

Build Your Own Debugger: Design Document

Overview

This system implements a source-level debugger similar to GDB that can attach to processes, set breakpoints, and inspect variables. The key architectural challenge is coordinating between multiple complex subsystems: ptrace-based process control, binary format parsing for debug information, and real-time program state inspection.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

Milestone(s): All milestones — this section establishes the foundation for understanding the entire debugger architecture

The Detective Analogy

Think of building a debugger like creating the ultimate detective system for software crimes. When a program misbehaves — crashes unexpectedly, produces wrong results, or hangs indefinitely — you need to investigate what went wrong. Just as a detective examines a crime scene, interviews witnesses, and pieces together evidence to reconstruct events, a debugger must examine a running program, inspect its memory and registers, and trace through its execution to understand what happened.

The parallels run deep. A detective arrives at a crime scene and immediately secures the area to prevent contamination — similarly, a debugger must attach to a process and gain control without disrupting its state. Detectives collect physical evidence like fingerprints, DNA samples, and photographs — debuggers collect register values, memory contents, and stack traces. Detectives interview witnesses to understand the sequence of events — debuggers step through code execution line by line to see how the program reached its current state.

Most importantly, detectives need specialized tools and knowledge to interpret evidence. A fingerprint means nothing without a database to match it against, just as a raw memory address means nothing without symbol tables to translate it back to variable names and source code locations. The detective's expertise lies not just in collecting evidence, but in knowing what evidence to look for, how to interpret it, and how to reconstruct the timeline of events.

This analogy reveals why debugger architecture is so complex: it must seamlessly bridge multiple layers of abstraction. The "crime scene" exists simultaneously at the hardware level (CPU registers and memory bytes), the operating system level (processes and signals), the binary format level (machine instructions and symbol tables), and the source code level (variables and line numbers). A debugger must be fluent in all these languages and translate between them effortlessly.

Core Technical Challenges

Building a debugger requires solving four fundamental technical challenges, each operating at different layers of the system stack. These challenges must work together seamlessly, but each brings its own complexity and failure modes.

Process Control Challenge

The first challenge is gaining control over another process's execution. In Unix-like systems, this means mastering the `ptrace` system call, which allows one process to observe and control another. Think of `ptrace` as a puppet master's strings — it gives you the ability to make the target process dance, but the strings are delicate and the choreography is complex.

The debugger must fork a child process, attach to it with `ptrace` before it begins executing the target program, and then orchestrate a careful dance of signals and system calls. The target process runs until it hits a breakpoint or receives a signal, at which point the kernel pauses it and notifies the debugger. The debugger can then examine the process state, modify memory or registers, and decide whether to continue execution or step one instruction at a time.

This challenge involves deep knowledge of Unix process lifecycle, signal handling, and the subtle race conditions that can occur during process startup and shutdown. A single mistake in signal handling can result in zombie processes, lost signals, or a debugger that hangs indefinitely waiting for events that never arrive.

Binary Format Parsing Challenge

The second challenge is understanding the target program's structure by parsing its binary format. Modern executable files contain multiple layers of information: the ELF (Executable and Linkable Format) structure describes how the program should be loaded into memory, while embedded DWARF (Debug With Arbitrary Record Formats) debug information provides the mapping between machine code and source code.

Parsing these formats is like archaeological work — you're excavating layers of encoded information to reconstruct the original source code structure. The ELF format describes sections like `.text` (executable code), `.data` (initialized variables), and `.debug_*` (debug information). Within the debug sections, DWARF uses a complex tree structure called Debug Information Entries (DIEs) to encode everything from function definitions to variable types to source line mappings.

This challenge requires understanding multiple binary formats, each with their own versioning, encoding rules, and optional extensions. DWARF alone has evolved through five major versions, each adding new capabilities and complexity. The parser must handle malformed or incomplete debug information gracefully, since optimizing compilers often produce partial or misleading debug data.

Symbol Resolution Challenge

The third challenge is building bidirectional mappings between the machine-level view and the source-level view of the program. When a user sets a breakpoint on "function main at line 42," the debugger must translate this to a specific memory address. Conversely, when the program crashes at address 0x401234, the debugger must translate this back to "file.c:15 in function calculate_sum."

This translation process involves multiple symbol tables and debug information structures. Function symbols provide name-to-address mappings, while DWARF line number tables provide address-to-source-line mappings. Variable symbols describe where each variable is stored (register, stack offset, or global address) and what type it represents.

The challenge intensifies with modern optimizing compilers that inline functions, eliminate dead code, and store variables in registers rather than memory. A single source line might correspond to multiple machine instructions scattered throughout the binary, while some variables might be "optimized out" entirely, existing only during certain phases of execution.

Real-Time State Inspection Challenge

The fourth challenge is safely reading and interpreting the target process's memory and register state while it's paused. This requires coordinating between the process control system (which manages when the process is stopped), the symbol resolution system (which knows where variables are located), and the binary format parsers (which know how to interpret raw bytes as typed data).

Reading a simple integer variable involves multiple steps: looking up the variable's location in the debug information, determining whether it's stored in a register or memory, using `ptrace` to read the appropriate location, and then formatting the raw bytes according to the variable's type information. Composite types like structs and arrays add layers of complexity, requiring recursive navigation through type hierarchies and careful attention to alignment and padding rules.

The challenge multiplies when dealing with complex location expressions in DWARF, which can describe variables that move between registers and memory during execution, or variables whose value must be computed from multiple locations. Modern compilers use these features extensively to describe optimized code, turning what seems like a simple "print variable" operation into a complex interpreter for stack machine bytecode.

Existing Debugger Architectures

Understanding how existing debuggers approach these challenges provides valuable insight into architectural patterns and trade-offs. The three major approaches — exemplified by GDB, LLDB, and newer language-specific debuggers — each make different decisions about modularity, extensibility, and performance.

GDB: The Monolithic Pioneer

GDB (GNU Debugger) represents the traditional monolithic approach to debugger architecture. Built over decades of incremental development, GDB contains all functionality within a single large executable that handles everything from process control to user interface within tightly coupled modules.

GDB's architecture centers around a global process state structure that all subsystems access directly. The symbol reading code directly modifies global symbol tables, the breakpoint manager directly manipulates process memory through `ptrace` calls, and the expression evaluator directly accesses both symbol information and process state. This tight coupling enables sophisticated features like conditional breakpoints that can evaluate complex expressions, but it also makes the codebase difficult to modify and extend.

The monolithic approach has several advantages: it minimizes overhead from inter-process communication, allows for complex interactions between subsystems (like expression evaluation that depends on both symbols and live process state), and provides a consistent user experience across all debugging scenarios. However, it also means that adding support for new binary formats or debugging protocols requires modifying the core debugger code, and testing individual subsystems in isolation becomes difficult.

GDB's handling of different architectures and binary formats demonstrates both the strengths and weaknesses of the monolithic approach. Adding support for a new CPU architecture requires implementing target description files and register handling code that integrates deeply with the core execution engine. While this tight integration enables sophisticated features like architecture-specific disassembly and register formatting, it also means that architectural extensions must be carefully coordinated with all other debugger subsystems.

LLDB: The Modular Evolution

LLDB represents a more modern, modular approach that emerged from lessons learned with GDB's architecture. Built as part of the LLVM project, LLDB separates concerns into distinct libraries that communicate through well-defined APIs, making the overall system more maintainable and extensible.

LLDB's core architecture separates process control (handled by the Process and Thread classes), symbol parsing (handled by SymbolFile plugins), and user interface (handled by CommandInterpreter and API layers) into independent modules. Each module has clearly defined responsibilities and interfaces, allowing them to evolve independently. The Process class abstracts away platform-specific details of process control, while SymbolFile plugins handle different debug information formats without affecting other parts of the system.

This modular approach enables several advanced features that would be difficult in a monolithic architecture. LLDB can load multiple SymbolFile plugins to handle different debug formats within the same debugging session, switch between different process control backends (local ptrace vs. remote debugging protocol) transparently, and expose its full functionality through both command-line and programmatic APIs for integration with IDEs.

The trade-off is increased complexity in the interaction protocols between modules. Where GDB might directly access global state, LLDB modules must coordinate through event systems and callback mechanisms. This indirection adds overhead but provides much better isolation for testing and development. Adding a new debug information format to LLDB means implementing a SymbolFile plugin interface, rather than modifying core debugger logic.

LLDB's architecture also demonstrates how modern debuggers handle the increasing complexity of debug information. Rather than trying to parse all debug information upfront, LLDB uses lazy loading extensively — symbol information is parsed on-demand as the user requests it. This approach scales better to large programs with extensive debug information, but requires careful coordination to ensure that related information (like type definitions and variable locations) remains consistent.

Language-Specific Debuggers: The Specialized Approach

Modern language ecosystems have spawned specialized debuggers that focus on single languages or runtime environments. Examples include the Go debugger (Delve), Rust debugger extensions, and JavaScript debuggers built into browsers. These represent a third architectural approach that sacrifices generality for deep language-specific optimization and features.

Delve, the Go debugger, demonstrates how language-specific knowledge can simplify debugger architecture. Since Go has a well-defined runtime with standardized goroutine structures and garbage collector metadata, Delve can implement goroutine-aware debugging without needing to parse complex debug information formats. It understands Go's calling conventions, knows how to navigate Go's specific stack layout, and can interpret Go-specific runtime structures directly.

This specialization enables features that would be difficult or impossible in general-purpose debuggers: displaying all goroutines and their states, understanding Go's escape analysis results for variable lifetime tracking, and providing Go-specific expression evaluation that understands interface conversions and method dispatch. However, it also means that Delve cannot debug programs written in other languages, limiting its applicability.

Language-specific debuggers often make different trade-offs in their architectures. Since they only need to support one language's debugging model, they can optimize their symbol parsing for that language's specific debug information patterns, implement expression evaluators that understand the language's type system natively, and provide user interfaces that match the language's development workflow expectations.

The browser-based JavaScript debuggers represent an extreme version of this approach, where the debugger is embedded directly into the runtime environment. This eliminates the need for separate process control mechanisms (since the debugger runs in the same process as the debugged code) and enables features like live code editing and hot-swapping that would be impossible with traditional process-based debugging.

Critical Insight: The choice between monolithic, modular, and specialized architectures reflects fundamental trade-offs between generality, performance, and maintainability. A monolithic architecture like GDB's enables sophisticated cross-cutting features but makes evolution difficult. A modular architecture like LLDB's provides better separation of concerns but adds complexity in module coordination. Specialized architectures enable deep language integration but sacrifice breadth of applicability.

Architecture Decision Comparison

Approach	Generality	Performance	Maintainability	Extensibility	Complexity
Monolithic (GDB)	High - supports many languages	High - minimal overhead	Low - tight coupling	Low - requires core changes	Medium
Modular (LLDB)	High - plugin architecture	Medium - API overhead	High - clear separation	High - plugin interfaces	High
Specialized (Delve)	Low - single language	Very High - optimized	Medium - focused scope	Medium - within domain	Low

For our educational debugger project, we'll adopt a simplified modular approach that balances learning value with implementation complexity. This choice emphasizes clear separation of concerns (making it easier to understand each component independently) while avoiding the full complexity of a plugin architecture. Each milestone corresponds to implementing one major module, allowing learners to build understanding incrementally.

Implementation Guidance

This implementation guidance focuses on architectural patterns and technology choices that will help organize the debugger implementation effectively across all four milestones.

Technology Recommendations

Component	Simple Option	Advanced Option	Reasoning
Process Control	Direct ptrace syscalls with signal handling	libprocstat or procfs parsing	Direct ptrace teaches fundamentals
Symbol Parsing	Custom ELF/DWARF parser	libdwarf or libelf integration	Custom parser reveals format details
Memory Management	malloc/free with manual tracking	Memory pools or garbage collection	Manual tracking forces attention to lifecycle
Error Handling	Return codes with errno checking	Exception-based error propagation	Return codes match Unix conventions
User Interface	Simple command-line with scanf	Readline library with command history	Command-line focus on core functionality
Build System	Simple Makefile	CMake or autotools	Makefile keeps build process transparent

Recommended Project Structure

The debugger implementation should follow a modular directory structure that maps directly to the four core components and their responsibilities:

```
debugger/
├── src/
│   ├── main.c                         ← Entry point and command loop
│   └── process/
│       ├── process_control.h           ← Milestone 1: Process Control
│       ├── process_control.c          ← Process control interface
│       ├── ptrace_wrapper.h           ← ptrace operations and signal handling
│       └── ptrace_wrapper.c          ← Safe ptrace operation wrappers
│   ├── breakpoint/
│       ├── breakpoint.h               ← Error handling for ptrace calls
│       ├── breakpoint.c              ← Milestone 2: Breakpoints
│       ├── instruction.h            ← Breakpoint data structures and interface
│       ├── instruction.c             ← Breakpoint lifecycle management
│       ├── symbol/
│           ├── elf_parser.h          ← Instruction parsing and patching
│           ├── elf_parser.c          ← INT3 insertion and restoration
│           ├── dwarf_parser.h         ← Milestone 3: Symbol Tables
│           ├── dwarf_parser.c         ← ELF format parsing interface
│           ├── symbol_table.h         ← ELF section and header parsing
│           └── symbol_table.c         ← DWARF debug information interface
│   ├── variable/
│       ├── variable_reader.h        ← DWARF DIE parsing and navigation
│       ├── variable_reader.c        ← Symbol storage and lookup interface
│       ├── type_formatter.h         ← Hash tables for symbol resolution
│       └── type_formatter.c         ← Milestone 4: Variable Inspection
│   └── common/
│       ├── error_handling.h         ← Variable location and reading interface
│       ├── error_handling.c         ← Memory and register access via ptrace
│       ├── memory_utils.h           ← Type-aware value formatting interface
│       └── memory_utils.c           ← Converting raw bytes to typed values
│   └── include/
│       ├── debugger_types.h         ← Shared utilities
│       └── debugger_api.h           ← Error codes and reporting
│   └── tests/
│       ├── test_programs/
│           ├── simple.c             ← Centralized error handling
│           ├── crasher.c             ← Safe memory allocation wrappers
│           └── loops.c               ← Memory leak tracking and debugging
│       ├── unit_tests/
│           └── integration_tests/
│       └── integration_tests/
│   └── docs/
│       ├── milestone_checkpoints.md ← Documentation
│       └── debugging_guide.md      ← Verification steps for each milestone
│   └── Makefile
└── README.md                           ← Common issues and solutions
                                         ← Build configuration
                                         ← Quick start and overview
```

This structure provides several benefits for learning:

- Each milestone corresponds to implementing one directory under `src/`
- Dependencies flow from `variable/` → `symbol/` → `breakpoint/` → `process/`, making build order clear
- The `common/` directory contains infrastructure code that learners can focus on later

- Test programs in `test_programs/` provide concrete targets for debugging exercises
- The separation between interface headers (`.h`) and implementation (`.c`) forces learners to think about API design

Core Data Structure Foundation

Every component will build upon a shared set of core data structures defined in `debugger_types.h`. These structures establish the common vocabulary for communication between components:

```
// Core process representation that all components will reference C

typedef struct {

    pid_t pid;                      // Process ID from fork/exec

    int status;                     // Current wait status from waitpid

    enum ProcessState state;        // RUNNING, STOPPED, EXITED, etc.

    struct user_regs_struct regs;   // CPU registers from ptrace GETREGS

    char *executable_path;          // Path to binary being debugged

    void *symbol_table;             // Opaque pointer to symbol information

    void *breakpoint_list;          // Opaque pointer to breakpoint collection

} DebuggedProcess;

// Breakpoint representation with lifetime tracking

typedef struct {

    uintptr_t address;              // Memory address where breakpoint is set

    uint8_t original_byte;          // Instruction byte replaced by INT3

    bool is_enabled;                // Whether breakpoint is currently active

    int hit_count;                  // Number of times this breakpoint was hit

    char *location_description;     // Human-readable location (e.g., "main:42")

} Breakpoint;
```

Error Handling Infrastructure

Since debugger operations involve many system calls that can fail in subtle ways, establish a consistent error handling pattern from the beginning:

```

// Error codes specific to debugger operations

typedef enum {

    DEBUG_SUCCESS = 0,
    DEBUG_PTRACE_FAILED,           // ptrace system call returned -1
    DEBUG_PROCESS_EXITED,          // Target process terminated unexpectedly
    DEBUG_INVALID_ADDRESS,         // Memory address is not accessible
    DEBUG_SYMBOL_NOT_FOUND,        // Requested symbol does not exist
    DEBUG_BREAKPOINT_EXISTS,       // Breakpoint already set at this address
    DEBUG_NO_DEBUG_INFO,           // Binary lacks debug information
    DEBUG_MALFORMED_DWARF         // Debug information is corrupted

} DebugResult;

// Error reporting with context for debugging the debugger

typedef struct {

    DebugResult code;

    char message[256];            // Human-readable error description

    const char *function;          // Function where error occurred

    int line;                     // Source line where error occurred

    int system_errno;              // Value of errno when error occurred

} DebugError;

```

Milestone Implementation Strategy

Each milestone should be implemented and verified independently before proceeding to the next:

Milestone 1 Checkpoint: After implementing process control, you should be able to:

1. Start the debugger with `./debugger test_programs/simple`
2. See output confirming process attachment: "Attached to process 12345"
3. Enter commands like `continue`, `step`, and `stop`
4. Observe the debugger correctly handling process termination

Milestone 2 Checkpoint: After implementing breakpoints, you should be able to:

1. Set breakpoints with commands like `break main` or `break *0x401000`
2. Run the program and see it stop at breakpoints with "Breakpoint 1 hit at 0x401000"
3. Verify that `continue` resumes execution correctly after breakpoint hits

4. Confirm that multiple breakpoints work independently

Milestone 3 Checkpoint: After implementing symbol parsing, you should be able to:

1. Query function addresses with `info address main`
2. Translate addresses to source locations with `info line *0x401000`
3. List all functions with `info functions`
4. See meaningful error messages when debug information is missing

Milestone 4 Checkpoint: After implementing variable inspection, you should be able to:

1. Print variable values with `print variable_name`
2. Display different data types correctly (int, float, char*, struct)
3. Access struct members with `print mystruct.field`
4. Handle variables stored in registers vs. memory transparently

Common Implementation Pitfalls

⚠ Pitfall: Ignoring ptrace Error Conditions Many ptrace operations can fail silently or return misleading success indicators. Always check return values and errno, especially for PTRACE_PEEKDATA operations that return -1 on error but also return -1 as valid data.

⚠ Pitfall: Forgetting Signal Forwarding When the debugged process receives signals (SIGINT, SIGTERM), the debugger must decide whether to handle them internally or forward them to the target process. Failing to forward expected signals breaks normal program behavior.

⚠ Pitfall: Race Conditions During Process Startup The window between fork() and ptrace(PTRACE_TRACEME) in the child process can allow the child to begin executing before the debugger gains control. Use PTRACE_TRACEME in the child, not PTRACE_ATTACH in the parent.

⚠ Pitfall: Breakpoint Restoration Timing After hitting a breakpoint, the instruction pointer points one byte past the INT3 instruction. You must restore the original instruction AND decrement the instruction pointer before continuing execution.

⚠ Pitfall: Endianness in Multi-Byte Reads When reading multi-byte values from process memory, ptrace returns data in the host's native byte order. For cross-debugging scenarios or when examining raw memory, explicit endianness handling becomes critical.

Goals and Non-Goals

Milestone(s): All milestones — this section defines the scope and boundaries that guide implementation decisions across all four components

The Foundation Blueprint Analogy

Think of this goals and non-goals section as the **foundation blueprint** for a house construction project. Just as an architect must decide whether to build a modest two-bedroom home or a sprawling mansion before laying the foundation, we must clearly define what our debugger will and will not do before writing a single line of code. A foundation built for a

small house cannot support a skyscraper, but overengineering a foundation for a simple cottage wastes time and resources. Similarly, our debugger's architecture must be sized appropriately for its intended capabilities.

The goals we establish here directly influence every subsequent design decision. If we aim to support multi-threaded debugging from day one, our process control component must handle thread synchronization from the start. If we explicitly exclude multi-threading, we can build simpler, more focused components that excel at single-threaded debugging. The non-goals are equally important — they prevent scope creep and feature bloat that can derail the project.

This section serves as our **contract with complexity**. Each feature we include multiplies the system's complexity exponentially. A debugger that handles breakpoints, variable inspection, multi-threading, remote debugging, and GUI interfaces simultaneously becomes a massive undertaking that few developers could complete. By explicitly limiting our scope, we create a achievable project that still teaches the fundamental principles of debugger construction.

Functional Requirements

Our debugger implements four core capabilities that represent the essential building blocks of any debugging system. These requirements directly correspond to our four milestones and form a complete, albeit minimal, debugging environment. Each requirement builds upon the previous ones, creating a natural progression from basic process control to sophisticated source-level debugging.

Process Attachment and Control

The debugger must establish complete control over a target process, enabling the fundamental start-stop-examine cycle that defines interactive debugging. This capability serves as the foundation upon which all other debugging features depend.

Capability	Description	Success Criteria
Process Launch	Fork and attach to a new process before it executes the target program	Child process stops immediately after fork, debugger gains control via <code>PTRACE_TRACEME</code>
Process Attachment	Attach to an existing running process for debugging	Debugger successfully attaches via <code>PTRACE_ATTACH</code> , target process enters stopped state
Execution Control	Start, stop, and resume process execution on demand	<code>PTRACE_CONT</code> resumes execution, <code>SIGSTOP</code> pauses execution, both operations complete within 100ms
Signal Management	Intercept and handle signals between debugger and target	All signals delivered to target process pass through debugger first, debugger can choose to forward or suppress

The process attachment mechanism must handle the delicate handoff between parent and child processes during debugging initialization. When launching a new process for debugging, our debugger forks a child that immediately calls `ptrace(PTRACE_TRACEME, 0, NULL, NULL)` before executing the target program. This ensures the debugger gains control before the target's first instruction executes, preventing any code from running unobserved.

For attaching to existing processes, the debugger uses `ptrace(PTRACE_ATTACH, pid, NULL, NULL)` followed by `waitpid()` to confirm the attachment succeeded. The target process receives a `SIGSTOP` signal and enters the stopped state, allowing the debugger to examine its current state and set up any necessary tracking structures.

Critical Design Constraint: The debugger must maintain exclusive control over the target process throughout the debugging session. No other process or debugger can attach simultaneously due to kernel-level ptrace restrictions.

Single-Step Execution

The debugger must provide precise, instruction-level execution control that allows developers to observe program behavior at the finest granularity. Single-stepping forms the basis for source-level stepping and helps developers understand exactly how their code executes at the machine level.

Operation	Implementation	Behavior
Step Single Instruction	<code>PTRACE_SINGLESTEP</code> followed by <code>waitpid()</code>	Process executes exactly one machine instruction, then stops and signals debugger
Instruction Pointer Tracking	<code>PTRACE_GETREGS</code> to read register state	Debugger can examine and display the current instruction pointer after each step
Step Count Tracking	Internal counter incremented on each step	Debugger maintains statistics on execution progress for user feedback
Step Boundary Detection	Compare instruction pointer before and after step	Debugger detects when stepping crosses function boundaries or source lines

The single-step implementation relies on the processor's trap flag functionality, accessed through the `PTRACE_SINGLESTEP` ptrace operation. When the debugger calls `ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL)`, the kernel sets the trap flag in the target process's flags register, causing the processor to generate a `SIGTRAP` after executing exactly one instruction.

The debugger must handle the resulting `SIGTRAP` carefully, distinguishing between single-step traps and breakpoint traps. Single-step traps occur immediately after instruction execution, while breakpoint traps occur before instruction execution. This timing difference affects how the debugger updates the instruction pointer and presents information to the user.

Software Breakpoint Management

The debugger implements software breakpoints using the INT3 instruction patching technique, providing the fundamental ability to pause execution at specific locations in the target program. This capability enables interactive debugging workflows where developers can examine program state at carefully chosen points.

Breakpoint Operation	Technical Implementation	Expected Behavior
Set Breakpoint	Replace first byte of target instruction with <code>0xCC</code> (INT3)	Original instruction byte saved, breakpoint becomes active immediately
Breakpoint Hit Detection	Handle <code>SIGTRAP</code> signal and examine instruction pointer	Debugger detects breakpoint hit, decrements instruction pointer by 1 byte
Temporary Breakpoint Removal	Restore original instruction byte	Instruction can execute normally, breakpoint can be re-enabled afterward
Continue After Breakpoint	Single-step over restored instruction, then re-enable breakpoint	Execution resumes without infinite breakpoint loop

Software breakpoints work by exploiting the x86 INT3 instruction, which generates a software interrupt that the kernel delivers as a `SIGTRAP` signal to the controlling debugger. The INT3 instruction is exactly one byte long (`0xCC`), making it perfect for overwriting the first byte of any instruction without affecting instruction alignment.

The breakpoint lifecycle requires careful state management. When setting a breakpoint at address `A`, the debugger reads the original byte at that address using `ptrace(PTRACE_PEEKDATA, pid, A, NULL)` and stores it in the breakpoint record. It then writes the INT3 byte using `ptrace(PTRACE_POKEDATA, pid, A, 0xCC)`. When the breakpoint hits, the processor stops after executing the INT3, leaving the instruction pointer pointing to the byte immediately after the breakpoint. The debugger must decrement the instruction pointer by one to point back to the breakpoint location.

Architecture Decision: Software vs Hardware Breakpoints

- **Context:** x86 processors support both software breakpoints (INT3 instruction patching) and hardware breakpoints (debug registers)
- **Options Considered:** Software-only, hardware-only, or hybrid approach
- **Decision:** Software breakpoints only
- **Rationale:** Software breakpoints provide unlimited quantity (hardware debug registers limit to 4 breakpoints), work identically across all x86 variants, and demonstrate the core instruction-patching technique used by professional debuggers
- **Consequences:** Breakpoints modify target program memory temporarily, but we gain unlimited breakpoint capability and simpler implementation

Symbol Resolution and Source Mapping

The debugger provides source-level debugging by parsing DWARF debug information to map between machine addresses and source code locations. This capability transforms the debugger from a low-level machine code inspector into a source-level development tool.

Symbol Resolution Feature	Data Source	Capability Provided
Function Name to Address	DWARF <code>.debug_info</code> subprogram DIEs	User can set breakpoints by function name instead of memory address
Address to Source Line	DWARF <code>.debug_line</code> line number table	Debugger displays current source file and line number during execution
Variable Name to Location	DWARF location expressions and type information	User can inspect variables by name instead of memory addresses
Scope-Aware Symbol Lookup	DWARF lexical block hierarchy	Variable names resolve correctly based on current execution context

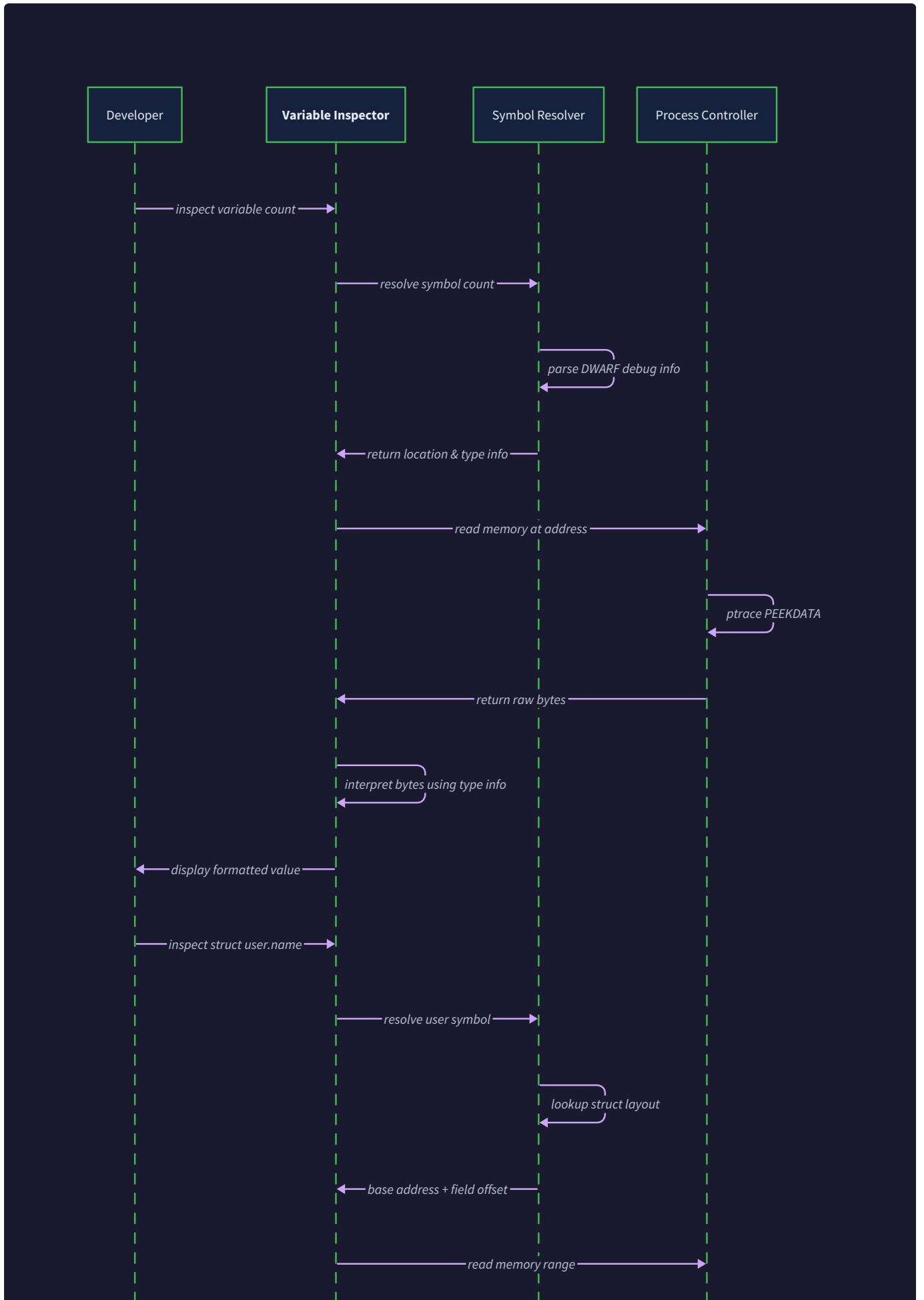
The symbol resolution system builds several mapping tables during initialization by parsing the target executable's DWARF debug information. The function name table maps from string identifiers like `"main"` or `"calculate_sum"` to their entry point addresses. The line number table provides bidirectional mapping between memory addresses and source locations specified as `(filename, line_number)` pairs.

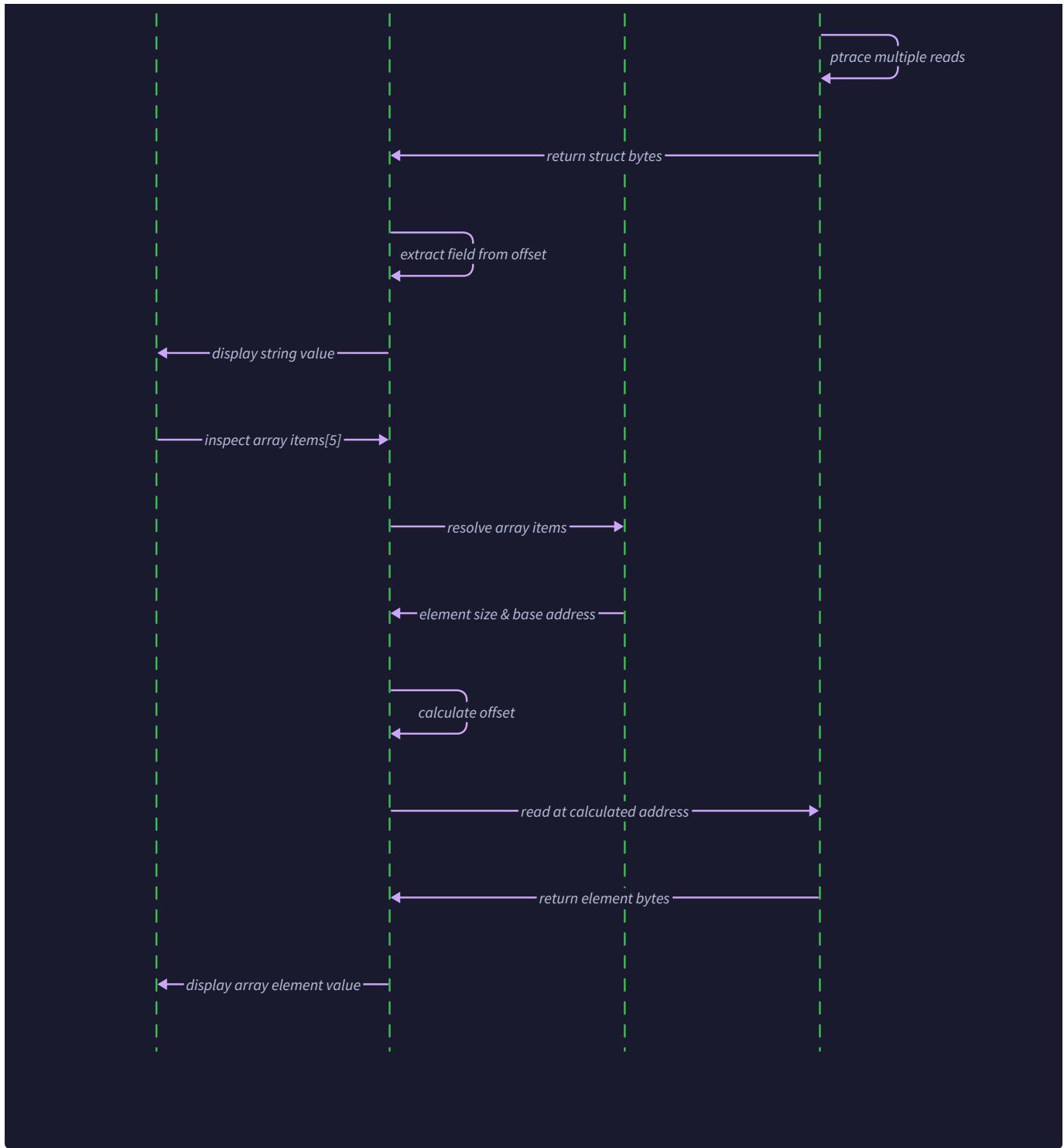
DWARF debug information is organized as a tree of Debug Information Entries (DIEs), each describing program entities like compilation units, functions, variables, and types. The debugger traverses this tree to extract relevant information and build efficient lookup structures. For example, a subprogram DIE contains attributes specifying the function's name, entry point address, parameter types, and local variable locations.

Key Design Insight: Symbol resolution must be **lazy and cached** because DWARF parsing is computationally expensive. The debugger parses only the information needed for current operations and caches results for subsequent lookups.

Variable Value Inspection

The debugger reads and displays variable values by combining DWARF location information with ptrace memory access capabilities. This feature completes the source-level debugging experience by allowing developers to examine program state using the same variable names they wrote in source code.





Variable Inspection Capability	Technical Approach	User Experience
Local Variable Access	Parse DWARF location expressions, read from stack or registers	User types <code>print local_var</code> , sees formatted value
Global Variable Access	Use symbol table addresses, read from process memory	Global variables accessible by name from any execution point
Struct Member Access	Calculate field offsets from DWARF type information	User can examine <code>person.age</code> or <code>node->next</code> with proper formatting

Variable Inspection Capability	Technical Approach	User Experience
Array Element Access	Apply stride calculations to base addresses	Array indexing works with proper bounds checking

Variable inspection requires coordinating three complex subsystems: DWARF location expression evaluation, ptrace memory access, and type-aware value formatting. DWARF location expressions specify how to find a variable's current storage location, which might be in a CPU register, on the stack at a specific offset, or at a global memory address. These expressions can be quite complex, involving arithmetic operations and conditional logic to handle compiler optimizations.

Once the debugger determines a variable's location, it uses ptrace operations to read the raw bytes. For memory locations, `ptrace(PTRACE_PEEKDATA, pid, address, NULL)` reads a word-sized value. For register locations, `ptrace(PTRACE_GETREGS, pid, NULL, ®isters)` retrieves the complete register set. The debugger then extracts the specific register or memory region containing the variable's value.

Type information from DWARF enables proper value interpretation and formatting. A sequence of bytes `0x42 0x00 0x00` `0x00` might represent the integer 66, the float 1.47e-43, a pointer to address 0x42, or four separate character values, depending on the variable's type. The debugger uses DWARF type DIEs to interpret raw bytes correctly and format them for human readability.

Explicit Non-Goals

Clearly defining what our debugger will NOT support is just as important as specifying its capabilities. These non-goals prevent scope creep and allow us to focus on implementing the core debugging primitives exceptionally well rather than attempting to build a comprehensive debugging environment.

Multi-Threading Support

Our debugger explicitly targets single-threaded programs only and will not implement any multi-threading debugging capabilities. This represents the most significant limitation but also the most important simplification for a learning project.

Multi-Threading Aspect	Complexity Introduced	Our Approach
Thread Creation/Destruction	Must track thread lifecycle events, handle <code>CLONE</code> ptrace events	Assume target program never creates threads
Thread-Specific State	Each thread has separate registers, stack, instruction pointer	Single global process state structure
Thread Synchronization	Breakpoints must coordinate across threads, stepping becomes complex	No coordination needed
Concurrent Memory Access	Variable inspection must handle concurrent modifications	Assume memory state is stable

Multi-threaded debugging introduces exponential complexity because threads can interact in unpredictable ways. When one thread hits a breakpoint, the debugger must decide whether to stop all threads or allow others to continue running. If other threads continue, they might modify memory that the developer is examining, leading to confusing and inconsistent debugging experiences.

Thread-specific breakpoints add another layer of complexity. Should a breakpoint set on a function stop execution when ANY thread calls that function, or only when a specific thread does? How should the debugger handle conditional breakpoints that depend on thread-local variables? These questions require sophisticated threading models that distract from learning core debugging concepts.

Scope Limitation Rationale: Multi-threading support would approximately triple the codebase size and require deep understanding of thread synchronization, scheduler internals, and concurrent programming models. These topics, while valuable, are orthogonal to the core goal of understanding debugger implementation.

Remote Debugging Protocol

Our debugger operates only on local processes running on the same machine as the debugger itself. We will not implement any network protocol or remote debugging capabilities.

Remote Debugging Feature	Implementation Requirements	Local-Only Alternative
GDB Remote Protocol	Complex packet-based protocol, network error handling	Direct ptrace system calls
Target Communication	Serialization, checksums, connection management	In-memory data structures
Network Security	Authentication, encryption, privilege management	Local process permissions
Platform Abstraction	Protocol must work across different architectures	Single local architecture

Remote debugging protocols like GDB's Remote Serial Protocol (RSP) require implementing a complete client-server architecture with packet serialization, error recovery, and connection management. The protocol itself includes dozens of packet types for different debugging operations, each with specific formatting requirements and error conditions.

Network communication introduces failure modes that don't exist in local debugging: connection timeouts, packet loss, network partitions, and latency issues. Handling these robustly requires significant additional code that doesn't contribute to understanding core debugging principles.

Graphical User Interface

Our debugger provides only a command-line interface (CLI) and will not include any graphical user interface components. This decision allows us to focus entirely on the debugging engine without the distraction of UI framework selection, event handling, and presentation logic.

GUI Component	Development Overhead	CLI Alternative
Window Management	Framework selection, layout design, event loops	Text-based command interface
Visual Breakpoint Display	Source code rendering, syntax highlighting	Address-based breakpoint listing
Variable Inspection Trees	Tree widgets, expandable nodes, data binding	Structured text output with indentation
Memory Visualization	Hex editors, memory maps, graphical layouts	Hexdump-style memory display

GUI development typically consumes 50-70% of application development time due to the complexity of user interaction handling, visual design decisions, and cross-platform compatibility issues. For a learning project focused on debugger internals, this represents a massive distraction from the core educational objectives.

Modern debugging workflows increasingly rely on IDE integration rather than standalone GUI debuggers anyway. Professional developers use debugging capabilities built into Visual Studio Code, IntelliJ, or similar IDEs rather than launching separate graphical debugging applications.

Advanced Optimization Handling

Our debugger assumes that target programs are compiled with minimal optimization (`-O0` or `-O1`) and will not attempt to handle the complex scenarios introduced by aggressive compiler optimizations.

Optimization Challenge	Why It's Complex	Our Assumption
Inlined Functions	No function call/return, breakpoints difficult to place	Functions exist as separate callable entities
Variable Elimination	Optimized-out variables have no storage location	All variables have concrete memory or register locations
Code Reordering	Source lines execute in different order than written	Source line order matches execution order
Register Allocation	Variables move between registers unpredictably	Variable locations remain stable during inspection

Heavily optimized code presents debugging challenges that require sophisticated compiler knowledge and heuristics to handle properly. Variables might be completely eliminated by the optimizer, leaving no way to inspect their values. Functions might be inlined, eliminating the function call boundaries that debuggers typically use for breakpoint placement and stack frame analysis.

DWARF debug information for optimized code becomes significantly more complex, including location lists that specify how variable locations change throughout function execution, and complex expressions that describe how optimized values can be reconstructed from available information.

Educational Trade-off: While understanding optimization-aware debugging is valuable for production debugger development, it requires deep compiler expertise that overshadows the fundamental debugging concepts we aim to teach.

Advanced Debugging Features

Several advanced debugging capabilities remain outside our scope, including watchpoints, conditional breakpoints, reverse debugging, and core file analysis.

Advanced Feature	Implementation Complexity	Learning Value vs Cost
Watchpoints	Hardware debug register management, memory protection	High complexity, minimal conceptual benefit
Conditional Breakpoints	Expression parser, evaluation engine	Significant scope expansion
Reverse Debugging	Complete execution state recording, replay engine	Massive implementation effort
Core File Analysis	Process state reconstruction, offline debugging	Different problem domain

Watchpoints require managing hardware debug registers and coordinating with the memory management unit to detect memory access patterns. Different processors provide different watchpoint capabilities, requiring architecture-specific code that complicates portability.

Conditional breakpoints need a complete expression parser and evaluation engine that can interpret user-provided conditions in the context of the current program state. This essentially requires building a subset of the target

programming language's expression evaluator within the debugger.

Reverse debugging demands recording complete program execution state so that execution can be reversed to previous points. This requires either complete memory state snapshots or deterministic replay systems, both of which represent massive engineering undertakings.

Architecture Decision Records

Our goals and non-goals drive several fundamental architecture decisions that shape the entire debugger design. Each decision represents a conscious trade-off that prioritizes simplicity and educational value over comprehensive functionality.

Decision: Single-Threaded Target Process Only

- **Context:** Multi-threaded debugging requires complex thread state management, synchronization between debugger and multiple threads, and handling of race conditions during debugging operations
- **Options Considered:** Full multi-threading support, limited threading (debug one thread at a time), or single-threaded only
- **Decision:** Single-threaded target processes only
- **Rationale:** Multi-threading would triple codebase complexity while teaching concepts orthogonal to core debugging principles. Single-threaded debugging demonstrates all fundamental techniques without threading complications
- **Consequences:** Debugger cannot debug multi-threaded applications, but implementation remains focused and manageable for learning purposes

Decision: Local Process Debugging Only

- **Context:** Remote debugging requires network protocols, serialization, error recovery, and cross-platform compatibility considerations
- **Options Considered:** Full remote debugging protocol, local-only debugging, or stub-based remote debugging
- **Decision:** Local process debugging only using direct ptrace calls
- **Rationale:** Network protocols add significant implementation complexity without teaching additional debugging concepts. Local debugging with ptrace provides full access to all debugging primitives
- **Consequences:** Cannot debug processes on remote machines, but eliminates network complexity and allows focus on core debugging algorithms

Decision: Command-Line Interface Only

- **Context:** User interface design represents a major implementation effort that doesn't contribute to understanding debugging internals
- **Options Considered:** Full GUI with visual debugging, web-based interface, or command-line interface
- **Decision:** Command-line interface with structured text output
- **Rationale:** GUI development typically consumes majority of application development time. CLI allows complete focus on debugging engine implementation while still providing full functionality
- **Consequences:** Less user-friendly than graphical debuggers, but development effort remains concentrated on educational objectives

Decision Factor	Weight	GUI Approach	CLI Approach	Winner
Development Time	High	6-12 months	2-3 months	CLI
Core Learning Value	High	Low (UI focused)	High (engine focused)	CLI
User Experience	Medium	Excellent	Good	GUI
Implementation Complexity	High	Very High	Moderate	CLI
Maintainability	Medium	Complex	Simple	CLI

Decision: DWARF Debug Information Only

- **Context:** Multiple debug information formats exist (DWARF, COFF, PDB) with different capabilities and complexity levels
- **Options Considered:** Multiple format support, DWARF only, or symbol tables only
- **Decision:** DWARF debug information format exclusively
- **Rationale:** DWARF is the standard debug format for Unix/Linux systems and provides comprehensive source-level debugging information. Supporting multiple formats adds complexity without educational benefit
- **Consequences:** Debugger works only with DWARF-equipped binaries (GCC/Clang output), but implementation can focus on thorough DWARF support rather than format abstraction

Implementation Guidance

The goals and non-goals established in this section directly influence every implementation decision throughout the debugger development process. The following guidance helps translate these high-level requirements into concrete development practices.

