FastMatchResult::Continue => Ok(true),
            FastMatchResult::Stop => Ok(false),
        }
    } else {
        self.match_by_line_slow(buf)
    }
}
```

The `FastMatchResult` enum allows the fast path to hand off to the slow path mid-search when conditions change.

Section 3: The Fast Line Search Algorithm

```rust
#[inline(always)]
fn find_by_line_fast(
    &mut self,
    buf: &[u8],
) -> Result<Option<Range>, S::Error> {
    debug_assert!(!self.searcher.multi_line_with_matcher(&self.matcher));
    debug_assert!(self.is_line_by_line_fast());

    let mut pos = self.pos();
    while !buf[pos..].is_empty() {
        if self.has_exceeded_match_limit() {
            return Ok(None);
        }
        // Ask matcher for candidate positions
        match self.matcher.find_candidate_line(&buf[pos..]) {
            Err(err) => return Err(S::Error::error_message(err)),
            Ok(None) => return Ok(None),  // No more candidates

            // Confirmed match - regex engine is certain
            Ok(Some(LineMatchKind::Confirmed(i))) => {
                let line = lines::locate(
                    buf,
                    self.config.line_term.as_byte(),
                    Range::zero(i).offset(pos),
                );
                // Edge case: match at buffer end isn't valid
                if line.start() == buf.len() {
                    pos = buf.len();
                    continue;
                }
                return Ok(Some(line));
            }

            // Candidate - needs verification
            Ok(Some(LineMatchKind::Candidate(i))) => {
                let line = lines::locate(
                    buf,
                    self.config.line_term.as_byte(),
                    Range::zero(i).offset(pos),
                );
                // Verify the candidate is a real match
                if self.is_match(&buf[line])? {
                    return Ok(Some(line));
                }
                // Not a match, advance past this line
                pos = line.end();
            }
        }
    }
    Ok(None)
}
```

The `lines::locate` function finds the full line containing a given position, handling the translation from match position to line boundaries.

```
fn is_match(&self, line: &[u8]) -> Result<bool, S::Error> {
    // Strip terminator to prevent regexes like (?m)^$ from
    // matching the empty position after the line terminator
    let line = lines::without_terminator(line, self.config.line_term);
    self.matcher.is_match(line).map_err(S::Error::error_message)
}
```

Section 4: Inverted Match Handling

```rust
#[inline(always)]
fn match_by_line_fast_invert(
    &mut self,
    buf: &[u8],
) -> Result<bool, S::Error> {
    assert!(self.config.invert_match);

    // Find the next matching line (which we want to exclude)
    let invert_match = match self.find_by_line_fast(buf)? {
        // No match found - everything from pos to end is non-matching
        None => {
            let range = Range::new(self.pos(), buf.len());
            self.set_pos(range.end());
            range
        }
        // Match found - non-matching region is from pos to match start
        Some(line) => {
            let range = Range::new(self.pos(), line.start());
            self.set_pos(line.end());  // Skip past the matching line
            range
        }
    };

    // Empty region means we hit a match immediately
    if invert_match.is_empty() {
        return Ok(true);
    }

    self.has_matched = true;

    // Handle context around the non-matching region
    if !self.after_context_by_line(buf, invert_match.start())? {
        return Ok(false);
    }
    if !self.before_context_by_line(buf, invert_match.start())? {
        return Ok(false);
    }

    // Iterate through each non-matching line
    let mut stepper = LineStep::new(
        self.config.line_term.as_byte(),
        invert_match.start(),
        invert_match.end(),
    );
    while let Some(line) = stepper.next_match(buf) {
        self.increment_count();  // Each line counts as a "match"
        if !self.sink_matched(buf, &line)? {
            return Ok(false);
        }
        // Check limit after each line
        if self.has_exceeded_match_limit() {
            return Ok(false);
        }
    }
```

```
    Ok(true)}
```

The inverted match region boundaries are computed by finding where actual matches occur, then treating everything between as the result.

Section 5: The Slow Path for Complex Scenarios

```
fn match_by_line_slow(&mut self, buf: &[u8]) -> Result<bool, S::Error> {
    debug_assert!(!self.searcher.multi_line_with_matcher(&self.matcher));

    let range = Range::new(self.pos(), buf.len());
    let mut stepper = LineStep::new(
        self.config.line_term.as_byte(),
        range.start(),
        range.end(),
    );

    while let Some(line) = stepper.next_match(buf) {
        // Early exit if we've hit limits and have no more context to emit
        if self.has_exceeded_match_limit()
            && !self.config.passthru
            && self.after_context_left == 0
        {
            return Ok(false);
        }

        // Check for match (with line terminator stripped)
        let matched = {
            let slice = lines::without_terminator(
                &buf[line],
                self.config.line_term,
            );
            self.shortest_match(slice)?.is_some()
        };
        self.set_pos(line.end());

        // XOR with invert_match to get actual success
        let success = matched != self.config.invert_match;

        if success {
            // Case 1: Line matches (or doesn't match in invert mode)
            self.has_matched = true;
            self.increment_count();
            if !self.before_context_by_line(buf, line.start())? {
                return Ok(false);
            }
            if !self.sink_matched(buf, &line)? {
                return Ok(false);
            }
        } else if self.after_context_left >= 1 {
            // Case 2: Emitting after-context from previous match
            if !self.sink_after_context(buf, &line)? {
                return Ok(false);
            }
        } else if self.config.passthru {
            // Case 3: Passthrough mode - emit as "other" context
            if !self.sink_other_context(buf, &line)? {
                return Ok(false);
            }
        }
```

```
        // Case 4: Stop on first non-match after a match        if
self.config.stop_on_nonmatch && !success && self.has_matched {            return
Ok(false);        }    }    Ok(true)}
```

The `shortest_match` method is an optimization—we only need to know if a match exists, not its full extent.

Quick Reference

FastMatchResult Enum

Variant	Meaning
`Continue`	Keep processing, buffer consumed
`Stop`	Stop searching entirely
`SwitchToSlow`	Fall back to slow path

Key State Fields

Field	Purpose
`pos`	Current position in buffer
`absolute_byte_offset`	Position in file (survives buffer rolls)
`last_line_counted`	Checkpoint for incremental line counting
`after_context_left`	Remaining context lines to emit
`count`	Total matches (for `--max-count`)

LineMatchKind Variants

```
enum LineMatchKind {
    Confirmed(usize),  // Definite match at offset
    Candidate(usize),  // Possible match, needs verification
}
```

Fast Path Requirements

The fast path is enabled when: 1. `passthru` mode is disabled 2. `stop_on_nonmatch` hasn't triggered 3. Matcher advertises line terminator support OR non-matching bytes include the terminator 4. Line terminator is not NUL (`\x00`)