ripgrep crates/core/main.rs: Code Companion

Reference code for the Application Entry Point lecture. Sections correspond to the lecture document.

Section 1: The Allocator Override

```
// Since Rust no longer uses jemalloc by default, ripgrep will, by default,
// use the system allocator. On Linux, this would normally be glibc's
// allocator, which is pretty good. In particular, ripgrep does not have a
// particularly allocation heavy workload, so there really isn't much
// difference (for ripgrep's purposes) between glibc's allocator and jemalloc.
//
// However, when ripgrep is built with musl, this means ripgrep will use musl's
// allocator, which appears to be substantially worse. (musl's goal is not to
// have the fastest version of everything. Its goal is to be small and amenable
// to static compilation.) Even though ripgrep isn't particularly allocation
// heavy, musl's allocator appears to slow down ripgrep quite a bit. Therefore,
// when building with musl, we use jemalloc.
//
// We don't unconditionally use jemalloc because it can be nice to use the
// system's default allocator by default. Moreover, jemalloc seems to increase
// compilation times by a bit.
//
// Moreover, we only do this on 64-bit systems since jemalloc doesn't support
// i686.

// Conditional compilation: only include this code for musl + 64-bit targets
#[cfg(all(target_env = "musl", target_pointer_width = "64"))]
#[global_allocator]  // Replaces the default allocator for ALL heap allocations
static ALLOC: tikv_jemallocator::Jemalloc = tikv_jemallocator::Jemalloc;
```

The `#[cfg(...)]` attribute uses boolean logic: `all()` requires both conditions. The `#[global_allocator]` attribute can only appear once per binary and completely replaces Rust's default allocation strategy.

Section 2: The Main Function and Exit Code Philosophy

```
use std::{io::Write, process::ExitCode};

/// Then, as it was, then again it will be.
fn main() -> ExitCode {
    match run(flags::parse()) {
        Ok(code) => code,
        Err(err) => {
            // Look for a broken pipe error. In this case, we generally want
            // to exit "gracefully" with a success exit code. This matches
            // existing Unix convention.
            for cause in err.chain() {
                // Walk the error chain looking for an io::Error
                if let Some(ioerr) = cause.downcast_ref::<std::io::Error>() {
                    if ioerr.kind() == std::io::ErrorKind::BrokenPipe {
                        return ExitCode::from(0);  // Graceful exit on pipe close
                    }
                }
            }
            // {:#} prints the full error chain with "caused by:" formatting
            eprintln_locked!("{:#}", err);
            ExitCode::from(2)  // Error exit code
        }
    }
}
```

The `err.chain()` method comes from the `anyhow` crate and iterates through the entire chain of wrapped errors. The `downcast_ref::<T>()` method attempts to cast a trait object back to a concrete type, returning `Some(&T)` on success.

Section 3: The Run Function and Mode Dispatch

```
/// The main entry point for ripgrep.
fn run(result: crate::flags::ParseResult<HiArgs>) -> anyhow::Result<ExitCode> {
    use crate::flags::{Mode, ParseResult};

    // Three-way match on parse result: error, special mode, or success
    let args = match result {
        ParseResult::Err(err) => return Err(err),
        ParseResult::Special(mode) => return special(mode),  // Help, version
        ParseResult::Ok(args) => args,
    };

    // Mode dispatch: determines what ripgrep actually does
    let matched = match args.mode() {
        // Short-circuit if pattern can never match anything
        Mode::Search(_) if !args.matches_possible() => false,
        // Single-threaded search (threads == 1)
        Mode::Search(mode) if args.threads() == 1 => search(&args, mode)?,
        // Multi-threaded search (default)
        Mode::Search(mode) => search_parallel(&args, mode)?,
        // File listing modes
        Mode::Files if args.threads() == 1 => files(&args)?,
        Mode::Files => files_parallel(&args)?,
        // Utility modes return directly
        Mode::Types => return types(&args),
        Mode::Generate(mode) => return generate(mode),
    };

    // Exit code calculation: 0 = match found, 1 = no match, 2 = error
    Ok(if matched && (args.quiet() || !messages::errored()) {
        ExitCode::from(0)
    } else if messages::errored() {
        ExitCode::from(2)
    } else {
        ExitCode::from(1)
    })
}
```

The match guard syntax (`if args.threads() == 1`) allows conditional branching within pattern arms. The `messages::errored()` function provides global error state tracking without threading error counts through every function.