Technology Recommendations

Component	Simple Option	Advanced Option	Recommended for Learning
Process Control	Direct ptrace system calls with basic error handling	libunwind for stack unwinding, advanced signal handling	Direct ptrace calls
Debug Information	Basic DWARF parsing with hardcoded structure offsets	libdwarf library for comprehensive DWARF support	Basic parsing with manual DIE traversal
Memory Access	ptrace PEEKDATA/POKEDATA with word-size operations	Process memory mapping via /proc/pid/mem	ptrace operations for portability
Command Interface	Simple scanf-based command parsing	Readline library with command completion and history	Simple parsing initially, readline later

The simple options provide complete functionality while keeping implementation complexity manageable. Advanced options offer better user experience and more robust operation but require understanding additional libraries and APIs that don't contribute to core debugging knowledge.

Recommended Project Structure

Our goals and non-goals suggest a straightforward project organization that separates the four main functional areas while maintaining simplicity:

```
debugger/
├── src/
│   ├── main.c
│   ├── process_control.c
│   ├── process_control.h
│   ├── breakpoints.c
│   ├── breakpoints.h
│   ├── symbols.c
│   ├── symbols.h
│   ├── variables.c
│   ├── variables.h
│   └── common.h
├── tests/
│   ├── test_programs/
│   │   ├── simple.c
│   │   ├── loops.c
│   │   └── structs.c
│   └── unit_tests/
└── docs/
    └── commands.md
└── Makefile
```

← CLI interface and command parsing
← Process attachment, ptrace operations
← Process control interface definitions
← Breakpoint management and INT3 handling
← Breakpoint data structures and interface
← DWARF parsing and symbol resolution
← Symbol table and debug info structures
← Variable inspection and value formatting
← Variable location and type definitions
← Shared data structures and error codes

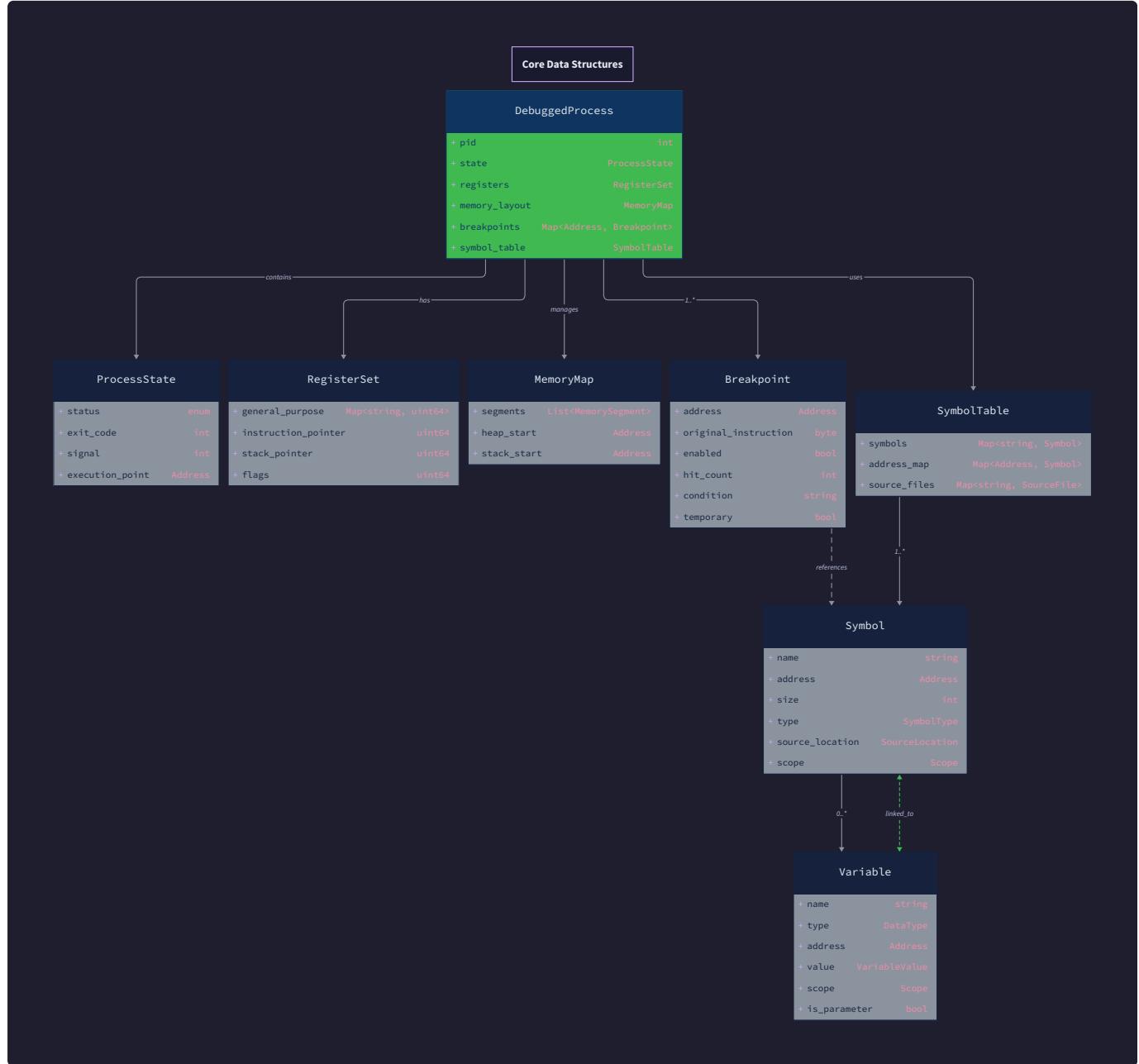
← Simple C programs for testing debugger
← Basic function calls and variables
← Loop constructs for stepping tests
← Struct and array variable tests
← Unit tests for individual components

← Command reference for CLI interface
← Build configuration with debug flags

This organization separates concerns cleanly while avoiding over-engineering. Each `.c` file corresponds to one of our four main functional requirements, making it easy to develop and test incrementally according to the milestone progression.

Core Data Structures

The functional requirements drive the design of several key data structures that appear throughout the system. These structures directly reflect our scope decisions and functional priorities.



Process Representation (process_control.h):

```
// Represents a debugged process with all necessary state for single-threaded debugging
C

typedef struct {

    pid_t pid;                      // Process ID for ptrace operations

    ProcessState state;              // RUNNING, STOPPED, EXITED

    struct user_regs_struct regs;   // CPU register state (x86-specific)

    char *executable_path;          // Path to executable for symbol loading

    SymbolTable *symbols;           // Parsed debug information

    BreakpointList *breakpoints;   // Active breakpoints in this process

    int signal_pending;             // Last signal received from process

} DebuggedProcess;

// TODO: Initialize process structure after successful ptrace attachment

// TODO: Update register state after each step or continue operation

// TODO: Clean up resources when process exits or debugging session ends
```

Breakpoint Management (breakpoints.h):

```
// Software breakpoint with full lifecycle management
C

typedef struct Breakpoint {

    uintptr_t address;               // Memory address where breakpoint is set

    uint8_t original_byte;           // Original instruction byte (for restoration)

    bool is_enabled;                 // Whether breakpoint is currently active

    uint32_t hit_count;              // Number of times breakpoint has been hit

    char *location_description;     // Human-readable location (e.g., "main+42")

    struct Breakpoint *next;         // Linked list for breakpoint collection

} Breakpoint;

// TODO: Implement breakpoint setting with INT3 instruction patching

// TODO: Handle breakpoint hits with instruction pointer adjustment

// TODO: Implement temporary breakpoint removal for single-step continuation
```

Error Handling Strategy

Our single-threaded, local-only scope simplifies error handling significantly compared to a full-featured debugger. However, robust error handling remains critical for a positive learning experience.

Error Category	Detection Method	Recovery Strategy	User Experience
ptrace Failures	Check return values, examine errno	Display specific error message, maintain debugger state	Clear error description with suggested fixes
Process Death	waitpid returns child exit status	Clean up debugger state, return to command prompt	Inform user of process termination with exit code
Invalid Commands	Command parsing validation	Display usage information	Show available commands and correct syntax
Symbol Lookup Failures	DWARF parsing errors, missing debug info	Fall back to address-based operations	Warn about missing debug information, continue with addresses

Infrastructure Code for Error Handling (common.h):

```
// Comprehensive error reporting for debugging operations C

typedef enum {

    DEBUG_SUCCESS = 0,

    DEBUG_PTRACE_FAILED,

    DEBUG_PROCESS_EXITED,

    DEBUG_INVALID_ADDRESS,

    DEBUG_SYMBOL_NOT_FOUND,

    DEBUG_BREAKPOINT_EXISTS,

    DEBUG_NO_DEBUG_INFO

} DebugResult;

typedef struct {

    DebugResult code;

    char message[256];

    const char *function;

    int line;

    int system_errno;

} DebugError;

// Complete error reporting infrastructure ready for use

#define DEBUG_ERROR(code, msg) debug_error_create(code, msg, __FUNCTION__, __LINE__, errno)

DebugError debug_error_create(DebugResult code, const char *message,
                             const char *function, int line, int sys_errno) {

    DebugError error = {0};

    error.code = code;

    error.function = function;

    error.line = line;

    error.system_errno = sys_errno;

    sprintf(error.message, sizeof(error.message), "%s", message);

    return error;
}
```

```
}

void debug_error_print(const DebugError *error) {

    fprintf(stderr, "Error in %s:%d - %s", error->function, error->line, error->message);

    if (error->system_errno != 0) {

        fprintf(stderr, " (system error: %s)", strerror(error->system_errno));

    }

    fprintf(stderr, "\n");

}
```

Command Interface Framework

Our CLI-only decision enables a simple but effective command processing framework that can be implemented quickly and extended incrementally.

Core Command Processing (main.c):

```
// Simple but extensible command processing for debugger CLI C

typedef struct {

    const char *name;

    const char *help;

    DebugResult (*handler)(DebuggedProcess *proc, const char *args);

} Command;

// TODO: Implement command parsing that splits input into command and arguments

// TODO: Add command lookup that finds handler function by name

// TODO: Add help system that displays available commands and usage

// TODO: Add command history for repeated operations (set multiple breakpoints)

// Skeleton command handlers for core functionality

DebugResult cmd_break(DebuggedProcess *proc, const char *args) {

    // TODO: Parse address or function name from args

    // TODO: Validate address is within executable memory range

    // TODO: Call breakpoint_set() with parsed address

    // TODO: Display confirmation message with breakpoint location

}

DebugResult cmd_continue(DebuggedProcess *proc, const char *args) {

    // TODO: Check if process is in stopped state

    // TODO: Call ptrace(PTRACE_CONT, ...) to resume execution

    // TODO: Call waitpid() to wait for next breakpoint or signal

    // TODO: Update process state based on wait result

}

DebugResult cmd_step(DebuggedProcess *proc, const char *args) {

    // TODO: Check if process is in stopped state

    // TODO: Call ptrace(PTRACE_SINGLESTEP, ...) for one instruction

    // TODO: Update register state after step completes

    // TODO: Display new instruction pointer and source location if available
```

```
}
```

Milestone Validation Checkpoints

Each functional requirement corresponds to a specific milestone with concrete validation criteria that verify correct implementation.

Milestone 1 Checkpoint - Process Control:

- Compile test program: `gcc -g -O0 test_programs/simple.c -o test_programs/simple`
- Run debugger: `./debugger test_programs/simple`
- Verify process attachment: Should see "Attached to process [pid]" message
- Test single stepping: `step` command should advance instruction pointer by small amount
- Test continue: `continue` command should run until program exits
- Expected behavior: Process runs under debugger control, all ptrace operations succeed

Milestone 2 Checkpoint - Breakpoints:

- Set breakpoint by address: `break 0x[main_address]` (find address with `objdump -t`)
- Set breakpoint by function: `break main`
- Run program: `continue` should stop at breakpoint with "Breakpoint hit at 0x..." message
- Verify instruction restoration: Continuing after breakpoint should execute normally
- Test multiple breakpoints: Setting 2-3 breakpoints should work independently

Milestone 3 Checkpoint - Symbol Resolution:

- Compile with debug info: Ensure `-g` flag produces DWARF information
- Verify debug sections: `readelf -S test_programs/simple | grep debug` shows debug sections
- Test function lookup: `break function_name` should resolve to correct address
- Test address-to-line mapping: Breakpoint hits should show source file and line number
- Validate symbol parsing: `info symbols` command should list available functions

Milestone 4 Checkpoint - Variable Inspection:

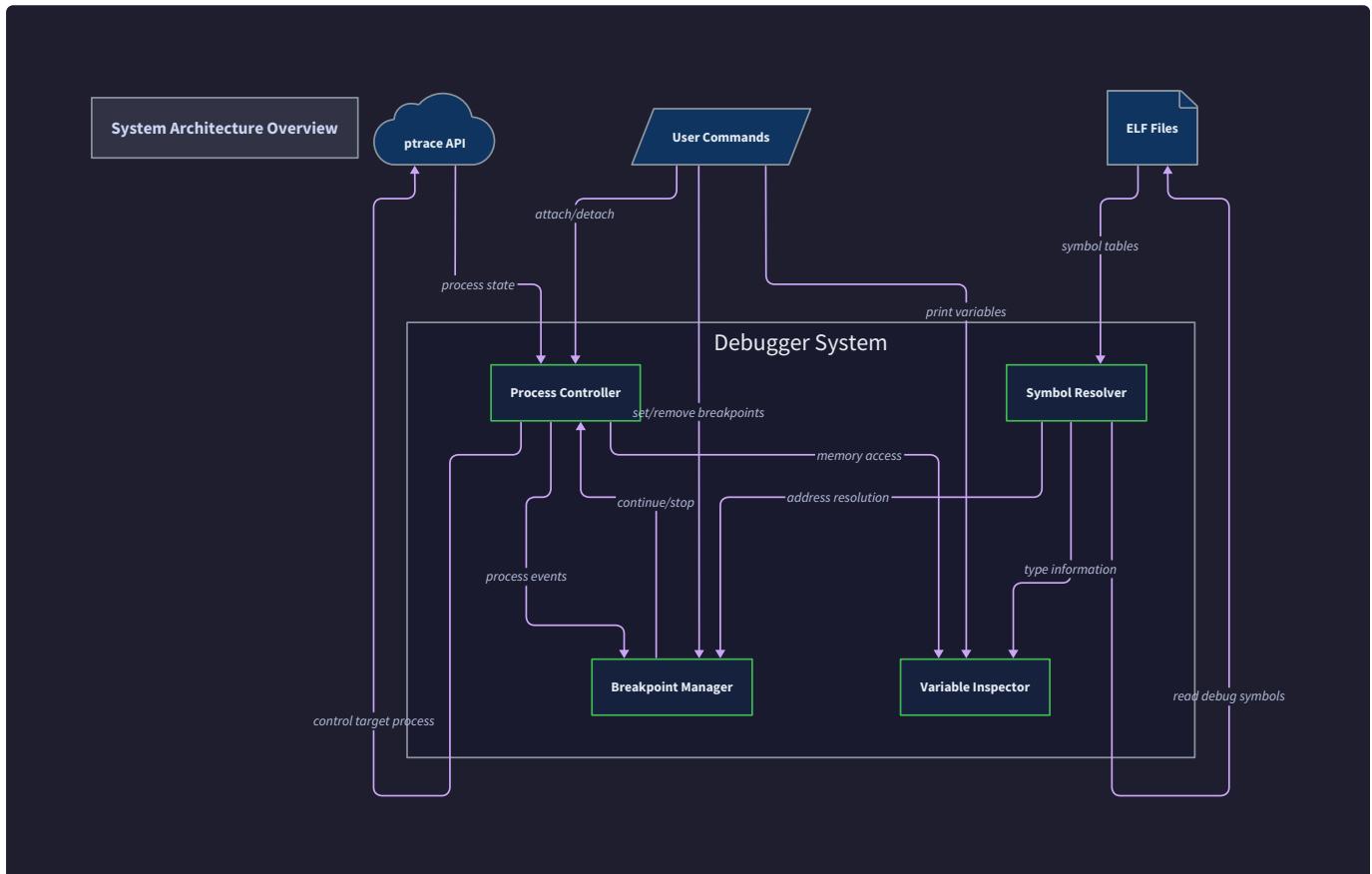
- Set breakpoint in function with local variables
- Hit breakpoint and examine variables: `print variable_name` shows correct value
- Test different types: int, char, float, and pointer variables display correctly
- Verify struct member access: `print struct_var.field_name` works correctly
- Test scope handling: Variables only accessible when in correct function scope

High-Level Architecture

Milestone(s): All milestones — this section establishes the architectural foundation and component relationships that will be implemented across all four milestones

Component Overview

Think of our debugger architecture like a modern emergency response system. When a 911 call comes in, it doesn't go to just one person who handles everything. Instead, the call is routed through a **dispatcher** (our main debugger loop) who coordinates with specialized teams: the **paramedics** (process control), **bomb squad** (breakpoint management), **detective unit** (symbol resolution), and **forensics team** (variable inspection). Each team has specialized tools and expertise, but they must work together seamlessly to handle the emergency effectively.



Our debugger follows this same principle of specialized coordination. The system is built around four core components, each with distinct responsibilities but tight integration points. This separation allows each component to focus on its domain expertise while providing clean interfaces for coordination.

The Process Controller acts as the primary interface to the operating system's process control mechanisms. It owns all interactions with ptrace system calls, manages the lifecycle of the debugged process, and handles Unix signals. Think of it as the puppet master that can start, stop, and manipulate the target process at the instruction level. This component encapsulates the complex world of process attachment, signal handling, and execution control into a clean interface that other components can use without understanding ptrace internals.

The Breakpoint Manager specializes in the delicate art of instruction patching and breakpoint lifecycle management. Like a skilled burglar who can pick locks without leaving traces, this component must carefully modify the target process's instruction stream, preserve original code, and restore everything seamlessly. It maintains the mapping between logical breakpoint locations and the physical INT3 patches in memory, handling the complex state transitions when breakpoints are hit, disabled, or removed.

The Symbol Resolver serves as the translation layer between the machine-level execution world and the human-readable source code world. It parses the complex DWARF debug information embedded in ELF binaries, building bidirectional mappings between memory addresses and source locations. This component transforms cryptic memory

addresses like `0x401234` into meaningful locations like `main.c:42` and resolves function names like `calculate_average` into their entry point addresses.

The Variable Inspector combines symbol information with live process state to provide real-time variable inspection. It interprets DWARF location expressions to determine where variables live (registers, stack offsets, or global memory), uses ptrace to read those locations, and formats the raw bytes according to type information. This component handles the complexity of different storage locations, data types, and composite structures to present variables in human-readable form.

The following table details each component's core responsibilities and interfaces:

Component	Primary Responsibility	Key Data Managed	External Dependencies
Process Controller	ptrace operations, process lifecycle, signal handling	Process state, register contents, execution mode	ptrace system calls, Unix signals, waitpid
Breakpoint Manager	Instruction patching, breakpoint state, hit detection	Breakpoint list, original instructions, hit counts	Process Controller for memory access
Symbol Resolver	Debug information parsing, address/symbol mapping	Function symbols, source line mappings, type info	ELF/DWARF parsing libraries, binary file access
Variable Inspector	Variable location resolution, value reading, type formatting	Variable cache, type definitions, location expressions	Symbol Resolver + Process Controller

Each component maintains internal state and provides well-defined interfaces to other components. The Process Controller is foundational — all other components depend on it for memory access and execution control. The Symbol Resolver is independent and builds its mappings from static debug information. The Breakpoint Manager and Variable Inspector both depend on the Process Controller but can operate independently of each other.

Design Insight: The component boundaries are drawn along expertise lines rather than data lines. Each component encapsulates a specific type of complexity (ptrace semantics, instruction patching, debug format parsing, or type interpretation) so that the other components don't need to understand these details.

Recommended File Structure

A well-organized file structure is crucial for managing the complexity of a multi-component system. Our debugger uses a hierarchical organization that separates concerns while keeping related functionality together. Think of it like organizing a professional kitchen — similar tools are grouped together, frequently used items are easily accessible, and there's a logical flow that makes sense to anyone working in the space.

The project structure reflects both the component architecture and the development workflow. Core components live in separate directories with clear boundaries, while shared utilities and data structures have their own spaces. This organization supports incremental development — you can build and test each component in isolation before integrating them together.

```
debugger/
├── src/                                # All source code
|   ├── main.c                            # Main entry point and command loop
|   ├── debugger.h                         # Main header with shared types
|   ├── process_control/                  # Process Controller component
|   |   ├── process_control.h             # Process control interface
|   |   ├── process_control.c            # ptrace operations and process lifecycle
|   |   ├── signal_handler.c            # Signal management and forwarding
|   |   └── process_state.c             # Process state tracking and registers
|   ├── breakpoints/                     # Breakpoint Manager component
|   |   ├── breakpoints.h              # Breakpoint interface and types
|   |   ├── breakpoint_manager.c       # Breakpoint lifecycle and storage
|   |   ├── instruction_patch.c       # INT3 patching and restoration
|   |   └── breakpoint_list.c          # Breakpoint collection management
|   ├── symbols/                          # Symbol Resolver component
|   |   ├── symbols.h                # Symbol resolution interface
|   |   ├── elf_parser.c              # ELF format parsing
|   |   ├── dwarf_parser.c            # DWARF debug information parsing
|   |   ├── symbol_table.c            # Symbol storage and lookup
|   |   └── address_mapping.c         # Address-to-source bidirectional mapping
|   ├── variables/                      # Variable Inspector component
|   |   ├── variables.h              # Variable inspection interface
|   |   ├── variable_inspector.c      # Main variable inspection logic
|   |   ├── location_resolver.c       # DWARF location expression interpreter
|   |   ├── type_formatter.c           # Type-aware value formatting
|   |   └── memory_reader.c           # Memory and register value reading
|   ├── common/                           # Shared utilities and data structures
|   |   ├── error_handling.h          # Error types and handling utilities
|   |   ├── error_handling.c          # Error creation and reporting
|   |   ├── debug_types.h             # Common types across all components
|   |   ├── memory_utils.c            # Memory allocation and management
|   |   └── string_utils.c            # String manipulation utilities
|   └── commands/                        # User command interface
|       ├── commands.h                # Command system interface
|       ├── command_parser.c          # Parse user input into commands
|       ├── command_dispatcher.c       # Route commands to appropriate handlers
|       └── builtin_commands.c        # Implementation of built-in commands
└── include/                            # Public headers for external use
    └── debugger_api.h                 # Public API if used as library
tests/
├── unit/                               # Unit tests for each component
|   ├── test_process_control.c
|   ├── test.breakpoints.c
|   ├── test_symbols.c
|   └── test_variables.c
|   └── integration/                  # Integration tests
|       ├── test_breakpoint_workflow.c
|       ├── test_symbol_resolution.c
|       └── test_debugging_session.c
|   └── fixtures/                     # Test programs and data files
|       ├── simple_program.c          # Basic test program with debug info
|       ├── complex_program.c         # Program with structs, arrays, functions
|       └── optimized_program.c       # Optimized code for edge case testing
└── tools/                               # Development and debugging utilities
    ├── elf_inspector.c              # Utility to examine ELF files
    ├── dwarf_dumper.c               # Utility to dump DWARF information
    └── memory_dumper.c              # Utility to examine process memory
docs/
└── api_reference.md                    # Component API documentation
```

```
|   └── troubleshooting.md      # Common problems and solutions
|   └── examples/              # Usage examples and tutorials
├── Makefile                  # Build configuration
└── README.md                 # Project overview and quick start
└── LICENSE                   # License information
```

This structure supports several important development practices:

Component Isolation: Each major component lives in its own directory with a clear header file defining the public interface. This makes it easy to understand what each component provides without diving into implementation details.

Incremental Development: You can build and test each component independently. The Makefile should support targets like `make process_control` or `make test_breakpoints` to build specific parts of the system.

Shared Infrastructure: Common utilities live in the `common/` directory and can be used by any component without creating circular dependencies. This includes error handling, memory management, and data structure utilities.

Testing Infrastructure: The test structure mirrors the source structure, making it easy to find tests for specific functionality. Integration tests verify that components work together correctly.

Development Tools: The `tools/` directory contains utilities for debugging the debugger itself — examining ELF files, dumping DWARF information, and inspecting memory layouts.

Best Practice: Each `.c` file should have a corresponding `.h` file that defines its public interface. Keep private functions and data structures out of header files to maintain clean component boundaries.

The header file organization follows a dependency hierarchy. Lower-level headers like `debug_types.h` define basic types used throughout the system. Component headers like `process_control.h` define component-specific interfaces. The main `debugger.h` header ties everything together and provides the unified interface used by the command loop.

Inter-Component Communication

The four components communicate through well-defined interfaces rather than direct access to each other's internal data structures. Think of this like departments in a hospital — the radiology department doesn't directly access patient charts from cardiology. Instead, they use standardized forms and procedures to request information and share results. This ensures that each department can change its internal processes without breaking other departments.

Our debugger uses a similar pattern of structured communication through interfaces, shared data structures, and event-driven coordination. The Process Controller serves as the central hub because all debugging operations ultimately require process memory access or execution control. However, the other components maintain their autonomy and can be developed, tested, and modified independently.

Communication Patterns and Data Flow

The communication follows several distinct patterns depending on the type of operation being performed. Understanding these patterns is crucial for implementing clean component boundaries and avoiding tight coupling that would make the system fragile and hard to maintain.

Communication Pattern	Components Involved	Data Flow Direction	Trigger
Command Dispatching	All components ← Commands	User command → appropriate component	User input
Symbol Lookup	Variable Inspector → Symbol Resolver	Request symbol info → return mappings	Variable inspection request
Memory Access	Breakpoint Manager → Process Controller	Request memory read/write → return data	Breakpoint operations
Address Resolution	Commands → Symbol Resolver	Function name → return address	Setting breakpoint by name
Variable Location	Variable Inspector → Symbol Resolver → Process Controller	Variable name → location → memory read	Print variable command
Breakpoint Notification	Process Controller → Breakpoint Manager → Commands	Signal received → breakpoint hit → user notification	SIGTRAP from debuggee

Shared Data Structures

Components share information through carefully designed data structures that act as contracts between components. These structures are defined in `common/debug_types.h` and provide stable interfaces that components can depend on without knowing implementation details.

The central data structure is `DebuggedProcess`, which represents the complete state of a debugging session:

Field	Type	Purpose	Owner Component
pid	pid_t	Process ID of debugged program	Process Controller
state	ProcessState	Current execution state (RUNNING/STOPPED/EXITED)	Process Controller
regs	struct user_regs_struct	CPU register contents	Process Controller
executable_path	char*	Path to executable being debugged	Process Controller
symbols	SymbolTable*	Symbol and debug information	Symbol Resolver
breakpoints	BreakpointList*	Active breakpoints	Breakpoint Manager
signal_pending	int	Pending signal number or 0	Process Controller

This structure is passed between components but each component only modifies the fields it owns. This provides coordination without tight coupling — components can read other components' data but must use interface functions to request changes.

Interface Definitions

Each component provides a clean interface that encapsulates its functionality and hides implementation details. These interfaces are designed to be stable — internal changes to a component shouldn't require changes to other components.

Process Controller Interface:

Function	Parameters	Returns	Purpose
process_attach	pid_t target_pid, DebuggedProcess* proc	DebugResult	Attach to existing process
process_start	char* executable_path, char** argv, DebuggedProcess* proc	DebugResult	Start new process under debugger control
process_continue	DebuggedProcess* proc	DebugResult	Resume execution until next signal
process_single_step	DebuggedProcess* proc	DebugResult	Execute exactly one instruction
process_read_memory	DebuggedProcess* proc, uintptr_t addr, void* buffer, size_t size	DebugResult	Read memory from debugger
process_write_memory	DebuggedProcess* proc, uintptr_t addr, void* data, size_t size	DebugResult	Write memory to debugger
process_get_registers	DebuggedProcess* proc	DebugResult	Update register contents in proc->regs

Breakpoint Manager Interface:

Function	Parameters	Returns	Purpose
breakpoint_set	DebuggedProcess* proc, uintptr_t address, char* description	DebugResult	Set breakpoint at address
breakpoint_remove	DebuggedProcess* proc, uintptr_t address	DebugResult	Remove breakpoint
breakpoint_enable	DebuggedProcess* proc, uintptr_t address	DebugResult	Enable existing breakpoint
breakpoint_disable	DebuggedProcess* proc, uintptr_t address	DebugResult	Disable breakpoint (keep in list)
breakpoint_handle_hit	DebuggedProcess* proc, uintptr_t address	DebugResult	Process breakpoint hit
breakpoint_list	DebuggedProcess* proc, Breakpoint** list, size_t* count	DebugResult	Get all breakpoints

Symbol Resolver Interface:

Function	Parameters	Returns	Purpose
symbols_load	char* executable_path, SymbolTable** symbols	DebugResult	Load debug information from ELF file
symbols_resolve_address	SymbolTable* symbols, uintptr_t address, char** filename, int* line	DebugResult	Convert address to source location
symbols_resolve_function	SymbolTable* symbols, char* function_name, uintptr_t* address	DebugResult	Convert function name to address
symbols_find_variable	SymbolTable* symbols, char* variable_name, uintptr_t pc, VariableInfo* info	DebugResult	Find variable information at given PC
symbols_get_function_bounds	SymbolTable* symbols, uintptr_t address, uintptr_t* start, uintptr_t* end	DebugResult	Get function start and end addresses

Variable Inspector Interface:

Function	Parameters	Returns	Purpose
variable_inspect	DebuggedProcess* proc, char* variable_name, char* output_buffer, size_t buffer_size	DebugResult	Inspect variable by name
variable_inspect_address	DebuggedProcess* proc, uintptr_t address, char* type_name, char* output_buffer, size_t buffer_size	DebugResult	Inspect memory location as specific type
variable_list_locals	DebuggedProcess* proc, char*** variable_names, size_t* count	DebugResult	List local variables in current scope
variable_read_register	DebuggedProcess* proc, int register_number, uint64_t* value	DebugResult	Read CPU register value

Event-Driven Coordination

The components coordinate through an event-driven pattern when handling runtime events like signals from the debugged process. This pattern avoids tight coupling while ensuring that relevant components can respond to important events.

When a signal arrives from the debugged process, the coordination follows this sequence:

1. **Process Controller** receives the signal through `waitpid` and determines the signal type
2. If the signal is `SIGTRAP`, it indicates a breakpoint hit or single-step completion
3. **Process Controller** reads the current instruction pointer from the process registers
4. **Process Controller** calls `breakpoint_handle_hit` to notify the **Breakpoint Manager**
5. **Breakpoint Manager** checks if there's a breakpoint at that address and handles the hit
6. **Breakpoint Manager** adjusts the instruction pointer back by one byte (INT3 is one byte)
7. **Process Controller** updates the process registers with the adjusted instruction pointer
8. The user interface is notified that a breakpoint was hit with location information

This sequence demonstrates how components maintain their responsibilities while coordinating effectively. The Process Controller handles signal management, the Breakpoint Manager handles breakpoint-specific logic, but neither component needs to know the other's implementation details.

Architecture Decision: Interface-Based Communication

- **Context:** Components need to share information and coordinate operations without creating tight coupling that would make the system fragile and hard to modify.
- **Options Considered:** Direct data structure access, message passing, interface functions
- **Decision:** Interface functions with shared data structures
- **Rationale:** Interface functions provide encapsulation and allow internal changes without breaking other components. Shared data structures avoid copying overhead while maintaining clear ownership. Message passing would add unnecessary complexity for a single-process debugger.
- **Consequences:** Clean component boundaries enable independent development and testing. Interface stability requirements mean careful design of function signatures. Slight performance overhead from function calls vs. direct access.

Error Propagation

All component interfaces use the standardized `DebugResult` enum for error reporting, which enables consistent error handling throughout the system. When an operation fails deep in one component, the error can be propagated up through the call chain with context preserved.

Error Code	Meaning	Recovery Strategy
DEBUG_SUCCESS	Operation completed successfully	Continue normal operation
DEBUG_PTRACE_FAILED	ptrace system call failed	Check process state, may need reattachment
DEBUG_PROCESS_EXITED	Target process has exited	End debugging session
DEBUG_INVALID_ADDRESS	Address not valid in target process	Report error to user, continue session
DEBUG_SYMBOL_NOT_FOUND	Symbol name not found in debug info	Report error to user, continue session
DEBUG_BREAKPOINT_EXISTS	Breakpoint already set at address	Report to user, no action needed
DEBUG_NO_DEBUG_INFO	ELF file lacks debug information	Limited functionality, warn user

The error handling system includes context information to help with debugging and user feedback:

DebugError Field	Type	Purpose
code	DebugResult	Primary error classification
message	char[256]	Human-readable error description
function	const char*	Function where error occurred
line	int	Source line where error occurred
system_errno	int	System errno if relevant

This comprehensive error system enables robust error handling while providing useful debugging information when things go wrong.

⚠ Pitfall: Component Circular Dependencies

A common mistake is creating circular dependencies between components, such as having the Symbol Resolver call back into the Process Controller directly. This creates tight coupling and makes testing difficult. Instead, use dependency injection — pass the needed interface functions as parameters or callbacks, or structure the call flow so that higher-level code coordinates between components rather than having them call each other directly.

⚠ Pitfall: Shared Mutable State

Another frequent error is allowing multiple components to modify the same data structure concurrently without coordination. For example, both the Process Controller and Breakpoint Manager might try to update the process state simultaneously. Establish clear ownership rules — only the owning component should modify its data fields, and other components should use interface functions to request changes.

Implementation Guidance

The debugger architecture requires careful coordination between system programming, binary format parsing, and user interface concerns. Understanding which technologies to use for each component and how to structure the code will save significant development time and prevent common architectural mistakes.

Technology Recommendations Table:

Component	Simple Option	Advanced Option
Process Control	Raw ptrace system calls with manual signal handling	libunwind or similar library for stack walking
Binary Parsing	Manual ELF parsing with mmap for file access	libdwarf or libelf for robust format handling
Memory Management	Standard malloc/free with manual cleanup	Memory pools with automatic cleanup on error
Error Handling	Return codes with errno for system errors	Structured error objects with stack traces
Build System	Simple Makefile with manual dependencies	CMake with automatic dependency detection
Testing Framework	Custom assert macros with manual test runners	Unity or similar C testing framework

For learning purposes, start with the simple options to understand the underlying mechanisms, then upgrade to advanced options as the codebase grows more complex.

Recommended File Structure Implementation:

Start with this minimal structure and expand as you implement each milestone:

```
debugger/
├── src/
|   ├── debugger.h          # Main header - start here
|   ├── main.c              # Entry point - minimal for now
|   ├── common/
|   |   ├── debug_types.h    # Core types - implement first
|   |   └── error_handling.c # Error system - critical foundation
|   └── process_control/
|       ├── process_control.h # First component interface
|       └── process_control.c # Milestone 1 implementation
└── tests/
    └── fixtures/
        └── simple_program.c # Basic test target
└── Makefile                # Start with basic build
```

Expand the structure as you complete each milestone, adding new component directories and moving shared code into `common/` as patterns emerge.

Core Data Structures Starter Code:

```
// src/common/debug_types.h

#ifndef DEBUG_TYPES_H
#define DEBUG_TYPES_H

#include <sys/types.h>
#include <sys/user.h>
#include <stdint.h>
#include <stdbool.h>

// Process execution states

typedef enum {
    PROCESS_RUNNING,
    PROCESS_STOPPED,
    PROCESS_EXITED
} ProcessState;

// Standardized error codes for all components

typedef enum {
    DEBUG_SUCCESS = 0,
    DEBUG_PTRACE_FAILED,
    DEBUG_PROCESS_EXITED,
    DEBUG_INVALID_ADDRESS,
    DEBUG_SYMBOL_NOT_FOUND,
    DEBUG_BREAKPOINT_EXISTS,
    DEBUG_NO_DEBUG_INFO
} DebugResult;

// Forward declarations for component data structures

typedef struct SymbolTable SymbolTable;
typedef struct BreakpointList BreakpointList;
typedef struct VariableInfo VariableInfo;
```

```
// Central state for debugging session

typedef struct {

    pid_t pid;

    ProcessState state;

    struct user_regs_struct regs;

    char* executable_path;

    SymbolTable* symbols;

    BreakpointList* breakpoints;

    int signal_pending;

} DebuggedProcess;

// Individual breakpoint representation

typedef struct Breakpoint {

    uintptr_t address;

    uint8_t original_byte;

    bool is_enabled;

    uint32_t hit_count;

    char* location_description;

    struct Breakpoint* next;

} Breakpoint;

// Error information with context

typedef struct {

    DebugResult code;

    char message[256];

    const char* function;

    int line;

    int system_errno;

} DebugError;

#endif // DEBUG_TYPES_H
```

Error Handling Infrastructure (Complete Implementation):

```
// src/common/error_handling.c

#include "error_handling.h"

#include <stdio.h>

#include <string.h>

#include <errno.h>

DebugError debug_error_create(DebugResult code, const char* message,
                             const char* function, int line, int system_errno) {

    DebugError error = {0};

    error.code = code;

    error.function = function;

    error.line = line;

    error.system_errno = system_errno;

    snprintf(error.message, sizeof(error.message), "%s", message);

    return error;
}

void debug_error_print(const DebugError* error) {

    const char* error_type;

    switch (error->code) {

        case DEBUG_SUCCESS: error_type = "SUCCESS"; break;

        case DEBUG_PTRACE_FAILED: error_type = "PTRACE_FAILED"; break;

        case DEBUG_PROCESS_EXITED: error_type = "PROCESS_EXITED"; break;

        case DEBUG_INVALID_ADDRESS: error_type = "INVALID_ADDRESS"; break;

        case DEBUG_SYMBOL_NOT_FOUND: error_type = "SYMBOL_NOT_FOUND"; break;

        case DEBUG_BREAKPOINT_EXISTS: error_type = "BREAKPOINT_EXISTS"; break;

        case DEBUG_NO_DEBUG_INFO: error_type = "NO_DEBUG_INFO"; break;

        default: error_type = "UNKNOWN"; break;
    }
}
```

```
}

fprintf(stderr, "[%s] %s", error_type, error->message);

if (error->function && error->line > 0) {

    fprintf(stderr, " (in %s:%d)", error->function, error->line);
}

if (error->system_errno != 0) {

    fprintf(stderr, " - %s", strerror(error->system_errno));
}

fprintf(stderr, "\n");

}

// Convenience macro for creating errors with automatic function/line info

#define DEBUG_ERROR(code, message) \
    debug_error_create(code, message, __func__, __LINE__, errno)
```

Component Interface Skeletons:

```
// src/process_control/process_control.h

#ifndef PROCESS_CONTROL_H
#define PROCESS_CONTROL_H

#include "../common/debug_types.h"

// Process lifecycle management

DebugResult process_start(const char* executable_path, char* const argv[],
                         DebuggedProcess* proc);

DebugResult process_attach(pid_t target_pid, DebuggedProcess* proc);

DebugResult process_detach(DebuggedProcess* proc);

// Execution control

DebugResult process_continue(DebuggedProcess* proc);

DebugResult process_single_step(DebuggedProcess* proc);

DebugResult process_stop(DebuggedProcess* proc);

// Memory and register access

DebugResult process_read_memory(DebuggedProcess* proc, uintptr_t address,
                                void* buffer, size_t size);

DebugResult process_write_memory(DebuggedProcess* proc, uintptr_t address,
                                 const void* data, size_t size);

DebugResult process_get_registers(DebuggedProcess* proc);

DebugResult process_set_registers(DebuggedProcess* proc);

// Signal handling

DebugResult process_wait_for_signal(DebuggedProcess* proc);

DebugResult process_handle_signal(DebuggedProcess* proc, int signal);

#endif // PROCESS_CONTROL_H
```

Process Control Component Skeleton:

```
// src/process_control/process_control.c

#include "process_control.h"

#include <sys/ptrace.h>

#include <sys/wait.h>

#include <unistd.h>

#include <errno.h>

DebugResult process_start(const char* executable_path, char* const argv[],

                         DebuggedProcess* proc) {

    // TODO 1: Fork a child process

    // TODO 2: In child: call PTRACE_TRACEME and exec the target program

    // TODO 3: In parent: wait for child to stop with SIGTRAP

    // TODO 4: Initialize DebuggedProcess structure with child PID

    // TODO 5: Set process state to PROCESS_STOPPED

    // Hint: Child will automatically stop after exec due to PTRACE_TRACEME

}

DebugResult process_continue(DebuggedProcess* proc) {

    // TODO 1: Check that process is in STOPPED state

    // TODO 2: Call ptrace(PTRACE_CONT, pid, NULL, NULL)

    // TODO 3: Update process state to RUNNING

    // TODO 4: Optionally call process_wait_for_signal to wait for next stop

    // Hint: PTRACE_CONT resumes execution until next signal

}

DebugResult process_single_step(DebuggedProcess* proc) {

    // TODO 1: Check that process is in STOPPED state

    // TODO 2: Call ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL)

    // TODO 3: Wait for process to stop again (should be immediate)

    // TODO 4: Update register contents with process_get_registers

    // Hint: SINGLESTEP executes exactly one instruction then stops
```

```
}

DebugResult process_read_memory(DebuggedProcess* proc, uintptr_t address,
                                void* buffer, size_t size) {

    // TODO 1: Check that process is attached and not exited

    // TODO 2: Use ptrace(PTRACE_PEEKDATA, ...) to read word-sized chunks

    // TODO 3: Handle partial words at the beginning and end of the range

    // TODO 4: Copy data into user buffer

    // Hint: PTRACE_PEEKDATA reads sizeof(long) bytes at word-aligned addresses

}

DebugResult process_write_memory(DebuggedProcess* proc, uintptr_t address,
                                 const void* data, size_t size) {

    // TODO 1: Check that process is attached and not exited

    // TODO 2: For partial words, read-modify-write to preserve surrounding bytes

    // TODO 3: Use ptrace(PTRACE_POKEDATA, ...) to write word-sized chunks

    // TODO 4: Handle alignment and partial words correctly

    // Hint: PTRACE_POKEDATA writes sizeof(long) bytes at word-aligned addresses

}
```

Build System Foundation:

```

# Makefile
CC = gcc
CFLAGS = -Wall -Wextra -g -std=c99 -D_GNU_SOURCE
INCLUDES = -Isrc/common -Isrc

# Directories
SRCDIR = src
OBJDIR = obj
TESTDIR = tests

# Find all source files
SOURCES = $(shell find $(SRCDIR) -name "*.c")
OBJECTS = $(SOURCES:$(SRCDIR)/%.c=$(OBJDIR)/%.o)

# Main targets
DEBUGGER = debugger
TEST_RUNNER = test_runner

.PHONY: all clean test

all: $(DEBUGGER)

$(DEBUGGER): $(OBJECTS)
    $(CC) $(OBJECTS) -o $@

$(OBJDIR)/%.o: $(SRCDIR)/%.c
    @mkdir -p $(dir $@)
    $(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

# Component-specific builds for development
process_control: $(OBJDIR)/process_control/process_control.o $(OBJDIR)/common/error_handling.o
    $(CC) $^ -o test_process_control

clean:
    rm -rf $(OBJDIR) $(DEBUGGER) $(TEST_RUNNER)

test: $(TEST_RUNNER)
    ./$(TEST_RUNNER)

# Debugging the debugger
debug: $(DEBUGGER)
    gdb ./$(DEBUGGER)

```

Milestone 1 Verification Checkpoint:

After implementing the Process Controller component, verify correct operation with these specific tests:

- 1. Basic Attachment Test:** Create a simple program that prints numbers in a loop. Your debugger should be able to start it and immediately receive a SIGTRAP when the process stops after exec.
- 2. Single Step Verification:** Single-step through the first few instructions. The instruction pointer should advance by small amounts (1-4 bytes per instruction on x86-64). Print the instruction pointer value after each step to verify progress.
- 3. Continue and Stop Test:** Use continue to let the program run, then send SIGSTOP to interrupt it. The debugger should regain control and be able to continue or single-step again.

4. **Memory Read Test:** Read a few bytes from the process's memory at the current instruction pointer. The bytes should match the machine code instructions of your test program (you can verify this with objdump).

Expected output for basic functionality:

```
$ ./debugger tests/fixtures/simple_program
Attached to process 12345
Process stopped at 0x401000
(debugger) step
Single step completed, PC now at 0x401003
(debugger) continue
Process continued, waiting for next stop...
```

Signs something is wrong and what to check:

- **Process won't stop after exec:** Check that child process calls `PTRACE_TRACEME` before `exec`
- **ptrace calls fail with EPERM:** Ensure debugger has permission, try running as same user as target
- **Single step doesn't advance:** Verify you're using `PTRACE_SINGLESTEP` and waiting for the process to stop
- **Memory reads return garbage:** Check address alignment for `PTRACE_PEEKDATA` and handle partial word reads

Language-Specific Implementation Hints:

For C development with ptrace:

- Use `#define _GNU_SOURCE` before system includes to access Linux-specific ptrace operations
- The `struct user_regs_struct` layout is architecture-specific; use `#ifdef` blocks for portability
- Always check ptrace return values — they return -1 on failure and set `errno`
- Use `waitpid()` with `WUNTRACED` flag to catch stopped (not just exited) processes
- Memory addresses from ptrace are in the target process's virtual address space, not your debugger's

Common C pitfalls to avoid:

- Forgetting to handle `EINTR` from system calls — retry interrupted operations
- Using `PTRACE_PEEKDATA` on unaligned addresses — it only works on word boundaries
- Not saving `errno` before making other system calls that might overwrite it
- Assuming little-endian byte order when reading multi-byte values from target process memory

This implementation foundation provides the architectural structure and infrastructure needed to build each component incrementally while maintaining clean boundaries and robust error handling throughout the system.

Data Model

Milestone(s): All milestones — this section defines the core data structures that will be built progressively, starting with basic process representation in Milestone 1 and extending through symbol tables in Milestone 3 and variable inspection in Milestone 4

The Blueprint Analogy

Think of the data model as the blueprint for a detective's case file system. Just as a detective maintains organized files containing suspect profiles, evidence catalogs, witness statements, and crime scene maps, our debugger maintains structured data representing the debugged process, breakpoint locations, symbol mappings, and variable information. Each data structure serves as a specialized filing system that allows quick access to critical information during the investigation. The relationships between these structures mirror how a detective cross-references evidence — a breakpoint address links to symbol information, which connects to source code locations, which relate to variable scope and values.

The data model forms the central nervous system of our debugger architecture. Every component — process control, breakpoint management, symbol resolution, and variable inspection — operates by reading and modifying these core data structures. The design must balance several competing requirements: fast lookup performance during debugging sessions, memory efficiency for large programs with extensive debug information, and extensibility for future features. Most critically, the data model must maintain consistency across concurrent operations, as ptrace operations and symbol lookups may happen simultaneously while the user inspects variables.

Process State Representation

The `DebuggedProcess` structure serves as the central hub containing all information about the process under debugger control. This structure represents the debugger's complete view of the target process, including its execution state, register contents, memory layout, and attached debugging metadata.

Decision: Single Process Structure vs Separate State Objects

- **Context:** We need to track multiple aspects of process state including execution status, CPU registers, loaded symbols, and active breakpoints
- **Options Considered:**
 1. Single monolithic `DebuggedProcess` structure containing all state
 2. Separate structures for each concern with loose coupling
 3. Hierarchical composition with separate state objects
- **Decision:** Single `DebuggedProcess` structure with embedded pointers to specialized collections
- **Rationale:** Debugging operations frequently need access to multiple state aspects simultaneously (e.g., hitting a breakpoint requires process state, register access, and symbol lookup). A central structure provides cache locality and simplifies state consistency, while pointer fields to specialized collections allow modular management of complex subsystems like symbol tables and breakpoint lists.
- **Consequences:** Enables atomic state updates and simplified error handling, but requires careful memory management for the embedded collections and may create tight coupling between components

Field Name	Type	Description
pid	pid_t	Process identifier assigned by the operating system for ptrace operations
state	ProcessState	Current execution state (RUNNING, STOPPED, EXITED) for controlling debugger operations
regs	struct user_regs_struct	Complete CPU register set captured via PTRACE_GETREGS for inspection and modification
executable_path	char*	Full filesystem path to the executable file for symbol table loading and display
symbols	SymbolTable*	Pointer to parsed debug information and symbol mappings for source-level debugging
breakpoints	BreakpointList*	Pointer to collection of active and disabled breakpoints with their metadata
signal_pending	int	Signal number waiting to be delivered to the process, or 0 if none pending

The `ProcessState` enumeration captures the fundamental execution states that drive debugger behavior and user interface updates. Each state represents a different capability set and determines which debugging operations are valid.

State	Description	Valid Operations	Transitions
RUNNING	Process executing normally under ptrace control	Signal delivery, forced stop via SIGSTOP	To STOPPED on signal/breakpoint, to EXITED on termination
STOPPED	Process halted and waiting for debugger commands	Single-step, continue, memory/register access, breakpoint modification	To RUNNING on continue/step, to EXITED on kill
EXITED	Process terminated, but debugger session still active	Symbol lookup, breakpoint inspection, session cleanup	Terminal state — no outbound transitions

The critical insight here is that process state determines the validity of ptrace operations. Attempting to read registers from a RUNNING process will fail, while trying to continue an EXITED process is meaningless. The state field serves as both a safety check and a user interface driver.

Process state transitions occur in response to specific events, and the debugger must handle each transition correctly to maintain system consistency:

1. **Process startup:** Fork creates child, child calls `PTRACE_TRACEME`, parent receives `SIGTRAP`, state becomes STOPPED
2. **Breakpoint hit:** Running process executes INT3, kernel delivers `SIGTRAP`, state transitions to STOPPED, instruction pointer adjusted
3. **Signal delivery:** External signal sent to process, kernel suspends execution, debugger receives notification via `waitpid`, state becomes STOPPED

4. **Single-step completion:** Process executes one instruction via `PTRACE_SINGLESTEP`, hardware trap fires, state returns to STOPPED
5. **Process termination:** Target process exits normally or crashes, kernel cleans up, debugger receives SIGCHLD, state becomes EXITED

The register structure `user_regs_struct` is architecture-specific but typically contains general-purpose registers, instruction pointer, stack pointer, and status flags. On x86-64, this includes RAX through R15, RIP, RSP, and RFLAGS. The debugger captures register state immediately after each stop event to provide accurate variable inspection and control flow analysis.

Decision: Embedded vs Referenced Register Storage

- **Context:** CPU registers change frequently during debugging and must be captured atomically after each process stop
- **Options Considered:**
 1. Embed `user_regs_struct` directly in `DebuggedProcess`
 2. Dynamically allocate register structure and store pointer
 3. Lazy loading with cache invalidation on process state changes
- **Decision:** Embed register structure directly in the process structure
- **Rationale:** Register access is extremely frequent during debugging operations (every variable lookup, stack trace, or instruction analysis requires register values). Embedding eliminates pointer indirection and memory allocation overhead. The structure size (~200 bytes on x86-64) is reasonable for stack allocation.
- **Consequences:** Simplifies memory management and improves cache performance, but increases `DebuggedProcess` size and couples the structure to architecture-specific layouts

Breakpoint Data Structures

Software breakpoints represent the core mechanism for interactive debugging, requiring careful management of instruction patching, original byte preservation, and hit detection. The `Breakpoint` structure encapsulates all metadata necessary to implement reliable breakpoint functionality.

Decision: Linked List vs Hash Table for Breakpoint Storage

- **Context:** Debuggers typically manage 1-50 breakpoints simultaneously, requiring both address-based lookup (for hit detection) and iteration (for display/management commands)
- **Options Considered:**
 1. Singly-linked list with linear search by address
 2. Hash table with address as key for O(1) lookup
 3. Sorted array with binary search for address lookup
- **Decision:** Singly-linked list with address-based linear search
- **Rationale:** Small breakpoint counts make hash table overhead unjustified. Linear search through 10-20 breakpoints is faster than hash computation and collision handling. Linked lists provide simple insertion/deletion and natural iteration order. Memory locality is less critical for infrequent operations like breakpoint management.
- **Consequences:** Enables simple implementation and memory management with O(n) lookup performance, but will degrade with very large breakpoint counts (100+) and requires pointer chasing

Field Name	Type	Description
address	uintptr_t	Memory address where INT3 instruction is patched, used for hit detection and restoration
original_byte	uint8_t	Original instruction byte that was replaced with 0xCC, preserved for execution restoration
is_enabled	bool	Whether breakpoint is currently active (INT3 patched) or temporarily disabled
hit_count	uint32_t	Number of times this breakpoint has been triggered during the debugging session
location_description	char*	Human-readable description like "main+0x42" or "foo.c:15" for user interface display
next	struct Breakpoint*	Pointer to next breakpoint in the linked list, NULL for list termination

The breakpoint lifecycle follows a precise sequence to maintain program correctness while providing debugging capabilities. Each breakpoint exists in one of several logical states that determine its behavior and management requirements:

1. **Allocation:** Breakpoint structure created with target address and description, but not yet installed in target process
2. **Installation:** Original byte read via `PTRACE_PEEKDATA`, saved in `original_byte` field, INT3 (0xCC) written via `PTRACE_POKEDATA`
3. **Armed:** Breakpoint active with INT3 instruction in place, waiting for process execution to reach the address
4. **Hit:** Process execution reaches breakpoint address, CPU executes INT3, kernel delivers SIGTRAP to debugger
5. **Restoration:** Original byte temporarily restored, instruction pointer decremented by 1, process ready for continuation
6. **Removal:** Original byte permanently restored, breakpoint structure deallocated, memory address returns to normal execution

The most subtle aspect of breakpoint management is the temporary restoration dance. When a breakpoint hits, the debugger must restore the original instruction, execute it, then potentially re-install the breakpoint. This requires careful coordination between instruction pointer manipulation and memory patching.

Breakpoint hit detection requires precise signal handling and address matching. The sequence occurs as follows:

1. Target process executes instruction at breakpoint address
2. CPU encounters INT3 (0xCC) byte, generates debug exception
3. Kernel converts exception to SIGTRAP signal, suspends process
4. Debugger's `waitpid` call returns with signal information
5. Debugger reads instruction pointer via `PTRACE_GETREGS`
6. Instruction pointer points one byte past the INT3 (on x86/x64)
7. Debugger decrements instruction pointer by 1 to get breakpoint address
8. Address lookup in breakpoint list identifies which breakpoint was hit
9. Hit count incremented, user notified, awaiting continuation command

The `BreakpointList` structure provides collection management for the linked list of breakpoints, encapsulating common operations and maintaining list integrity:

Operation	Parameters	Returns	Description
<code>breakpoint_list_create</code>	None	<code>BreakpointList*</code>	Allocates and initializes empty breakpoint list with NULL head pointer
<code>breakpoint_list_add</code>	<code>list</code> , <code>address</code> , <code>description</code>	<code>DebugResult</code>	Creates new breakpoint, checks for duplicates, links into list
<code>breakpoint_list_find</code>	<code>list</code> , <code>address</code>	<code>Breakpoint*</code>	Linear search by address, returns matching breakpoint or NULL
<code>breakpoint_list_remove</code>	<code>list</code> , <code>address</code>	<code>DebugResult</code>	Finds breakpoint by address, unlinks from list, deallocates memory
<code>breakpoint_list_enable</code>	<code>list</code> , <code>address</code>	<code>DebugResult</code>	Sets <code>is_enabled</code> to true, patches INT3 into target process memory
<code>breakpoint_list_disable</code>	<code>list</code> , <code>address</code>	<code>DebugResult</code>	Sets <code>is_enabled</code> to false, restores original byte to target process

Symbol and Debug Information

Debug information forms the bridge between machine-level execution and source code, enabling the debugger to translate addresses to function names, line numbers, and variable locations. The symbol resolution system must efficiently manage large amounts of DWARF data while providing fast lookup operations during interactive debugging sessions.

Decision: Eager vs Lazy DWARF Parsing

- **Context:** Modern programs contain megabytes of DWARF debug information, but debuggers typically access only small subsets during any session
- **Options Considered:**
 1. Eager parsing: Load all DWARF information at startup into memory structures
 2. Lazy parsing: Parse DWARF sections on-demand as symbols are requested
 3. Hybrid approach: Parse compilation unit headers eagerly, DIEs lazily
- **Decision:** Hybrid approach with compilation unit indexing and lazy DIE parsing
- **Rationale:** Startup time is critical for debugger usability — parsing megabytes of DWARF can take seconds. However, symbol lookups during debugging must be fast (sub-100ms). Indexing compilation units provides fast source file lookup, while lazy DIE parsing avoids memory bloat for unused symbols.
- **Consequences:** Enables fast startup and efficient memory usage with acceptable lookup performance, but complicates the parsing implementation and requires careful cache management for frequently accessed symbols

The `SymbolTable` structure provides the primary interface for symbol resolution operations, maintaining both parsed DWARF data and lookup acceleration structures:

Field Name	Type	Description
<code>compilation_units</code>	<code>CompilationUnit*</code>	Array of compilation unit headers for source file organization
<code>cu_count</code>	<code>size_t</code>	Number of compilation units in the executable for array bounds checking
<code>function_index</code>	<code>FunctionSymbol*</code>	Hash table mapping function names to entry addresses for breakpoint-by-name
<code>address_ranges</code>	<code>AddressRange*</code>	Sorted array of address ranges mapping memory locations to compilation units
<code>line_table</code>	<code>LineTableEntry*</code>	Address-to-line-number mapping for source-level stepping and display
<code>line_count</code>	<code>size_t</code>	Number of line table entries for binary search operations
<code>variable_cache</code>	<code>VariableInfo*</code>	LRU cache of recently accessed variable location information
<code>dwarf_sections</code>	<code>DwarfSectionInfo</code>	Pointers and sizes of DWARF sections (.debug_info, .debug_line, etc.)

Compilation units represent the fundamental organizational structure of DWARF debug information, corresponding to individual source files compiled into the executable. Each compilation unit contains debug information entries (DIEs) describing functions, variables, and types defined in that source file:

Field Name	Type	Description
source_filename	char*	Primary source file name for this compilation unit (e.g., "main.c")
compilation_directory	char*	Working directory during compilation for relative path resolution
low_pc	uintptr_t	Lowest memory address covered by this compilation unit
high_pc	uintptr_t	Highest memory address covered by this compilation unit
dies_offset	size_t	Byte offset into .debug_info section where DIE tree begins
line_table_offset	size_t	Byte offset into .debug_line section for this compilation unit
functions	FunctionSymbol*	Linked list of functions defined in this compilation unit
global_variables	VariableInfo*	Linked list of global variables defined in this compilation unit

Function symbols provide the mapping between human-readable function names and their memory addresses, enabling users to set breakpoints by name rather than address:

Field Name	Type	Description
name	char*	Function name as it appears in source code (e.g., "calculate_distance")
mangled_name	char*	C++ mangled name for overloaded functions, NULL for C functions
entry_address	uintptr_t	Memory address of function's first instruction for breakpoint placement
end_address	uintptr_t	Memory address after function's last instruction for range checking
source_file	char*	Source filename where function is defined
declaration_line	uint32_t	Line number where function is declared in source file
parameter_count	uint8_t	Number of function parameters for call stack analysis
return_type	TypeInfo*	Pointer to return type information for value formatting

Address-to-line mapping enables the debugger to translate memory addresses back to source code locations, supporting source-level stepping and current location display. The line table is derived from DWARF .debug_line section information:

Field Name	Type	Description
address	uintptr_t	Memory address of the instruction generated from this source line
source_file	char*	Source filename containing this line
line_number	uint32_t	Line number within the source file (1-based)
column_number	uint32_t	Column number within the line (1-based), 0 if unknown
is_statement	bool	Whether this address represents a source statement boundary
is_basic_block	bool	Whether this address begins a new basic block for optimization analysis

The line table is sorted by address to enable binary search lookup. When the debugger needs to display the current source location, it searches for the largest address less than or equal to the current instruction pointer. This handles cases where multiple instructions map to the same source line due to compiler optimization.

Symbol resolution operations provide the core functionality for translating between machine and source representations:

Operation	Parameters	Returns	Description
<code>symbols_load_from_elf</code>	<code>executable_path</code> , <code>symbols</code>	<code>DebugResult</code>	Parse ELF headers, locate DWARF sections, build compilation unit index
<code>symbols_resolve_address</code>	<code>symbols</code> , <code>address</code> , <code>filename</code> , <code>line</code>	<code>DebugResult</code>	Binary search line table to find source location for given address
<code>symbols_resolve_function</code>	<code>symbols</code> , <code>function_name</code> , <code>address</code>	<code>DebugResult</code>	Hash table lookup to find entry address for named function
<code>symbols_find_functions_in_file</code>	<code>symbols</code> , <code>filename</code> , <code>functions</code>	<code>DebugResult</code>	Return all functions defined in specified source file
<code>symbols_get_function_at_address</code>	<code>symbols</code> , <code>address</code> , <code>function</code>	<code>DebugResult</code>	Find function containing the specified address using range lookup
<code>symbols_lookup_variable</code>	<code>symbols</code> , <code>name</code> , <code>scope_address</code> , <code>variable</code>	<code>DebugResult</code>	Find variable information within lexical scope at given address

The variable information structure captures the metadata necessary for locating and interpreting variable values within the debugged process:

Field Name	Type	Description
<code>name</code>	<code>char*</code>	Variable name as it appears in source code
<code>type_info</code>	<code>TypeInfo*</code>	Pointer to type information for size and interpretation
<code>location_type</code>	<code>LocationType</code>	Whether variable is in register, memory, or stack frame
<code>location_data</code>	<code>union LocationData</code>	Register number, memory address, or stack frame offset
<code>scope_start</code>	<code>uintptr_t</code>	First address where variable is in scope and accessible
<code>scope_end</code>	<code>uintptr_t</code>	Last address where variable is in scope and accessible
<code>is_parameter</code>	<code>bool</code>	Whether this variable is a function parameter vs local variable
<code>is_optimized_out</code>	<code>bool</code>	Whether compiler optimization eliminated this variable

Decision: Union vs Separate Fields for Variable Location

- **Context:** Variables can be stored in CPU registers, memory addresses, or stack frame offsets, but only one location type applies per variable
- **Options Considered:**
 1. Separate fields for each location type (`register_num`, `memory_addr`, `stack_offset`)
 2. Union with discriminated access based on `location_type` field
 3. Void pointer with type-specific casting
- **Decision:** Tagged union with `LocationType` discriminator
- **Rationale:** Memory efficiency is important for large programs with thousands of variables. Separate fields waste space since only one is valid per variable. Union provides type safety compared to void pointers while clearly indicating which field is valid.
- **Consequences:** Reduces structure size by ~12 bytes per variable with type-safe access, but requires careful union member access and complicates serialization if needed for remote debugging

Common Pitfalls

⚠ **Pitfall: Process State Inconsistency** Many implementations fail to update process state consistently across ptrace operations. For example, calling `process_continue()` but forgetting to update `state` to `RUNNING`, or handling a `SIGTRAP` but leaving state as `RUNNING` instead of `STOPPED`. This leads to operations being attempted on processes in invalid states (like trying to read registers from a running process), causing ptrace failures. Always update the `state` field atomically with ptrace operations, and validate state before attempting operations.

⚠ **Pitfall: Register Structure Staleness** The `regs` field in `DebuggedProcess` contains a snapshot of CPU registers from the last `PTRACE_GETREGS` call. Many implementations forget to refresh this data after single-stepping or continuing execution, leading to stale register values being displayed to users. Always call `PTRACE_GETREGS` immediately after receiving a `SIGTRAP` or other stop signal to ensure register data reflects the current process state.

⚠ **Pitfall: Breakpoint Address Arithmetic Errors** On x86/x64, when a breakpoint hits, the instruction pointer points one byte past the INT3 instruction (not at the breakpoint address). Many implementations fail to adjust for this offset, leading

to incorrect breakpoint identification and restoration. Always decrement the instruction pointer by 1 on x86/x64 when processing SIGTRAP signals to get the actual breakpoint address.

⚠ Pitfall: Original Byte Loss If the debugger crashes or exits improperly while breakpoints are installed, the original instruction bytes remain overwritten with INT3, corrupting the target program. Always implement cleanup handlers that restore original bytes before debugger termination. Consider storing breakpoint information persistently or in shared memory for recovery across debugger restarts.

⚠ Pitfall: Symbol Table Memory Leaks DWARF parsing allocates substantial amounts of memory for strings, structures, and lookup tables. Many implementations fail to properly deallocate this memory when the debugging session ends, causing memory leaks. Implement proper cleanup routines that walk through all symbol structures and free allocated memory. Consider using memory pools or reference counting for complex data structures.

⚠ Pitfall: Case Sensitivity in Symbol Lookup Function and variable names are case-sensitive in most languages, but users often expect case-insensitive lookup (especially for functions). Failing to handle this leads to "symbol not found" errors for valid symbols. Implement both exact-match and case-insensitive lookup modes, or provide helpful error messages suggesting similar symbol names when exact matches fail.

Implementation Guidance

The data model serves as the foundation that all other components build upon. Focus on getting these structures correct early, as changing them later requires updating all components simultaneously.

Technology Recommendations

Component	Simple Option	Advanced Option
Memory Management	Manual malloc/free with cleanup functions	Memory pools with automatic cleanup
String Handling	Standard C strings with strdup/free	String interning with hash table
DWARF Parsing	libdwarf library (if available)	Custom parser reading ELF sections directly
Hash Tables	Simple linear probing	Robin Hood hashing or cuckoo hashing
Error Handling	Error codes with errno	Structured error objects with stack traces

Recommended File Structure

```
project-root/  
  C  
  src/  
    debugger.h          ← Main header with all data structure definitions  
    process.c           ← DebuggedProcess management functions  
    process.h           ← Process-related structure declarations  
    breakpoint.c        ← Breakpoint list operations and management  
    breakpoint.h         ← Breakpoint structure definitions  
    symbols.c           ← Symbol table loading and lookup  
    symbols.h           ← Symbol and DWARF structure definitions  
    variables.c         ← Variable location and type handling  
    variables.h         ← Variable inspection structure definitions  
    error.c             ← Error handling and reporting utilities  
    error.h             ← Error code definitions and structures  
  
  include/  
    debugger_types.h   ← Public type definitions for library users  
  
  tests/  
    test_process.c     ← Unit tests for process management  
    test_breakpoints.c ← Unit tests for breakpoint operations  
    test_symbols.c     ← Unit tests for symbol resolution  
    sample_program.c   ← Test target with known symbols and variables
```

Core Data Structure Definitions

```
// debugger.h - Core type definitions C

#ifndef DEBUGGER_H

#define DEBUGGER_H


#include <sys/types.h>

#include <sys/user.h>

#include <stdint.h>

#include <stdbool.h>

// Forward declarations for complex types

typedef struct SymbolTable SymbolTable;

typedef struct BreakpointList BreakpointList;

typedef struct VariableInfo VariableInfo;

// Process execution states

typedef enum {

    PROCESS_RUNNING,

    PROCESS_STOPPED,

    PROCESS_EXITED

} ProcessState;

// Main process representation

typedef struct {

    pid_t pid;

    ProcessState state;

    struct user_regs_struct regs;

    char* executable_path;

    SymbolTable* symbols;

    BreakpointList* breakpoints;

    int signal_pending;

}
```

```
    } DebuggedProcess;

    // Individual breakpoint metadata

    typedef struct Breakpoint {

        uintptr_t address;

        uint8_t original_byte;

        bool is_enabled;

        uint32_t hit_count;

        char* location_description;

        struct Breakpoint* next;

    } Breakpoint;

    // Result codes for all debugger operations

    typedef enum {

        DEBUG_SUCCESS,

        DEBUG_PTRACE_FAILED,

        DEBUG_PROCESS_EXITED,

        DEBUG_INVALID_ADDRESS,

        DEBUG_SYMBOL_NOT_FOUND,

        DEBUG_BREAKPOINT_EXISTS,

        DEBUG_NO_DEBUG_INFO

    } DebugResult;

    // Structured error information

    typedef struct {

        DebugResult code;

        char message[256];

        const char* function;

        int line;

        int system_errno;

    } DebugError;
```

```
#endif // DEBUGGER_H
```

Process Management Infrastructure

```
// process.c - Complete process management implementation C

#include "debugger.h"

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/ptrace.h>

#include <sys/wait.h>

#include <errno.h>

// Complete process creation and initialization

DebugResult process_create(const char* executable_path, char* const argv[], DebuggedProcess* proc) {

    // TODO 1: Allocate and initialize process structure

    // TODO 2: Fork child process for target program

    // TODO 3: In child: call ptrace(PTRACE_TRACEME) then exec target program

    // TODO 4: In parent: wait for initial SIGTRAP from child

    // TODO 5: Initialize process state to STOPPED

    // TODO 6: Capture initial register state with PTRACE_GETREGS

    // TODO 7: Load symbol table from executable if debug info available

    // TODO 8: Initialize empty breakpoint list

    // Hint: Use fork(), ptrace(PTRACE_TRACEME), execv(), waitpid()

    return DEBUG_SUCCESS;
}

// Process state management with validation

DebugResult process_continue(DebuggedProcess* proc) {

    // TODO 1: Validate process is in STOPPED state

    // TODO 2: Check if any signals are pending delivery

    // TODO 3: Call ptrace(PTRACE_CONT) with pending signal or 0

    // TODO 4: Update process state to RUNNING
```

```
// TODO 5: Return immediately - don't wait for next stop

// Hint: Only waitpid() when you want to block for the next event

return DEBUG_SUCCESS;

}

// Single instruction execution

DebugResult process_single_step(DebuggedProcess* proc) {

    // TODO 1: Validate process is in STOPPED state

    // TODO 2: Call ptrace(PTRACE_SINGLESTEP) with any pending signal

    // TODO 3: Call waitpid() to wait for step completion

    // TODO 4: Check wait status for SIGTRAP indicating step completion

    // TODO 5: Update register state with PTRACE_GETREGS

    // TODO 6: Keep process state as STOPPED

    // Hint: PTRACE_SINGLESTEP automatically stops after one instruction

    return DEBUG_SUCCESS;

}

// Memory access operations

DebugResult process_read_memory(DebuggedProcess* proc, uintptr_t address, void* buffer, size_t size)
{
    // TODO 1: Validate process state allows memory access

    // TODO 2: Read memory word-by-word using PTRACE_PEEKDATA

    // TODO 3: Handle partial word reads at end of buffer

    // TODO 4: Copy data into user buffer with proper alignment

    // Hint: PTRACE_PEEKDATA reads sizeof(long) bytes at aligned addresses

    return DEBUG_SUCCESS;
}
```

Breakpoint Management Infrastructure

```
// breakpoint.c - Complete breakpoint implementation

C

#include "debugger.h"

#include <stdlib.h>

#include <string.h>

#include <sys/ptrace.h>

// Breakpoint list structure (opaque type)

struct BreakpointList {

    Breakpoint* head;

    size_t count;

};

// Complete breakpoint installation

DebugResult breakpoint_set(DebuggedProcess* proc, uintptr_t address, const char* description) {

    // TODO 1: Check if breakpoint already exists at this address

    // TODO 2: Read original byte from target address using PTRACE_PEEKDATA

    // TODO 3: Create and initialize new Breakpoint structure

    // TODO 4: Write INT3 (0xCC) byte using PTRACE_POKEDATA

    // TODO 5: Add breakpoint to process breakpoint list

    // TODO 6: Set is_enabled to true and hit_count to 0

    // Hint: Use (data & 0xFF) to extract single byte from ptrace word

    return DEBUG_SUCCESS;

}

// Breakpoint hit detection and handling

DebugResult breakpoint_handle_hit(DebuggedProcess* proc, uintptr_t hit_address) {

    // TODO 1: Find breakpoint by address in the breakpoint list

    // TODO 2: Increment hit_count for the breakpoint

    // TODO 3: Temporarily restore original byte at the address

    // TODO 4: Decrement instruction pointer by 1 (x86/x64 specific)
```

```
// TODO 5: Update process registers with PTRACE_SETREGS

// TODO 6: Print breakpoint hit message with location description

// Hint: Instruction pointer is in regs.rip (x86-64) or regs.eip (x86)

return DEBUG_SUCCESS;

}

// Breakpoint list search operation

Breakpoint* breakpoint_find_by_address(BreakpointList* list, uintptr_t address) {

    // TODO 1: Iterate through linked list starting from head

    // TODO 2: Compare each breakpoint's address with target address

    // TODO 3: Return pointer to matching breakpoint or NULL if not found

    // Hint: Simple linear search is sufficient for typical breakpoint counts

    return NULL;

}
```

Symbol Table Infrastructure

```
// symbols.c - Symbol table management
// C

#include "debugger.h"

#include <fcntl.h>

#include <sys/mman.h>

#include <elf.h>

// Symbol table structure (opaque type)

struct SymbolTable {

    char* executable_path;

    void* elf_mapping;

    size_t elf_size;

    Elf64_Ehdr* elf_header;

    Elf64_Shdr* section_headers;

    char* section_names;

    // DWARF section pointers

    void* debug_info;

    size_t debug_info_size;

    void* debug_line;

    size_t debug_line_size;

    // Symbol lookup tables

    FunctionSymbol* functions;

    LineTableEntry* line_table;

    size_t line_count;

};

// Complete symbol table loading

DebugResult symbols_load_from_elf(const char* executable_path, SymbolTable** symbols) {

    // TODO 1: Open executable file and memory-map it

    // TODO 2: Validate ELF header magic and architecture
```

```

// TODO 3: Parse section header table and locate section name strings

// TODO 4: Find .debug_info, .debug_line, .debug_abbrev sections

// TODO 5: Parse DWARF compilation units and build function index

// TODO 6: Extract line number table for address-to-source mapping

// TODO 7: Build hash table for function name lookups

// Hint: Use mmap() for efficient large file access

return DEBUG_SUCCESS;

}

// Address to source location mapping

DebugResult symbols_resolve_address(SymbolTable* symbols, uintptr_t address, char** filename,
uint32_t* line) {

    // TODO 1: Binary search line table for largest address <= target

    // TODO 2: Extract source filename and line number from entry

    // TODO 3: Allocate and copy filename string for caller

    // TODO 4: Handle case where address is not in any line table entry

    // Hint: Line table is sorted by address for binary search

    return DEBUG_SUCCESS;

}

```

Milestone Checkpoints

Milestone 1 Checkpoint: After implementing basic data structures, create a simple test program:

```

gcc -g -o test_target test_target.c                                BASH

./debugger test_target

```

Expected behavior: Debugger should attach to process, display "Process attached (PID: XXXX)", show STOPPED state, and accept single-step commands that advance instruction pointer.

Milestone 2 Checkpoint: Test breakpoint functionality:

```
(debugger) break main  
  
Breakpoint set at 0x401126 (main+0x0)  
  
(debugger) continue  
  
Breakpoint hit at 0x401126 (main+0x0)
```

BASH

Expected behavior: Process should stop at breakpoint, display hit message, and allow examination of registers and memory.

Milestone 3 Checkpoint: Test symbol resolution:

```
(debugger) info functions  
  
main (0x401126-0x401145)  
  
calculate (0x401146-0x40115a)  
  
(debugger) break calculate  
  
Breakpoint set at 0x401146 (calculate+0x0)
```

BASH

Expected behavior: Debugger should list all functions, resolve function names to addresses, and display current source location when stopped.

Language-Specific Hints

- Use `mmap()` for memory-mapping ELF files — more efficient than reading large debug sections into memory
- `ptrace()` returns -1 on error with errno set — always check return values and preserve errno
- Use `WIFEXITED()`, `WIFSTOPPED()`, `WTERMSIG()` macros to decode waitpid status
- Structure packing with `__attribute__((packed))` may be needed for binary format parsing
- Consider using `valgrind` to detect memory leaks in symbol table and breakpoint management
- Use `objdump -d` and `readelf -w` to verify your symbol parsing against known-good tools

Process Control Component

Milestone(s): Milestone 1 (Process Control) — this section establishes the foundational process control mechanisms using `ptrace` that enable all subsequent debugging capabilities

The Puppet Master Analogy

Think of the Process Control Component as a skilled puppeteer controlling a marionette. The debugger is the puppeteer, and the target program being debugged is the marionette. Just as a puppeteer uses strings to control every movement of their puppet — making it walk, pause, turn, or perform specific actions — our debugger uses the `ptrace` system call to control every aspect of the target process's execution.

The puppet strings in our analogy are the ptrace requests: `PTRACE_SINGLESTEP` pulls the string to make the puppet take exactly one step forward, `PTRACE_CONT` releases the strings to let the puppet dance freely until something interesting happens, and `PTRACE_ATTACH` is how we first connect our control strings to the puppet. The puppet cannot move on its own while under our control — every action happens only when we explicitly command it.

This mental model captures several crucial aspects of process control. First, the relationship is hierarchical — the puppeteer has complete authority over the puppet's movements. Second, the control is granular — we can dictate individual steps or allow free movement as needed. Third, the puppet can signal back to the puppeteer through its movements (process signals), just as a marionette's position tells the puppeteer what's happening. Finally, if the puppeteer stops paying attention or makes a mistake with the strings, the puppet might fall or become tangled — representing the race conditions and synchronization challenges we must handle carefully.

The Process Control Component serves as the foundation layer for all debugging operations. Without reliable process control, we cannot set breakpoints (the puppet would keep running past them), inspect variables (we couldn't pause to examine memory), or step through code (we couldn't advance instruction by instruction). Every other component depends on our ability to start, stop, and precisely control the target process's execution.

Process Attachment Lifecycle

The process attachment lifecycle represents the sequence of operations required to establish debugger control over a target process. This lifecycle has two primary paths: creating a new process under debugger control, or attaching to an already-running process. Both paths must result in the same end state — a process that is stopped and ready to accept debugger commands.

The **create-and-attach path** begins with the debugger forking itself to create a child process. The child immediately calls `ptrace(PTRACE_TRACEME, 0, NULL, NULL)` to indicate it wants to be traced by its parent, then raises `SIGSTOP` to halt execution before calling `exec` to replace itself with the target program. The parent debugger waits for the `SIGSTOP` signal using `waitpid`, at which point it has full control over the child before any target program code has executed. This approach ensures clean initialization with no race conditions.

The **attach-to-existing path** uses `ptrace(PTRACE_ATTACH, target_pid, NULL, NULL)` to forcibly attach to a running process. The kernel automatically sends `SIGSTOP` to the target process and notifies the debugger when attachment succeeds. However, this path introduces complexity because the target process may be in an unknown state, potentially holding locks or in the middle of critical operations. The debugger must be prepared to handle various process states and signal conditions during attachment.

Attachment Step	Create Path	Attach Path
Process Creation	<code>fork()</code> creates child	Target already exists
Trace Setup	Child calls <code>PTRACE_TRACEME</code>	Parent calls <code>PTRACE_ATTACH</code>
Initial Stop	Child raises <code>SIGSTOP</code>	Kernel sends <code>SIGSTOP</code> to target
Program Loading	Child calls <code>exec(target)</code>	Program already loaded
Debugger Control	Parent waits for <code>SIGSTOP</code>	Parent waits for attachment signal
State Validation	Clean initial state	Must inspect existing state

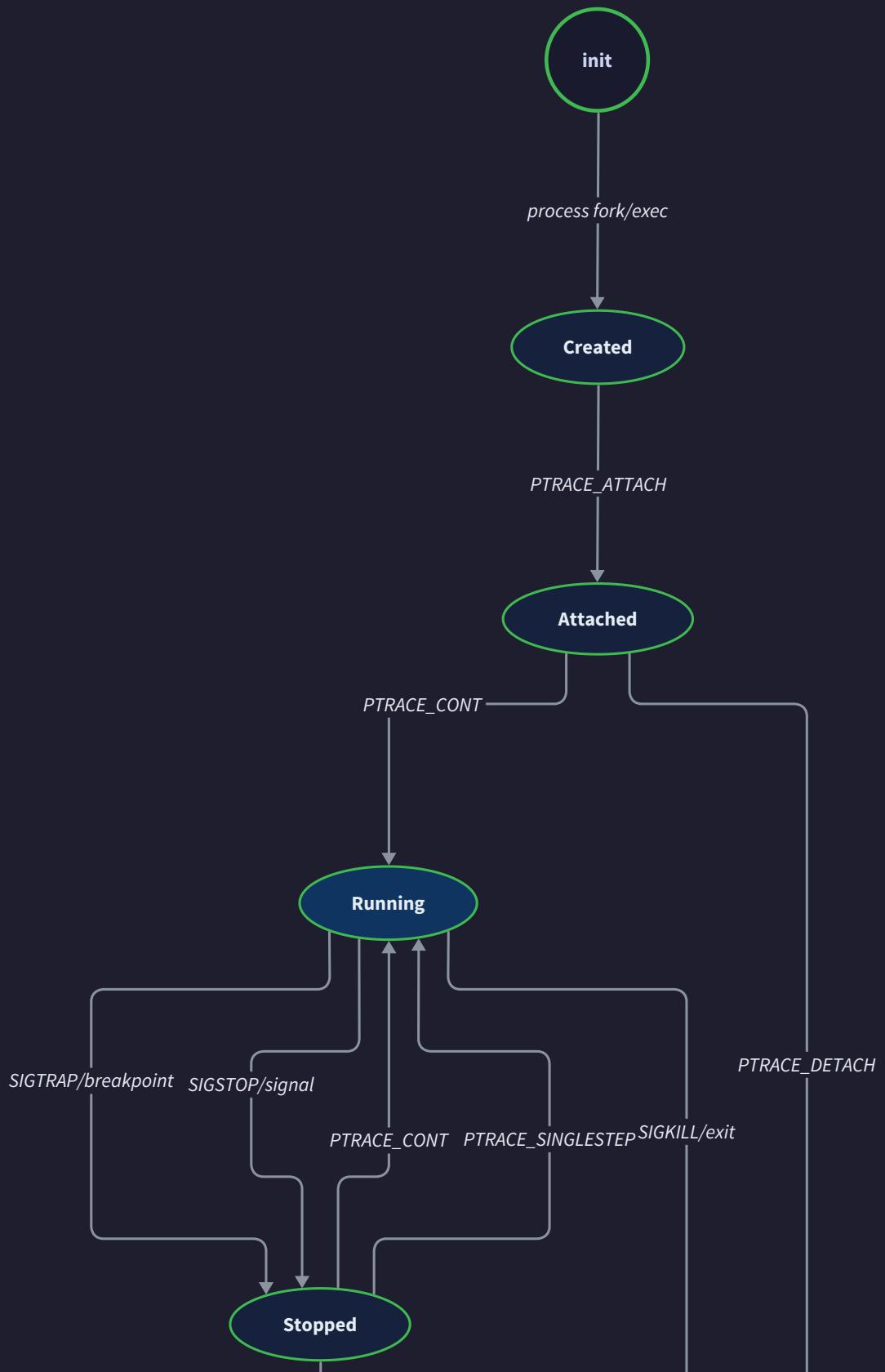
The initialization sequence after successful attachment involves several critical steps. First, the debugger must call `waitpid(pid, &status, 0)` to wait for the initial `SIGSTOP` signal, which confirms the target process is stopped and under debugger control. Next, it reads the process's initial register state using `ptrace(PTRACE_GETREGS, pid, NULL, ®s)` to establish the starting program counter and stack pointer values. The debugger then reads the process memory layout from `/proc/pid/maps` to understand which memory regions are executable, writable, or contain shared libraries.

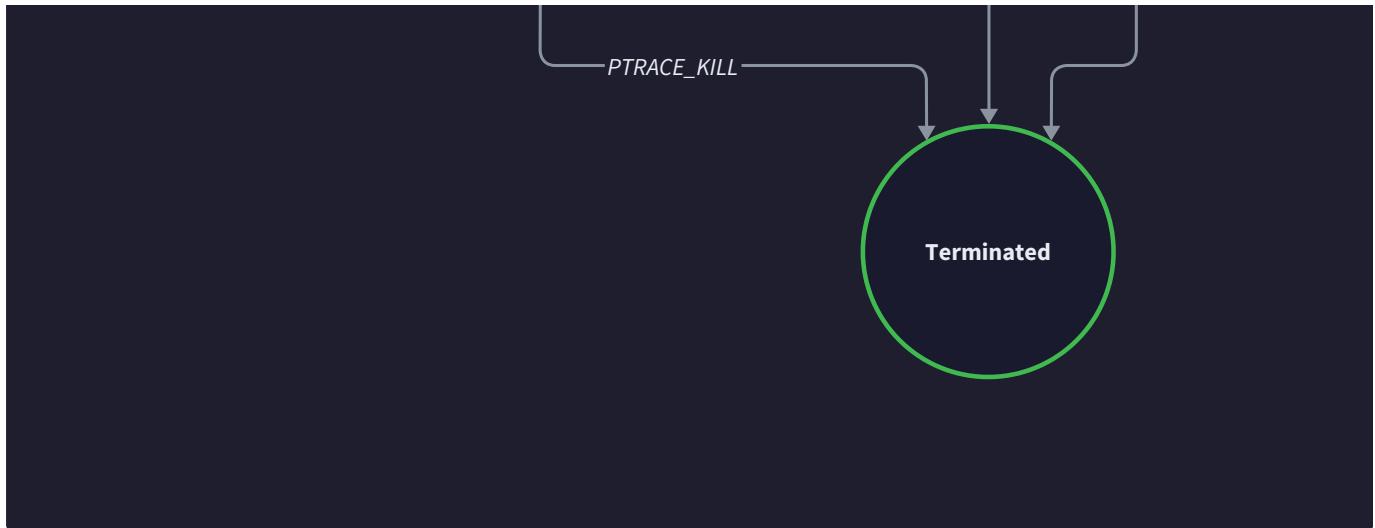
Once basic process information is captured, the debugger loads symbol information from the target's executable file. This involves parsing ELF headers to locate debug sections and building initial symbol tables that map memory addresses to function names. The symbol loading process is critical because it enables address-to-source-line mapping and function name resolution for user commands like "break main" or "list function_name".

The final initialization step involves setting up the debugger's internal data structures to track process state. The `DebuggedProcess` structure must be populated with the process ID, current execution state, register snapshot, executable path, and empty collections for future breakpoints and variable watches. Signal handling must be configured to properly forward or handle signals sent to the target process. At this point, the process is ready to accept debugging commands.

Process State Field	Type	Description
<code>pid</code>	<code>pid_t</code>	Process ID of debugged target
<code>state</code>	<code>ProcessState</code>	Current execution state (RUNNING/STOPPED/EXITED)
<code>regs</code>	<code>struct user_regs_struct</code>	CPU register snapshot when process stopped
<code>executable_path</code>	<code>char*</code>	Path to executable file for symbol loading
<code>symbols</code>	<code>SymbolTable*</code>	Symbol table parsed from debug information
<code>breakpoints</code>	<code>BreakpointList*</code>	Collection of active breakpoints
<code>signal_pending</code>	<code>int</code>	Signal number waiting to be delivered

Design Insight: The attachment lifecycle must handle both paths uniformly because higher-level components shouldn't need to know whether a process was created fresh or attached to an existing instance. This abstraction allows the same debugging commands to work regardless of how the debugger gained control.





Execution Control Operations

Execution control operations provide the fundamental primitives for controlling target process execution flow. These operations — single-step, continue, and stop — form the basic vocabulary for implementing higher-level debugging features like breakpoint handling and source-level stepping. Each operation involves specific ptrace requests and signal handling protocols that must be implemented correctly to maintain reliable debugger control.

Single-step execution advances the target process by exactly one machine instruction. This operation uses `ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL)` to configure the processor's trap flag, which generates a `SIGTRAP` signal after the next instruction executes. The debugger calls `waitpid` to block until the single-step completes, then reads the updated register state to determine the new program counter value. Single-stepping is essential for source-level debugging because it allows precise control over program execution without requiring breakpoints at every instruction.

The single-step implementation must handle several edge cases. Instructions that modify the program counter directly (jumps, calls, returns) may land on addresses that don't correspond to source code lines. System calls may be interrupted by single-stepping, requiring the debugger to handle `SYSCALL` signals appropriately. Multi-byte instructions on x86 architectures must be stepped as atomic units — the processor handles this automatically, but the debugger must be prepared for the program counter to advance by varying amounts.

Continue execution resumes normal program execution until the next signal or breakpoint is encountered. This operation uses `ptrace(PTRACE_CONT, pid, NULL, signal_to_deliver)` where the final parameter specifies whether to deliver any pending signal to the target process. The debugger then calls `waitpid` to block until something interesting happens — typically a breakpoint hit (`SIGTRAP`), a crash signal (`SIGSEGV`, `SIGABRT`), or a user interrupt (`SIGINT`).

The continue operation requires careful signal handling logic. If the target process was stopped due to a signal other than `SIGTRAP` (such as `SIGTERM` from another process), the debugger must decide whether to forward that signal to the target or suppress it. Generally, signals like `SIGTERM`, `SIGUSR1`, and `SIGUSR2` should be forwarded because the target application expects to receive them. However, debugging-related signals like `SIGTRAP` should not be forwarded because they represent debugger control events rather than application signals.