Section 4: Single-Threaded Search Architecture

```rust
/// The top-level entry point for single-threaded search.
fn search(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    let started_at = std::time::Instant::now();

    // Build the search infrastructure using HiArgs builder methods
    let haystack_builder = args.haystack_builder();
    let unsorted = args
        .walk_builder()?
        .build()
        .filter_map(|result| haystack_builder.build_from_result(result));
    let haystacks = args.sort(unsorted);  // Optional sorting

    let mut matched = false;
    let mut searched = false;
    let mut stats = args.stats();  // Option<Stats> - None if not requested

    // Compose the search worker from matcher, searcher, and printer
    let mut searcher = args.search_worker(
        args.matcher()?,
        args.searcher()?,
        args.printer(mode, args.stdout()),
    )?;

    for haystack in haystacks {
        searched = true;
        let search_result = match searcher.search(&haystack) {
            Ok(search_result) => search_result,
            // Broken pipe means graceful termination
            Err(err) if err.kind() == std::io::ErrorKind::BrokenPipe => break,
            Err(err) => {
                err_message!("{}: {}", haystack.path().display(), err);
                continue;  // Log error but keep searching other files
            }
        };
        matched = matched || search_result.has_match();

        // Accumulate stats only if requested (Option pattern)
        if let Some(ref mut stats) = stats {
            *stats += search_result.stats().unwrap();
        }

        // Support --quiet: stop after first match
        if matched && args.quit_after_match() {
            break;
        }
    }

    // Helpful error when ignore rules filter everything
    if args.has_implicit_path() && !searched {
        eprint_nothing_searched();
    }

    // Print stats if requested
    if let Some(ref stats) = stats {
```

```
        let wtr = searcher.printer().get_mut();        let _ = print_stats(mode,
 stats, started_at, wtr);    }    Ok(matched)}
```

The `filter_map` combinator transforms and filters in one step—`None` values are discarded. The `if let Some(ref mut stats)` pattern provides zero-cost optional feature handling.

Section 5: Parallel Search and Thread Coordination

```
/// The top-level entry point for multi-threaded search.
fn search_parallel(args: &HiArgs, mode: SearchMode) -> anyhow::Result<bool> {
    use std::sync::atomic::{AtomicBool, Ordering};

    let started_at = std::time::Instant::now();
    let haystack_builder = args.haystack_builder();
    let bufwtr = args.buffer_writer();  // Thread-safe buffered writer

    // Shared state wrapped in appropriate synchronization primitives
    let stats = args.stats().map(std::sync::Mutex::new);
    let matched = AtomicBool::new(false);
    let searched = AtomicBool::new(false);

    // Create a prototype searcher that will be cloned per-thread
    let mut searcher = args.search_worker(
        args.matcher()?,
        args.searcher()?,
        args.printer(mode, bufwtr.buffer()),
    )?;

    // Parallel directory walk with worker closure
    args.walk_builder()?.build_parallel().run(|| {
        // Capture shared state by reference
        let bufwtr = &bufwtr;
        let stats = &stats;
        let matched = &matched;
        let searched = &searched;
        let haystack_builder = &haystack_builder;
        let mut searcher = searcher.clone();  // Each thread gets its own clone

        // Return a boxed closure that processes one directory entry
        Box::new(move |result| {
            let haystack = match haystack_builder.build_from_result(result) {
                Some(haystack) => haystack,
                None => return WalkState::Continue,
            };

            searched.store(true, Ordering::SeqCst);
            searcher.printer().get_mut().clear();  // Clear buffer for reuse

            let search_result = match searcher.search(&haystack) {
                Ok(search_result) => search_result,
                Err(err) => {
                    err_message!("{}: {}", haystack.path().display(), err);
                    return WalkState::Continue;
                }
            };

            if search_result.has_match() {
                matched.store(true, Ordering::SeqCst);
            }

            // Stats protected by mutex for thread-safe accumulation
            if let Some(ref locked_stats) = *stats {
```

```
                let mut stats = locked_stats.lock().unwrap();
*stats += search_result.stats().unwrap();            }            // Print
through buffer writer (handles synchronization)            if let Err(err) =
bufwtr.print(searcher.printer().get_mut()) {                if err.kind() ==
std::io::ErrorKind::BrokenPipe {                    return WalkState::Quit;  //
Stop all threads                }                err_message!("{}: {}",
haystack.path().display(), err);            }            // Check if we should
stop (--quiet mode)            if matched.load(Ordering::SeqCst) &&
args.quit_after_match() {                WalkState::Quit            } else {
WalkState::Continue            }        })    });    // Post-processing (runs
after all threads complete)    if args.has_implicit_path() &&
!searched.load(Ordering::SeqCst) {        eprint_nothing_searched();    }    if
let Some(ref locked_stats) = stats {        let stats =
locked_stats.lock().unwrap();        let mut wtr = searcher.printer().get_mut();
let _ = print_stats(mode, &stats, started_at, &mut wtr);        let _ =
bufwtr.print(&mut wtr);    }    Ok(matched.load(Ordering::SeqCst))}
```