Stop operations forcibly halt a running target process by sending `SIGSTOP` using `kill(pid, SIGSTOP)`. This is typically used to implement user interrupts when someone presses Ctrl+C in the debugger interface. The stop signal is delivered asynchronously, so the debugger must call `waitpid` to wait for the process to actually stop before attempting other operations. Once stopped, the debugger can read register state and perform memory inspection.

Operation	ptrace Request	Expected Signal	Purpose
Single Step	PTRACE_SINGLESTEP	SIGTRAP	Execute exactly one instruction
Continue	PTRACE_CONT	Various	Resume until next breakpoint or signal
Stop	kill(SIGSTOP)	SIGSTOP	Forcibly halt execution
Read Memory	PTRACE_PEEKDATA	None	Read word from process address space
Write Memory	PTRACE_POKEDATA	None	Write word to process address space
Get Registers	PTRACE_GETREGS	None	Read CPU register state
Set Registers	PTRACE_SETREGS	None	Write CPU register state

The execution control loop forms the heart of the debugger's operation. This loop continuously waits for the target process to stop, determines why it stopped, and takes appropriate action before potentially resuming execution. The basic structure involves calling `waitpid` to block until a signal arrives, examining the signal type and exit status, updating internal debugger state, notifying higher-level components (like the breakpoint manager), and deciding whether to continue automatically or return control to the user.

1. Call `waitpid(pid, &status, 0)` to wait for target process signal
2. Check if process exited using `WIFEXITED(status)` or `WIFSIGNALED(status)`
3. If process stopped, extract signal number using `WSTOPSIG(status)`
4. Update `DebuggedProcess` state based on signal type and reason
5. If `SIGTRAP`, check if it was caused by breakpoint or single-step
6. Forward non-debugging signals to target using `ptrace` continuation
7. Notify higher-level components about process state change
8. Return control to user or continue execution based on context

Critical Design Principle: Every execution control operation must be synchronous from the debugger's perspective — when `process_single_step()` or `process_continue()` returns, the operation is complete and the process state is known. This synchronous behavior simplifies higher-level logic and prevents race conditions.

Signal Management

Signal management in a debugger requires understanding the complex interaction between the debugger process, the target process, and the kernel's signal delivery mechanism. The debugger must intercept, interpret, and selectively forward signals while maintaining proper process control. This involves distinguishing between debugging-related signals generated by the debugger's own actions and genuine application signals that should be delivered to the target program.

The kernel delivers signals to stopped processes differently than to running processes. When a process is under `ptrace` control and receives a signal, the kernel stops the process and notifies the debugger via `waitpid`. The debugger can then examine the signal, decide whether to forward it, and resume the process using `ptrace(PTRACE_CONT, pid, NULL, signal_to_forward)`. If the fourth parameter is zero, the signal is suppressed. If it contains a signal number, that signal is delivered to the target when execution resumes.

Debugging signals like `SIGTRAP` are generated by the processor or kernel in response to debugger actions. A `SIGTRAP` from single-stepping should never be forwarded to the target because it represents completion of a debugger

operation, not an application event. Similarly, `SIGTRAP` from breakpoint hits must be handled by the breakpoint manager and not forwarded. The debugger must examine the instruction pointer and breakpoint tables to distinguish between different types of `SIGTRAP` signals.

Application signals like `SIGTERM`, `SIGUSR1`, `SIGINT`, and `SIGPIPE` represent legitimate inter-process communication or system events that the target application expects to receive. These signals should generally be forwarded unless the user has explicitly configured signal handling rules. However, the debugger may choose to catch `SIGINT` (Ctrl+C) to implement user interrupts rather than forwarding it to the target.

Crash signals like `SIGSEGV`, `SIGABRT`, `SIGFPE`, and `SIGBUS` indicate serious program errors. The debugger should catch these signals, display diagnostic information about the crash location and register state, and give the user an opportunity to inspect the program state before the process terminates. These signals should usually be forwarded after investigation because they represent genuine program failures.

Signal Type	Examples	Debugger Action	Forward to Target?
Debugging	<code>SIGTRAP</code> from breakpoints/steps	Handle internally	No
Application	<code>SIGUSR1</code> , <code>SIGUSR2</code> , <code>SIGTERM</code>	Log and forward	Yes
User Interrupt	<code>SIGINT</code> (Ctrl+C)	Stop execution, return to prompt	Usually No
Crash	<code>SIGSEGV</code> , <code>SIGABRT</code> , <code>SIGBUS</code>	Display crash info, allow inspection	Yes (after analysis)
System	<code>SIGCHLD</code> , <code>SIGPIPE</code>	Forward transparently	Yes

The signal handling workflow begins when `waitpid` returns indicating the target process has stopped. The debugger examines the wait status using macros like `WIFSTOPPED()`, `WSTOPSIG()`, and `WIFEXITED()` to determine what happened. For stopped processes, `WSTOPSIG(status)` extracts the signal number that caused the stop. The debugger then consults its signal handling policy to determine the appropriate response.

Signal delivery timing introduces subtle race conditions. If multiple signals arrive while the process is stopped, only one will be reported by `waitpid`. Additional signals may be queued or lost depending on their type. The debugger must be prepared for signal coalescing and should not assume it will see every signal individually. Real-time signals have different queuing behavior than standard signals, adding another layer of complexity for applications that use them.

The debugger must also handle signal delivery during breakpoint processing. When a breakpoint is hit, the processor generates `SIGTRAP` and stops at the breakpoint instruction. If another signal arrives while the debugger is processing the breakpoint, that signal may be delivered when execution resumes. The signal handling logic must coordinate with breakpoint restoration to ensure signals are delivered at the correct instruction boundaries.

Signal Handling Principle: The debugger should be invisible to the target process's normal signal handling. From the target's perspective, signals should arrive as if no debugger were attached, except for explicit user interrupts and debugging-related stops.

Architecture Decisions

The Process Control Component requires several critical architecture decisions that affect both implementation complexity and debugging capabilities. These decisions involve trade-offs between portability, performance, reliability, and feature completeness. Each decision has long-term consequences for the debugger's behavior and maintainability.

Decision: ptrace vs Alternative Process Control APIs

- **Context:** Unix systems provide several mechanisms for process control including ptrace, /proc filesystem, and specialized debugging libraries. Different approaches offer varying levels of control, portability, and complexity.
- **Options Considered:** ptrace system call, /proc-based control, libunwind integration
- **Decision:** Use ptrace as the primary process control mechanism with /proc as supplementary information source
- **Rationale:** ptrace provides the most comprehensive process control capabilities across Unix systems, with standardized behavior for breakpoints, single-stepping, and memory access. While /proc offers easier access to some information, it lacks execution control primitives. libunwind would add significant dependency complexity for marginal benefit.
- **Consequences:** Enables full debugging feature set with relatively simple implementation, but limits portability to Unix-like systems and requires careful signal handling.

Option	Pros	Cons
ptrace	Complete control, standardized, minimal dependencies	Unix-only, complex signal handling
/proc filesystem	Easy memory/register access, human-readable	Limited execution control, Linux-specific
Debugging libraries	Higher-level abstraction, feature-rich	Heavy dependencies, reduced control

Decision: Process Creation vs Attachment Priority

- **Context:** Debuggers can either create new processes under control or attach to existing processes. The implementation must handle both cases, but one approach should be the primary design target.
- **Options Considered:** Creation-first design, attachment-first design, equal priority
- **Decision:** Design primarily for process creation with attachment as secondary feature
- **Rationale:** Creating processes under debugger control provides cleaner initialization, no race conditions, and guaranteed symbol access. Attachment introduces complexities with unknown process state, potential security restrictions, and limited symbol information.
- **Consequences:** Simpler core implementation and more reliable debugging, but reduced utility for debugging already-running production processes.

Decision: Synchronous vs Asynchronous Process Control

- **Context:** Process control operations can be implemented synchronously (blocking until completion) or asynchronously (returning immediately with completion notification later). This affects both API design and internal state management.
- **Options Considered:** Synchronous operations, asynchronous with callbacks, hybrid approach
- **Decision:** Implement all process control operations synchronously
- **Rationale:** Synchronous operations simplify state management, eliminate race conditions, and make the API easier to use correctly. Debugger operations are inherently interactive with human response times, so blocking for milliseconds is acceptable.
- **Consequences:** Simpler implementation and fewer bugs, but potential UI responsiveness issues during long-running operations.

Decision: Signal Forwarding Policy

- **Context:** The debugger must decide which signals to forward to the target process and which to handle internally. This policy affects both application compatibility and debugging experience.
- **Options Considered:** Forward all signals, selective forwarding based on signal type, user-configurable policies
- **Decision:** Implement selective forwarding with sensible defaults and future extensibility for user configuration
- **Rationale:** Blind forwarding breaks debugging (SIGTRAP would confuse applications), while blocking all signals breaks application functionality. Selective forwarding based on signal semantics provides the best balance.
- **Consequences:** Applications behave naturally under debugger control, but implementation requires detailed signal knowledge and careful testing.

The **error handling strategy** for process control operations must balance robustness with usability. ptrace operations can fail for numerous reasons: permission denied, process exited, invalid addresses, or kernel resource exhaustion. The component design distinguishes between recoverable errors (temporarily invalid addresses) and fatal errors (process terminated unexpectedly). Recoverable errors are reported to higher-level components for potential retry logic, while fatal errors trigger cleanup and user notification.

Process control errors fall into several categories that require different handling approaches. **Permission errors** typically indicate insufficient privileges or security policy restrictions. These are usually permanent failures that should be reported clearly to the user. **Resource errors** like `ENOMEM` or `EAGAIN` may be temporary and could succeed if retried. **State errors** occur when operations are attempted on processes in inappropriate states, such as trying to single-step an already-running process. **System errors** represent kernel or hardware problems that are typically unrecoverable.

Error Category	Example Conditions	Recovery Strategy
Permission	<code>EPERM</code> , <code>EACCES</code>	Report to user, abort operation
Resource	<code>ENOMEM</code> , <code>EAGAIN</code>	Retry with exponential backoff
State	Operation on running process	Synchronize state, retry
System	Kernel failures, hardware faults	Clean shutdown, report error
Process Gone	<code>ESRCH</code> , <code>ECHILD</code>	Update state to EXITED, notify user

Common Pitfalls

Process control implementation contains several subtle pitfalls that can lead to unreliable debugger behavior, race conditions, or security vulnerabilities. Understanding these pitfalls is crucial for building a robust debugging system that works consistently across different target programs and system conditions.

⚠️ Pitfall: Race Condition Between Fork and PTRACE_TRACEME

Many implementations attempt to have the parent call `ptrace(PTRACE_ATTACH)` on the child after `fork()` returns. This creates a race condition where the child might call `exec()` and start executing target code before the parent gains control. The child process could run to completion or crash before the debugger attaches, making debugging impossible.

Why it's wrong: The window between `fork()` and successful attachment can be large enough for the target program to execute significant code, especially on multi-core systems. If the target program exits quickly or crashes during startup, the debugger will never gain control.

How to fix: Always use the `PTRACE_TRACEME` approach where the child process requests tracing before calling `exec()`. The child should call `ptrace(PTRACE_TRACEME, 0, NULL, NULL)`, then `raise(SIGSTOP)`, then `exec()`. This ensures the parent gains control before any target code executes.

⚠️ Pitfall: Incorrect Signal Forwarding Logic

A common mistake is forwarding `SIGTRAP` signals to the target process or suppressing legitimate application signals. This happens when developers don't distinguish between debugging-related signals (generated by `ptrace` operations) and application signals (sent by other processes or the kernel for non-debugging reasons).

Why it's wrong: Forwarding `SIGTRAP` from breakpoints or single-steps can confuse target applications that have their own `SIGTRAP` handlers. Conversely, suppressing legitimate signals like `SIGTERM` or `SIGUSR1` breaks normal inter-process communication and can make applications hang or behave incorrectly.

How to fix: Implement signal classification logic that examines both the signal type and the context in which it occurred. `SIGTRAP` should only be forwarded if it didn't result from a debugger operation. Check the instruction pointer against active breakpoints and recent single-step operations to determine signal origin.

⚠️ Pitfall: Zombie Process Creation

When the target process exits, it becomes a zombie until the parent debugger calls `waitpid()` to reap it. If the debugger fails to handle process exit properly or crashes before reaping children, zombie processes accumulate and consume system resources.

Why it's wrong: Zombie processes consume process table entries and can eventually exhaust system resources if created in large numbers. They also remain visible in process lists, confusing users and system administrators.

How to fix: Always install a `SIGCHLD` handler in the debugger process to automatically reap child processes. Additionally, ensure that all exit paths in the debugger code call `waitpid()` on active target processes. Use `waitpid(-1, NULL, WNOHANG)` in the signal handler to reap any available children without blocking.

⚠️ Pitfall: Memory Access During Process Execution

Attempting to read or write target process memory while the process is running can produce inconsistent results or fail entirely. Some implementations assume they can access memory at any time without checking process state.

Why it's wrong: `ptrace` memory access operations require the target process to be stopped. Accessing memory from a running process may return stale data, fail with errors, or read partially-updated data structures during concurrent

modifications.

How to fix: Always verify the target process is in `STOPPED` state before attempting memory operations. If the process is running, stop it first using `PTRACE_INTERRUPT` or `kill(SIGSTOP)`, perform the memory operation, then resume execution if appropriate.

Pitfall: Platform-Specific Register Structure Assumptions

Different CPU architectures have different register layouts and naming conventions. Code that assumes x86-64 register names and offsets will fail on ARM, RISC-V, or other architectures.

Why it's wrong: Hard-coding register offsets or names makes the debugger non-portable and causes compilation failures or runtime errors on different architectures. Even within the same architecture family, 32-bit vs 64-bit variants have different register layouts.

How to fix: Use the platform-provided `struct user_regs_struct` from `<sys/user.h>` and access registers through their standardized field names. Implement architecture-specific register access functions that abstract register reading and writing behind a common interface.

Pitfall: Inadequate Error Handling for ptrace Operations

ptrace system calls can fail for many reasons, but some implementations only check for obvious errors like invalid PIDs while ignoring subtler failure modes like temporary resource exhaustion or permission changes.

Why it's wrong: Inadequate error handling leads to mysterious debugger crashes, silent failures where operations appear to succeed but don't, or cascading failures where one small error causes widespread debugger malfunction.

How to fix: Check the return value of every ptrace call and examine `errno` for specific error conditions. Implement retry logic for transient errors like `EAGAIN` or `EINTR`. Provide meaningful error messages that help users understand what went wrong and how to fix it.

Implementation Guidance

The Process Control Component serves as the foundational layer that all other debugger components depend on. This implementation guidance provides both complete infrastructure code for ancillary functions and skeleton code for the core process control logic that learners should implement themselves.

Technology Recommendations

Component	Simple Option	Advanced Option
Process Control	Direct ptrace syscalls with signal handling	libunwind integration with enhanced stack unwinding
Error Handling	Simple errno checking with perror()	Structured error objects with context and recovery hints
Process State	Basic state enum with manual tracking	State machine with automatic transition validation
Signal Management	Manual signal filtering with lookup tables	Configurable signal policies with user customization

Recommended File Structure

```
debugger/
├── src/
│   ├── process/
│   │   ├── process_control.c      ← Core process control implementation
│   │   ├── process_control.h     ← Process control public interface
│   │   ├── signal_handler.c     ← Signal management and forwarding logic
│   │   └── process_utils.c       ← Helper functions for process operations
│   ├── common/
│   │   ├── debug_error.c        ← Error handling infrastructure
│   │   ├── debug_error.h        ← Error type definitions
│   │   └── platform.h           ← Platform-specific definitions
│   └── main.c                  ← Debugger entry point
└── tests/
    ├── test_process_control.c   ← Unit tests for process control
    └── test_programs/
        ├── simple_loop.c         ← Sample programs for testing
        └── signal_test.c          ← Basic test program
                                     ← Program for signal handling tests
└── Makefile
```

Infrastructure Starter Code

Error Handling Infrastructure (`src/common/debug_error.c`):

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include "debug_error.h"

DebugError debug_error_create(DebugResult code, const char* message,
                             const char* function, int line, int system_errno) {

    DebugError error;

    error.code = code;
    error.function = function;
    error.line = line;
    error.system_errno = system_errno;

    snprintf(error.message, sizeof(error.message), "%s", message);

    return error;
}

void debug_error_print(const DebugError* error) {
    fprintf(stderr, "Debug Error: %s\n", error->message);
    fprintf(stderr, " Location: %s:%d\n", error->function, error->line);

    if (error->system_errno != 0) {
        fprintf(stderr, " System Error: %s\n", strerror(error->system_errno));
    }

    const char* code_name = "UNKNOWN";
    switch (error->code) {
        case DEBUG_SUCCESS: code_name = "SUCCESS"; break;
        case DEBUG_PTRACE_FAILED: code_name = "PTRACE_FAILED"; break;
        case DEBUG_PROCESS_EXITED: code_name = "PROCESS_EXITED"; break;
        case DEBUG_INVALID_ADDRESS: code_name = "INVALID_ADDRESS"; break;
    }
}
```

```
        case DEBUG_SYMBOL_NOT_FOUND: code_name = "SYMBOL_NOT_FOUND"; break;

        case DEBUG_BREAKPOINT_EXISTS: code_name = "BREAKPOINT_EXISTS"; break;

        case DEBUG_NO_DEBUG_INFO: code_name = "NO_DEBUG_INFO"; break;

    }

    fprintf(stderr, " Error Code: %s\n", code_name);

}

#define DEBUG_ERROR(code, msg) debug_error_create(code, msg, __func__, __LINE__, errno)
```

Platform Definitions (`src/common/platform.h`):

```
#ifndef PLATFORM_H
```

C

```
#define PLATFORM_H
```

```
#include <sys/types.h>
```

```
#include <sys/ptrace.h>
```

```
#include <sys/user.h>
```

```
#include <sys/wait.h>
```

```
#include <signal.h>
```

```
#include <stdint.h>
```

```
#include <stdbool.h>
```

```
// Process state tracking
```

```
typedef enum {
```

```
    PROCESS_RUNNING,
```

```
    PROCESS_STOPPED,
```

```
    PROCESS_EXITED
```

```
} ProcessState;
```

```
// Core process structure
```

```
typedef struct {
```

```
    pid_t pid;
```

```
    ProcessState state;
```

```
    struct user_regs_struct regs;
```

```
    char* executable_path;
```

```
    struct SymbolTable* symbols; // Forward declaration
```

```
    struct BreakpointList* breakpoints; // Forward declaration
```

```
    int signal_pending;
```

```
} DebuggedProcess;
```

```
// Error handling types
```

```
typedef enum {
```

```
    DEBUG_SUCCESS,
```

```
    DEBUG_PTRACE_FAILED,  
  
    DEBUG_PROCESS_EXITED,  
  
    DEBUG_INVALID_ADDRESS,  
  
    DEBUG_SYMBOL_NOT_FOUND,  
  
    DEBUG_BREAKPOINT_EXISTS,  
  
    DEBUG_NO_DEBUG_INFO  
  
} DebugResult;  
  
typedef struct {  
  
    DebugResult code;  
  
    char message[256];  
  
    const char* function;  
  
    int line;  
  
    int system_errno;  
  
} DebugError;  
  
#endif
```

Signal Classification Helper (src/process/signal_handler.c):

```
#include <signal.h>
#include <stdbool.h>
#include "process_control.h"

static const int DEBUGGING_SIGNALS[] = { SIGTRAP, 0 };

static const int APPLICATION_SIGNALS[] = { SIGUSR1, SIGUSR2, SIGTERM, SIGPIPE, SIGALRM, 0 };

static const int CRASH_SIGNALS[] = { SIGSEGV, SIGABRT, SIGBUS, SIGFPE, SIGILL, 0 };

bool is_debugging_signal(int signum) {

    for (int i = 0; DEBUGGING_SIGNALS[i] != 0; i++) {

        if (DEBUGGING_SIGNALS[i] == signum) return true;

    }

    return false;

}

bool is_application_signal(int signum) {

    for (int i = 0; APPLICATION_SIGNALS[i] != 0; i++) {

        if (APPLICATION_SIGNALS[i] == signum) return true;

    }

    return false;

}

bool is_crash_signal(int signum) {

    for (int i = 0; CRASH_SIGNALS[i] != 0; i++) {

        if (CRASH_SIGNALS[i] == signum) return true;

    }

    return false;

}

int determine_signal_action(int signum, bool from_breakpoint) {

    if (is_debugging_signal(signum) && from_breakpoint) {

        return 0; // Suppress debugging signals from breakpoints
```

```
}

if (is_crash_signal(signum)) {

    printf("Process crashed with signal %d (%s)\n", signum, strsignal(signum));

    return signum; // Forward crash signals after notification

}

if (is_application_signal(signum)) {

    return signum; // Forward application signals transparently

}

return 0; // Suppress unknown signals by default
}
```

Core Logic Skeleton Code

Process Control Interface (`src/process/process_control.h`):

```
#ifndef PROCESS_CONTROL_H
#define PROCESS_CONTROL_H

#include "../common/platform.h"
#include "../common/debug_error.h"

// Process lifecycle functions

DebugResult process_create(const char* executable_path, char* const argv[], DebuggedProcess* proc);

DebugResult process_attach(pid_t target_pid, DebuggedProcess* proc);

DebugResult process_detach(DebuggedProcess* proc);

// Execution control functions

DebugResult process_continue(DebuggedProcess* proc);

DebugResult process_single_step(DebuggedProcess* proc);

DebugResult process_stop(DebuggedProcess* proc);

// Memory access functions

DebugResult process_read_memory(DebuggedProcess* proc, uintptr_t address,
                                void* buffer, size_t size);

DebugResult process_write_memory(DebuggedProcess* proc, uintptr_t address,
                                 const void* data, size_t size);

// Register access functions

DebugResult process_get_registers(DebuggedProcess* proc);

DebugResult process_set_registers(DebuggedProcess* proc);

// Signal handling

DebugResult process_handle_signal(DebuggedProcess* proc, int signum);

#endif
```

Core Process Control Implementation (`src/process/process_control.c`):

```
#include "process_control.h"
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

DebugResult process_create(const char* executable_path, char* const argv[], DebuggedProcess* proc) {

    // TODO 1: Initialize the DebuggedProcess structure with default values

    // Set pid to 0, state to PROCESS_STOPPED, clear registers, copy executable_path

    // TODO 2: Fork the current process to create child and parent

    // Handle fork() failure by returning DEBUG_PTRACE_FAILED with errno

    // TODO 3: In child process: call ptrace(PTRACE_TRACEME) to enable tracing

    // If PTRACE_TRACEME fails, exit child with error code

    // TODO 4: In child process: raise SIGSTOP to halt before exec

    // This ensures parent gains control before target code executes

    // TODO 5: In child process: call exec() to replace with target program

    // Use execv(executable_path, argv) and handle failure with exit()

    // TODO 6: In parent process: wait for child's SIGSTOP using waitpid()

    // Check that waitpid succeeds and child is actually stopped

    // TODO 7: In parent process: read initial register state using PTRACE_GETREGS

    // Store registers in proc->regs for future reference

    // TODO 8: Set process state to PROCESS_STOPPED and return DEBUG_SUCCESS
```

```
// The process is now ready to accept debugging commands
}

DebugResult process_continue(DebuggedProcess* proc) {

    // TODO 1: Validate that process is in STOPPED state

    // Return DEBUG_PROCESS_EXITED if process is not stoppable

    // TODO 2: Call ptrace(PTRACE_CONT) with any pending signal to deliver

    // Use proc->signal_pending as the signal parameter

    // TODO 3: Clear the pending signal since it's being delivered

    // Set proc->signal_pending = 0

    // TODO 4: Update process state to RUNNING

    // Set proc->state = PROCESS_RUNNING

    // TODO 5: Wait for next signal using waitpid() with WUNTRACED

    // This blocks until the process stops again or exits

    // TODO 6: Check waitpid result - did process exit or stop?

    // Use WIFEXITED(), WIFSIGNALED(), WIFSTOPPED() to determine outcome

    // TODO 7: If process stopped, extract signal number using WSTOPSIG()

    // Update proc->signal_pending with the signal that caused the stop

    // TODO 8: Update process state and registers based on stop reason

    // Set state to STOPPED or EXITED, refresh register snapshot

}

DebugResult process_single_step(DebuggedProcess* proc) {
```

```
// TODO 1: Verify process is in STOPPED state

// Single-stepping only works on stopped processes


// TODO 2: Call ptrace(PTRACE_SINGLESTEP) with pending signal

// This executes exactly one instruction then stops


// TODO 3: Clear pending signal since it's being delivered

// Set proc->signal_pending = 0


// TODO 4: Wait for SIGTRAP signal indicating step completion

// Use waitpid() and verify the signal is SIGTRAP from single-step


// TODO 5: Read updated register state using PTRACE_GETREGS

// The program counter will have advanced by one instruction


// TODO 6: Handle case where single-step hits a breakpoint

// Check if new PC location has an active breakpoint


// TODO 7: Return DEBUG_SUCCESS if step completed normally

// Process should remain in STOPPED state after single-step

}

DebugResult process_read_memory(DebuggedProcess* proc, uintptr_t address,
                                void* buffer, size_t size) {

    // TODO 1: Validate that process is stopped and address is reasonable

    // Memory access requires stopped process for consistency


    // TODO 2: Use ptrace(PTRACE_PEEKDATA) to read memory word by word

    // ptrace reads in sizeof(long) chunks, handle partial reads
```

```

    // TODO 3: Handle misaligned addresses by reading extra bytes

    // Copy only the requested bytes to output buffer


    // TODO 4: Check for ptrace errors on each read operation

    // Some addresses may be unreadable (unmapped, protected)

    // TODO 5: Return number of bytes successfully read

    // Partial reads are common and should be handled gracefully

}

```

Language-Specific Hints

ptrace System Call Usage:

- Always check return values: `ptrace()` returns -1 on failure and sets `errno`
- Use `PTRACE_O_EXITKILL` option to automatically kill target if debugger exits
- Memory access with `PTRACE_PEEKDATA / PTRACE_POKEDATA` works in `sizeof(long)` units
- Register access requires `struct user_regs_struct` from `<sys/user.h>`

Signal Handling Best Practices:

- Install `SIGCHLD` handler to automatically reap zombie processes
- Use `sigaction()` instead of `signal()` for reliable signal handling
- Block signals during critical sections using `sigprocmask()`
- Always check `WIFSTOPPED()` before accessing stop signal with `WSTOPSIG()`

Error Handling Patterns:

- Save and restore `errno` around debug logging calls
- Use `strerror()` to convert `errno` values to human-readable messages
- Implement retry logic for `EINTR` and `EAGAIN` errors
- Distinguish between recoverable and fatal ptrace errors

Milestone Checkpoint

After implementing the Process Control Component, verify correct operation with these tests:

Test Command: `make test_process_control && ./test_process_control`

Expected Behavior:

1. **Process Creation Test:** Create a simple loop program under debugger control. The process should start in stopped state before executing any target code.
2. **Single Step Test:** Single-step through the first few instructions. Each step should advance the program counter by exactly one instruction.

3. **Continue Test:** Set the process running and interrupt it with Ctrl+C. The process should stop and return control to debugger.
4. **Signal Forwarding Test:** Send `SIGUSR1` to the target process and verify it's forwarded correctly while `SIGTRAP` signals are handled internally.

Signs of Problems:

- Process starts running immediately instead of stopping: Check `PTRACE_TRACEME` and `SIGSTOP` sequence
- Single-step advances by wrong amounts: Verify register reading and instruction boundary handling
- Debugger hangs on continue: Check `waitpid()` usage and signal handling logic
- Zombie processes after exit: Install proper `SIGCHLD` handler

Debugging the Debugger:

- Use `strace -f ./your_debugger target_program` to see all ptrace system calls
- Check `/proc/PID/status` to see actual process state during debugging
- Use `gdb` on your debugger process to debug infinite loops or crashes
- Verify signal handling with `kill -USR1 PID` on the target process

Breakpoint Manager Component

Milestone(s): Milestone 2 (Breakpoints) — this section implements the breakpoint management system that enables setting, hitting, and managing software breakpoints using INT3 instruction patching

The Tripwire Analogy

Think of breakpoints as **security tripwires** strategically placed along a path. When someone walks down the path, they trigger the tripwire, which immediately alerts the security system and stops all movement. The security guard can then examine the scene, ask questions, and decide whether to let the person continue or take different action.

In our debugger, the "path" is the program's execution flow through machine instructions. The "tripwires" are breakpoints we place at specific memory addresses. When the CPU tries to execute an instruction at a breakpoint location, it triggers our "security system" (the debugger) through a special interrupt. The debugger gains control, examines the program's state, and waits for the user to decide what to do next.

Just like physical tripwires, our breakpoints must be:

- **Invisible to the target** — the original program logic remains unchanged
- **Instantly triggering** — execution stops immediately when hit
- **Restorable** — we can remove the tripwire and let execution continue normally
- **Non-destructive** — the original "path" (instruction) is preserved and can be restored

The key insight is that breakpoints are not passive markers — they're active intervention points that fundamentally alter the program's execution flow in a controlled, reversible way.

Critical Design Principle: Software breakpoints work by temporarily replacing the target instruction with a special "trap" instruction (INT3 on x86), causing the processor to generate a signal that the debugger catches. This is instruction patching — we modify the running program's code in memory, but preserve the original instruction so we can restore it later.

Software Breakpoint Implementation

The **Breakpoint Manager** implements software breakpoints through a sophisticated instruction patching mechanism. Unlike hardware breakpoints (which use special CPU registers), software breakpoints work by temporarily modifying the target program's machine code in memory.

The core mechanism relies on the x86 `INT3` instruction (opcode `0xcc`), which is a single-byte interrupt instruction specifically designed for debuggers. When the CPU encounters `INT3`, it immediately generates a `SIGTRAP` signal and transfers control to the operating system, which forwards the signal to our debugger process.

Here's how the patching mechanism works in detail:

Breakpoint Setting Process:

1. **Address Validation** — The debugger first validates that the target address contains the start of a valid instruction. Setting a breakpoint in the middle of a multi-byte instruction would corrupt the instruction stream.
2. **Original Byte Preservation** — The debugger reads the first byte of the instruction at the target address using `ptrace(PTRACE_PEEKDATA)` and stores it in the `Breakpoint` structure's `original_byte` field.
3. **INT3 Injection** — The debugger writes the `INT3` opcode (`0xcc`) to the target address using `ptrace(PTRACE_POKEDATA)`, replacing the first byte of the original instruction.
4. **Breakpoint Registration** — The debugger adds the breakpoint to its internal breakpoint list with metadata including the address, original byte, and enabled state.

Breakpoint Hit Detection:

When the target process executes the patched instruction, the CPU encounters the `INT3` and generates `SIGTRAP`. The debugger's main control loop, waiting in `waitpid()`, receives this signal and must:

1. **Signal Classification** — Determine that the `SIGTRAP` was caused by a breakpoint hit (not single-step or other trap)
2. **Address Identification** — Read the process's instruction pointer using `ptrace(PTRACE_GETREGS)` to determine which breakpoint was hit
3. **Instruction Pointer Adjustment** — The `INT3` instruction advances the instruction pointer by one byte, so the debugger must decrement it to point back to the original instruction location
4. **Breakpoint Lookup** — Find the `Breakpoint` structure corresponding to the hit address and increment its `hit_count`

Breakpoint Restoration and Continuation:

To continue execution past a breakpoint, the debugger must temporarily restore the original instruction:

1. **Original Instruction Restoration** — Write the `original_byte` back to the breakpoint address, replacing the `INT3`

2. **Single-Step Execution** — Use `ptrace(PTRACE_SINGLESTEP)` to execute exactly one instruction (the original instruction that was replaced)
3. **Breakpoint Re-patching** — After the single step completes, write the `INT3` back to the address so the breakpoint remains active for future hits
4. **Normal Continuation** — Resume normal execution with `ptrace(PTRACE_CONT)`

Breakpoint Field	Type	Description
address	uintptr_t	Memory address where breakpoint is set
original_byte	uint8_t	First byte of original instruction, preserved for restoration
is_enabled	bool	Whether breakpoint is currently active (INT3 patched)
hit_count	uint32_t	Number of times this breakpoint has been triggered
location_description	char*	Human-readable description (e.g., "main.c:42 in function foo")
next	struct Breakpoint*	Next breakpoint in linked list (for breakpoint collection)

Key Insight: The fundamental challenge of software breakpoints is that they modify the program being debugged. This creates a chicken-and-egg problem — to continue execution past a breakpoint, we must restore the original instruction, but then we lose the breakpoint. The solution is the restore-step-repatch dance that temporarily removes the breakpoint for exactly one instruction execution.

The instruction patching approach has several important implications:

Memory Modification Requirements — Software breakpoints require write access to the target process's code pages. This works because the debugger has `ptrace` privileges, but it means breakpoints cannot be set in truly read-only code segments.

Single-Byte Limitation — Since `INT3` is only one byte, we can only replace the first byte of an instruction. This works for most instruction sets, but it means we cannot distinguish between breakpoints set at different offsets within the same word on some architectures.

Performance Characteristics — Software breakpoints have minimal overhead when not hit (just a single-byte instruction replacement), but hitting a breakpoint requires several `ptrace` operations and system calls, making it relatively expensive.

Breakpoint Lifecycle Management

The **Breakpoint Manager** must carefully track the state of each breakpoint throughout its entire lifecycle. A breakpoint progresses through several distinct phases, each requiring specific management actions and state transitions.

Lifecycle States and Transitions:

Current State	Event	Next State	Actions Taken
Non-existent	Set breakpoint command	Enabled	Validate address, save original byte, patch INT3, add to list
Enabled	Breakpoint hit	Hit-Suspended	Signal debugger, adjust instruction pointer, notify user
Hit-Suspended	Continue command	Temporarily-Disabled	Restore original byte, prepare for single-step
Temporarily-Disabled	Single-step complete	Enabled	Re-patch INT3, resume normal execution
Enabled	Disable breakpoint command	Disabled	Restore original byte, mark as disabled
Disabled	Enable breakpoint command	Enabled	Re-patch INT3, mark as enabled
Any State	Delete breakpoint command	Non-existent	Restore original byte if needed, remove from list

State Management Data Structures:

The lifecycle management requires sophisticated state tracking beyond the basic `Breakpoint` structure. The debugger maintains additional metadata to coordinate state transitions:

Breakpoint Manager Field	Type	Description
active_breakpoints	BreakpointList*	Linked list of all breakpoint structures
hit_breakpoint	Breakpoint*	Currently hit breakpoint awaiting user action
single_step_restore	Breakpoint*	Breakpoint temporarily disabled for single-step
next_breakpoint_id	uint32_t	Counter for assigning unique breakpoint identifiers
breakpoint_count	size_t	Total number of active breakpoints for validation

Setting Breakpoints — Detailed Algorithm:

The breakpoint setting process requires careful validation and error handling to ensure debugger stability:

- Address Validation** — Check that the target address falls within the executable's code segments and is properly aligned for the target architecture.
- Duplicate Detection** — Search the existing breakpoint list to ensure no breakpoint already exists at this address, preventing double-patching.
- Instruction Boundary Verification** — Use disassembly information (if available) to confirm the address points to the start of an instruction, not the middle of a multi-byte instruction.
- Memory Access Test** — Attempt to read from the target address using `ptrace(PTRACE_PEEKDATA)` to ensure the memory is accessible and the process is in a traceable state.

5. **Original Byte Preservation** — Read and store the original instruction byte that will be replaced by `INT3`.
6. **Atomic Patching** — Write the `INT3` opcode to the target address using `ptrace(PTRACE_POKEDATA)` in a single operation.
7. **Breakpoint Registration** — Create a new `Breakpoint` structure, initialize all fields, and add it to the active breakpoint list.
8. **User Notification** — Provide feedback to the user confirming successful breakpoint creation with address and description.

Handling Breakpoint Hits — Signal Processing:

When a breakpoint is hit, the debugger must handle the resulting `SIGTRAP` signal with precise timing and state management:

1. **Signal Reception** — The debugger's main loop receives `SIGTRAP` through `waitpid()` and extracts the signal information from the status word.
2. **Context Capture** — Read the complete register state using `ptrace(PTRACE_GETREGS)` to determine the current instruction pointer and processor state.
3. **Breakpoint Identification** — Search the active breakpoint list for a breakpoint matching the instruction pointer address (accounting for the post-INT3 increment).
4. **Instruction Pointer Correction** — Decrement the instruction pointer by one byte to point back to the original instruction location.
5. **Hit Count Update** — Increment the `hit_count` field in the matching `Breakpoint` structure for debugging statistics.
6. **State Transition** — Mark the breakpoint as "Hit-Suspended" and store a reference to it in the breakpoint manager's `hit_breakpoint` field.
7. **User Notification** — Display breakpoint hit information including address, hit count, and source location (if debug symbols are available).
8. **Control Return** — Return control to the debugger's command processing loop, allowing the user to examine state or continue execution.

Continuing Past Breakpoints — Restore-Step-Repatch:

The most complex part of breakpoint lifecycle management is safely continuing execution past a hit breakpoint:

1. **Restoration Preparation** — Verify that there is indeed a hit breakpoint awaiting continuation and that the process is in the correct stopped state.
2. **Original Instruction Restoration** — Write the saved `original_byte` back to the breakpoint address, temporarily removing the `INT3` patch.
3. **State Tracking** — Move the breakpoint to "Temporarily-Disabled" state and store a reference in `single_step_restore` for post-step cleanup.
4. **Single-Step Execution** — Use `ptrace(PTRACE_SINGLESTEP)` to execute exactly one instruction, which will be the original instruction that was replaced.

5. **Single-Step Completion** — Wait for the single-step to complete via `waitpid()`, handling any signals or errors that occur during the step.
6. **Re-patching** — Write the `INT3` opcode back to the breakpoint address, restoring the breakpoint for future hits.
7. **State Restoration** — Return the breakpoint to "Enabled" state and clear the `single_step_restore` reference.
8. **Normal Continuation** — Use `ptrace(PTRACE_CONT)` to resume normal execution until the next signal or breakpoint.

Critical Timing Window: The restore-step-repatch sequence creates a brief window where the breakpoint is not active. If the program branches back to the same address during the single-step (which is possible with certain jump instructions), the breakpoint will not trigger. This is a known limitation of software breakpoints that debugger users must understand.

Multiple Breakpoint Coordination

Managing collections of breakpoints introduces significant coordination challenges. Unlike single breakpoints, multiple breakpoints can interact in unexpected ways, creating race conditions, memory conflicts, and state consistency problems that the **Breakpoint Manager** must handle gracefully.

Concurrent Breakpoint Access Patterns:

Multiple breakpoints create several coordination scenarios that require careful management:

Sequential Breakpoint Hits — When execution flows from one breakpoint to another in rapid succession, the debugger must ensure that the restore-step-repatch sequence for the first breakpoint completes before the second breakpoint can be hit.

Breakpoint Clustering — When breakpoints are set at nearby addresses (within a few instructions of each other), the single-step operation used for breakpoint restoration might immediately trigger another breakpoint before the first one can be re-patched.

Overlapping Instruction Boundaries — On variable-length instruction architectures like x86, it's possible to set breakpoints that would affect the same memory word, requiring conflict detection and resolution.

Dynamic Breakpoint Modification — Users may add, remove, or modify breakpoints while the debugger is processing a hit from another breakpoint, requiring atomic updates to the breakpoint collection.

Breakpoint Collection Management:

The breakpoint manager maintains breakpoints in a linked list structure that supports efficient insertion, deletion, and lookup operations:

BreakpointList Operation	Parameters	Returns	Description
breakpoint_list_create	void	BreakpointList*	Initialize empty breakpoint collection with proper synchronization
breakpoint_list_add	list, breakpoint	DebugResult	Add breakpoint to collection with duplicate detection
breakpoint_list_remove	list, address	DebugResult	Remove breakpoint at address and restore original instruction
breakpoint_list_find	list, address	Breakpoint*	Locate breakpoint by address, return NULL if not found
breakpoint_list_clear	list	DebugResult	Remove all breakpoints and restore original instructions
breakpoint_list_count	list	size_t	Return number of active breakpoints for validation

Address Conflict Detection:

Before setting a new breakpoint, the manager must check for potential conflicts with existing breakpoints:

- Exact Address Duplicate** — Check if a breakpoint already exists at the exact target address, which would cause double-patching.
- Instruction Overlap Detection** — On architectures with variable-length instructions, verify that the new breakpoint wouldn't interfere with existing breakpoints in nearby addresses.
- Memory Page Boundaries** — Ensure that breakpoint addresses don't cross memory page boundaries in ways that could cause access violations.
- Code Segment Validation** — Confirm that all breakpoint addresses remain within executable code segments and haven't been invalidated by dynamic loading.

Atomic Breakpoint Operations:

To prevent race conditions and state corruption, breakpoint operations must be atomic with respect to process execution:

Operation	Atomicity Requirement	Implementation Strategy
Set Multiple	All succeed or all fail	Validate all addresses first, then patch in sequence
Remove Multiple	Safe partial completion	Remove in reverse address order to avoid execution gaps
Enable/Disable Toggle	Consistent state visibility	Update enabled flag before patching to prevent inconsistency
Hit Processing	Single active hit	Use hit_breakpoint field to serialize hit handling

Breakpoint Iteration and Batch Operations:

The debugger must support efficient iteration over all breakpoints for operations like listing, enabling/disabling groups, or cleanup on process termination:

```

Breakpoint Iteration Algorithm:
1. Acquire breakpoint list lock (if threading support added later)
2. Start with list head pointer
3. For each breakpoint:
   a. Validate breakpoint structure integrity
   b. Check if breakpoint is still at valid address
   c. Apply operation (list, enable, disable, etc.)
   d. Handle any errors without corrupting list structure
4. Update breakpoint count and consistency markers
5. Release lock and return operation results

```

Memory Consistency During Updates:

When modifying multiple breakpoints, the manager must ensure memory consistency so the target process never sees corrupted instruction bytes:

1. **Batch Validation** — Validate all requested changes before making any memory modifications
2. **Ordered Updates** — Apply changes in address order to prevent execution from encountering partially updated state
3. **Rollback Capability** — Maintain enough state to undo partial operations if errors occur during batch updates
4. **Verification Pass** — After batch operations, verify that all breakpoints are in expected states

Coordination Insight: The fundamental challenge of multiple breakpoints is maintaining the illusion that each breakpoint operates independently, while actually coordinating their interactions behind the scenes. This requires sophisticated state machines and error recovery to handle the complex interactions between process execution and dynamic code patching.

Error Recovery in Multi-Breakpoint Scenarios:

Complex breakpoint interactions can lead to error conditions that require sophisticated recovery:

Error Condition	Detection Method	Recovery Strategy
Double-patched address	Duplicate detection fails	Restore original byte, re-scan all breakpoints
Lost original byte	Breakpoint structure corruption	Attempt recovery from backup or disable breakpoint
Inconsistent hit state	Multiple breakpoints claim same hit	Clear all hit states, re-examine instruction pointer
Memory access violation	ptrace operations fail	Remove problematic breakpoint, continue with remainder

Architecture Decisions

The **Breakpoint Manager** design involves several critical architectural decisions that significantly impact performance, reliability, and extensibility. Each decision represents a trade-off between competing requirements and constraints.

Decision: Software vs Hardware Breakpoints

- **Context:** Modern CPUs provide both software breakpoints (instruction patching) and hardware breakpoints (debug registers). We must choose the primary implementation approach.
- **Options Considered:**
 1. Software-only (INT3 patching)
 2. Hardware-only (debug registers)
 3. Hybrid approach (hardware when available, software fallback)
- **Decision:** Software breakpoints as primary implementation with hardware breakpoint awareness
- **Rationale:** Software breakpoints provide unlimited quantity (only constrained by memory), work on all x86 processors, and match the learning objectives of understanding instruction patching. Hardware breakpoints are limited to 4 on x86 and require privileged access to debug registers.
- **Consequences:** Higher overhead per breakpoint hit due to restore-step-repatch sequence, but unlimited breakpoint quantity and simpler implementation for learning purposes.

Breakpoint Type	Pros	Cons	Chosen?
Software (INT3)	Unlimited quantity, universally supported, educational value	Higher hit overhead, modifies target code, complex restoration	✓ Primary
Hardware (DR0-3)	No code modification, faster hits, processor-supported	Limited to 4 breakpoints, requires privileged access, architecture-specific	Future extension
Hybrid Approach	Best of both worlds, optimal performance	Complex implementation, difficult testing, feature interaction bugs	Not selected

Decision: Breakpoint Storage Strategy

- **Context:** Breakpoints must be stored in a data structure that supports efficient lookup by address, iteration for batch operations, and dynamic modification.
- **Options Considered:**
 1. Linked list (simple insertion/deletion)
 2. Hash table (fast address lookup)
 3. Sorted array (cache-friendly iteration)
- **Decision:** Linked list for initial implementation
- **Rationale:** Simplicity for learning implementation, typical debugger sessions have fewer than 50 breakpoints making $O(n)$ lookup acceptable, and dynamic insertion/deletion is more important than lookup speed in interactive debugging.
- **Consequences:** $O(n)$ lookup performance, but simpler memory management and easier debugging of the debugger itself.

Storage Option	Pros	Cons	Chosen?
Linked List	Simple implementation, dynamic size, easy insertion/deletion	$O(n)$ lookup time, poor cache locality, pointer traversal overhead	✓ Selected
Hash Table	$O(1)$ average lookup, good for many breakpoints	Complex implementation, memory overhead, hash collision handling	Future optimization
Sorted Array	Cache-friendly, binary search possible, compact memory	$O(n)$ insertion/deletion, fixed size or reallocation complexity	Not selected

Decision: Breakpoint Hit Detection Strategy

- **Context:** When SIGTRAP is received, the debugger must determine if it was caused by a breakpoint hit, single-step completion, or other trap source.
- **Options Considered:**
 1. Instruction pointer matching (check if IP points to known breakpoint)
 2. Signal analysis (examine detailed signal information)
 3. Hybrid approach (combine multiple detection methods)
- **Decision:** Instruction pointer matching with signal validation
- **Rationale:** Instruction pointer provides definitive breakpoint identification, signal analysis alone cannot distinguish between different INT3 sources, and the combination provides robust detection with reasonable complexity.
- **Consequences:** Requires careful instruction pointer adjustment and breakpoint address tracking, but provides reliable hit detection.

Detection Method	Pros	Cons	Chosen?
IP Matching	Definitive breakpoint identification, simple logic	Requires precise IP tracking, sensitive to IP adjustment errors	✓ Primary
Signal Analysis	Additional context, distinguishes trap sources	Cannot identify specific breakpoints, Linux signal info limited	Supporting
Exception Analysis	Detailed processor state, hardware support	Platform-specific, complex implementation, not portable	Not selected

Decision: Breakpoint Enable/Disable Mechanism

- **Context:** Users need to temporarily disable breakpoints without losing them entirely, requiring a mechanism to toggle breakpoints on and off.
- **Options Considered:**
 1. Physical patching (remove/restore INT3 on disable/enable)
 2. Logical flagging (keep INT3, ignore hits when disabled)
 3. List management (move between active/inactive lists)
- **Decision:** Physical patching with logical state tracking
- **Rationale:** Physical patching ensures disabled breakpoints have zero runtime impact and cannot accidentally trigger. Logical flagging would still cause SIGTRAP signals that must be processed and ignored, creating unnecessary overhead.
- **Consequences:** Enable/disable operations require memory writes to target process, but disabled breakpoints have no performance impact on target execution.

Enable/Disable Method	Pros	Cons	Chosen?
Physical Patching	Zero overhead when disabled, clean target memory	Requires memory writes, potential for patch/unpatch errors	✓ Selected
Logical Flagging	Fast enable/disable, no memory operations	Disabled breakpoints still cause signals, runtime overhead	Not selected
List Management	Clean separation, efficient iteration	Complex bookkeeping, potential for list inconsistencies	Not selected

Decision: Error Handling Philosophy

- **Context:** Breakpoint operations can fail due to memory access issues, invalid addresses, or process state problems. The error handling approach affects both debugger stability and user experience.
- **Options Considered:**
 1. Fail-fast (abort operation on first error)
 2. Best-effort (continue with partial success)
 3. Transactional (all-or-nothing with rollback)
- **Decision:** Best-effort with detailed error reporting
- **Rationale:** Interactive debugging sessions benefit from partial success rather than complete failure. Users can work with successfully set breakpoints while addressing failures individually. Transactional semantics add complexity without matching typical debugging workflows.
- **Consequences:** Some breakpoint operations may succeed while others fail, requiring clear user communication about partial results and error details.

Error Approach	Pros	Cons	Chosen?
Fail-Fast	Clear success/failure, simple error handling	All-or-nothing behavior, poor user experience	Not selected
Best-Effort	Maximizes successful operations, flexible user workflow	Partial failures create complex state, unclear error boundaries	✓ Selected
Transactional	Consistent state, predictable behavior	Complex rollback logic, may reject operations unnecessarily	Not selected

Decision: Instruction Pointer Adjustment Timing

- **Context:** When INT3 is hit, the instruction pointer advances past the breakpoint. The debugger must decide when to adjust it back to the breakpoint location.
- **Options Considered:**
 1. Immediate adjustment (fix IP as soon as hit is detected)
 2. Lazy adjustment (fix IP only when continuing execution)
 3. Context-dependent (adjust based on user's next action)
- **Decision:** Immediate adjustment with continuation-time restoration
- **Rationale:** Immediate adjustment ensures that user commands like "print current address" show the expected breakpoint location. The IP is adjusted back to the breakpoint address immediately, then handled properly during continuation.
- **Consequences:** All breakpoint hit handling must account for the adjusted instruction pointer, but user-visible behavior is intuitive and consistent.

Common Pitfalls

The **Breakpoint Manager** implementation involves several subtle pitfalls that frequently trip up developers building their first debugger. These issues often manifest as seemingly random crashes, missed breakpoints, or corrupted program execution.

⚠️ Pitfall: Forgetting Instruction Pointer Adjustment After INT3

When the CPU executes the `INT3` instruction, it automatically advances the instruction pointer by one byte (since `INT3` is a single-byte instruction). This means that when the debugger receives the `SIGTRAP` signal, the instruction pointer is pointing to the byte *after* the breakpoint, not at the breakpoint itself.

Why this breaks: If the debugger doesn't adjust the instruction pointer back, several problems occur:

- User commands like "show current location" display the wrong address
- Single-stepping from the breakpoint starts at the wrong instruction
- The restore-step-repatch sequence restores the original byte but then steps from the wrong location

How to detect: The instruction pointer read from `ptrace(PTRACE_GETREGS)` is one byte past any known breakpoint addresses. Breakpoint lookup by instruction pointer fails.

Correct fix: Immediately after detecting a breakpoint hit, decrement the instruction pointer by one byte and write it back using `ptrace(PTRACE_SETREGS)`. This makes the instruction pointer point to the actual breakpoint location.