The `WalkState` enum controls parallel traversal: `Continue` proceeds normally, `Quit` signals all threads to stop. `SeqCst` ordering provides the strongest memory ordering guarantees, ensuring visibility across all threads.

Section 6: File Listing Modes

```
/// Single-threaded file listing
fn files(args: &HiArgs) -> anyhow::Result<bool> {
    let haystack_builder = args.haystack_builder();
    let unsorted = args
        .walk_builder()?
        .build()
        .filter_map(|result| haystack_builder.build_from_result(result));
    let haystacks = args.sort(unsorted);

    let mut matched = false;
    let mut path_printer = args.path_printer_builder().build(args.stdout());

    for haystack in haystacks {
        matched = true;
        if args.quit_after_match() {
            break;
        }
        if let Err(err) = path_printer.write(haystack.path()) {
            if err.kind() == std::io::ErrorKind::BrokenPipe {
                break;
            }
            return Err(err.into());
        }
    }
    Ok(matched)
}

/// Multi-threaded file listing with channel-based output
fn files_parallel(args: &HiArgs) -> anyhow::Result<bool> {
    use std::{
        sync::{atomic::{AtomicBool, Ordering}, mpsc},
        thread,
    };

    let haystack_builder = args.haystack_builder();
    let mut path_printer = args.path_printer_builder().build(args.stdout());
    let matched = AtomicBool::new(false);

    // Channel for sending paths from workers to printer thread
    let (tx, rx) = mpsc::channel::<crate::haystack::Haystack>();

    // Single printing thread prevents write tearing
    let print_thread = thread::spawn(move || -> std::io::Result<()> {
        for haystack in rx.iter() {
            path_printer.write(haystack.path())?;
        }
        Ok(())
    });

    args.walk_builder()?.build_parallel().run(|| {
        let haystack_builder = &haystack_builder;
        let matched = &matched;
        let tx = tx.clone();  // Clone sender for each worker
```

```
        Box::new(move |result| {            let haystack = match
haystack_builder.build_from_result(result) {                Some(haystack) =>
haystack,                None => return WalkState::Continue,            };
matched.store(true, Ordering::SeqCst);            if args.quit_after_match() {
WalkState::Quit            } else {                match tx.send(haystack) {
Ok(_) => WalkState::Continue,                    Err(_) => WalkState::Quit,  //
Receiver dropped                }            }        })    });    drop(tx);  //
Close channel, allowing print_thread to finish    if let Err(err) =
print_thread.join().unwrap() {        if err.kind() !=
std::io::ErrorKind::BrokenPipe {            return Err(err.into());        }    }
Ok(matched.load(Ordering::SeqCst))}
```

The `drop(tx)` is critical: it closes the sending end of the channel, allowing the receiver's `iter()` to terminate. Without this, the print thread would block forever waiting for more messages.

Section 7: Utility Modes and Statistics

```
/// The top-level entry point for `--type-list`.
fn types(args: &HiArgs) -> anyhow::Result<ExitCode> {
    let mut count = 0;
    let mut stdout = args.stdout();
    for def in args.types().definitions() {
        count += 1;
        stdout.write_all(def.name().as_bytes())?;
        stdout.write_all(b": ")?;

        let mut first = true;
        for glob in def.globs() {
            if !first {
                stdout.write_all(b", ")?;
            }
            stdout.write_all(glob.as_bytes())?;
            first = false;
        }
        stdout.write_all(b"\n")?;
    }
    // Exit 1 if no types defined, 0 otherwise
    Ok(ExitCode::from(if count == 0 { 1 } else { 0 }))
}

/// Generate shell completions and man pages
fn generate(mode: crate::flags::GenerateMode) -> anyhow::Result<ExitCode> {
    use crate::flags::GenerateMode;

    let output = match mode {
        GenerateMode::Man => flags::generate_man_page(),
        GenerateMode::CompleteBash => flags::generate_complete_bash(),
        GenerateMode::CompleteZsh => flags::generate_complete_zsh(),
        GenerateMode::CompleteFish => flags::generate_complete_fish(),
        GenerateMode::CompletePowerShell =>
flags::generate_complete_powershell(),
    };
    writeln!(std::io::stdout(), "{}", output.trim_end())?;
    Ok(ExitCode::from(0))
}

/// Print statistics in text or JSON format
fn print_stats<W: Write>(
    mode: SearchMode,
    stats: &grep::printer::Stats,
    started: std::time::Instant,
    mut wtr: W,
) -> std::io::Result<()> {
    let elapsed = std::time::Instant::now().duration_since(started);

    if matches!(mode, SearchMode::JSON) {
        serde_json::to_writer(
            &mut wtr,
            &serde_json::json!({
                "type": "summary",
                "data": {
```

```
                    "stats": stats,                    "elapsed_total": {
"secs": elapsed.as_secs(),                        "nanos":
elapsed.subsec_nanos(),                       "human": format!("{:0.6}s",
elapsed.as_secs_f64()),                    },                }            }),
)?;        write!(wtr, "\n")    } else {        write!(wtr, "{matches}
matches{lines} matched lines{searches_with_match} files contained
matches{searches} files searched{bytes_printed} bytes printed{bytes_searched}
bytes searched{search_time:0.6} seconds spent searching{process_time:0.6} seconds
total",            matches = stats.matches(),            lines =
stats.matched_lines(),            // ... remaining fields        )    }}
```

The `matches!` macro is a concise way to test if a value matches a pattern without binding. The generic `W: Write` bound allows the same function to write to any destination implementing `Write`.

Quick Reference

Exit Codes

Code	Meaning
0	Matches found (or broken pipe)
1	No matches found
2	Error occurred

Mode Dispatch

Mode	Single-Threaded	Multi-Threaded
Search	`search()`	`search_parallel()`
Files	`files()`	`files_parallel()`
Types	`types()`	N/A

Generate	`generate()`	N/A

Thread Synchronization Primitives

```
// Atomic boolean for simple flags
use std::sync::atomic::{AtomicBool, Ordering};
let flag = AtomicBool::new(false);
flag.store(true, Ordering::SeqCst);
flag.load(Ordering::SeqCst);

// Mutex for complex data requiring exclusive access
use std::sync::Mutex;
let data = Mutex::new(Stats::new());
let mut guard = data.lock().unwrap();
*guard += new_stats;

// Channel for producer-consumer patterns
use std::sync::mpsc;
let (tx, rx) = mpsc::channel();
tx.send(item)?;
for item in rx.iter() { /* ... */ }
```

WalkState Control Flow

```
use ignore::WalkState;

// In parallel walk closure:
WalkState::Continue  // Keep processing files
WalkState::Quit      // Signal all threads to stop
WalkState::Skip      // Skip current directory (not used here)
```