```
// In breakpoint hit handling:  
  
// After reading registers with PTRACE_GETREGS  
  
regs.rip -= 1; // Adjust instruction pointer back to breakpoint  
  
ptrace(PTRACE_SETREGS, proc->pid, NULL, &regs);
```

⚠ Pitfall: Double-Patching the Same Address

Setting a breakpoint at an address that already has a breakpoint causes the debugger to save the `INT3` byte (0xCC) as the "original instruction," instead of the actual original instruction byte.

Why this breaks: When the breakpoint is later removed or hit, the debugger restores 0xCC instead of the original instruction. This leaves a permanent `INT3` in the code, causing the program to trap every time it reaches that address, even after breakpoint removal.

How to detect: The original_byte field in multiple breakpoints contains 0xCC, or removing all breakpoints still causes SIGTRAP at those addresses.

Correct fix: Before setting a breakpoint, search the existing breakpoint list for the same address. If found, either reject the duplicate or increment a reference count instead of double-patching.

```
// Before setting breakpoint:  
  
if (breakpoint_list_find(proc->breakpoints, address) != NULL) {  
  
    return DEBUG_BREAKPOINT_EXISTS; // Reject duplicate  
  
}
```

⚠ Pitfall: Not Re-patching After Single-Step

The restore-step-repatch sequence requires re-installing the `INT3` byte after the single-step completes. Forgetting this step permanently disables the breakpoint — it will never trigger again.

Why this breaks: After continuing past a breakpoint, the original instruction is restored for the single-step, but if the `INT3` isn't written back, the breakpoint becomes inactive. Future execution through that address will not stop at the debugger.

How to detect: Breakpoints stop working after being hit once. The breakpoint appears enabled in the debugger's list, but execution passes through without stopping.

Correct fix: After the single-step operation completes (detected via `waitpid()` receiving the single-step signal), immediately re-patch the `INT3` byte at the breakpoint address.

```
// After single-step completes:  
  
if (single_step_restore != NULL) {  
  
    // Re-patch the INT3 to reactivate breakpoint  
  
    ptrace(PTRACE_POKEDATA, proc->pid, single_step_restore->address,  
  
        (original_data & ~0xFF) | INT3);  
  
    single_step_restore = NULL;  
  
}
```

⚠ Pitfall: Incorrect Multi-byte Instruction Handling

On x86, instructions can be 1-15 bytes long, but `INT3` only replaces the first byte. Setting a breakpoint in the middle of a multi-byte instruction corrupts the instruction stream and causes undefined behavior.

Why this breaks: If a breakpoint is set at offset +2 of a 5-byte instruction, the `INT3` overwrites part of the instruction's operand or displacement. When the instruction is restored, it becomes a different instruction entirely, causing crashes or incorrect behavior.

How to detect: Program crashes or behaves incorrectly after hitting certain breakpoints. Disassembly shows corrupted instructions around breakpoint addresses.

Correct fix: Validate that breakpoint addresses align with instruction boundaries. This requires either disassembly capabilities or conservative checking (e.g., only allow breakpoints at function entry points from symbol tables).

```
// Basic validation - ensure address is at start of basic block  
  
if (!symbols_is_instruction_start(proc->symbols, address)) {  
  
    return DEBUG_INVALID_ADDRESS;  
  
}
```

⚠ Pitfall: Race Conditions in Signal Handling

The debugger's main loop uses `waitpid()` to receive signals from the target process. If signals arrive faster than they can be processed, or if the signal handling code isn't reentrant, race conditions can cause missed breakpoints or corrupted state.

Why this breaks: Multiple breakpoints hit in rapid succession, or signals arriving during breakpoint processing, can cause the debugger to lose track of which breakpoint was hit or to process the same signal multiple times.

How to detect: Breakpoints are sometimes missed, or the debugger reports hitting the wrong breakpoint. Multiple SIGTRAP signals seem to be received for a single breakpoint hit.

Correct fix: Process exactly one signal per `waitpid()` call, and ensure that signal processing is atomic with respect to breakpoint state modifications. Use a state machine to track signal processing progress.

```
// Main loop with proper signal handling:  
  
while (proc->state != EXITED) {  
  
    int status;  
  
    pid_t result = waitpid(proc->pid, &status, 0);  
  
  
    if (WIFSTOPPED(status) && WSTOPSIG(status) == SIGTRAP) {  
  
        handle_single_sigtrap(proc); // Process exactly one signal  
  
    }  
  
}
```

C

⚠ Pitfall: Memory Corruption in Breakpoint Lists

Linked list management for breakpoints is prone to classic pointer errors: use-after-free when removing breakpoints, memory leaks when process exits, and corrupted next pointers causing infinite loops during list traversal.

Why this breaks: Corrupted breakpoint lists cause the debugger to crash when listing breakpoints, or cause breakpoints to be lost or duplicated. Memory leaks accumulate over multiple debugging sessions.

How to detect: Debugger crashes when listing or modifying breakpoints. Memory usage grows continuously. List traversal never terminates or skips entries.

Correct fix: Use defensive programming for all list operations: null-check pointers, clear freed memory, and validate list structure integrity after modifications.

```

// Safe breakpoint removal:

void breakpoint_list_remove(BreakpointList* list, uintptr_t address) {

    Breakpoint** current = &list->head;

    while (*current != NULL) {

        if ((*current)->address == address) {

            Breakpoint* to_remove = *current;

            *current = (*current)->next; // Unlink from list

            // Restore original instruction before freeing

            restore_original_instruction(to_remove);

            // Clear and free memory

            memset(to_remove, 0, sizeof(Breakpoint));

            free(to_remove);

            list->count--;

            return;
        }

        current = &(*current)->next;
    }
}

```

! Pitfall: Assuming Process State During Operations

Breakpoint operations like setting, removing, or hitting assume the target process is in a specific state (usually stopped). If the process is running, exited, or in an unexpected state, ptrace operations will fail in confusing ways.

Why this breaks: `ptrace()` calls return `ESRCH` (no such process) or `EPERM` (operation not permitted) when the process state doesn't match the operation requirements. The debugger may attempt to patch memory in a running process, causing race conditions.

How to detect: ptrace operations fail with permission or process-not-found errors. Breakpoint setting appears to succeed but breakpoints don't trigger.

Correct fix: Always verify process state before performing breakpoint operations. Add state checking to all breakpoint functions.

```

DebugResult breakpoint_set(DebuggedProcess* proc, uintptr_t address, const char* description) {
    C

    if (proc->state != STOPPED) {

        return debug_error_create(DEBUG_PROCESS_NOT_STOPPED,
            "Cannot set breakpoint while process is running",
            __FUNCTION__, __LINE__, 0);
    }

    // Proceed with breakpoint setting...
}

}

```

Implementation Guidance

This subsection provides concrete implementation guidance for building the **Breakpoint Manager** component. The focus is on providing complete, working infrastructure code and detailed skeletons for the core breakpoint logic that students should implement themselves.

Technology Recommendations:

Component	Simple Option	Advanced Option
Memory Access	Direct ptrace PEEK/POKE operations	ptrace wrapper with caching and validation
Instruction Analysis	Basic byte-level patching	Capstone disassembly library integration
Error Handling	Simple return codes with errno	Structured error objects with context
Data Structures	Manual linked list implementation	Generic container library (e.g., uthash)
Testing	Manual test programs with known breakpoints	Automated test suite with mock ptrace

File Structure:

```

debugger/
├── src/
│   ├── breakpoint_manager.c      ← Core breakpoint logic (student implements)
│   ├── breakpoint_manager.h     ← Public interface and data structures
│   ├── process_memory.c        ← Memory access utilities (provided)
│   ├── process_memory.h
│   ├── debug_error.c           ← Error handling utilities (provided)
│   └── main.c                  ← Integration with command loop
└── tests/
    ├── test_breakpoints.c       ← Breakpoint-specific tests
    ├── sample_programs/
    │   ├── simple_loop.c         ← Basic breakpoint testing
    │   └── nested_calls.c       ← Complex control flow testing
    └── Makefile

```

Infrastructure Code — Process Memory Access (Complete Implementation):

```
// process_memory.h - Memory access utilities
// C

#ifndef PROCESS_MEMORY_H
#define PROCESS_MEMORY_H

#include <sys/types.h>
#include <stdint.h>
#include <stdbool.h>

// Read a single byte from process memory
// Returns true on success, false on failure
bool process_read_byte(pid_t pid, uintptr_t address, uint8_t* value);

// Write a single byte to process memory
// Returns true on success, false on failure
bool process_write_byte(pid_t pid, uintptr_t address, uint8_t value);

// Read multiple bytes from process memory
// Returns number of bytes successfully read
size_t process_read_memory(pid_t pid, uintptr_t address, void* buffer, size_t size);

// Write multiple bytes to process memory
// Returns number of bytes successfully written
size_t process_write_memory(pid_t pid, uintptr_t address, const void* data, size_t size);

// Validate that an address is readable/writable
bool process_validate_address(pid_t pid, uintptr_t address, bool write_access);

#endif // PROCESS_MEMORY_H
```

```
// process_memory.c - Complete implementation

#include "process_memory.h"

#include <sys/ptrace.h>

#include <errno.h>

#include <string.h>

bool process_read_byte(pid_t pid, uintptr_t address, uint8_t* value) {

    errno = 0;

    long data = ptrace(PTRACE_PEEKDATA, pid, address & ~7, NULL);

    if (data == -1 && errno != 0) {

        return false;

    }

    // Extract the specific byte from the word

    int byte_offset = address & 7;

    *value = (data >> (byte_offset * 8)) & 0xFF;

    return true;

}

bool process_write_byte(pid_t pid, uintptr_t address, uint8_t value) {

    // Read the current word

    errno = 0;

    uintptr_t aligned_addr = address & ~7;

    long data = ptrace(PTRACE_PEEKDATA, pid, aligned_addr, NULL);

    if (data == -1 && errno != 0) {

        return false;

    }

    // Modify the specific byte
```

```
int byte_offset = address & 7;

long mask = 0xFFL << (byte_offset * 8);

data = (data & ~mask) | ((long)value << (byte_offset * 8));

// Write back the modified word

if (ptrace(PTRACE_POKEDATA, pid, aligned_addr, data) == -1) {

    return false;

}

return true;

}

size_t process_read_memory(pid_t pid, uintptr_t address, void* buffer, size_t size) {

    uint8_t* buf = (uint8_t*)buffer;

    size_t bytes_read = 0;

    for (size_t i = 0; i < size; i++) {

        if (!process_read_byte(pid, address + i, &buf[i])) {

            break;

        }

        bytes_read++;

    }

    return bytes_read;

}

size_t process_write_memory(pid_t pid, uintptr_t address, const void* data, size_t size) {

    const uint8_t* buf = (const uint8_t*)data;

    size_t bytes_written = 0;
```

```
for (size_t i = 0; i < size; i++) {

    if (!process_write_byte(pid, address + i, buf[i])) {
        break;
    }

    bytes_written++;
}

return bytes_written;
}

bool process_validate_address(pid_t pid, uintptr_t address, bool write_access) {

    uint8_t test_byte;

    // Test read access

    if (!process_read_byte(pid, address, &test_byte)) {
        return false;
    }

    // Test write access if requested

    if (write_access) {
        // Write the same byte back - should be safe

        if (!process_write_byte(pid, address, test_byte)) {
            return false;
        }
    }

    return true;
}
```

Infrastructure Code — Error Handling (Complete Implementation):

```
// debug_error.h - Structured error handling

#ifndef DEBUG_ERROR_H

#define DEBUG_ERROR_H

#include <errno.h>

typedef enum {

    DEBUG_SUCCESS = 0,
    DEBUG_PTRACE_FAILED,
    DEBUG_PROCESS_EXITED,
    DEBUG_INVALID_ADDRESS,
    DEBUG_SYMBOL_NOT_FOUND,
    DEBUG_BREAKPOINT_EXISTS,
    DEBUG_NO_DEBUG_INFO,
    DEBUG_PROCESS_NOT_STOPPED,
    DEBUG_BREAKPOINT_NOT_FOUND,
    DEBUG_MEMORY_ACCESS_FAILED,
    DEBUG_INSTRUCTION_BOUNDARY_ERROR
} DebugResult;

typedef struct {

    DebugResult code;
    char message[256];
    const char* function;
    int line;
    int system_errno;
} DebugError;

// Macro for creating errors with automatic context

#define DEBUG_ERROR(code, msg) debug_error_create(code, msg, __FUNCTION__, __LINE__, errno)

// Create a structured error with context
```

```
DebugError debug_error_create(DebugResult code, const char* message,
                             const char* function, int line, int system_errno);

// Print formatted error message

void debug_error_print(const DebugError* error);

// Convert DebugResult to human-readable string

const char* debug_result_string(DebugResult result);

#endif // DEBUG_ERROR_H
```

```
// debug_error.c - Complete implementation

#include "debug_error.h"

#include <stdio.h>

#include <string.h>

DebugError debug_error_create(DebugResult code, const char* message,
                             const char* function, int line, int system_errno) {

    DebugError error;

    error.code = code;

    error.function = function;

    error.line = line;

    error.system_errno = system_errno;

    strncpy(error.message, message, sizeof(error.message) - 1);

    error.message[sizeof(error.message) - 1] = '\0';

    return error;
}

void debug_error_print(const DebugError* error) {

    fprintf(stderr, "Error: %s (%s)\n", error->message, debug_result_string(error->code));

    fprintf(stderr, " Location: %s:%d\n", error->function, error->line);

    if (error->system_errno != 0) {

        fprintf(stderr, " System error: %s\n", strerror(error->system_errno));
    }
}

const char* debug_result_string(DebugResult result) {

    switch (result) {

        case DEBUG_SUCCESS: return "Success";

```

```
        case DEBUG_PTRACE_FAILED: return "ptrace operation failed";

        case DEBUG_PROCESS_EXITED: return "Process has exited";

        case DEBUG_INVALID_ADDRESS: return "Invalid memory address";

        case DEBUG_SYMBOL_NOT_FOUND: return "Symbol not found";

        case DEBUG_BREAKPOINT_EXISTS: return "Breakpoint already exists";

        case DEBUG_NO_DEBUG_INFO: return "No debug information available";

        case DEBUG_PROCESS_NOT_STOPPED: return "Process is not stopped";

        case DEBUG_BREAKPOINT_NOT_FOUND: return "Breakpoint not found";

        case DEBUG_MEMORY_ACCESS_FAILED: return "Memory access failed";

        case DEBUG_INSTRUCTION_BOUNDARY_ERROR: return "Instruction boundary error";

        default: return "Unknown error";
    }

}
```

Core Logic Skeleton — Breakpoint Manager (Student Implementation):

```
// breakpoint_manager.h - Public interface

#ifndef BREAKPOINT_MANAGER_H
#define BREAKPOINT_MANAGER_H

#include "debug_error.h"

#include <stdint.h>
#include <stdbool.h>
#include <sys/types.h>

#define INT3 0xCC

typedef struct Breakpoint {

    uintptr_t address;
    uint8_t original_byte;
    bool is_enabled;
    uint32_t hit_count;
    char* location_description;
    struct Breakpoint* next;
} Breakpoint;

typedef struct {

    Breakpoint* head;
    size_t count;
    Breakpoint* hit_breakpoint;           // Currently hit breakpoint
    Breakpoint* single_step_restore;     // Breakpoint being restored
} BreakpointList;

// Initialize empty breakpoint list

BreakpointList* breakpoint_list_create(void);

// Clean up breakpoint list and restore all original instructions

void breakpoint_list_destroy(BreakpointList* list, pid_t pid);
```

```
// Set a new breakpoint at the specified address

DebugResult breakpoint_set(pid_t pid, BreakpointList* list, uintptr_t address,
                         const char* description);

// Remove breakpoint at address

DebugResult breakpoint_remove(pid_t pid, BreakpointList* list, uintptr_t address);

// Handle breakpoint hit (called when SIGTRAP received)

DebugResult breakpoint_handle_hit(pid_t pid, BreakpointList* list, uintptr_t hit_address);

// Continue execution past current breakpoint hit

DebugResult breakpoint_continue_past_hit(pid_t pid, BreakpointList* list);

// Enable/disable breakpoint without removing it

DebugResult breakpoint_enable(pid_t pid, BreakpointList* list, uintptr_t address);

DebugResult breakpoint_disable(pid_t pid, BreakpointList* list, uintptr_t address);

// Find breakpoint by address

Breakpoint* breakpoint_find(BreakpointList* list, uintptr_t address);

// List all breakpoints (for user display)

void breakpoint_list_all(BreakpointList* list);

#endif // BREAKPOINT_MANAGER_H
```

```
// breakpoint_manager.c - Student implementation skeleton
```

```
#include "breakpoint_manager.h"
```

```
#include "process_memory.h"
```

```
#include "debug_error.h"
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <stdio.h>
```

```
BreakpointList* breakpoint_list_create(void) {
```

```
    // TODO 1: Allocate memory for BreakpointList structure
```

```
    // TODO 2: Initialize all fields (head = NULL, count = 0, hit_breakpoint = NULL, etc.)
```

```
    // TODO 3: Return pointer to initialized list
```

```
    // Hint: Use calloc to initialize all fields to zero
```

```
    return NULL; // Student replaces this
```

```
}
```

```
void breakpoint_list_destroy(BreakpointList* list, pid_t pid) {
```

```
    // TODO 1: Check if list is NULL and return early if so
```

```
    // TODO 2: Iterate through all breakpoints in the linked list
```

```
    // TODO 3: For each breakpoint, restore original instruction if it's currently patched
```

```
    // TODO 4: Free the location_description string if allocated
```

```
    // TODO 5: Free the Breakpoint structure itself
```

```
    // TODO 6: Free the BreakpointList structure
```

```
    // Hint: Use a temporary pointer when traversing to avoid use-after-free
```

```
}
```

```
DebugResult breakpoint_set(pid_t pid, BreakpointList* list, uintptr_t address,
```

```
                        const char* description) {
```

```
    // TODO 1: Validate parameters (list not NULL, address not 0)
```

```
    // TODO 2: Check if breakpoint already exists at this address
```

```
    // TODO 3: Validate that the address is accessible for reading/writing
```

```
// TODO 4: Read the original byte at the target address

// TODO 5: Write the INT3 (0xCC) byte to the target address

// TODO 6: Create new Breakpoint structure and initialize all fields

// TODO 7: Copy the description string (allocate memory)

// TODO 8: Add breakpoint to the linked list (at head for simplicity)

// TODO 9: Increment the breakpoint count

// TODO 10: Return DEBUG_SUCCESS

// Hint: Use process_read_byte and process_write_byte from process_memory.h

return DEBUG_SUCCESS; // Student replaces this

}
```

```
DebugResult breakpoint_remove(pid_t pid, BreakpointList* list, uintptr_t address) {

    // TODO 1: Validate parameters

    // TODO 2: Find the breakpoint in the linked list

    // TODO 3: If not found, return DEBUG_BREAKPOINT_NOT_FOUND

    // TODO 4: If breakpoint is currently enabled, restore the original byte

    // TODO 5: Remove breakpoint from linked list (update prev->next or head)

    // TODO 6: Free the location_description string

    // TODO 7: Free the Breakpoint structure

    // TODO 8: Decrement the breakpoint count

    // TODO 9: Return DEBUG_SUCCESS

    // Hint: Need to track previous pointer for linked list removal

    return DEBUG_SUCCESS; // Student replaces this

}
```

```
Breakpoint* breakpoint_find(BreakpointList* list, uintptr_t address) {

    // TODO 1: Check if list is NULL

    // TODO 2: Iterate through linked list starting from head

    // TODO 3: Compare each breakpoint's address with target address

    // TODO 4: Return pointer to breakpoint if found, NULL if not found
```

```

// Hint: Simple linear search is sufficient for typical breakpoint counts

return NULL; // Student replaces this

}

DebugResult breakpoint_handle_hit(pid_t pid, BreakpointList* list, uintptr_t hit_address) {

    // TODO 1: Find the breakpoint that was hit

    // TODO 2: If no breakpoint found at hit_address, return error (shouldn't happen)

    // TODO 3: Increment the hit_count for this breakpoint

    // TODO 4: Store reference to hit breakpoint in list->hit_breakpoint

    // TODO 5: Display hit information to user (address, hit count, description)

    // TODO 6: Return DEBUG_SUCCESS

    // Hint: The hit_address should already be adjusted by the caller (IP - 1)

    return DEBUG_SUCCESS; // Student replaces this

}

DebugResult breakpoint_continue_past_hit(pid_t pid, BreakpointList* list) {

    // TODO 1: Check if there is a currently hit breakpoint

    // TODO 2: Get the hit breakpoint from list->hit_breakpoint

    // TODO 3: Restore the original byte at the breakpoint address

    // TODO 4: Store reference for re-patching after single step

    // TODO 5: Clear the hit_breakpoint field

    // TODO 6: Set single_step_restore to point to this breakpoint

    // TODO 7: Return DEBUG_SUCCESS (caller handles actual single-step)

    // Hint: This function prepares for single-step; caller does PTRACE_SINGLESTEP

    return DEBUG_SUCCESS; // Student replaces this

}

DebugResult breakpoint_enable(pid_t pid, BreakpointList* list, uintptr_t address) {

    // TODO 1: Find breakpoint at the specified address

    // TODO 2: If not found, return DEBUG_BREAKPOINT_NOT_FOUND

    // TODO 3: If already enabled, return DEBUG_SUCCESS (no-op)

```

```

// TODO 4: Write INT3 byte to the address (patch the instruction)

// TODO 5: Set is_enabled flag to true

// TODO 6: Return DEBUG_SUCCESS

return DEBUG_SUCCESS; // Student replaces this

}

DebugResult breakpoint_disable(pid_t pid, BreakpointList* list, uintptr_t address) {

    // TODO 1: Find breakpoint at the specified address

    // TODO 2: If not found, return DEBUG_BREAKPOINT_NOT_FOUND

    // TODO 3: If already disabled, return DEBUG_SUCCESS (no-op)

    // TODO 4: Write original byte back to the address (unpatch the instruction)

    // TODO 5: Set is_enabled flag to false

    // TODO 6: Return DEBUG_SUCCESS

    return DEBUG_SUCCESS; // Student replaces this

}

void breakpoint_list_all(BreakpointList* list) {

    // TODO 1: Check if list is NULL or empty

    // TODO 2: Print header for breakpoint listing

    // TODO 3: Iterate through all breakpoints in the linked list

    // TODO 4: For each breakpoint, print: address, enabled status, hit count, description

    // TODO 5: Print total breakpoint count

    // Hint: Use printf with format specifiers for addresses (0x%lx)

}

// Helper function for re-patching after single step (called by main debug loop)

DebugResult breakpoint_complete_single_step(pid_t pid, BreakpointList* list) {

    // TODO 1: Check if there is a breakpoint awaiting re-patch

    // TODO 2: Get breakpoint from list->single_step_restore

    // TODO 3: Write INT3 byte back to re-enable the breakpoint

    // TODO 4: Clear the single_step_restore field

```

```
// TODO 5: Return DEBUG_SUCCESS

// Hint: This is called after PTRACE_SINGLESTEP completes

return DEBUG_SUCCESS; // Student replaces this

}
```

Language-Specific Implementation Hints:

- **Memory Management:** Use `malloc/free` for dynamic allocation, `calloc` for zero-initialized memory
- **String Handling:** Use `strdup()` for copying description strings, or manual `malloc + strcpy`
- **ptrace Error Checking:** Always check `ptrace()` return value; -1 indicates error, check `errno`
- **Address Formatting:** Use `0x%lx` format specifier for displaying addresses in hex
- **Linked List Traversal:** Always use temporary pointer when freeing nodes to avoid use-after-free

Milestone Checkpoint:

After implementing the breakpoint manager, verify functionality with these steps:

1. **Compile and Link:** `gcc -g -o debugger src/*.c`

2. **Create Test Program:**

```
// test_program.c

#include <stdio.h>

int main() {

    printf("Line 1\n");

    printf("Line 2\n"); // Set breakpoint here

    printf("Line 3\n");

    return 0;

}
```

3. **Test Breakpoint Setting:**

- Start debugger with test program
- Set breakpoint at main function entry
- Expected: Breakpoint created, INT3 patched at address
- Command: `break main` or `break *0x[address]`

4. **Test Breakpoint Hit:**

- Continue execution after setting breakpoint
- Expected: Program stops at breakpoint, displays hit information
- Command: `continue`

- Output should show: "Breakpoint hit at 0x[address], hit count: 1"

5. Test Continue Past Breakpoint:

- Continue after breakpoint hit
- Expected: Program executes normally, breakpoint remains active for future hits
- Command: `continue`

6. Signs of Problems:

- **SIGSEGV when setting breakpoint:** Address validation failed or ptrace permissions issue
- **Breakpoint never hits:** INT3 not properly written, or process not properly stopped
- **Program crashes after breakpoint:** Original byte not properly restored, or instruction pointer not adjusted
- **Breakpoint hits only once:** Re-patching after single-step not working

Debugging the Debugger:

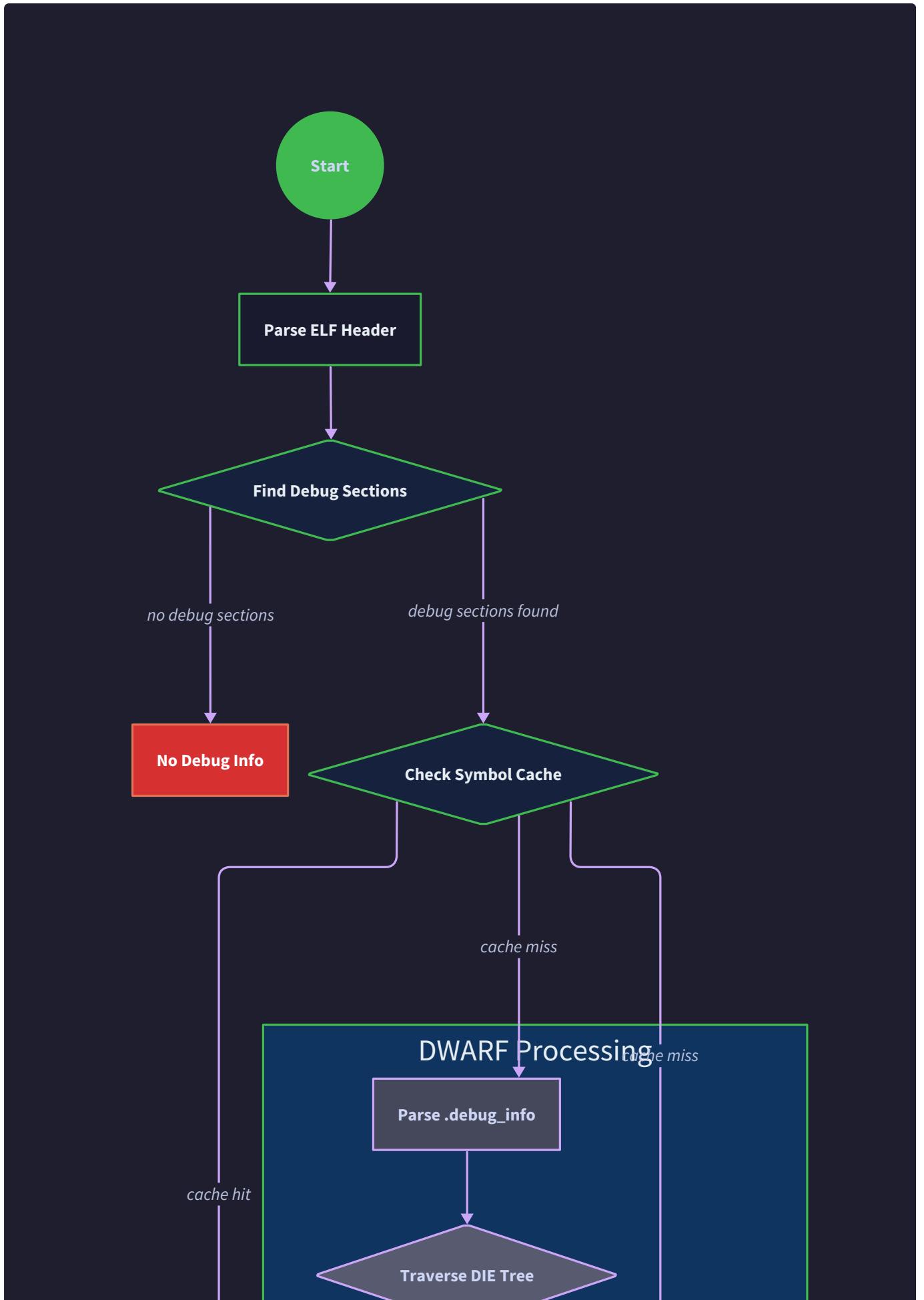
Symptom	Likely Cause	How to Diagnose	Fix
ptrace returns ESRCH	Process not attached or exited	Check process state with <code>ps</code>	Ensure proper process attachment
Breakpoint address shows 0xCC	Double-patching same address	Check original_byte field values	Add duplicate detection before patching
Program crashes at random locations	Corrupted instruction restoration	Use <code>objdump -d</code> to examine instructions	Verify original byte preservation
Infinite loop in breakpoint listing	Corrupted linked list pointers	Add debug prints during list operations	Implement defensive pointer checking
Memory access violations	Invalid address or permissions	Test with <code>process_validate_address</code>	Add address validation before all operations

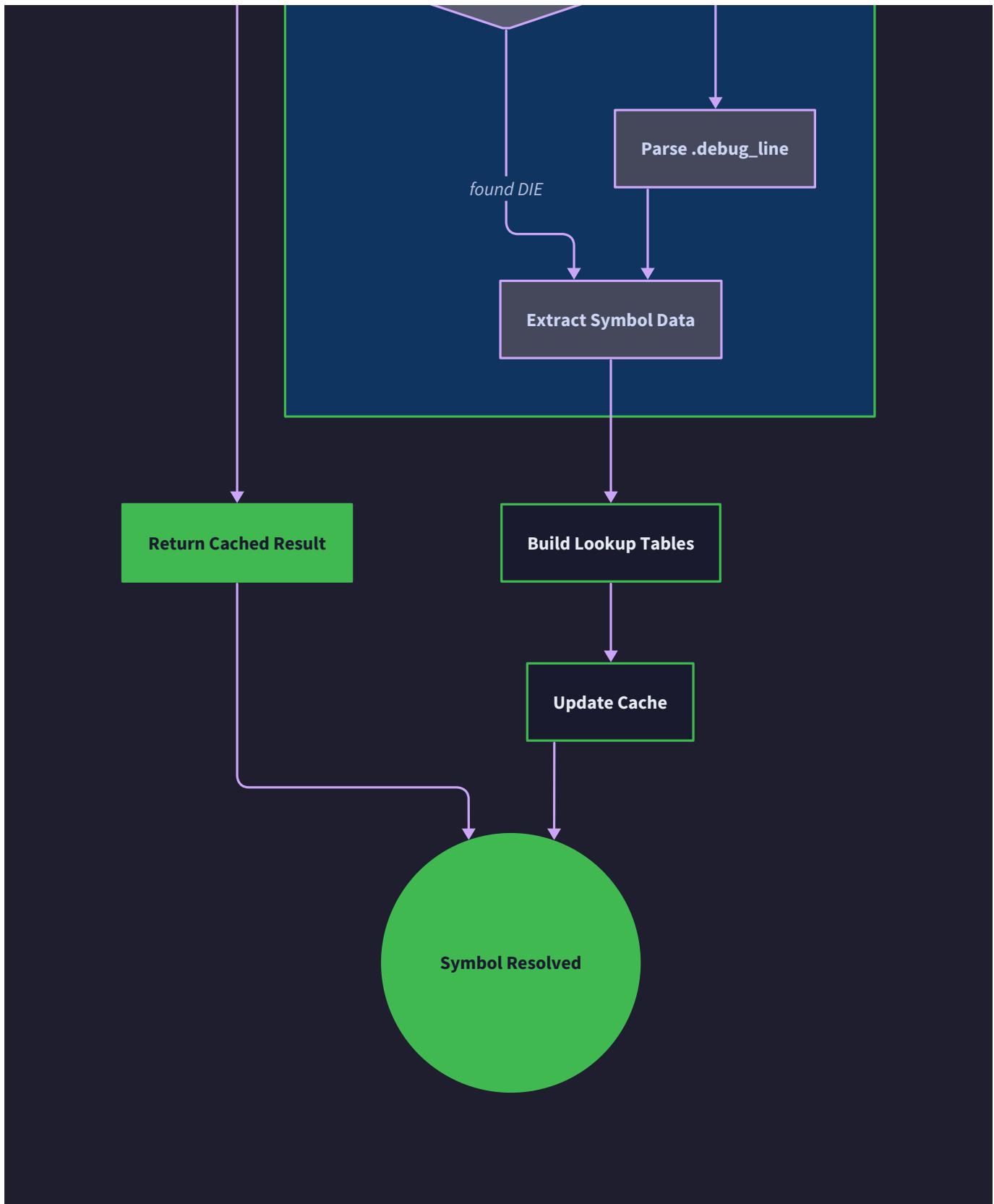
Symbol Resolver Component

Milestone(s): Milestone 3 (Symbol Tables) — this section implements the symbol resolution system that parses ELF and DWARF debug information to enable source-level debugging capabilities including address-to-source mapping and function name resolution

The Rosetta Stone Analogy

Think of the Symbol Resolver as a **Rosetta Stone** for your debugger. Just as the ancient Rosetta Stone contained the same text written in three different scripts (hieroglyphic, demotic, and ancient Greek), allowing scholars to decode Egyptian hieroglyphs by comparing them to known Greek text, debug information serves as a translation dictionary between three different representations of the same program: machine code addresses, assembly instructions, and human-readable source code.





When you compile a C program with the `-g` flag, the compiler acts like a scribe creating this Rosetta Stone. It embeds **debug information** in special sections of the ELF binary that establish correspondence between every machine instruction address and its original source file location, function name, variable name, and data type. Without this translation layer, our debugger would be like an archaeologist staring at hieroglyphs without any way to understand their meaning — we could see the raw bytes and assembly instructions, but we couldn't tell the user "you're currently stopped at line 42 in `main.c` inside the `calculate_fibonacci` function."

The Symbol Resolver's job is to parse this embedded Rosetta Stone and build fast lookup tables that can instantly translate in both directions: given a memory address like `0x401234`, tell me the source file and line number; given a function name like `fibonacci`, tell me its entry point address so I can set a breakpoint there.

The complexity comes from the fact that this translation isn't simple — modern compilers perform optimizations that can inline functions, eliminate variables, reorder instructions, and create multiple machine code sequences for the same source line. The debug information format (DWARF) has evolved to handle these complexities, but parsing it correctly requires understanding a sophisticated hierarchical data structure with cross-references, compression, and multiple encoding formats.

ELF Format Parsing

The **Executable and Linkable Format (ELF)** serves as the container for both the executable machine code and the debug information. Think of an ELF file like a filing cabinet with clearly labeled drawers — each section has a specific

purpose and location within the file. Before we can parse any debug information, we must first understand how to navigate this filing cabinet and locate the drawers that contain what we need.

An ELF file begins with a header that acts like a table of contents, followed by a series of sections that contain different types of data. Our Symbol Resolver is primarily interested in sections whose names start with `.debug_` — these contain the DWARF debug information. However, we also need the traditional symbol table sections like `.symtab` and `.strtab` for basic function and global variable names.

Critical Insight: ELF parsing must handle both 32-bit and 64-bit formats, different endianness, and various DWARF versions. The parser needs to be robust enough to gracefully degrade when debug information is incomplete or corrupted.

The ELF parsing process follows a systematic approach that builds up our understanding of the file structure layer by layer:

- 1. ELF Header Validation:** Read the first 16 bytes to verify the ELF magic number (`0x7F` followed by "ELF"), determine the file's architecture (32-bit vs 64-bit), endianness (little vs big endian), and ELF version. This information determines how to interpret all subsequent data structures in the file.
- 2. Section Header Table Location:** The ELF header contains an offset to the section header table, which is an array of entries describing each section in the file. Each entry contains the section's name, type, memory address, file offset, size, and flags.
- 3. String Table Resolution:** Section names are stored as indices into a string table section (usually `.shstrtab`). We must locate and read this string table first so we can resolve section names from their numeric indices.
- 4. Debug Section Discovery:** Iterate through all section headers, resolving their names and identifying the debug-related sections we need. The critical sections for source-level debugging include `.debug_info` (compilation unit structure), `.debug_line` (address-to-line mappings), `.debug_abbrev` (debug information entry templates), and `.debug_str` (debug string table).
- 5. Symbol Table Sections:** Traditional ELF symbol tables in `.symtab` provide function and global variable names with their addresses. These complement the more detailed DWARF information and serve as a fallback when DWARF parsing fails.

Section Name	Purpose	Content Format
<code>.debug_info</code>	Primary debug information	DWARF compilation units and debug information entries
<code>.debug_line</code>	Address-to-source mapping	Line number program bytecode
<code>.debug_abbrev</code>	DIE template definitions	Abbreviation declarations for <code>.debug_info</code>
<code>.debug_str</code>	Debug string table	Null-terminated strings referenced by debug info
<code>.symtab</code>	Traditional symbol table	ELF symbol table entries with names and addresses
<code>.strtab</code>	Symbol name strings	Null-terminated strings for symbol names
<code>.debug_ranges</code>	Address range lists	Non-contiguous address ranges for functions/variables
<code>.debug_loc</code>	Location lists	Variable location expressions for different address ranges

The parsing implementation must handle several architectural considerations. Different CPU architectures store multi-byte values in different byte orders (endianness), so all integer reads must respect the endianness flag from the ELF header. The ELF format supports both 32-bit and 64-bit variants with different structure sizes — addresses are 4 bytes in 32-bit ELF but 8 bytes in 64-bit ELF.

Design Insight: Rather than loading entire debug sections into memory at once, implement lazy loading that reads section headers immediately but defers section content loading until needed. This keeps memory usage reasonable for large binaries while still providing fast access to frequently-used information.

Error handling during ELF parsing must distinguish between recoverable and fatal errors. A missing `.debug_info` section means we can't provide source-level debugging but can still offer basic function names from `.syms`. However, a corrupted ELF header or inaccessible file represents a fatal error that prevents any symbol resolution.

Architecture Decision: ELF Parser Architecture

Decision: Layered ELF Parser with Lazy Loading

- **Context:** ELF files can be very large (hundreds of megabytes), and debug sections may never be accessed during a debugging session. Loading everything eagerly wastes memory and startup time.
- **Options Considered:**
 1. Eager loading: Parse and cache all sections at startup
 2. On-demand loading: Parse sections only when first accessed
 3. Hybrid approach: Parse headers eagerly, load content lazily
- **Decision:** Hybrid approach with lazy section content loading
- **Rationale:** Section headers are small (typically a few KB) and needed for navigation, while section contents can be large and may never be accessed. Lazy loading provides good memory efficiency while maintaining reasonable access performance.
- **Consequences:** Enables handling large debug binaries efficiently but requires careful error handling when section loading fails during symbol resolution operations.

Option	Memory Usage	Startup Time	Access Performance	Implementation Complexity
Eager Loading	High	Slow	Fastest	Low
Pure On-Demand	Lowest	Fastest	Slower	Medium
Hybrid (Chosen)	Medium	Medium	Fast	Medium

DWARF Debug Information Processing

DWARF (Debug With Arbitrary Record Format) is the industry-standard format for encoding debug information in Unix-like systems. Think of DWARF as a compressed database embedded within the ELF binary that describes the complete structure of your source program using a hierarchical tree of records called **Debug Information Entries (DIEs)**.

Each compilation unit (typically one source file) creates a tree of DIEs that describes all the functions, variables, types, and source locations within that unit. A DIE is like a database record with a type tag and a list of attributes. For example, a

function DIE might have attributes for its name, start address, end address, return type, parameter list, and source file location.

The genius of DWARF is its use of **abbreviation tables** to compress this information. Instead of storing the complete attribute list with every DIE, DWARF defines abbreviation codes that specify which attributes a particular type of DIE will have. This is like having a template that says "DIE type 15 always has a name attribute, a start address attribute, and a line number attribute" — then each function DIE just stores the abbreviation code 15 followed by the actual values.

The DWARF processing pipeline transforms this compressed hierarchical data into the lookup tables our debugger needs:

1. **Compilation Unit Discovery:** The `.debug_info` section contains multiple compilation units, each representing one source file that was compiled. Each compilation unit begins with a header specifying its length, DWARF version, abbreviation table offset, and address size.
2. **Abbreviation Table Parsing:** For each compilation unit, parse its associated abbreviation table from `.debug_info`. Each abbreviation entry specifies a DIE tag (like `DW_TAG_subprogram` for functions), a list of attribute names and encoding formats, and whether this DIE has children in the tree.
3. **DIE Tree Traversal:** Parse the DIE tree for each compilation unit using a depth-first traversal. Each DIE begins with an abbreviation code that determines its structure. Use the abbreviation table to decode the attribute list and determine whether this DIE has children.
4. **Attribute Value Extraction:** DWARF attributes can be encoded in many formats — constants, strings, references to other DIES, location expressions, or offsets into other sections. The attribute form determines how to decode the raw bytes into meaningful values.
5. **Cross-Reference Resolution:** Many DIES reference other DIES by offset. For example, a function parameter DIE references a type DIE to specify its data type. Build a map from DIE offsets to parsed DIE structures to resolve these references efficiently.

DWARF DIE Tag	Purpose	Key Attributes	Children
<code>DW_TAG_compile_unit</code>	Source file compilation unit	name, comp_dir, producer, language	All top-level definitions
<code>DW_TAG_subprogram</code>	Function or method	name, low_pc, high_pc, type	Parameters, local variables
<code>DW_TAG_variable</code>	Global or local variable	name, type, location	None
<code>DW_TAG_formal_parameter</code>	Function parameter	name, type, location	None
<code>DW_TAG_base_type</code>	Primitive data type	name, byte_size, encoding	None
<code>DW_TAG_pointer_type</code>	Pointer type	type, byte_size	None
<code>DW_TAG_structure_type</code>	Struct or class	name, byte_size	Member variables
<code>DW_TAG_member</code>	Struct field	name, type, data_member_location	None

The most complex aspect of DWARF processing is handling **location expressions**. Variables and parameters don't always live in simple memory locations — they might be in CPU registers, computed by adding offsets to stack pointers,

split across multiple registers, or completely optimized away. DWARF encodes these locations using a stack-based virtual machine language with operators like `DW_OP_reg0` (value is in register 0), `DW_OP_fbreg` (add offset to frame base), and `DW_OP_piece` (value spans multiple locations).

Design Insight: The key to efficient DWARF parsing is building the right intermediate data structures. Rather than exposing the raw DIE tree to the rest of the debugger, transform it into purpose-built lookup tables optimized for the queries your debugger needs to perform.

Architecture Decision: DWARF Version Support Strategy

Decision: Support DWARF versions 2, 3, and 4 with graceful degradation

- **Context:** Different compiler versions and optimization levels produce different DWARF versions. DWARF 5 introduces significant format changes but isn't widely deployed yet. Our debugger needs to work with binaries compiled over the last decade.
- **Options Considered:**
 1. Support only DWARF 4 (latest stable version)
 2. Support DWARF 2-4 with version-specific parsers
 3. Support DWARF 2-5 with unified parser
- **Decision:** Support DWARF 2-4 with unified parser and version detection
- **Rationale:** DWARF 2-4 covers the vast majority of binaries in use today. The format changes between these versions are incremental and can be handled by a single parser with version checks. DWARF 5 support can be added later without architectural changes.
- **Consequences:** Enables debugging most real-world binaries while keeping parser complexity manageable. Missing DWARF 5 features limits effectiveness with newest compilers.

DWARF Version	Compiler Support	Key Features	Parsing Complexity
DWARF 2	GCC 3.x, older systems	Basic debug info	Low
DWARF 3	GCC 4.x, widespread	Improved expressions	Medium
DWARF 4 (Target)	GCC 4.5+, current default	Better compression	Medium
DWARF 5	GCC 7+, cutting edge	Major restructuring	High

The DWARF parsing implementation must handle several error conditions gracefully. Corrupted or truncated debug information should not crash the debugger but should fall back to whatever partial information was successfully parsed. Unsupported DWARF extensions should be skipped rather than causing parse failures.

Address-to-Source Mapping

The **address-to-source mapping** is one of the most frequently used capabilities of the Symbol Resolver. Every time the debugger stops execution — whether at a breakpoint, after single-stepping, or due to a signal — it needs to instantly tell the user exactly where they are in their source code. This requires translating a machine code address (program counter value) into a source filename and line number.

Think of this mapping like a **GPS system for your code**. Just as GPS translates your geographic coordinates into a street address and turn-by-turn directions, the address-to-source mapping translates your program counter coordinates into source code locations and navigation information. The challenge is that this mapping is not one-to-one — compiler optimizations can cause a single source line to generate multiple non-contiguous machine code sequences, or multiple source lines to be combined into a single optimized instruction sequence.

DWARF encodes this mapping information in the `.debug_line` section using a compressed format called the **Line Number Program**. This is a bytecode program that, when executed, generates a complete table mapping every machine instruction address to its corresponding source file and line number. The bytecode format exists because storing explicit address-to-line pairs for every instruction would be enormous — a typical function might have hundreds of instructions but only dozens of distinct source lines.

The Line Number Program uses a virtual machine with registers that track the current state as it processes the bytecode:

Register	Purpose	Initial Value
<code>address</code>	Current machine code address	0
<code>file</code>	Current source file index	1
<code>line</code>	Current source line number	1
<code>column</code>	Current column number	0
<code>is_stmt</code>	Is this a statement boundary?	true
<code>basic_block</code>	Is this a basic block start?	false
<code>end_sequence</code>	Is this the end of sequence?	false

The bytecode program consists of opcodes that modify these registers and emit address-to-line mapping entries.

Standard opcodes like `DW_LNS_advance_pc` increment the address register, `DW_LNS_advance_line` modifies the line register, and `DW_LNS_copy` emits a mapping entry with the current register values. Special opcodes combine common operations — incrementing both address and line — into single bytes for better compression.

Building an efficient address-to-source mapping requires careful data structure selection. The naive approach of storing every mapping entry in a linear array would require binary search for each lookup. However, most lookups query addresses within the same function, so a two-level structure performs better:

- 1. Function-Level Index:** Map address ranges to compilation units and functions. This provides coarse-grained location quickly and helps scope the detailed search.
- 2. Instruction-Level Mapping:** Within each function's address range, store the detailed instruction-to-line mappings in a sorted array optimized for binary search.
- 3. File Name Resolution:** Line number program entries reference source files by index into a file table stored in the compilation unit header. Build a mapping from file indices to actual filenames.

The implementation must handle several complex scenarios that arise from compiler optimizations:

Inlined Functions: Modern compilers inline function calls to eliminate call overhead. DWARF represents inlined functions using `DW_TAG_inlined_subroutine` DIEs with address range information. When looking up an address that falls within an inlined function, the Symbol Resolver must return a call stack showing both the inlined function and its call site.

Non-Contiguous Functions: Link-time optimization can split functions into hot and cold sections placed at different addresses. DWARF uses `DW_AT_ranges` attributes to specify multiple address ranges for a single function. The address lookup must check all ranges when determining function membership.

Optimized-Away Code: Aggressive optimization can eliminate entire code paths, leaving gaps in the address space where no source mapping exists. The Symbol Resolver should detect these gaps and report them clearly rather than returning incorrect mappings.

Design Insight: Cache recently-used address-to-source lookups since debugger operations often query nearby addresses repeatedly. A small LRU cache of 100-200 entries can eliminate 80-90% of expensive DWARF lookups during typical debugging sessions.

Architecture Decision: Address Mapping Data Structure

Decision: Interval tree with function-level caching

- **Context:** Address-to-source lookups happen on every single-step and breakpoint hit, making them performance-critical. Functions can have complex address ranges due to optimization, and lookups often cluster around the current execution point.
- **Options Considered:**
 1. Linear array with binary search
 2. Hash table with address bucketing
 3. Interval tree with range queries
- **Decision:** Interval tree with LRU cache for recent lookups
- **Rationale:** Interval trees handle non-contiguous address ranges efficiently and provide $O(\log n)$ lookup time. The LRU cache eliminates repeated expensive lookups during single-stepping through the same function.
- **Consequences:** Provides excellent lookup performance for typical debugging patterns, but requires more complex implementation and higher memory usage than simpler approaches.

Approach	Lookup Time	Memory Usage	Range Support	Implementation Complexity
Linear Array	$O(\log n)$	Low	Poor	Low
Hash Table	$O(1)$ average	Medium	Poor	Medium
Interval Tree (Chosen)	$O(\log n)$	Higher	Excellent	High

Symbol Table Management

Symbol table management encompasses the broader task of maintaining and providing access to all the symbolic information extracted from ELF and DWARF sources. Think of the Symbol Resolver as a **librarian managing a vast reference collection** — it must organize information from multiple sources (traditional ELF symbol tables, DWARF debug information, dynamic symbol tables) into a unified system that can quickly answer diverse queries from other debugger components.

The Symbol Resolver must handle two primary types of symbol lookup operations that flow in opposite directions:

Name-to-Address Resolution: When a user types "break fibonacci", the debugger needs to find the memory address where the `fibonacci` function begins. This requires searching through function symbols by name and returning the entry point address where a breakpoint can be set.

Address-to-Name Resolution: When the debugger hits a breakpoint or displays a stack trace, it needs to determine which function contains a given address. This requires range-based lookup to find the symbol whose address range encompasses the query address.

The complexity arises because symbols come from multiple sources with different capabilities and coverage:

Symbol Source	Coverage	Information Available	Lookup Performance
<code>.syms</code> ELF section	Global functions, global variables	Name, address, size, binding	Fast hash lookup
DWARF <code>DW_TAG_subprogram</code>	All functions (including static)	Name, address range, parameters, locals	Tree traversal
DWARF <code>DW_TAG_variable</code>	All variables (including locals)	Name, type, location expression	Context-dependent
<code>.dynsym</code> ELF section	Dynamically linked functions	Name, address (may be PLT stub)	Fast hash lookup

A robust symbol table implementation must integrate information from all these sources while handling conflicts and gaps:

- Symbol Deduplication:** The same function might appear in both `.syms` and DWARF information with slightly different addresses or names. The Symbol Resolver must detect and merge duplicate symbols, typically preferring DWARF information when available since it's more accurate.
- Scope Handling:** DWARF provides hierarchical scope information — local variables exist only within specific functions and address ranges, while static functions are visible only within their compilation unit. The symbol lookup must respect these scope rules.
- Name Mangling:** C++ compilers mangle function and method names to encode parameter types and namespaces. The Symbol Resolver should store both mangled and demangled names and support lookup by either form.
- Weak Symbols and Overrides:** ELF supports weak symbols that can be overridden by strong symbols with the same name. The Symbol Resolver must implement the correct precedence rules when building its lookup tables.

The symbol table data structures must support efficient lookup in both directions while minimizing memory usage:

```

Symbol Table Components:
├─ Name-to-Address Hash Table
|  ├─ Function names → Symbol records
|  ├─ Variable names → Symbol records
|  └─ Demangled C++ names → Symbol records
├─ Address Range Tree
|  ├─ Function address ranges → Symbol records
|  ├─ Variable scope ranges → Symbol records
|  └─ Compilation unit ranges → DIE pointers
└─ Type Information Database
    ├─ Base types (int, float, char, etc.)
    ├─ Composite types (struct, union, array)
    └─ Type relationships (pointer-to, array-of)

```

Incremental Symbol Loading: Rather than parsing all debug information at startup, implement demand-driven loading that parses symbols as needed. This is particularly important for large programs with extensive debug information — parsing everything eagerly can take seconds and consume hundreds of megabytes of memory.

The loading strategy should prioritize the most commonly-needed information:

1. Load ELF symbol tables immediately (they're small and provide basic coverage)
2. Parse DWARF compilation unit headers to build a directory of available information
3. Parse individual compilation units only when symbols from that unit are requested
4. Cache parsed compilation units using an LRU policy to balance memory usage and access speed

Critical Insight: Symbol table management is not just about parsing and storage — it's about providing the right abstractions to the rest of the debugger. The Breakpoint Manager shouldn't need to understand DWARF DIE trees; it should work with clean APIs like `symbols_resolve_function(name, &address)` and `symbols_resolve_address(address, &filename, &line)`.

Architecture Decision: Symbol Caching Strategy

Decision: Multi-tier caching with compilation-unit granularity

- **Context:** DWARF parsing is expensive (can take 10-50ms per compilation unit), but most debugging sessions access symbols from only a small subset of compilation units. Memory usage must remain reasonable even for large programs.
- **Options Considered:**
 1. Parse everything at startup, no caching needed
 2. Parse on-demand, cache individual symbols
 3. Parse on-demand, cache at compilation unit level
- **Decision:** On-demand parsing with compilation unit level caching and LRU eviction
- **Rationale:** Compilation units are the natural granularity for DWARF parsing — parsing partial units is complex and error-prone. Most debugging sessions touch 5-10 compilation units out of hundreds, making LRU caching very effective.
- **Consequences:** Excellent memory efficiency and good performance for typical usage, but first access to a new compilation unit may have noticeable latency.

Strategy	Startup Time	Memory Usage	Access Latency	Implementation Complexity
Eager Parsing	Very Slow	Very High	None	Low
Individual Symbol Cache	Fast	Medium	Medium	High
Compilation Unit Cache (Chosen)	Fast	Low	Low after first access	Medium

Architecture Decisions

The Symbol Resolver component requires several critical architectural decisions that affect both performance and maintainability. These decisions establish the foundation for how debug information is processed, stored, and accessed throughout the debugger's lifetime.

Architecture Decision: DWARF Parser Architecture

Decision: Streaming parser with incremental state building

- **Context:** DWARF debug information can be enormous (100+ MB for large C++ programs) and contains complex cross-references between DIEs. Memory usage must remain reasonable while providing efficient access to parsed information.
- **Options Considered:**
 1. DOM-style parser: Build complete in-memory tree of all DIEs
 2. SAX-style streaming parser: Process DIEs on-demand without retention
 3. Hybrid approach: Stream processing with selective caching
- **Decision:** Streaming parser that builds focused lookup tables rather than retaining raw DIE structures
- **Rationale:** Most debugging operations need specific lookup operations (name → address, address → line) rather than full DIE tree traversal. Building optimized lookup tables during streaming provides better performance and memory efficiency than retaining the raw parsed structure.
- **Consequences:** Enables handling very large debug information efficiently, but makes some advanced debugging features (like complete type introspection) more difficult to implement.

Approach	Memory Usage	Parse Time	Lookup Performance	Implementation Complexity
Full DOM Tree	Very High	Slow	Fast	Medium
Pure Streaming	Very Low	Fast	Very Slow	High
Selective Caching (Chosen)	Medium	Medium	Fast	Medium

Architecture Decision: Multi-Version DWARF Support

Decision: Plugin-based version handlers with shared infrastructure

- **Context:** DWARF has evolved through multiple versions (2, 3, 4, 5) with different attribute encodings, opcodes, and section layouts. Supporting multiple versions without code duplication requires careful abstraction.
- **Options Considered:**
 1. Monolithic parser with version switches throughout
 2. Separate parser implementation for each version
 3. Shared infrastructure with version-specific handlers
- **Decision:** Shared parsing infrastructure with pluggable version-specific handlers for differences
- **Rationale:** Most DWARF parsing logic is identical across versions — the same DIE tree traversal, abbreviation table processing, and attribute extraction patterns apply. Version differences are concentrated in specific areas like attribute encoding and opcode sets.
- **Consequences:** Enables supporting multiple DWARF versions without excessive code duplication, but requires careful interface design to isolate version-specific behavior.

Architecture Decision: Error Recovery Strategy

Decision: Best-effort parsing with graceful degradation

- **Context:** Real-world binaries often have corrupted or incomplete debug information due to toolchain bugs, link-time stripping, or file system corruption. The debugger must remain functional even with imperfect debug information.
- **Options Considered:**
 1. Strict parsing: Fail completely on any debug information error
 2. Best-effort parsing: Skip corrupted sections but continue processing
 3. Redundant parsing: Use multiple debug information sources with fallbacks
- **Decision:** Best-effort parsing with multiple fallback layers
- **Rationale:** Partial debug information is better than no debug information. Users can still debug effectively with basic symbol tables even if DWARF line number information is corrupted. The debugger should degrade gracefully rather than becoming completely unusable.
- **Consequences:** Provides robust operation with real-world binaries, but makes it harder to detect subtle debug information corruption that could lead to incorrect symbol resolution.

Common Pitfalls

Symbol resolution involves parsing complex binary formats and managing large amounts of interconnected data, creating numerous opportunities for subtle errors that can cause incorrect debugging behavior or crashes.

⚠ Pitfall: Incorrect Endianness Handling

One of the most common errors in ELF parsing is failing to respect the byte order specified in the ELF header. When reading multi-byte integers from the ELF file, you must check the `e_ident[EI_DATA]` field and swap bytes appropriately for big-endian files on little-endian systems (or vice versa). **Why it's wrong:** Reading a big-endian 32-bit address `0x12345678` as little-endian produces `0x78563412`, causing symbol lookups to fail with completely wrong addresses.

How to fix: Implement byte-swapping functions and use them consistently for all multi-byte reads based on the ELF header's endianness flag.

⚠ Pitfall: Assuming Contiguous Address Ranges

Many developers assume that functions occupy single, contiguous address ranges, but link-time optimization can split functions into multiple non-contiguous segments. Setting a breakpoint at "function + offset" might land in unrelated code if the offset falls in a gap between function segments. **Why it's wrong:** The debugger might set breakpoints in wrong functions or report incorrect source locations when execution hits optimized code sections. **How to fix:** Always use DWARF range lists (`DW_AT_ranges`) when available, and validate that computed addresses fall within the actual function boundaries rather than assuming linear layout.

⚠ Pitfall: Memory Leaks in DIE Tree Parsing

DWARF DIE trees contain complex cross-references and recursive structures that make memory management challenging. Failing to properly free DIE structures, especially when parsing fails partway through a compilation unit, leads to memory leaks that accumulate over long debugging sessions. **Why it's wrong:** Each leaked compilation unit can consume several megabytes, eventually causing the debugger to exhaust system memory. **How to fix:** Use reference counting or garbage collection for DIE structures, and ensure that all allocated DIE trees are freed even when parsing encounters errors.

⚠ Pitfall: Ignoring DWARF Attribute Form Encoding

DWARF attributes can be encoded in multiple forms (constants, strings, references, etc.), and the attribute form determines how to decode the raw bytes. Using the wrong decoding logic produces garbage values that cause symbol resolution to return incorrect information. **Why it's wrong:** A function name encoded as a string offset might be interpreted as a direct string pointer, causing the debugger to display random memory contents as function names. **How to fix:** Always check the attribute form from the abbreviation table and use the appropriate decoding logic for each form type.

⚠ Pitfall: Race Conditions in Lazy Symbol Loading

When implementing on-demand symbol loading, multiple debugger threads might simultaneously request symbols from the same compilation unit, causing duplicate parsing work or corrupted symbol tables if proper synchronization isn't implemented. **Why it's wrong:** Duplicate parsing wastes CPU time and memory, while corrupted symbol tables cause incorrect breakpoint placement and variable inspection failures. **How to fix:** Use proper locking around symbol loading operations, or implement atomic flag-based loading that ensures each compilation unit is parsed exactly once.

⚠ Pitfall: Incorrect Source Path Resolution

DWARF debug information often contains relative source file paths that must be resolved against the compilation directory (`DW_AT_comp_dir`). Failing to handle this correctly causes source file lookups to fail when the debugger runs from a different directory than where compilation occurred. **Why it's wrong:** The debugger can't display source code or set breakpoints by line number, severely limiting its usefulness for source-level debugging. **How to fix:** Always combine relative source paths with the compilation directory, and implement fallback logic to search common source locations when the original path is no longer valid.

⚠ Pitfall: Assuming All Functions Have Debug Information

Not all functions in a binary have complete DWARF debug information — some might be compiled without debug symbols, come from libraries compiled without `-g`, or be generated by the linker itself (like PLT stubs). Assuming complete debug info is available causes crashes when accessing non-existent DIE structures. **Why it's wrong:** Attempting to access DWARF information for functions that don't have it causes null pointer dereferences or array bounds

violations. **How to fix:** Always check whether debug information exists for a symbol before accessing it, and provide fallback behavior using basic ELF symbol table information when DWARF data is unavailable.

Implementation Guidance

The Symbol Resolver represents the most complex parsing and data management component in the debugger. This implementation guidance provides the foundation code and structure needed to handle ELF and DWARF parsing without overwhelming beginning implementers.

Technology Recommendations

Component	Simple Option	Advanced Option
ELF Parsing	Manual struct reading with <code>fread()</code>	libelf library for ELF manipulation
DWARF Parsing	Custom parser with manual DIE traversal	libdwarf library for debug information
Symbol Storage	Hash table with chaining (<code>uthash.h</code>)	Red-black tree with range queries
Memory Management	Manual malloc/free with careful tracking	Memory pool allocator for DIE structures
String Handling	C strings with <code>strdup()</code> for symbol names	String interning table to reduce duplicates

Recommended File Structure

```
debugger/ C

├── src/
│   ├── symbol_resolver.c      ← main symbol resolution logic
│   ├── elf_parser.c          ← ELF format parsing implementation
│   ├── dwarf_parser.c        ← DWARF debug information parsing
│   ├── symbol_cache.c        ← symbol table and caching management
│   └── location_expr.c       ← DWARF location expression evaluator
├── include/
│   ├── symbol_resolver.h     ← public symbol resolution API
│   ├── elf_parser.h          ← ELF parsing structures and functions
│   ├── dwarf_parser.h        ← DWARF parsing structures and functions
│   └── debug_types.h         ← common debug information data types
└── tests/
    ├── test_symbols/          ← test binaries with debug information
    ├── symbol_resolver_test.c ← symbol resolution functionality tests
    └── parser_test.c          ← ELF and DWARF parsing unit tests
```

Infrastructure Starter Code

ELF Basic Structures (`include/elf_parser.h`):

```
#include <stdint.h>
#include <stdio.h>

// ELF header structure for 64-bit binaries

typedef struct {

    uint8_t e_ident[16];          // ELF identification

    uint16_t e_type;              // Object file type

    uint16_t e_machine;           // Architecture

    uint32_t e_version;            // Object file version

    uint64_t e_entry;              // Entry point virtual address

    uint64_t e_phoff;              // Program header table file offset

    uint64_t e_shoff;              // Section header table file offset

    uint32_t e_flags;              // Processor-specific flags

    uint16_t e_ehsize;             // ELF header size in bytes

    uint16_t e_phentsize;           // Program header table entry size

    uint16_t e_phnum;               // Program header table entry count

    uint16_t e_shentsize;           // Section header table entry size

    uint16_t e_shnum;                // Section header table entry count

    uint16_t e_shstrndx;             // Section header string table index

} Elf64_Ehdr;

// Section header structure for 64-bit binaries

typedef struct {

    uint32_t sh_name;              // Section name (string table offset)

    uint32_t sh_type;               // Section type

    uint64_t sh_flags;              // Section flags

    uint64_t sh_addr;                // Section virtual address at execution

    uint64_t sh_offset;              // Section file offset

    uint64_t sh_size;                // Section size in bytes

    uint32_t sh_link;                  // Link to another section

    uint32_t sh_info;                  // Additional section information
```

```

    uint64_t sh_addralign;      // Section alignment

    uint64_t sh_entsize;        // Entry size if section holds table

} Elf64_Shdr;

// ELF file context for parsing operations

typedef struct {

    FILE *file;

    Elf64_Ehdr header;

    Elf64_Shdr *section_headers;

    char *section_names;

    int is_64bit;

    int is_little_endian;

} ElfFile;

// ELF parsing functions

DebugResult elf_open(const char *filename, ElfFile *elf);

DebugResult elf_find_section(ElfFile *elf, const char *name, Elf64_Shdr **section);

DebugResult elf_read_section_data(ElfFile *elf, Elf64_Shdr *section, void **data, size_t *size);

void elf_close(ElfFile *elf);

```

Symbol Table Data Structures (`include/debug_types.h`):

```
// Symbol information record C

typedef struct Symbol {

    char *name;                      // Symbol name (function or variable)

    char *demangled_name;            // Demangled C++ name (if applicable)

    uintptr_t address;               // Symbol address (start for functions)

    uintptr_t size;                  // Symbol size (function length)

    int symbol_type;                // Function, variable, etc.

    int binding;                    // Local, global, weak

    struct Symbol *next;             // Hash table chaining

} Symbol;

// Source location information

typedef struct SourceLocation {

    char *filename;                 // Source file path

    char *directory;                // Compilation directory

    uint32_t line_number;           // Line number

    uint32_t column_number;         // Column number (if available)

} SourceLocation;

// Address-to-source mapping entry

typedef struct AddressMapping {

    uintptr_t address;              // Machine code address

    SourceLocation location;        // Corresponding source location

} AddressMapping;

// Symbol table with hash-based name lookup

typedef struct SymbolTable {

    Symbol **name_hash_table;       // Hash table for name-to-address lookup

    size_t hash_table_size;          // Hash table bucket count

    AddressMapping *address_mappings; // Sorted array for address-to-source lookup

    size_t mapping_count;            // Number of address mappings

}
```

```
    size_t mapping_capacity;           // Allocated mapping array size  
} SymbolTable;
```

Symbol Resolution Core Functions (`src/symbol_resolver.c`):

```
#include "symbol_resolver.h"
#include "elf_parser.h"
#include <stdlib.h>
#include <string.h>

// Initialize symbol table with specified hash table size

DebugResult symbols_create(SymbolTable **symbols, size_t hash_size) {

    *symbols = malloc(sizeof(SymbolTable));
    if (!*symbols) {
        return DEBUG_MEMORY_ACCESS_FAILED;
    }

    (*symbols)->hash_table_size = hash_size;
    (*symbols)->name_hash_table = calloc(hash_size, sizeof(Symbol*));
    (*symbols)->address_mappings = malloc(1000 * sizeof(AddressMapping));
    (*symbols)->mapping_count = 0;
    (*symbols)->mapping_capacity = 1000;

    if (!(*symbols)->name_hash_table || !(*symbols)->address_mappings) {
        return DEBUG_MEMORY_ACCESS_FAILED;
    }

    return DEBUG_SUCCESS;
}

// Simple hash function for symbol names

static uint32_t hash_symbol_name(const char *name, size_t table_size) {

    uint32_t hash = 5381;
    while (*name) {
        hash = ((hash << 5) + hash) + *name++;
    }
}
```

```
    return hash % table_size;

}

// Add symbol to name-to-address hash table

DebugResult symbols_add_symbol(SymbolTable *symbols, const char *name,
                               uintptr_t address, size_t size, int type) {

    // TODO: Implement symbol addition with hash table chaining

    // TODO: Handle duplicate symbols by preferring DWARF over ELF symbols

    // TODO: Store both original and demangled names for C++ symbols

    return DEBUG_SUCCESS;
}
```

Core Logic Skeleton Code

ELF Section Discovery (`src/elf_parser.c`):

```
// Parse ELF header and section headers to locate debug sections C

DebugResult elf_parse_headers(ElfFile *elf) {

    // TODO 1: Read and validate ELF header magic number and format

    // TODO 2: Determine file architecture (32/64-bit) and endianness

    // TODO 3: Read section header table from offset specified in ELF header

    // TODO 4: Read section name string table to resolve section names

    // TODO 5: Build map of section names to section header structures

    // Hint: Check e_ident[EI_MAG0-3] for ELF magic bytes 0x7F, 'E', 'L', 'F'

    // Hint: Use e_ident[EI_CLASS] to distinguish 32-bit vs 64-bit format

}

// Locate specific debug sections needed for symbol resolution

DebugResult elf_find_debug_sections(ElfFile *elf, Elf64_Shdr **debug_info,
                                    Elf64_Shdr **debug_line, Elf64_Shdr **debug_abbrev) {

    // TODO 1: Iterate through section headers looking for debug section names

    // TODO 2: Match section names ".debug_info", ".debug_line", ".debug_abbrev"

    // TODO 3: Store pointers to found sections in output parameters

    // TODO 4: Return error if critical sections (.debug_info) are missing

    // TODO 5: Handle case where some optional sections are not present

    // Hint: Use strcmp() to match section names from section name string table

}
```

DWARF Compilation Unit Parsing (`src/dwarf_parser.c`):

```

// Parse DWARF compilation unit header and abbreviation table

C

DebugResult dwarf_parse_compilation_unit(const uint8_t *debug_info_data, size_t offset,
                                         const uint8_t *debug_abbrev_data) {

    // TODO 1: Read compilation unit header (length, version, abbrev offset, address size)

    // TODO 2: Validate DWARF version is supported (2, 3, or 4)

    // TODO 3: Load abbreviation table from debug_abbrev section at specified offset

    // TODO 4: Parse abbreviation entries to build lookup table by abbreviation code

    // TODO 5: Begin DIE tree traversal starting with compilation unit root DIE

    // Hint: Compilation unit length includes version, abbrev offset, and address size fields

    // Hint: Abbreviation table entries end with abbreviation code 0

}

// Parse individual Debug Information Entry using abbreviation table

DebugResult dwarf_parse_die(const uint8_t *data, size_t *offset,
                           uint32_t abbrev_code, /* abbreviation table */) {

    // TODO 1: Look up abbreviation entry for this DIE's abbreviation code

    // TODO 2: Extract DIE tag (subprogram, variable, base_type, etc.)

    // TODO 3: Iterate through abbreviation's attribute list

    // TODO 4: For each attribute, read value based on attribute form encoding

    // TODO 5: Process important attributes (name, low_pc, high_pc, type references)

    // TODO 6: Recursively parse child DIEs if abbreviation indicates children present

    // Hint: Attribute forms determine value encoding (string, address, reference, etc.)

    // Hint: Child DIE list ends with abbreviation code 0

}

```

Symbol Resolution APIs (`src/symbol_resolver.c`):

```
// Load symbol information from ELF file with debug information

C

DebugResult symbols_load_from_elf(const char *executable_path, SymbolTable **symbols) {

    // TODO 1: Open ELF file and parse headers to locate debug sections

    // TODO 2: Parse traditional ELF symbol table (.symtab) for basic function names

    // TODO 3: Parse DWARF debug information for detailed symbol information

    // TODO 4: Build name-to-address hash table from extracted symbols

    // TODO 5: Build address-to-source mapping from DWARF line number program

    // TODO 6: Sort address mappings for efficient binary search during lookups

    // Hint: Prefer DWARF symbols over ELF symbols when both are available

    // Hint: Handle case where debug information is missing or corrupted gracefully

}

// Resolve function name to entry point address for breakpoint setting

DebugResult symbols_resolve_function(SymbolTable *symbols, const char *function_name,
                                    uintptr_t *address) {

    // TODO 1: Compute hash of function name for hash table lookup

    // TODO 2: Search hash table bucket for symbol with matching name

    // TODO 3: Handle hash collisions by traversing collision chain

    // TODO 4: Return symbol address if found, error if not found

    // TODO 5: Support lookup by both original and demangled C++ names

    // Hint: Use case-sensitive string comparison for exact matches

    // Hint: Consider implementing fuzzy matching for partial names

}

// Convert memory address to source file and line number

DebugResult symbols_resolve_address(SymbolTable *symbols, uintptr_t address,
                                    char **filename, uint32_t *line_number) {

    // TODO 1: Binary search address mapping array for entry containing target address

    // TODO 2: Find mapping entry where entry.address <= target < next_entry.address

    // TODO 3: Extract source filename and line number from found mapping

    // TODO 4: Handle case where address falls outside any known mapping
```

```
// TODO 5: Return allocated filename string that caller must free  
  
// Hint: Address mappings are sorted by address for efficient binary search  
  
// Hint: Last mapping entry extends to end of function or compilation unit  
  
}
```

Language-Specific Hints

File I/O and Binary Parsing:

- Use `fread()` with size validation to read ELF structures safely
- Check `fseek()` and `fseek()` return values to detect file corruption
- Use `mmap()` for large debug sections to avoid copying entire sections into memory
- Always validate that read operations don't exceed section boundaries

Memory Management:

- Use `valgrind` to detect symbol table memory leaks during development
- Implement reference counting for shared symbol structures to avoid double-free errors
- Consider using memory pools for DIE structures to reduce allocation overhead
- Free symbol name strings when destroying symbol table to prevent leaks

String Handling:

- Use `strdup()` to copy symbol names from temporary parsing buffers
- Implement string interning for frequently-repeated debug strings to save memory
- Handle both null-terminated strings and length-prefixed strings in DWARF data
- Validate string table indices to prevent buffer overflow attacks

Endianness Handling:

```

// Byte-swapping functions for cross-platform ELF parsing

C

uint16_t swap_uint16(uint16_t val) {

    return (val << 8) | (val >> 8);
}

uint32_t swap_uint32(uint32_t val) {

    return ((val << 24) & 0xff000000) |
           ((val << 16) & 0x00ff0000) |
           ((val >> 16) & 0x0000ff00) |
           ((val >> 24) & 0x000000ff);
}

// Use based on ELF header e_ident[EI_DATA] field

#define NEEDS_SWAP(elf) ((elf)->is_little_endian != is_host_little_endian())

```

Milestone Checkpoint

After implementing the Symbol Resolver component, verify correct functionality with these specific tests:

Test Command: `./debugger test_program` **Expected Behavior:**

1. Debugger loads and parses ELF/DWARF information without errors
2. Command `info functions` lists all available functions with addresses
3. Command `break main` successfully sets breakpoint at main function entry
4. When breakpoint hits, debugger displays correct source filename and line number

Verification Steps:

1. Compile test program: `gcc -g -o test_program test_program.c`
2. Run debugger: `./debugger test_program`
3. Set breakpoint: `break fibonacci` should respond with address like "Breakpoint set at 0x401234"
4. Start execution: `run` should stop at main with output like "Stopped at test_program.c:15"
5. Check symbol resolution: `info line` should display current source location

Signs of Problems:

- "No debug information found" indicates ELF section parsing failed
- Wrong source locations suggest DWARF line number program parsing errors
- "Symbol not found" for known functions indicates symbol table construction problems
- Segmentation faults during symbol lookup indicate hash table or memory management bugs

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
All symbol lookups fail	ELF sections not found	Use <code>readelf -S binary</code> to verify debug sections exist	Check section name string table parsing
Wrong function addresses	Endianness handling error	Compare addresses with <code>objdump -t binary</code>	Implement proper byte swapping for target architecture
Missing local functions	Only parsing .syms, not DWARF	Use <code>objdump -g binary</code> to verify DWARF info	Implement DWARF subprogram DIE parsing
Incorrect source locations	Line number program parsing error	Compare with <code>addr2line -e binary address</code>	Debug DWARF line number state machine execution
Memory corruption in symbol table	Hash table collision handling bug	Run under valgrind to detect buffer overflows	Fix hash table chaining and memory allocation
Slow symbol lookup performance	Linear search instead of hash lookup	Profile with gprof to identify bottlenecks	Verify hash table distribution and size

Variable Inspector Component

Milestone(s): Milestone 4 (Variable Inspection) — this section implements the variable inspection system that reads and displays variable values using debug information and process memory access, completing the debugger's core functionality

The X-Ray Vision Analogy

Think of the Variable Inspector as a medical X-ray machine combined with a skilled radiologist. When a doctor needs to examine a patient's internal structure, they use X-ray imaging to see through skin and muscle to reveal bones, organs, and foreign objects. The X-ray machine provides the raw imaging capability, while the radiologist interprets the images using their knowledge of anatomy to identify what each shadow and shape represents.

Similarly, the Variable Inspector combines raw memory access capabilities with sophisticated interpretation skills. The "X-ray machine" component uses `ptrace` to peer through the process boundary and read raw bytes from the debugged process's memory and registers. The "radiologist" component uses DWARF debug information as its "anatomy textbook" to interpret those raw bytes as meaningful variable values — understanding that these four bytes represent an integer, those eight bytes form a pointer, and this complex structure contains multiple fields at specific offsets.

Just as a radiologist must understand different imaging techniques (X-ray, CT scan, MRI) and anatomical variations, the Variable Inspector must handle different storage locations (registers, stack, heap) and type representations (integers, floats, pointers, structures). The debug information serves as the critical "medical knowledge" that transforms meaningless byte patterns into comprehensible variable values that developers can understand and analyze.

This analogy captures three essential aspects of variable inspection: the non-invasive observation capability (reading without modifying), the need for specialized interpretation knowledge (DWARF debug information), and the transformation from low-level raw data (memory bytes) into high-level meaningful information (typed variable values).

Variable Location Resolution

The Variable Inspector's first challenge is determining where a variable actually lives in the running process. Unlike source code where variables have simple names, compiled programs store variables in various locations depending on optimization level, scope, and usage patterns. The DWARF debug information contains **location expressions** — specialized bytecode programs that calculate where to find each variable at any given program counter.

DWARF location expressions use a stack-based virtual machine with operations that can reference CPU registers, perform arithmetic, and handle complex addressing scenarios. The most common location expression operations include direct register access (`DW_OP_reg0` through `DW_OP_reg31`), frame-relative addressing (`DW_OP_fbreg` with signed offset), and memory dereferencing (`DW_OP_deref`). More complex expressions might combine multiple operations, such as loading a base address from one register, adding an offset from debug information, and dereferencing the result to find the final variable location.

The Variable Inspector maintains a **location expression evaluator** that executes these DWARF bytecode programs to produce concrete memory addresses or register numbers. This evaluator operates as a simple stack machine, processing each operation in sequence while maintaining an evaluation stack for intermediate values. When evaluating `DW_OP_fbreg`, the evaluator must first determine the current frame base pointer, which might itself require consulting the call frame information from the `.debug_frame` section.

Architecture Decision: Location Expression Caching Strategy

Decision: Cache Evaluated Locations Per Stack Frame

- **Context:** Location expressions must be re-evaluated every time the program counter changes, as variable locations can change throughout function execution
- **Options Considered:**
 - No caching (re-evaluate every access)
 - Cache by program counter (one location per PC)
 - Cache by stack frame (one location per frame)
- **Decision:** Cache evaluated locations per stack frame with invalidation on frame changes
- **Rationale:** Most variable accesses occur while stopped at the same location, making frame-level caching effective while avoiding the complexity of fine-grained PC tracking
- **Consequences:** Reduces redundant location calculations but requires frame change detection to invalidate cached results

Location Expression Type	DWARF Operation	Evaluation Process	Example Use Case
Register Direct	DW_OP_reg0 to DW_OP_reg31	Return register number directly	Function parameters in registers
Frame Relative	DW_OP_fbreg + offset	Add offset to frame base pointer	Local variables on stack
Memory Address	DW_OP_addr + address	Use absolute memory address	Global variables
Register Indirect	DW_OP_breg0 + offset	Add offset to register content	Pointer dereferencing
Expression Stack	DW_OP_lit1 DW_OP_plus	Execute stack operations	Complex address calculations
Composite Location	DW_OP_piece	Combine multiple locations	Variables split across locations

The location resolution process follows a systematic evaluation sequence. First, the Variable Inspector retrieves the variable's DWARF information, including its type and location expression. Second, it obtains the current execution context, including the program counter and stack frame information. Third, it evaluates the location expression using the current context to produce a concrete location specification. Fourth, it validates that the computed location is accessible and within reasonable bounds. Finally, it caches the evaluated location for potential reuse within the same stack frame.

Complex variables may have **composite locations** where different parts of the variable reside in different places. For example, a 64-bit value might have its lower 32 bits in one register and upper 32 bits in another register, described using DW_OP_piece operations. The Variable Inspector must handle these multi-part locations by reading from multiple sources and reassembling the complete value according to the target architecture's endianness rules.

The critical insight for location resolution is that DWARF expressions are programs, not just addresses. They must be executed with the current machine state to produce meaningful results.

Memory and Register Access

Once the Variable Inspector determines where a variable resides, it must safely read that memory or register content from the debugged process. This access occurs across process boundaries using `ptrace` system calls, which provide controlled access to another process's address space and CPU state. The access patterns differ significantly between register-resident variables, stack-resident variables, and heap-resident variables, each requiring different `ptrace` operations and error handling strategies.

Register access uses the `PTRACE_GETREGS` operation to retrieve the complete CPU register set as a `struct user_regs_struct`. This structure contains all general-purpose registers, the instruction pointer, stack pointer, and various status flags. The Variable Inspector maps DWARF register numbers to the appropriate fields within this structure, handling architecture-specific register layouts and naming conventions. Reading registers is generally reliable since registers are always accessible when the process is stopped, but the Variable Inspector must account for register reuse and temporary values that might not represent meaningful variable content.

Memory access uses `PTRACE_PEEKDATA` to read word-sized chunks from the process's virtual address space. Since ptrace operates on word boundaries (typically 8 bytes on 64-bit systems), reading arbitrary variable sizes requires careful alignment handling. The Variable Inspector implements a **buffered memory reader** that fetches aligned words and extracts the requested bytes, handling cases where variables span word boundaries. This approach minimizes ptrace system calls while ensuring correct byte extraction regardless of variable size or alignment.

The memory access component must handle several challenging scenarios. **Invalid addresses** can occur when optimized code reuses stack space or when pointers contain garbage values. The Variable Inspector detects these situations by catching `EFAULT` errors from ptrace and reporting them as "address not accessible" rather than crashing.

Partially accessible regions can arise when variables span page boundaries where some pages are mapped but others are not, requiring byte-by-byte fallback reading when word-level access fails.

Memory Access Pattern	Ptrace Operation	Error Conditions	Recovery Strategy
Single Word Read	<code>PTRACE_PEEKDATA</code>	<code>EFAULT</code> (bad address)	Report address inaccessible
Multi-Word Variable	Multiple <code>PTRACE_PEEKDATA</code>	Partial failure	Read available bytes, mark rest invalid
Unaligned Access	Aligned reads + byte extraction	Alignment calculation error	Retry with byte-level access
Register Access	<code>PTRACE_GETREGS</code>	Process not stopped	Ensure process is in stopped state
String Reading	Iterative <code>PTRACE_PEEKDATA</code>	Infinite loop on bad pointer	Limit maximum string length

Architecture Decision: Memory Access Caching Strategy

Decision: Implement Write-Through Cache with Page-Level Granularity

- **Context:** Memory reads via ptrace are expensive system calls, but debugged process memory can change during execution
- **Options Considered:**
 - No caching (every access hits ptrace)
 - Write-through cache (cache reads, invalidate on process resume)
 - Write-back cache (cache reads and writes, sync periodically)
- **Decision:** Write-through cache with page-level granularity and invalidation on process state changes
- **Rationale:** Balances performance with correctness — caches expensive reads while ensuring fresh data after process execution
- **Consequences:** Significantly reduces ptrace overhead for repeated variable access while maintaining memory consistency

The Variable Inspector implements **safe memory access patterns** that gracefully handle common error conditions. For pointer dereferencing, it first validates that the pointer value falls within reasonable bounds (not NULL, not obviously invalid) before attempting to read the target memory. For string variables, it implements **bounded string reading** that

reads characters incrementally until finding a null terminator or reaching a maximum length limit, preventing infinite loops on corrupted string pointers.

Register mapping requires architecture-specific knowledge to translate DWARF register numbers into the correct fields of `struct user_regs_struct`. On x86-64, register 0 corresponds to RAX, register 1 to RDX, and so forth, but other architectures use different numbering schemes. The Variable Inspector maintains architecture-specific register mapping tables that translate abstract DWARF register numbers into concrete struct field offsets.

The key principle for memory and register access is defensive programming — assume that any address might be invalid and any register might contain garbage, then handle these cases gracefully rather than crashing.

Type-Aware Value Interpretation

Raw bytes from memory and registers become meaningful only when interpreted according to their intended data types. The Variable Inspector uses DWARF type information to transform byte sequences into properly formatted values, handling architecture-specific representation details like endianness, alignment, and floating-point formats. This type-aware interpretation converts meaningless hex dumps into readable variable displays that developers can understand and analyze.

Primitive type interpretation handles the fundamental data types supported by the target language and architecture. Integer types require endianness conversion when the target process uses different byte ordering than the debugger host. The Variable Inspector determines the target endianness from the ELF header and applies appropriate byte swapping when necessary. Floating-point values require IEEE 754 format interpretation, converting the raw bits into decimal representations while preserving precision and handling special values like infinity and NaN.

The Variable Inspector maintains a **type interpretation engine** that processes DWARF type information to determine how to format each variable. This engine handles type hierarchies, following type references (`DW_AT_type` attributes) to locate the ultimate base type definition. For `typedef` and `const`-qualified types, it unwraps the type layers to find the underlying representation type. For enumeration types, it translates numeric values back to their symbolic names when possible, providing more meaningful displays than raw integer values.

DWARF Base Type	Encoding	Size (bytes)	Interpretation Logic	Display Format
<code>DW_ATE_signed</code>	Two's complement	1, 2, 4, 8	Apply endianness conversion	Decimal with sign
<code>DW_ATE_unsigned</code>	Binary	1, 2, 4, 8	Apply endianness conversion	Decimal unsigned
<code>DW_ATE_float</code>	IEEE 754	4, 8	Convert to host float format	Decimal with precision
<code>DW_ATE_boolean</code>	Binary	1	Zero is false, nonzero is true	"true" or "false"
<code>DW_ATE_address</code>	Binary	4, 8	Interpret as pointer value	Hexadecimal with 0x prefix
<code>DW_ATE_signed_char</code>	ASCII/UTF-8	1	Handle printable and escape sequences	Character with escape notation

Pointer type interpretation involves two levels of meaning: the pointer value itself and the data it references. The Variable Inspector displays both the pointer's address value and, when possible, the content it points to. For string pointers (`char*`), it automatically dereferences and reads the null-terminated string content. For structure pointers, it can optionally dereference and display the referenced structure's fields. The dereference operation includes safety checks to prevent crashes on NULL or invalid pointers.

Enumeration value translation enhances readability by converting numeric enumeration values back to their symbolic names. The Variable Inspector extracts enumeration member definitions from DWARF information, building lookup tables that map numeric values to their corresponding identifiers. When displaying an enumeration variable, it first attempts symbolic lookup and falls back to numeric display for values not defined in the enumeration.

Architecture Decision: Type Information Caching

Decision: Cache Parsed Type Definitions with Lazy Loading

- **Context:** DWARF type information is complex to parse but relatively stable, and the same types are referenced repeatedly
- **Options Considered:**
 - Parse types on every access (simple but slow)
 - Eager loading of all types (fast access but high memory usage)
 - Lazy loading with persistent cache (balanced approach)
- **Decision:** Lazy loading with persistent cache, indexed by DWARF type offset
- **Rationale:** Most debugging sessions only examine a subset of available types, making lazy loading more efficient than eager loading
- **Consequences:** First access to each type incurs parsing overhead, but subsequent accesses are fast and memory usage remains proportional to actual usage

The type interpretation process handles **bit fields** within structures, which require special parsing to extract the correct bits from the containing word. DWARF information specifies bit field offset and size, allowing the Variable Inspector to mask and shift the appropriate bits from the containing integer. This bit-level extraction ensures accurate display of packed structures and hardware register representations.

Character and string handling accommodates different character encodings and null-termination conventions. The Variable Inspector attempts to detect printable ASCII characters and displays them directly, while using escape sequence notation for control characters and non-printable bytes. For wide character strings, it handles multi-byte encodings according to the target system's locale settings, though this can be complex in cross-platform debugging scenarios.

The essence of type-aware interpretation is using metadata (DWARF type information) to impose structure and meaning on raw bytes, transforming incomprehensible hex dumps into readable variable displays.

Composite Type Handling

Complex data structures like arrays, structures, and unions require sophisticated handling beyond simple type interpretation. The Variable Inspector must navigate these composite types by understanding their internal layout, field relationships, and memory organization. This navigation capability enables developers to inspect not just primitive values but also complex data structures that form the backbone of most programs.

Structure member access involves parsing DWARF structure definitions to locate individual fields within the composite type. Each structure member has a name, type, and byte offset within the containing structure. The Variable Inspector builds **field lookup tables** from DWARF information, enabling efficient field access by name or sequential enumeration of all fields. When displaying a structure, it reads the memory at each field's offset and applies the appropriate type interpretation for that specific field.

Structure handling must account for **memory alignment and padding** inserted by the compiler to satisfy architectural alignment requirements. DWARF information includes the actual byte offsets used by the compiler, which may differ from naive field size summation due to inserted padding. The Variable Inspector relies on these DWARF-specified offsets rather than attempting to recalculate alignment, ensuring compatibility with the compiler's actual memory layout decisions.

Array traversal requires understanding the array's element type, element count, and stride (bytes between consecutive elements). DWARF array type definitions specify these parameters, allowing the Variable Inspector to calculate the memory address of any array element using base address plus index times stride. For multi-dimensional arrays, it handles nested indexing by processing each dimension's bounds and stride calculations in the correct order.

Composite Type	DWARF Representation	Navigation Method	Display Strategy
Structure	<code>DW_TAG_structure_type</code> with members	Field offset table lookup	Show all fields with names and values
Union	<code>DW_TAG_union_type</code> with members	Same memory location, multiple interpretations	Show all possible interpretations
Array	<code>DW_TAG_array_type</code> with bounds	Base address + index * stride	Show elements with index annotations
Pointer	<code>DW_TAG_pointer_type</code>	Dereference to target type	Show address and dereferenced value
Function Pointer	<code>DW_TAG_pointer_type</code> to <code>DW_TAG_subroutine_type</code>	Symbol table lookup for name	Show address and function name if known

Union handling presents a unique challenge since the same memory location can be interpreted as different types depending on the intended usage. The Variable Inspector displays all possible union member interpretations simultaneously, allowing developers to choose the most appropriate view for their current debugging context. This multi-interpretation approach helps when debugging polymorphic data structures or low-level code that reinterprets memory content.

Dynamic array and pointer traversal enables inspection of data structures whose size is determined at runtime. For pointer-based linked lists, the Variable Inspector can follow pointer chains, displaying each node's contents while detecting cycles to prevent infinite traversal. For dynamic arrays allocated with `malloc`, it relies on programmer-provided size information or heuristics to determine reasonable bounds for element display.

Architecture Decision: Composite Type Display Depth

Decision: Implement Configurable Recursion Depth with Smart Truncation

- **Context:** Composite types can contain deeply nested structures or large arrays that overwhelm the display and consume excessive memory
- **Options Considered:**
 - Fixed depth limit (simple but inflexible)
 - Unlimited recursion (thorough but potentially overwhelming)
 - Configurable depth with smart truncation (balanced approach)
- **Decision:** Configurable maximum depth with intelligent truncation based on type complexity and size
- **Rationale:** Allows detailed inspection when needed while preventing display overflow and excessive memory usage in common cases
- **Consequences:** Requires more complex display logic but provides better user experience for both simple and complex debugging scenarios

The Variable Inspector implements **circular reference detection** to prevent infinite recursion when traversing self-referential data structures like linked lists with back pointers or tree structures with parent links. It maintains a **traversal stack** that tracks currently active memory addresses, detecting when the same address appears twice in the current traversal path. When a circular reference is detected, it displays a reference indicator rather than recursing infinitely.

Flexible member access patterns support both named field access ("show me variable.field") and exploratory browsing ("show me all fields of this structure"). For named access, the Variable Inspector performs hash table lookup in its field tables for O(1) access time. For exploratory browsing, it iterates through all structure members in declaration order, providing a comprehensive view of the data structure's current state.

The key insight for composite type handling is that complex data structures are built from simple primitives using well-defined layout rules, so understanding those rules enables systematic navigation of arbitrarily complex types.

Architecture Decisions

The Variable Inspector's architecture reflects several critical design decisions that balance functionality, performance, and maintainability. These decisions establish the component's approach to type system representation, value formatting strategies, and optimized variable handling, creating a foundation that supports both basic variable inspection and advanced debugging scenarios.

Architecture Decision: Type System Representation Strategy

Decision: Unified Type Descriptor with Lazy Attribute Resolution

- **Context:** DWARF type information is complex and hierarchical, with types referencing other types through various attribute relationships
- **Options Considered:**
 - Direct DWARF structure mirroring (preserves all information but complex to use)
 - Flattened type system (easier to use but loses important relationships)
 - Unified type descriptor with lazy resolution (balanced approach)
- **Decision:** Implement unified `TypeDescriptor` structure with lazy attribute resolution and reference following
- **Rationale:** Provides clean programming interface while preserving full DWARF semantics and enabling efficient memory usage through lazy loading
- **Consequences:** Requires sophisticated reference resolution logic but enables both simple and complex type operations through consistent interface

The type system representation centers around a **unified type descriptor** that abstracts DWARF complexity while preserving semantic information necessary for accurate variable interpretation. This descriptor includes the essential type properties (size, alignment, encoding) along with lazy-loaded attributes that provide detailed information only when required. The lazy loading mechanism reduces memory consumption by deferring expensive DWARF parsing operations until specific type details are actually needed.

Type System Component	Purpose	Implementation Strategy	Memory Impact
Base Type Cache	Store primitive type definitions	Hash table indexed by DWARF offset	Low - fixed set of base types
Composite Type Registry	Track structure/array definitions	Lazy-loaded tree with reference counting	Medium - grows with usage
Type Attribute Resolver	Follow DWARF type references	On-demand parsing with caching	Low - cached results after first access
Field Layout Calculator	Compute member offsets	Pre-computed tables for active types	Medium - proportional to accessed types

Architecture Decision: Value Formatting Strategy

Decision: Pluggable Formatter Architecture with Default Implementations

- **Context:** Different debugging contexts require different value display formats (hex vs decimal, full precision vs rounded, raw bytes vs interpreted)
- **Options Considered:**
 - Fixed formatting rules (simple but inflexible)
 - Context-sensitive formatting (flexible but complex)
 - Pluggable formatter architecture (most flexible)
- **Decision:** Implement pluggable value formatters with sensible defaults and context-aware selection
- **Rationale:** Enables customization for different debugging scenarios while providing good defaults for common cases
- **Consequences:** Requires formatter registry and selection logic but enables extensibility for specialized debugging needs

The value formatting architecture employs **pluggable formatters** that can be selected based on variable type, debugging context, and user preferences. Each formatter implements a common interface but applies different presentation logic — for example, displaying integers as decimal, hexadecimal, or binary representations. The formatter selection process considers the variable's type, its role in the program, and the current debugging context to choose the most appropriate display format automatically.

Architecture Decision: Optimized Variable Handling

Decision: Multi-Tier Caching with Invalidation Strategies

- **Context:** Variable inspection involves expensive operations (ptrace calls, DWARF parsing, type resolution) that benefit from caching, but cached data can become stale
- **Options Considered:**
 - No caching (simple but slow)
 - Simple time-based cache expiration (fast but potentially stale)
 - Event-driven cache invalidation (optimal but complex)
- **Decision:** Multi-tier caching with different invalidation strategies per tier
- **Rationale:** Balances performance optimization with data freshness by using appropriate invalidation strategies for different types of cached data
- **Consequences:** Requires sophisticated cache management but provides optimal performance while maintaining correctness

The multi-tier caching system employs different caching strategies for different types of data based on their volatility and access patterns. **Type information cache** uses persistent caching since DWARF type definitions never change during a debugging session. **Location cache** uses frame-based invalidation since variable locations typically remain stable within a stack frame but change across function calls. **Value cache** uses execution-based invalidation since variable values can change whenever the debugged process executes.

Cache Tier	Data Type	Invalidation Trigger	Access Pattern	Performance Impact
Type Information	DWARF type definitions	Never (persistent)	Read-heavy, shared	High - eliminates repeated DWARF parsing
Location Resolution	Variable location expressions	Stack frame changes	Medium frequency	Medium - reduces location calculations
Memory Content	Raw memory bytes	Process execution	High frequency	High - eliminates repeated ptrace calls
Formatted Values	Display-ready strings	Value or format changes	High frequency	Low - fast to regenerate

Architecture Decision: Error Propagation and Recovery

Decision: Hierarchical Error Handling with Graceful Degradation

- **Context:** Variable inspection can fail at multiple levels (ptrace failures, invalid addresses, corrupted debug information) but partial success is often possible
- **Options Considered:**
 - Fail-fast approach (simple but loses partial information)
 - Best-effort with silent failures (preserves information but hides problems)
 - Hierarchical error handling with explicit partial results (complex but most informative)
- **Decision:** Implement hierarchical error handling that reports partial success and specific failure reasons
- **Rationale:** Maximizes information available to developers while clearly indicating what information might be unreliable or incomplete
- **Consequences:** Requires sophisticated error classification and partial result handling but provides superior debugging experience

The error handling architecture distinguishes between **recoverable errors** (where partial information is still useful) and **fatal errors** (where no meaningful information can be extracted). For recoverable errors, the Variable Inspector continues processing other structure fields or array elements while marking the failed components as inaccessible. This approach ensures that developers can still inspect the accessible portions of complex data structures even when some members cannot be read due to optimization or memory access restrictions.

The architectural principle guiding these decisions is progressive enhancement — provide basic functionality reliably while offering advanced features that gracefully degrade when underlying systems cannot support them.

Common Pitfalls

The Variable Inspector component presents several challenging implementation scenarios that frequently trip up developers building debuggers. These pitfalls arise from the complex interaction between compiled code optimization, debug information limitations, and the inherent difficulties of cross-process memory access.

⚠️ Pitfall: Assuming Variables Always Have Simple Locations

Many developers initially assume that variables have fixed memory addresses or register assignments throughout their scope, leading to naive location resolution that fails with optimized code. In reality, compilers frequently move variables between locations, store them in different places during different parts of their lifetime, or eliminate them entirely through optimization. DWARF location expressions can specify different locations for different program counter ranges, and some locations may only be valid for single instructions.

The problem manifests when the debugger displays stale or incorrect values because it cached a location that is no longer valid. For example, a local variable might be stored in a register at the function entry but moved to the stack after a function call that needs to use that register for parameter passing. The debugger must re-evaluate location expressions whenever the program counter changes significantly.

Fix: Implement location expression evaluation that considers the current program counter when resolving variable locations. Maintain location validity ranges and re-evaluate locations when the program counter moves outside the previously valid range. Cache location results only within their valid program counter ranges.

Pitfall: Incorrect Endianness Handling in Cross-Architecture Debugging

When the debugger runs on a different architecture than the debugged process, or when examining core dumps from different systems, endianness mismatches can cause incorrect value interpretation. This is particularly problematic for multi-byte values like integers, pointers, and floating-point numbers, where byte order affects the interpreted value significantly.

The issue often goes unnoticed during testing because most development occurs on the same architecture as the target system. Problems surface when debugging embedded systems, cross-compiled code, or core dumps from different architectures. Symptoms include wildly incorrect integer values, invalid-looking pointer addresses, and floating-point values that appear as garbage.

Fix: Always check the target process's endianness from the ELF header and apply appropriate byte swapping when the debugger host and target process use different endianness. Implement endianness-aware read functions that automatically convert multi-byte values from target byte order to host byte order.

Pitfall: Following Invalid Pointers Without Safety Checks

Optimized code often leaves garbage values in pointer variables when the variable is not currently in use, or optimization might reuse memory locations for different purposes at different times. Blindly dereferencing these pointer values can cause the debugger to attempt reading from invalid memory addresses, resulting in ptrace failures or, worse, reading random memory that happens to be mapped.

This problem is particularly acute with optimized builds where register allocation is aggressive and memory locations are reused frequently. A pointer variable that contained a valid address earlier in its scope might contain an arbitrary bit pattern when examined at a different program location.

Fix: Implement pointer validation that checks for obviously invalid values (NULL, addresses in kernel space, addresses outside the process's memory map) before attempting dereferencing operations. Provide user control over automatic pointer dereferencing versus displaying just the pointer value. Use safe memory access patterns that handle ptrace failures gracefully.

Pitfall: Ignoring DWARF Location Expression Complexity

Simple location expressions like "variable is at stack offset -16" are straightforward to handle, but DWARF supports complex expressions that can perform arithmetic, conditionally select different locations, and even describe variables that

exist only in pieces across multiple registers. Developers often implement only the simple cases, leading to incorrect or missing variable information for optimized code.

Complex location expressions become common with higher optimization levels, where compilers use sophisticated register allocation and may keep different parts of a single variable in different locations. For example, a 64-bit integer might have its lower 32 bits in one register and upper 32 bits in another register, described using `DW_OP_piece` operations.

Fix: Implement a complete DWARF location expression evaluator that handles the full range of operations defined in the DWARF specification. For operations not yet implemented, provide clear error messages indicating that the variable location is too complex to evaluate, rather than displaying incorrect information.

Pitfall: Inadequate Memory Access Error Handling

Cross-process memory access through ptrace can fail for numerous reasons: the target address might be unmapped, the process might not be stopped, the address might be in a protected memory region, or the process might have exited. Developers often handle only the common success case, leading to debugger crashes or hangs when memory access fails.

The problem is exacerbated because memory access patterns that work perfectly on simple test programs can fail in complex real-world scenarios with dynamic memory allocation, memory mapping, and protection changes. Error handling code paths are often under-tested because they require specific system conditions to trigger.

Fix: Implement comprehensive error handling for all ptrace operations, with specific handling for different failure modes. Provide meaningful error messages that help users understand why variable access failed. Implement timeout mechanisms for ptrace operations that might hang if the target process is in an unexpected state.

Pitfall: Inefficient Type Information Processing

DWARF type information forms complex graphs with type references, inheritance relationships, and circular dependencies. Naive implementations might re-parse the same type information repeatedly or fail to handle circular type references properly. This leads to poor performance or infinite loops when processing complex type hierarchies.

The performance problems become apparent when debugging programs with large numbers of types, such as C++ programs with template instantiations or programs that include large system headers. The debugger might become unresponsive when attempting to display complex structures or perform type lookups.

Fix: Implement proper type caching with reference counting and circular reference detection. Use lazy loading to defer expensive type processing until actually needed. Build type resolution caches that persist across multiple variable inspection operations within the same debugging session.

Error Condition	Symptom	Root Cause	Detection Method	Recovery Strategy
Invalid location expression	Wrong variable values	Cached stale location	PC range validation	Re-evaluate location with current PC
Endianness mismatch	Garbage integer values	Cross-architecture debugging	ELF header check	Apply byte swapping during reads
Invalid pointer dereference	Ptrace EFAULT errors	Optimized/reused memory	Address validation	Display pointer value without dereferencing
Complex DWARF operations	Missing variable info	Incomplete expression evaluator	Unsupported operation codes	Report "location too complex" error
Memory access failure	Debugger crash/hang	Process state assumptions	Ptrace return value checking	Graceful error reporting with context
Type processing loops	Debugger freeze	Circular type references	Traversal cycle detection	Break cycles with reference indicators

The fundamental principle for avoiding Variable Inspector pitfalls is defensive programming — assume that every memory access might fail, every type might be corrupted, and every optimization might change expected behavior patterns.

Implementation Guidance

The Variable Inspector component requires careful coordination between low-level memory access, complex data structure interpretation, and user-friendly value formatting. This implementation guidance provides practical starting points for building each aspect of the variable inspection system while avoiding common implementation mistakes.

Technology Recommendations

Component	Simple Option	Advanced Option
DWARF Parsing	Custom parser for basic DIES	libdw or elfutils integration
Type System	Flat type lookup table	Hierarchical type graph with lazy loading
Memory Access	Direct ptrace wrapper	Buffered memory reader with caching
Value Formatting	Printf-style formatters	Pluggable formatter registry
Location Evaluation	Simple offset calculations	Full DWARF expression evaluator
Error Handling	Basic error codes	Structured error context with recovery hints

Recommended File Structure

```
debugger/
├── src/
│   ├── variable_inspector/
│   │   ├── inspector.c           ← main Variable Inspector interface
│   │   ├── inspector.h          ← public API definitions
│   │   ├── location_resolver.c    ← DWARF location expression evaluator
│   │   ├── location_resolver.h    ← location resolution interface
│   │   ├── memory_reader.c       ← ptrace memory access wrapper
│   │   ├── memory_reader.h       ← memory access interface
│   │   ├── type_interpreter.c    ← type-aware value formatting
│   │   ├── type_interpreter.h    ← type interpretation interface
│   │   ├── value_formatter.c     ← display formatting logic
│   │   └── value_formatter.h     ← formatter interface definitions
│   ├── dwarf_parser/           ← from Symbol Resolver component
│   ├── process_control/        ← from Process Controller component
│   └── common/
│       ├── debug_error.h        ← error handling definitions
│       └── platform.h          ← architecture-specific definitions
└── tests/
    ├── variable_inspector_test.c  ← unit tests for inspector logic
    ├── test_programs/
    │   ├── simple_types.c         ← compiled test programs with debug info
    │   ├── complex_structs.c      ← basic variable types test
    │   └── optimized_code.c       ← composite type test
    └── test_data/                ← optimization handling test
                                    ← expected output and debug info samples
```

Infrastructure Starter Code

Memory Reader with Caching


```

// Check if request is in cache

if (reader->cache.is_valid &&

    address >= reader->cache.base_address &&

    address + size <= reader->cache.base_address + reader->cache.valid_bytes) {

    size_t offset = address - reader->cache.base_address;

    memcpy(buffer, reader->cache.data + offset, size);

    reader->cache_hits++;

    return DEBUG_SUCCESS;

}

// Cache miss - read from ptrace

reader->cache_misses++;

reader->cache.base_address = address & ~0xFFF; // Align to page boundary

reader->cache.valid_bytes = 0;

reader->cache.is_valid = false;

// Read word-aligned data using ptrace

for (size_t offset = 0; offset < 4096; offset += sizeof(long)) {

    errno = 0;

    long word = ptrace(PTRACE_PEEKDATA, reader->target_pid,

                       reader->cache.base_address + offset, 0);

    if (word == -1 && errno != 0) {

        if (offset == 0) {

            return DEBUG_MEMORY_ACCESS_FAILED; // Can't read any data

        }

        break; // Partial page read

    }

    memcpy(reader->cache.data + offset, &word, sizeof(long));

    reader->cache.valid_bytes += sizeof(long);

}

```

```
}

reader->cache.is_valid = true;

// Extract requested bytes from cache

if (address >= reader->cache.base_address &&
    address + size <= reader->cache.base_address + reader->cache.valid_bytes) {

    size_t offset = address - reader->cache.base_address;
    memcpy(buffer, reader->cache.data + offset, size);

    return DEBUG_SUCCESS;
}

return DEBUG_MEMORY_ACCESS_FAILED;
}

void memory_reader_invalidate_cache(MemoryReader* reader) {

    reader->cache.is_valid = false;
}

void memory_reader_destroy(MemoryReader* reader) {

    if (reader) {
        free(reader);
    }
}
```

Register Access Helper

```
#include "register_reader.h"
#include <sys/user.h>
#include <sys/ptrace.h>

// Complete implementation for register access

DebugResult register_reader_get_all(pid_t pid, struct user_regs_struct* regs) {

    if (ptrace(PTRACE_GETREGS, pid, 0, regs) == -1) {
        return DEBUG_PTRACE_FAILED;
    }

    return DEBUG_SUCCESS;
}

DebugResult register_reader_get_by_dwarf_number(pid_t pid, int dwarf_reg_num,
                                                uint64_t* value) {

    struct user_regs_struct regs;

    DebugResult result = register_reader_get_all(pid, &regs);

    if (result != DEBUG_SUCCESS) {
        return result;
    }

    // x86-64 DWARF register number mapping

    switch (dwarf_reg_num) {
        case 0: *value = regs.rax; break;
        case 1: *value = regs.rdx; break;
        case 2: *value = regs.rcx; break;
        case 3: *value = regs.rbx; break;
        case 4: *value = regs.rsi; break;
        case 5: *value = regs.rdi; break;
        case 6: *value = regs.rbp; break;
        case 7: *value = regs.rsp; break;
        case 8: *value = regs.r8; break;
    }
}
```

```
        case 9: *value = regs.r9; break;
        case 10: *value = regs.r10; break;
        case 11: *value = regs.r11; break;
        case 12: *value = regs.r12; break;
        case 13: *value = regs.r13; break;
        case 14: *value = regs.r14; break;
        case 15: *value = regs.r15; break;
        case 16: *value = regs.rip; break;
    default:
        return DEBUG_INVALID_ADDRESS; // Unsupported register
    }

    return DEBUG_SUCCESS;
}
```

Core Logic Skeleton

Variable Inspector Main Interface

```
// variable_inspect_by_name - Locate and display a variable by name at current PC

// Returns formatted string representation of variable value

DebugResult variable_inspect_by_name(DebuggedProcess* proc, const char* var_name,
                                      char* output_buffer, size_t buffer_size) {

    // TODO 1: Look up variable in current scope using symbol table
    //         Use symbols_find_variable(proc->symbols, var_name, proc->regs.rip)
    //         Handle case where variable name is not found in current scope

    // TODO 2: Get variable's DWARF type information and location expression
    //         Extract DW_AT_type attribute to determine variable type
    //         Extract DW_AT_location attribute for location expression

    // TODO 3: Evaluate location expression to determine where variable is stored
    //         Call location_resolver_evaluate(location_expr, &proc->regs, &location)
    //         Handle complex expressions involving multiple locations or pieces

    // TODO 4: Read variable value from the determined location
    //         If location is register: use register_reader_get_by_dwarf_number()
    //         If location is memory: use memory_reader_read_bytes()
    //         Handle cases where location is not accessible

    // TODO 5: Interpret raw bytes according to variable's type information
    //         Call type_interpreter_format_value(type_info, raw_bytes, formatted_output)
    //         Apply appropriate endianness conversion if needed

    // TODO 6: Format the result for display and copy to output buffer
    //         Include variable name, type, and formatted value
    //         Handle buffer size limits and truncation if necessary
}
```

C

Location Expression Evaluator

```
// location_resolver_evaluate - Execute DWARF location expression to find variable
// Implements stack-based evaluation of DWARF bytecode operations

DebugResult location_resolver_evaluate(const uint8_t* expr_bytes, size_t expr_length,
                                       struct user_regs_struct* regs,
                                       VariableLocation* result) {

    // TODO 1: Initialize evaluation stack for DWARF expression operations

    // Create stack with reasonable depth limit (e.g., 32 entries)

    // Initialize program counter to start of expression bytes

    // TODO 2: Loop through expression bytes and decode each operation

    // Read operation code byte and advance program counter

    // Switch on operation code to determine required action

    // TODO 3: Implement stack operations (DW_OP_lit*, DW_OP_const*, DW_OP_dup, etc.)

    // Push literal values onto evaluation stack

    // Handle signed and unsigned constant operations

    // Implement stack manipulation operations

    // TODO 4: Implement register operations (DW_OP_reg*, DW_OP_breg*, DW_OP_fbreg)

    // For DW_OP_reg*: set result type to register and store register number

    // For DW_OP_breg*: read register value, add offset, push to stack

    // For DW_OP_fbreg: get frame base pointer, add offset, push to stack

    // TODO 5: Implement arithmetic and memory operations

    // Handle DW_OP_plus, DW_OP_minus, DW_OP_deref operations

    // Pop operands from stack, perform operation, push result

    // For deref: treat top of stack as address and read memory at that address

    // TODO 6: Handle piece operations (DW_OP_piece, DW_OP_bit_piece) for composite locations

    // Track multiple location pieces when variable spans registers/memory
```

```
//      Set result type to composite and store piece information

// TODO 7: Set final result based on evaluation state

//      If stack has one value and no pieces: location is computed address

//      If register operation was last: location is register

//      If pieces were used: location is composite with piece list

}
```

Type-Aware Value Formatter

```
// type_interpreter_format_value - Convert raw bytes to readable string using type info
// Handles endianness, signedness, and composite type navigation

DebugResult type_interpreter_format_value(const TypeInfo* type, const uint8_t* raw_bytes,
                                         size_t byte_count, char* output, size_t output_size) {

    // TODO 1: Determine the base type from DWARF type information
    //           Follow DW_AT_type references until reaching base type
    //           Handle typedef, const, and pointer type unwrapping

    // TODO 2: Check for endianness requirements and convert if necessary
    //           Compare target endianness from ELF header with host endianness
    //           Apply byte swapping for multi-byte values if endianness differs

    // TODO 3: Format primitive types according to their encoding
    //           For DW_ATE_signed: interpret as signed integer with appropriate size
    //           For DW_ATE_unsigned: interpret as unsigned integer
    //           For DW_ATE_float: interpret as IEEE 754 floating point
    //           For DW_ATE_boolean: display as "true" or "false"

    // TODO 4: Handle pointer types with dereferencing option
    //           Display pointer value in hexadecimal format
    //           Optionally attempt to dereference and show pointed-to value
    //           Include safety checks for NULL and invalid pointers

    // TODO 5: Process composite types (structures, arrays, unions)
    //           For structures: iterate through members and format each field
    //           For arrays: format elements with index indicators
    //           For unions: show all possible interpretations

    // TODO 6: Handle special cases and error conditions
    //           Truncate output if it exceeds buffer size
```

```
//           Mark values as "<optimized out>" if location is not accessible  
  
//           Handle partially readable composite types gracefully  
  
}
```

Language-Specific Hints

- Use `ptrace(PTRACE_PEEKDATA, pid, addr, 0)` for reading process memory word by word
- Check `errno` after ptrace calls to distinguish between legitimate zero values and errors
- Handle word-boundary alignment by reading aligned words and extracting required bytes
- Use `struct user_regs_struct` from `<sys/user.h>` for register access on x86-64
- Implement endianness detection using ELF header `e_ident[EI_DATA]` field
- Cache parsed DWARF type information using hash tables indexed by type offset
- Use `snprintf` with size limits for safe string formatting to prevent buffer overflows
- Implement circular reference detection using address sets during structure traversal

Milestone Checkpoint

After implementing the Variable Inspector component, verify correct functionality:

Test Command: Compile test program with debug information and inspect variables

```
gcc -g -O0 test_programs/simple_types.c -o test_simple  
  
../debugger test_simple  
  
(debugger) break main  
  
(debugger) run  
  
(debugger) print local_int  
  
(debugger) print local_struct.field1
```

BASH

Expected Output:

```
Breakpoint hit at main+0 (simple_types.c:15)  
(debugger) print local_int  
local_int: (int) 42  
(debugger) print local_struct.field1  
local_struct.field1: (char*) "hello world"
```

Verification Steps:

1. Variables display correct values with appropriate type information
2. Structure members are accessible by name with proper field values
3. Pointer variables show both address and dereferenced content when appropriate
4. Error messages appear for optimized-out or inaccessible variables
5. Complex types (arrays, nested structures) display in readable hierarchical format

Common Issues and Diagnosis:

- "Variable not found" → Check DWARF parsing and symbol table building
- Wrong values displayed → Verify endianness handling and location resolution
- Segmentation fault → Add bounds checking to memory access operations
- "Optimized out" for simple variables → Test with `-O0` compilation first

Interactions and Data Flow

Milestone(s): All milestones — this section describes how the four components work together during typical debugging operations, showing the complete interaction patterns from Milestone 1 through Milestone 4

The Orchestra Conductor Analogy

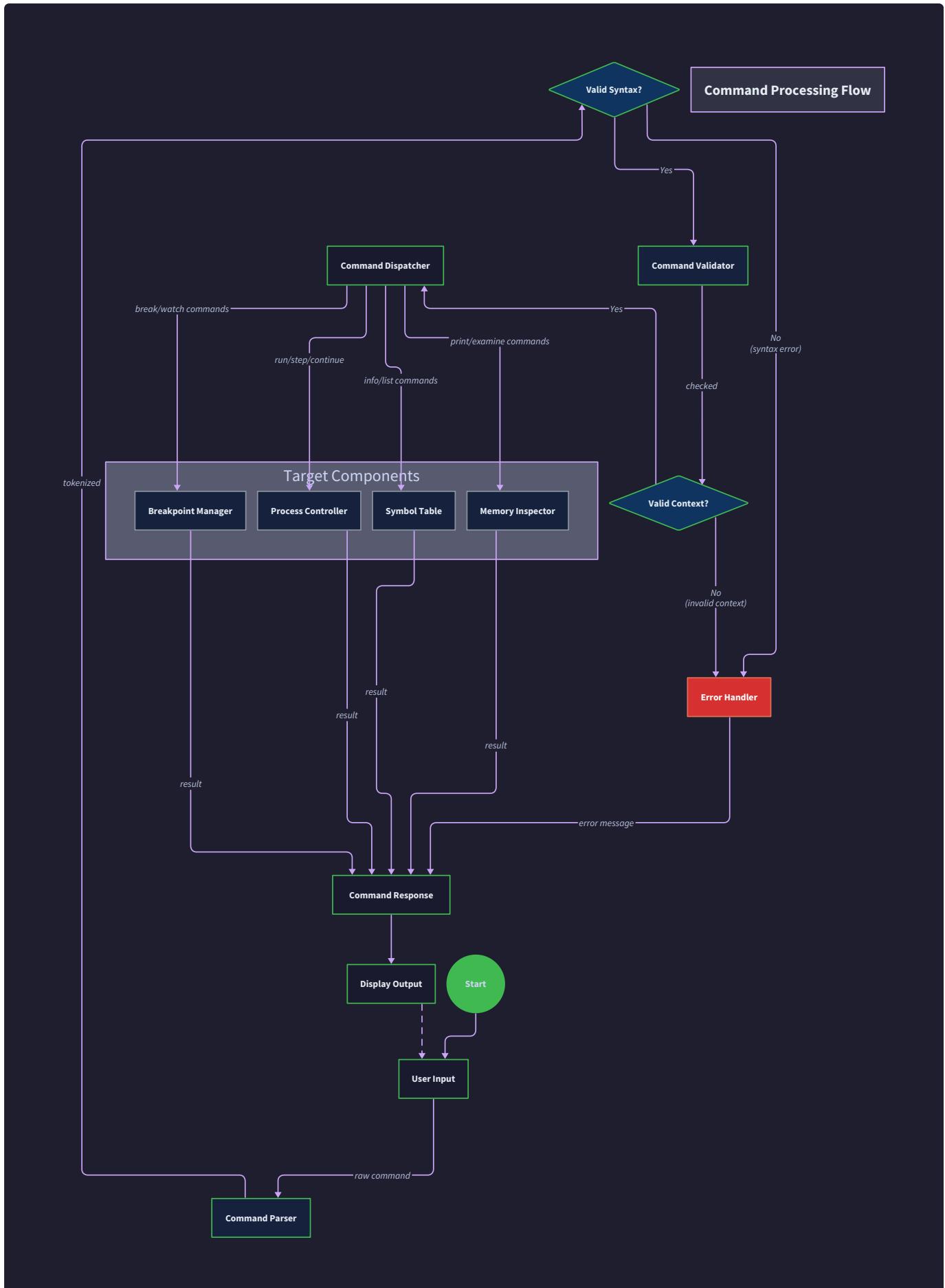
Think of debugger interactions like a symphony orchestra performance. The user commands are like the conductor's baton movements — they trigger coordinated responses from different instrument sections (components). The **Process Controller** is like the rhythm section, providing the fundamental timing and control. The **Breakpoint Manager** acts like the percussion section, creating precise interruptions at exactly the right moments. The **Symbol Resolver** functions as the string section, providing rich harmonic information that gives context and meaning. The **Variable Inspector** serves as the wind section, adding detailed expression and nuance to the performance.

Just as musicians must listen to each other and respond to the conductor's cues in precise timing, our debugger components must coordinate their actions through well-defined interfaces and data flows. A single user command like "step into function" triggers a cascade of interactions: symbol lookup to find the function entry point, breakpoint setting to stop at the destination, process control to begin execution, and finally variable inspection to show the new context. Each component has its specialized role, but the magic happens in their coordination.

The conductor (command processor) doesn't play any instruments directly — instead, it understands the musical score (user intent) and translates it into specific cues that each section can understand and execute. Similarly, our command processing flow doesn't perform low-level operations itself, but orchestrates the components that do.

Command Processing Flow

The **command processing flow** represents the primary interface between user intent and system action. When a user types a command like `break main` or `step`, that text string must be parsed, validated, and translated into a series of component operations that achieve the desired debugging behavior.



The flow begins with **command parsing** in the main debugger loop. Raw input text gets tokenized into command name and arguments, then matched against a registry of available commands. Each registered command has a handler function that knows how to interpret its specific argument format and delegate to appropriate components.

Design Insight: The command registry pattern allows for clean separation between user interface concerns and core debugging logic. New commands can be added without modifying the parsing infrastructure, and components can register their own command handlers without tight coupling to the main loop.

Architecture Decision: Command Dispatch Strategy

Decision: Function Pointer Registry vs Switch Statement

- **Context:** Need to route parsed commands to appropriate handlers while maintaining extensibility
- **Options Considered:** Giant switch statement, function pointer registry, object-oriented dispatch
- **Decision:** Function pointer registry with `Command` structure containing handler function
- **Rationale:** Provides O(1) lookup, allows components to register their own commands, avoids massive switch statement that would require modification for every new command
- **Consequences:** Enables modular command registration but requires careful memory management for function pointers

Command Flow Stage	Input	Processing	Output	Error Handling
Input Parsing	Raw string	Tokenize, extract command and args	<code>Command</code> struct with handler	Syntax error message
Command Lookup	Command name	Hash table lookup in registry	Function pointer to handler	"Unknown command" error
Permission Check	Current process state	Verify command valid for state	Boolean permission	"Process not stopped" error
Handler Execution	Parsed arguments	Call component-specific logic	<code>DebugResult</code> status	Component-specific error
Result Display	Handler return value	Format for user display	Formatted output string	Error message formatting

The **command validation** step occurs after lookup but before execution. Many commands require the debugged process to be in a specific state — for example, `step` only works when the process is stopped, while `attach` only works when no process is currently attached. This validation prevents invalid operations and provides clear error messages.

Handler execution represents the core delegation to components. Each command handler function receives a pointer to the `DebuggedProcess` structure and the parsed command arguments. The handler is responsible for calling the appropriate component methods in the correct sequence and handling any errors that occur during execution.

The **error propagation** mechanism ensures that failures at any level bubble up with sufficient context for user diagnosis. Components return `DebugResult` enum values with optional `DebugError` structures containing detailed failure information. The command processing layer translates these into user-friendly messages.

Command Execution Sequence Example

Consider the user command `break main`. The processing sequence involves multiple component interactions:

1. **Input parsing** tokenizes the string into command name `"break"` and argument `"main"`
2. **Command lookup** finds the `cmd_break` handler function in the registry
3. **State validation** checks that the process is attached (breakpoints require active process)
4. **Symbol resolution** calls `symbols_resolve_function(symbols, "main", &address)` to find function entry point
5. **Address validation** verifies the resolved address points to executable code
6. **Breakpoint creation** calls `breakpoint_set(proc->pid, proc->breakpoints, address, "main")` to install the INT3 patch
7. **Result formatting** creates success message with breakpoint location and address

Each step can fail with specific error conditions. If symbol resolution fails because debug information is missing, the error propagates back with `DEBUG_NO_DEBUG_INFO`. If the breakpoint already exists at that address, `DEBUG_BREAKPOINT_EXISTS` is returned. The command processor translates these into messages like "No debug information found for symbol 'main'" or "Breakpoint already exists at 0x401000".

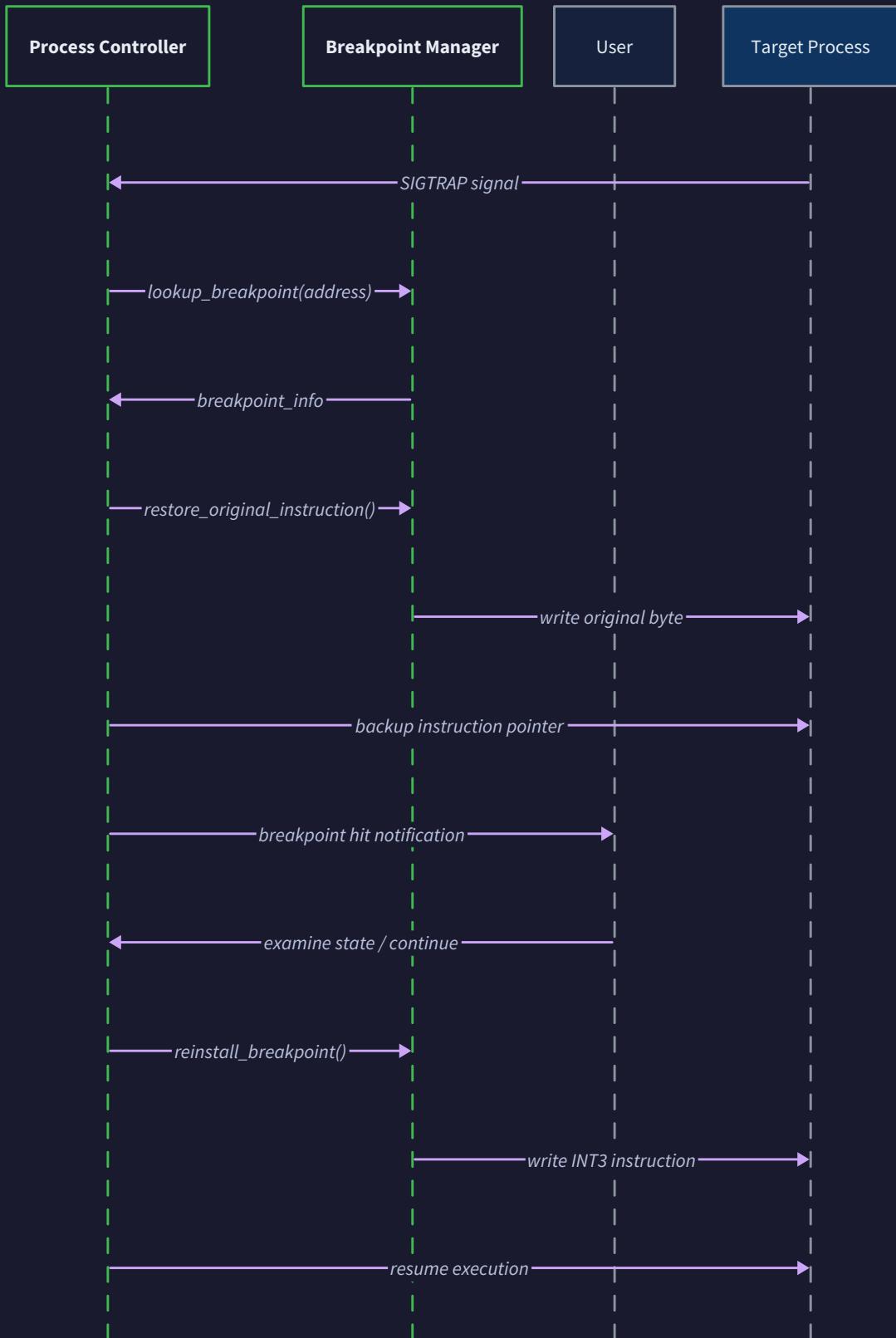
Asynchronous Command Handling

Some commands like `continue` start asynchronous operations. The `continue` command resumes process execution and returns control to the user, but the debugger must remain responsive to signals and breakpoint hits. This requires careful coordination between the command processing thread and signal handling.

The main debugger loop alternates between reading user commands and checking for process status changes using `waitpid` with `WNOHANG`. When a signal is received, the appropriate handler is called to process breakpoint hits, crashes, or normal termination. This asynchronous model allows users to interrupt running processes with Ctrl+C or examine the current state while execution continues.

Breakpoint Hit Workflow

The **breakpoint hit workflow** represents one of the most complex interaction sequences in the debugger. When the debugged process hits a breakpoint, a carefully orchestrated sequence of operations must occur to properly handle the interrupt, provide user feedback, and prepare for subsequent commands.



The Traffic Light Analogy

Think of a breakpoint hit like a traffic control system managing a busy intersection. The INT3 instruction acts like a red light that immediately stops traffic (process execution). The debugger serves as the traffic control operator who must: identify which intersection triggered the signal, record the incident in the log, direct traffic around the blocked lane, and then decide when to turn the light green again. Just as a traffic operator must coordinate with multiple city systems (emergency services, traffic monitoring, road maintenance), the debugger must coordinate between multiple components to handle the breakpoint properly.

The **signal detection** phase begins when the debugged process executes the INT3 instruction at a breakpoint location. This generates a `SIGTRAP` signal that is delivered to the debugging process through the ptrace interface. The Process Controller's main loop detects this signal via `waitpid` returning with a status indicating signal delivery.

Signal Processing Sequence

The signal handling sequence follows a precise protocol to maintain debugger state consistency:

Processing Step	Component Responsible	Action Taken	State Changes	Error Conditions
Signal Detection	Process Controller	<code>waitpid</code> returns with <code>SIGTRAP</code>	Process marked as <code>STOPPED</code>	Process died, signal lost
Address Calculation	Process Controller	Read instruction pointer, subtract 1	IP adjusted to breakpoint location	Memory access failure
Breakpoint Lookup	Breakpoint Manager	Search breakpoint list by address	Breakpoint marked as hit	Unknown breakpoint address
Hit Count Update	Breakpoint Manager	Increment <code>hit_count</code> field	Statistics updated	Overflow protection needed
User Notification	Command Processor	Format and display breakpoint info	User sees location details	Display formatting errors
Command Processing	Command Processor	Wait for user input	Interactive mode active	User disconnection

The **instruction pointer adjustment** is a critical step that often confuses implementors. When the processor executes the INT3 instruction, it advances the instruction pointer past the single-byte INT3 before delivering the signal. However, the original instruction at that location might be multiple bytes long. The debugger must subtract 1 from the instruction pointer to point back to the beginning of the original instruction location.

Breakpoint Restoration Decision

When a breakpoint is hit, the Breakpoint Manager must decide how to handle the subsequent execution. Two main strategies exist:

Decision: Immediate Restoration vs Lazy Restoration

- **Context:** After breakpoint hit, must decide when to restore original instruction
- **Options Considered:** Restore immediately and single-step, restore only when continuing, keep INT3 and step over
- **Decision:** Lazy restoration — restore only when user continues execution
- **Rationale:** Avoids unnecessary instruction patching if user examines state without continuing, reduces race conditions in multi-threaded scenarios
- **Consequences:** Requires careful state tracking but improves performance for examination-heavy debugging sessions

The **breakpoint state tracking** maintains several important pieces of information during a hit:

- **Hit breakpoint reference:** The `BreakpointList.hit_breakpoint` field points to the breakpoint that was just triggered
- **Original instruction preservation:** The `Breakpoint.original_byte` contains the instruction byte that was replaced by INT3
- **Restoration pending flag:** Internal state indicating whether the breakpoint needs restoration before continuing
- **Hit count statistics:** The `Breakpoint.hit_count` tracks how many times this breakpoint has been triggered

Continue-Past-Breakpoint Sequence

When the user issues a `continue` command after a breakpoint hit, a complex sequence ensures proper execution resumption:

1. **Instruction restoration:** Call `process_write_byte(pid, address, original_byte)` to replace INT3 with original instruction
2. **Single-step execution:** Use `PTRACE_SINGLESTEP` to execute the restored instruction exactly once
3. **Instruction re-patching:** Write INT3 back to the address to maintain the breakpoint for future hits
4. **Normal continuation:** Use `PTRACE_CONT` to resume normal execution speed

This **restore-step-repatch** sequence is essential because simply continuing with the original instruction restored would bypass the breakpoint permanently. The single-step ensures the original instruction executes once, then the INT3 is restored to catch future executions.

Error Handling in Breakpoint Workflow

Several failure modes can occur during breakpoint hit processing:

⚠ Pitfall: Race Condition in Multi-threaded Targets

If the target process has multiple threads, one thread might hit a breakpoint while another thread is executing at the same address. The restoration process must be coordinated carefully to avoid corrupting instruction memory. Our single-threaded debugger design avoids this complexity, but it's important to understand why multi-threading adds significant architectural challenges.

⚠ Pitfall: Breakpoint on Last Instruction

Setting a breakpoint on the last instruction of a function can cause the single-step restoration to step into the function return sequence, potentially changing the call stack before the user expects. The debugger should detect this condition

and warn the user about potential side effects.

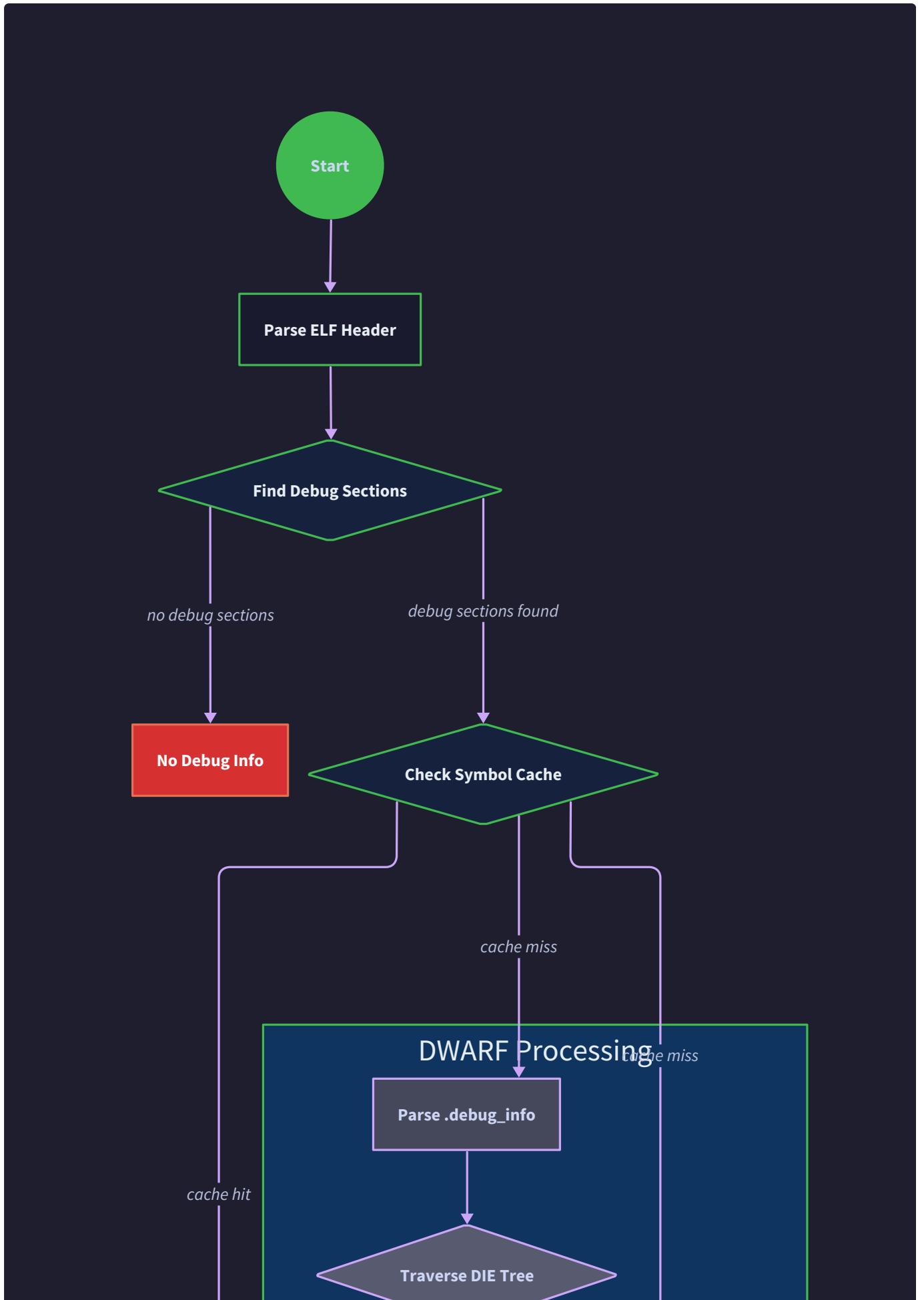
The **user interaction phase** after breakpoint hit provides an opportunity for state examination. Users typically want to see the current location, examine variables, or inspect the call stack. The debugger must maintain the stopped state while providing access to Symbol Resolver and Variable Inspector services.

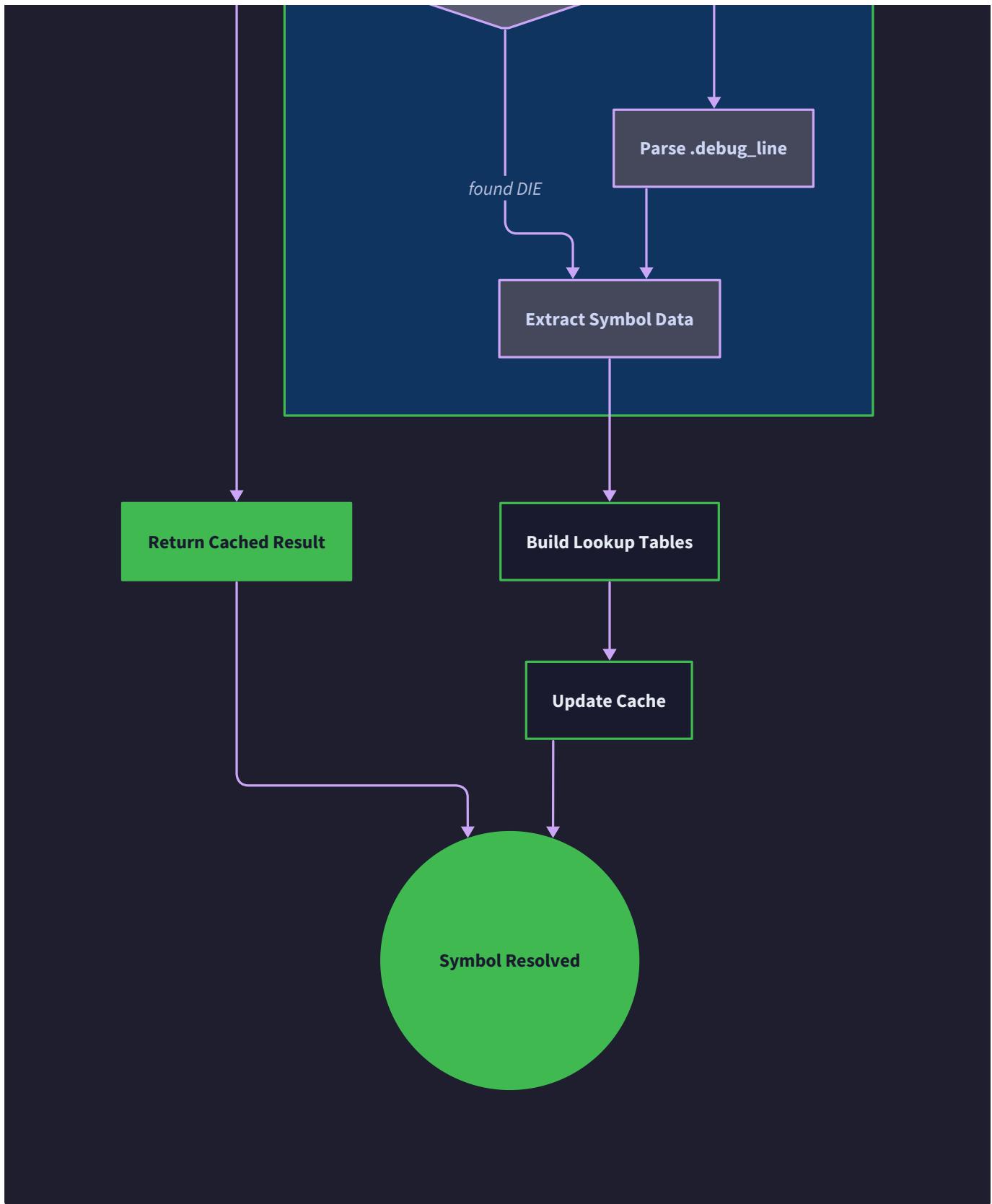
Breakpoint Hit Statistics and Debugging

The hit count tracking serves multiple purposes beyond simple statistics. High hit counts might indicate breakpoints in tight loops that significantly slow execution. Zero hit counts might indicate breakpoints set at unreachable code or incorrect addresses. The debugger can provide this information to help users optimize their debugging sessions.

Symbol Lookup Flow

The **symbol lookup flow** provides the critical translation services that enable source-level debugging. When users want to set breakpoints by function name, examine variables by name, or understand what source code corresponds to a memory address, the Symbol Resolver must efficiently search through potentially large symbol databases.





The Library Catalog Analogy

Think of symbol lookup like using a sophisticated library catalog system. The **DWARF debug information** is like the library's complete collection — thousands of books (compilation units) organized into sections (debug info, line numbers, abbreviations). The **symbol table** acts like the card catalog, providing quick name-based access to specific books and their locations on the shelves. The **address mappings** function like a detailed floor plan that can tell you exactly which aisle and shelf position corresponds to any location in the library.

Just as a librarian might use multiple catalog systems depending on whether you're searching by author name, subject, or Dewey decimal number, the Symbol Resolver maintains multiple indexes optimized for different types of lookups. A name-to-address lookup uses the hash table for fast symbol name resolution. An address-to-source lookup uses the sorted address mapping array for efficient binary search.

The **caching strategy** in symbol lookup resembles a librarian's desk with frequently requested books kept within easy reach. Popular symbols and recently accessed address mappings remain in fast-access data structures, while less common lookups require full index traversal.

Symbol Resolution Request Types

The Symbol Resolver handles several distinct types of lookup requests, each with different performance characteristics and data requirements:

Lookup Type	Input	Output	Index Used	Performance	Cache Strategy
Function name to address	Function name string	Entry point address	Hash table	O(1) average	MRU name cache
Address to source location	Memory address	Filename, line, column	Sorted address array	O(log n)	Address range cache
Variable name to location	Variable name, scope	DWARF location expression	Scope-aware hash	O(1) average	Scope-specific cache
Type name to descriptor	Type name	DWARF type information	Type hash table	O(1) average	Type descriptor cache
Line number to address	Filename, line number	Memory address	Line number table	O(log n)	Line mapping cache

Name-to-Address Resolution Workflow

The most common symbol lookup occurs when users set breakpoints by function name. The `symbols_resolve_function` call triggers a multi-stage resolution process:

- 1. Hash table lookup:** Calculate hash of function name and search the `SymbolTable.name_hash_table` for matching `Symbol` entries
- 2. Name comparison:** Handle hash collisions by comparing actual symbol names using string comparison
- 3. Symbol filtering:** Verify the symbol represents a function (not variable or type) by checking `symbol_type` field
- 4. Address validation:** Confirm the symbol's address falls within the executable's valid memory ranges
- 5. Demangling check:** If no match found with original name, attempt lookup with demangled C++ symbol names
- 6. Cache update:** Store successful lookup results in the name cache for future rapid access

The **hash collision handling** is particularly important for large symbol tables. Multiple symbols might hash to the same bucket, requiring iteration through the collision chain. The hash table design uses chaining with linked lists of `Symbol` structures, where the `next` field points to the next symbol in the same hash bucket.

Address-to-Source Resolution Workflow

Converting memory addresses to source locations requires a different approach because addresses form a continuous space rather than discrete names. The `symbols_resolve_address` function uses the `AddressMapping` array for efficient lookups:

- 1. Binary search:** Use the sorted `AddressMapping` array to find the mapping entry with the largest address \leq target address
- 2. Range validation:** Verify the target address falls within the valid range for the found mapping entry
- 3. Source location extraction:** Return the `SourceLocation` structure containing filename, line number, and column
- 4. Directory resolution:** Combine the relative filename with the compilation directory from DWARF info
- 5. Line number refinement:** For addresses between line mappings, interpolate to find the most accurate source location

The **binary search optimization** takes advantage of the fact that address mappings are naturally sorted by memory address. This provides O(log n) lookup time even for programs with thousands of functions.

DWARF Information Lazy Loading

Large programs can have megabytes of debug information, making it impractical to parse everything during debugger startup. The Symbol Resolver implements a **lazy loading strategy** that parses debug information on-demand:

Decision: Eager vs Lazy DWARF Parsing

- **Context:** Debug information can be very large (100MB+), affecting startup time and memory usage
- **Options Considered:** Parse everything at startup, parse nothing until needed, hybrid approach with essential symbols only
- **Decision:** Lazy loading with essential symbol table parsed at startup
- **Rationale:** Provides fast startup while ensuring good performance for accessed symbols, balances memory usage with response time
- **Consequences:** First access to new symbols has higher latency, but overall better resource utilization

The lazy loading implementation maintains **parsing state** for each compilation unit. When a symbol lookup requires detailed DWARF information that hasn't been parsed yet, the Symbol Resolver:

1. **Locate compilation unit:** Find the DWARF compilation unit containing the target symbol
2. **Parse DIE tree:** Read and parse the Debug Information Entries for that compilation unit only
3. **Extract symbol information:** Build symbol table entries and address mappings from the parsed DIES
4. **Cache parsed results:** Store the parsed information to avoid re-parsing the same compilation unit
5. **Update indexes:** Add newly discovered symbols to the hash table and address mapping array

Symbol Lookup Caching Strategy

The Symbol Resolver implements multiple levels of caching to optimize performance for common debugging patterns:

Name lookup cache stores recently resolved function names with their addresses. Debugging sessions often involve setting multiple breakpoints in related functions, so caching function addresses avoids repeated hash table searches.

Address range cache maintains a small cache of recently accessed address-to-source mappings. Users frequently step through code in localized areas, so caching recent address translations improves stepping performance.

Type information cache stores parsed DWARF type descriptors for recently inspected variables. Variable inspection often involves examining multiple variables of the same type, so cached type information accelerates the display formatting process.

Cache Type	Size Limit	Eviction Policy	Hit Rate Target	Cache Key
Name lookup	256 entries	LRU	85%+	Function name hash
Address range	64 entries	LRU	70%+	Address range start
Type descriptors	128 entries	LRU	90%+	DWARF type offset
Scope context	32 entries	Stack-based	95%+	Current frame pointer

Error Handling in Symbol Lookup

Symbol resolution can fail for several reasons, each requiring different error handling strategies:

Pitfall: Optimized Code Symbol Loss

Compiler optimizations can eliminate or relocate symbols, causing lookup failures for variables that exist in source code but not in the optimized binary. The Symbol Resolver should detect this condition by checking DWARF optimization flags and provide helpful error messages explaining that the variable may be optimized away.

Pitfall: DWARF Version Incompatibilities

Different compiler versions generate different DWARF formats. The parser must gracefully handle version differences by checking the version field in DWARF headers and either adapting the parsing strategy or providing clear error messages about unsupported DWARF versions.

The **incremental symbol loading** approach allows the debugger to start quickly and expand its symbol knowledge as debugging progresses. This provides good user experience while handling large programs efficiently.

Symbol Lookup Performance Monitoring

The Symbol Resolver tracks performance metrics to help identify optimization opportunities:

- **Cache hit ratios** for each cache type, helping tune cache sizes
- **Lookup latencies** for different symbol types, identifying bottlenecks
- **DWARF parsing frequency**, showing which compilation units are accessed most often
- **Hash table collision rates**, indicating when hash table resizing might be beneficial

These metrics help both debugger developers and users understand the performance characteristics of their debugging sessions and optimize accordingly.

Implementation Guidance

This implementation guidance provides the infrastructure and patterns needed to coordinate the four debugger components effectively. The focus is on the communication pathways and data flow patterns that enable smooth interaction between Process Controller, Breakpoint Manager, Symbol Resolver, and Variable Inspector.

Technology Recommendations

Component	Simple Option	Advanced Option
Command Processing	Function pointer array + linear search	Hash table with chained dispatch
Inter-component Communication	Direct function calls with shared state	Event bus with message passing
Error Propagation	Return codes with global error state	Structured error objects with stack traces
Asynchronous Coordination	Signal-driven polling loop	Event-driven state machine
Symbol Lookup Caching	Fixed-size LRU array	Adaptive hash table with statistics

Recommended File Structure

```
debugger/
├── src/
│   ├── main.c           ← Main event loop and command processing
│   ├── command_processor.c    ← Command parsing and dispatch
│   ├── command_processor.h    ← Command registry and handler types
│   ├── process_control/
│   │   ├── process_controller.c    ← Process Controller component
│   │   └── process_controller.h
│   ├── breakpoint_manager/
│   │   ├── breakpoint_manager.c    ← Breakpoint Manager component
│   │   └── breakpoint_manager.h
│   ├── symbol_resolver/
│   │   ├── symbol_resolver.c    ← Symbol Resolver component
│   │   └── symbol_resolver.h
│   │   ├── elf_parser.c        ← ELF format handling
│   │   └── dwarf_parser.c      ← DWARF format handling
│   ├── variable_inspector/
│   │   ├── variable_inspector.c    ← Variable Inspector component
│   │   └── variable_inspector.h
│   ├── common/
│   │   ├── debug_error.c        ← Shared infrastructure
│   │   ├── debug_error.h        ← Error handling utilities
│   │   ├── data_structures.h    ← Core data structure definitions
│   │   └── constants.h         ← Shared constants and enums
│   └── tests/
│       ├── test_program.c      ← Test programs and fixtures
│       ├── complex_program.c    ← Simple program for testing breakpoints
│       └── Makefile            ← Program with variables for inspection
                           ← Test program build rules
└── include/
    └── debugger.h           ← Public headers
└── Makefile               ← Main API definitions
└── README.md              ← Build configuration
                           ← Build and usage instructions
```

Infrastructure Starter Code: Command Registry

This complete command registry implementation handles command parsing, validation, and dispatch:

```
// command_processor.h

#ifndef COMMAND_PROCESSOR_H

#define COMMAND_PROCESSOR_H

#include "common/data_structures.h"

#include "common/debug_error.h"

typedef DebugResult (*CommandHandler)(DebuggedProcess* proc, const char* args);

typedef struct Command {

    const char* name;

    const char* help;

    CommandHandler handler;

} Command;

// Initialize command registry with built-in commands

DebugResult command_processor_init(void);

// Register a new command with the processor

DebugResult command_processor_register(const char* name, const char* help,
                                       CommandHandler handler);

// Process a complete command line input

DebugResult command_processor_execute(DebuggedProcess* proc, const char* input);

// Display help for all commands or specific command

void command_processor_show_help(const char* specific_command);

// Cleanup command registry

void command_processor_cleanup(void);

#endif // COMMAND_PROCESSOR_H
```

```
// command_processor.c

#include "command_processor.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX_COMMANDS 64

#define MAX_INPUT_LENGTH 512

static Command command_registry[MAX_COMMANDS];

static size_t command_count = 0;

// Forward declarations for built-in command handlers

static DebugResult cmd_help(DebuggedProcess* proc, const char* args);

static DebugResult cmd_quit(DebuggedProcess* proc, const char* args);

DebugResult command_processor_init(void) {

    command_count = 0;

    // Register built-in commands

    command_processor_register("help", "Show help for commands", cmd_help);

    command_processor_register("quit", "Exit the debugger", cmd_quit);

    command_processor_register("exit", "Exit the debugger", cmd_quit);

    return DEBUG_SUCCESS;
}

DebugResult command_processor_register(const char* name, const char* help,
                                      CommandHandler handler) {

    if (command_count >= MAX_COMMANDS) {

        return DEBUG_ERROR(DEBUG_SYSTEM_ERROR, "Command registry full");
    }
}
```

```
}

// Check for duplicate command names

for (size_t i = 0; i < command_count; i++) {

    if (strcmp(command_registry[i].name, name) == 0) {

        return DEBUG_ERROR(DEBUG_INVALID_ARGUMENT, "Command already exists");

    }

}

command_registry[command_count].name = name;

command_registry[command_count].help = help;

command_registry[command_count].handler = handler;

command_count++;



return DEBUG_SUCCESS;

}

DebugResult command_processor_execute(DebuggedProcess* proc, const char* input) {

    char command_buffer[MAX_INPUT_LENGTH];

    char* command_name;

    char* arguments;




    // Copy input to avoid modifying original

    strncpy(command_buffer, input, MAX_INPUT_LENGTH - 1);

    command_buffer[MAX_INPUT_LENGTH - 1] = '\0';




    // Skip leading whitespace

    command_name = command_buffer;

    while (isspace(*command_name)) command_name++;
```

```
// Find end of command name

arguments = command_name;

while (*arguments && !isspace(*arguments)) arguments++;

// Split command name and arguments

if (*arguments) {

    *arguments = '\0';

    arguments++;

    while (isspace(*arguments)) arguments++;

}

// Handle empty command

if (*command_name == '\0') {

    return DEBUG_SUCCESS;

}

// Look up command in registry

for (size_t i = 0; i < command_count; i++) {

    if (strcmp(command_registry[i].name, command_name) == 0) {

        return command_registry[i].handler(proc, arguments);

    }

}

printf("Unknown command: %s\n", command_name);

printf("Type 'help' for available commands.\n");

return DEBUG_SUCCESS;

}

void command_processor_show_help(const char* specific_command) {

    if (specific_command && *specific_command) {
```

```

// Show help for specific command

for (size_t i = 0; i < command_count; i++) {

    if (strcmp(command_registry[i].name, specific_command) == 0) {

        printf("%s: %s\n", command_registry[i].name, command_registry[i].help);

        return;
    }
}

printf("Unknown command: %s\n", specific_command);

} else {

    // Show all commands

    printf("Available commands:\n");

    for (size_t i = 0; i < command_count; i++) {

        printf("  %-12s - %s\n", command_registry[i].name, command_registry[i].help);
    }
}

}

static DebugResult cmd_help(DebuggedProcess* proc, const char* args) {

    command_processor_show_help(args);

    return DEBUG_SUCCESS;
}

static DebugResult cmd_quit(DebuggedProcess* proc, const char* args) {

    printf("Goodbye!\n");

    exit(0);
}

void command_processor_cleanup(void) {

    command_count = 0;
}

```

```
// common/debug_error.h

#ifndef DEBUG_ERROR_H

#define DEBUG_ERROR_H

#include "data_structures.h"

#include <errno.h>

// Create error with context information

DebugError debug_error_create(DebugResult code, const char* message,
                             const char* function, int line, int system_errno);

// Print formatted error message

void debug_error_print(const DebugError* error);

// Convenience macro for creating errors with automatic context

#define DEBUG_ERROR(code, msg) \
    debug_error_create(code, msg, __FUNCTION__, __LINE__, errno)

// Check if DebugResult indicates success

#define DEBUG_SUCCESS(result) ((result) == DEBUG_SUCCESS)

// Check if DebugResult indicates failure

#define DEBUG_FAILED(result) ((result) != DEBUG_SUCCESS)

#endif // DEBUG_ERROR_H
```

Core Logic Skeleton: Main Event Loop

```
// main.c - Main debugger event loop skeleton

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>
#include <readline/readline.h>
#include <readline/history.h>

#include "command_processor.h"
#include "process_control/process_controller.h"
#include "breakpoint_manager/breakpoint_manager.h"
#include "symbol_resolver/symbol_resolver.h"
#include "variable_inspector/variable_inspector.h"

// Main debugger event loop with command processing and signal handling

int main(int argc, char* argv[]) {
    DebuggedProcess proc = {0};
    DebugResult result;
    char* input_line;

    // TODO 1: Initialize all debugger components
    // Call command_processor_init(), register component command handlers
    // Initialize process controller, breakpoint manager, symbol resolver, variable inspector

    // TODO 2: Handle command line arguments for target program
    // If argc > 1, treat argv[1] as target program path
    // Call process_create() to start target under debugger control
    // Load symbols using symbols_load_from_elf()

    printf("Debugger ready. Type 'help' for commands.\n");
}
```

C

```
// Main command loop

while (1) {

    // TODO 3: Check for process status changes (if process attached)

    // Use waitpid(proc.pid, &status, WNOHANG) to check for signals

    // If SIGTRAP received, call breakpoint_handle_hit()

    // If process exited, clean up process state


    // TODO 4: Read user command input

    // Use readline() for command line editing and history

    // Handle NULL return (EOF/Ctrl-D) by breaking from loop

    input_line = readline("(debugger) ");

    if (!input_line) {

        printf("\n");

        break;

    }

    // Add non-empty commands to history

    if (*input_line) {

        add_history(input_line);

    }

    // TODO 5: Process the command through command processor

    // Call command_processor_execute() with current process state

    // Handle any errors returned from command execution

    result = command_processor_execute(&proc, input_line);

    if (DEBUG_FAILED(result)) {
```

```
    printf("Command failed: %d\n", result);

}

free(input_line);

}

// TODO 6: Cleanup debugger state

// Detach from process if still attached

// Free symbol tables, breakpoint lists

// Call component cleanup functions

return 0;
}
```

Core Logic Skeleton: Breakpoint Hit Handler

```
// breakpoint_hit_coordinator.c - Coordinates breakpoint hit across components C

DebugResult handle_breakpoint_hit_workflow(DebuggedProcess* proc, int signal_status) {

    uintptr_t hit_address;

    Breakpoint* hit_breakpoint;

    SourceLocation location = {0};

    // TODO 1: Verify this is actually a breakpoint-related SIGTRAP

    // Check WSTOPSIG(signal_status) == SIGTRAP

    // Exclude other SIGTRAP sources (single-step completion, manual SIGTRAP)

    if (WSTOPSIG(signal_status) != SIGTRAP) {

        return DEBUG_ERROR(DEBUG_PROCESS_ERROR, "Expected SIGTRAP signal");

    }

    // TODO 2: Read current instruction pointer and adjust for INT3

    // Call process_read_registers() to get current CPU state

    // Subtract 1 from instruction pointer to account for INT3 execution

    // Store adjusted address in hit_address variable

    // TODO 3: Look up which breakpoint was hit

    // Call breakpoint_find(proc->breakpoints, hit_address)

    // If no breakpoint found, this might be a spurious SIGTRAP

    // Return appropriate error if breakpoint lookup fails

    hit_breakpoint = breakpoint_find(proc->breakpoints, hit_address);

    if (!hit_breakpoint) {

        return DEBUG_ERROR(DEBUG_BREAKPOINT_NOT_FOUND,
                           "SIGTRAP at address with no breakpoint");

    }

}
```

```

// TODO 4: Update breakpoint hit statistics

// Increment hit_breakpoint->hit_count

// Record hit_breakpoint in proc->breakpoints->hit_breakpoint for later reference

// Mark process state as stopped at breakpoint


// TODO 5: Resolve hit location to source code (if debug info available)

// Call symbols_resolve_address() to get source file and line number

// Store result in location structure for user display

// Handle gracefully if no debug information available


// TODO 6: Display breakpoint hit information to user

// Show breakpoint number, address, hit count

// Show source location if available (filename:line)

// Show current function name if available


printf("\nBreakpoint hit at address 0x%lx\n", hit_address);

if (location.filename) {

    printf("File: %s:%d\n", location.filename, location.line_number);

}

printf("Hit count: %u\n", hit_breakpoint->hit_count);


// TODO 7: Prepare for user interaction

// Process is now stopped at breakpoint

// User can examine variables, set more breakpoints, continue, or step

// Return success to indicate breakpoint hit was handled properly


return DEBUG_SUCCESS;
}

```

Milestone Checkpoint: Command Flow Integration

After implementing the command processing flow:

Test Command 1: `make && ./debugger test_program`

- **Expected:** Debugger starts, loads test_program, shows prompt
- **Verify:** Type `help` and see list of available commands
- **Check:** Symbol loading messages appear if debug info present

Test Command 2: `break main` (in debugger prompt)

- **Expected:** "Breakpoint set at 0x[address] (main)" message
- **Verify:** Address should be non-zero and reasonable (0x400000+ range typical)
- **Check:** Symbol resolution succeeded, breakpoint creation succeeded

Test Command 3: `continue` followed by examining stopped state

- **Expected:** Process runs until breakpoint, shows hit message
- **Verify:** Process stops at main function entry point
- **Check:** Breakpoint hit count shows 1, process marked as stopped

Signs of Problems:

- **Command not found:** Check command registration in component init functions
- **Symbol resolution fails:** Verify ELF file has debug information (`readelf -S test_program | grep debug`)
- **Process hangs:** Check signal handling and waitpid usage for proper blocking/non-blocking behavior
- **Breakpoint not hit:** Verify INT3 patching with `objdump -d test_program` and check addresses match

Debugging Tips

Symptom	Likely Cause	Diagnosis Method	Fix Approach
Commands ignored	Handler not registered	Check command_count after init	Add missing registration calls
Symbol lookup fails	ELF sections missing	Use <code>readelf -S binary</code> to verify	Compile with <code>-g</code> debug flag
Breakpoint never hits	Wrong address calculation	Print addresses during setting	Verify symbol resolution accuracy
Process hangs on continue	Signal handling broken	Use <code>strace</code> on debugger	Fix waitpid status checking
Memory access errors	Invalid address reads	Check ptrace return values	Add address validation
Component coordination fails	State inconsistency	Add debug prints to data flow	Implement state validation

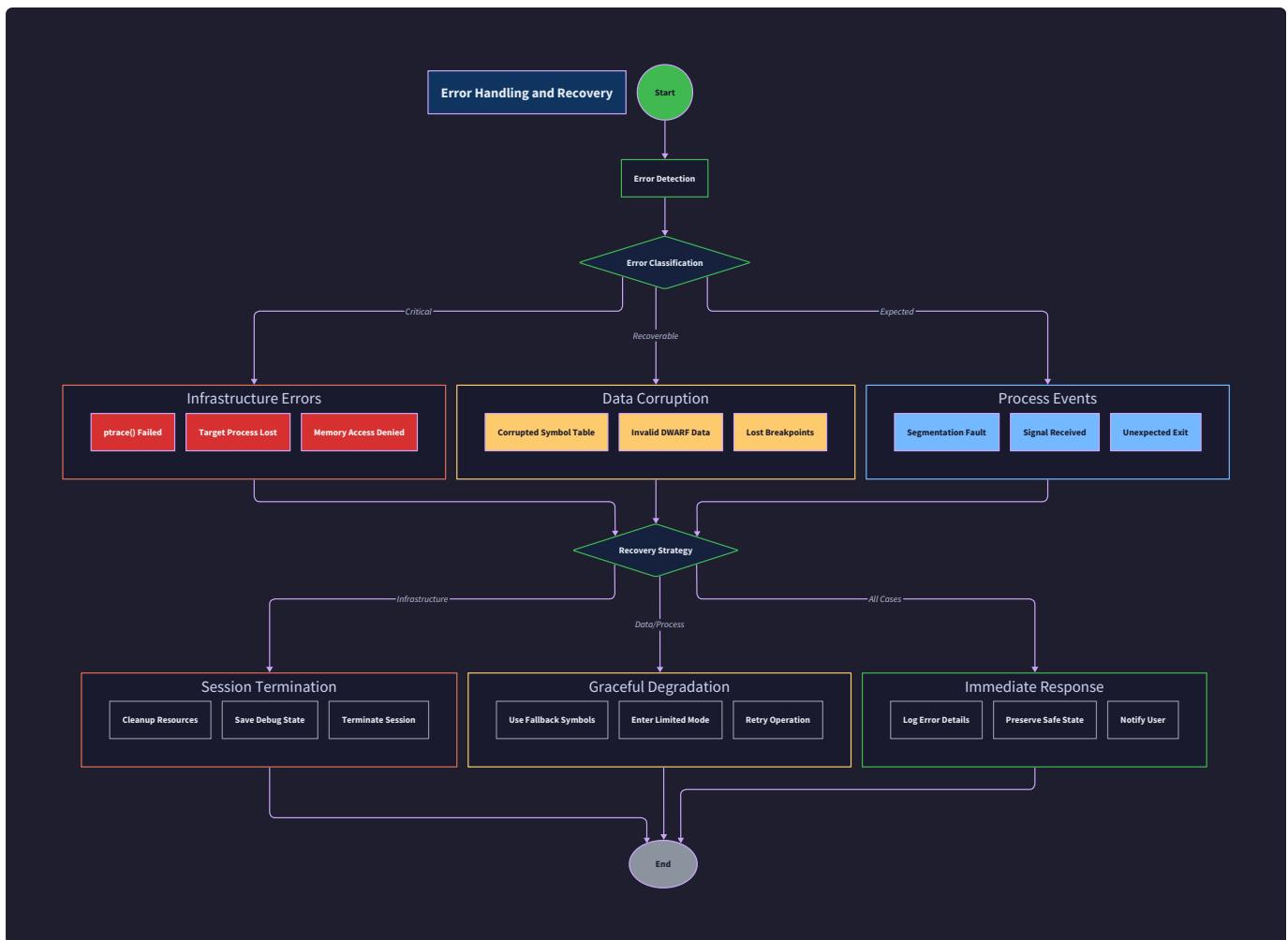
The interaction patterns established here form the foundation for all debugging operations. Understanding how commands flow through the system, how breakpoints coordinate between components, and how symbol lookups integrate with the other subsystems is essential for building a robust debugger that provides a smooth user experience.

Implementation Guidance

The command processing, breakpoint workflow, and symbol lookup flows represent the heart of debugger coordination. This implementation guidance provides the essential infrastructure that enables smooth component interaction and user experience.

Error Handling and Edge Cases

Milestone(s): All milestones — this section defines error handling strategies that apply throughout debugger implementation, with specific focus on robust failure detection and recovery mechanisms



The Emergency Response System Analogy

Think of debugger error handling like a comprehensive emergency response system for a complex building. Just as a building has smoke detectors for fires, security alarms for intrusions, and structural monitoring for earthquakes, our debugger needs different types of error detection for different categories of failures. Each type of failure requires its own detection mechanism, classification system, and response protocol.

Like an emergency response system, our error handling operates at multiple levels: immediate local responses (like a sprinkler system activating automatically), coordinated system responses (like evacuating a building section), and recovery procedures (like damage assessment and restoration). The key insight is that different failure modes require

different response strategies — you don't evacuate the entire building for a minor water leak, but you also can't ignore structural damage.

The debugger faces three primary categories of emergencies: **infrastructure failures** (the building's foundation is unstable — ptrace operations fail), **data corruption** (the emergency maps are wrong — debug information is malformed), and **tenant emergencies** (a specific office has a problem — the target process crashes). Each requires different detection methods, escalation procedures, and recovery strategies.

Error Classification and Response Strategy

Our debugger implements a hierarchical error handling strategy that matches the severity and scope of different failure modes. At the foundation level, we have infrastructure errors that affect the debugger's ability to control and inspect the target process. These are typically unrecoverable within the current debugging session and require clean shutdown or process restart. The middle layer consists of data parsing errors where debug information is incomplete or malformed — these often allow graceful degradation with reduced functionality. The top layer handles target process issues where the debuggee crashes or behaves unexpectedly — these can sometimes be recovered by restarting the target while preserving debugger state.

Each error type flows through a standard detection-classification-response pipeline. Detection mechanisms range from system call return value checking for ptrace operations, to structural validation for parsed debug information, to signal analysis for target process failures. Classification involves determining error severity, scope of impact, and recovery possibilities. Response strategies include immediate local fixes, graceful degradation, clean shutdown with state preservation, and user notification with actionable guidance.

The error handling system maintains detailed error context including the exact operation that failed, relevant system state at the time of failure, and suggested recovery actions. This context proves invaluable both for immediate error recovery and for debugging the debugger itself during development.

Design Principle: Errors should be detected as close to their source as possible, classified by their impact scope, and handled at the appropriate architectural level. Local errors stay local when possible, but system-wide issues trigger coordinated responses across all components.

Error Context and Reporting Framework

The debugger implements a comprehensive error reporting framework that captures rich contextual information for every failure. This framework serves multiple purposes: immediate error handling and recovery, user feedback and guidance, and developer debugging during system development. Each error captures not just what went wrong, but where it happened, what the system was doing at the time, and what recovery options are available.

Error context includes the specific function and line number where the error occurred, relevant system state such as process ID and current operation, the underlying system error code when applicable, and a human-readable message explaining the failure and potential causes. This rich context enables sophisticated error handling strategies that go beyond simple failure notification to provide actionable guidance for both users and the debugger system itself.

The error reporting system uses structured error objects that can be easily extended with additional context as needed. This approach allows different components to add their own relevant context while maintaining a consistent error handling interface across the entire system.

Error Context Field	Type	Purpose	Example Value
code	DebugResult	Standardized error classification	DEBUG_PTRACE_FAILED
message	char[256]	Human-readable error description	"Failed to read memory at 0x400000: permission denied"
function	const char*	Source function where error occurred	"process_read_memory"
line	int	Source line number for error location	156
system_errno	int	Underlying system error code	EPERM

Architecture Decision: Structured Error Objects vs Return Codes

- **Context:** Need to communicate rich error information across component boundaries while maintaining performance
- **Options Considered:** Simple integer return codes, exception-based error handling, structured error objects with context
- **Decision:** Structured error objects with embedded context information
- **Rationale:** C language limitations preclude exceptions, simple return codes lose critical context information, structured objects provide rich context while maintaining explicit error handling
- **Consequences:** Slightly more complex error handling code, but dramatically improved debugging and recovery capabilities, consistent error reporting across all components

ptrace Operation Failures

The process control subsystem faces numerous potential failures when interacting with the operating system through ptrace. These failures represent some of the most critical errors in the debugger because they directly impact the fundamental ability to control and inspect the target process. Understanding the failure modes, their causes, and appropriate responses is essential for building a robust debugger.

Permission and Access Failures

Permission-related failures occur when the debugger lacks sufficient privileges to perform the requested ptrace operation. The most common scenario involves attempting to attach to a process owned by a different user or a process with elevated privileges. Modern Linux systems also implement additional security restrictions through mechanisms like YAMA ptrace scope settings that can prevent even same-user process attachment.

When permission failures occur, the debugger must distinguish between temporary permission issues that might be resolved through user action (like running with elevated privileges) and permanent restrictions that cannot be overcome (like kernel security policies). The error handling strategy involves immediate failure detection through ptrace return value checking, classification of the specific permission issue through errno analysis, and user guidance on potential resolution steps.

The debugger implements a permission checking strategy that attempts to provide helpful guidance rather than generic error messages. When attachment fails with permission errors, the system checks common causes like process ownership, YAMA ptrace settings, and SELinux policies to provide specific guidance on resolving the issue.

Permission Failure Type	Detection Method	Recovery Strategy	User Guidance
Different user ownership	<code>ptrace()</code> returns -1, <code>errno == EPERM</code>	Cannot recover automatically	"Process owned by different user. Try running debugger with sudo or as the process owner."
YAMA ptrace restrictions	<code>ptrace()</code> returns -1, <code>errno == EPERM</code> , check <code>/proc/sys/kernel/yama/ptrace_scope</code>	User must modify system settings	"System has restricted ptrace access. Run: echo 0 sudo tee /proc/sys/kernel/yama/ptrace_scope"
SELinux denials	<code>ptrace()</code> returns -1, <code>errno == EACCES</code> , check audit logs	Cannot recover in debugger	"SELinux denied ptrace access. Check audit logs with: ausearch -m avc -ts recent"
Process already traced	<code>ptrace()</code> returns -1, <code>errno == EPERM</code> , process has tracer	Wait or find existing tracer	"Process already being debugged. Check for existing debugger with: ps -eo pid,cmd grep debugger"

Process State and Lifecycle Failures

Process lifecycle failures occur when the target process changes state unexpectedly or when the debugger attempts operations that are invalid for the current process state. These failures often indicate race conditions, timing issues, or fundamental misunderstandings about process state management. Proper handling requires careful state tracking and validation of preconditions before attempting ptrace operations.

The most common lifecycle failure involves attempting to control a process that has already exited. This can happen when the target process crashes or terminates while the debugger is preparing to send a control command. The debugger must detect this condition quickly to avoid hanging indefinitely waiting for responses from a non-existent process.

Another critical failure mode involves state transitions that occur between the debugger's state checks and its ptrace operations. For example, the debugger might check that a process is stopped, then attempt to read its memory, but the process could resume execution (due to a signal or another debugger) in the brief window between these operations.

Race condition handling requires atomic state checking combined with ptrace operations where possible, timeout mechanisms for operations that might hang indefinitely, and graceful handling of state transition errors with appropriate user notification.

Process State Failure	Detection Signals	Recovery Actions	Prevention Strategy
Process exited during operation	<code>waitpid()</code> returns with <code>WIFEXITED()</code> true	Clean up debugger state, notify user of exit	Check process existence before each operation
Process continued by external signal	<code>ptrace()</code> operation returns <code>ESRCH</code>	Re-attach and verify state	Use <code>PTRACE_SEIZE</code> for more robust attachment
Process became zombie	<code>waitpid()</code> succeeds but process unresponsive	Collect exit status and clean up	Monitor for <code>SIGCHLD</code> signals continuously
Debugger lost control	ptrace operations return <code>EPERM</code> after working	Attempt reattachment or graceful shutdown	Implement control validation before operations

System Resource and Capability Failures

System resource failures occur when the operating system cannot fulfill ptrace requests due to resource exhaustion or capability limitations. These failures are often transient but can indicate system-wide issues that require careful handling to avoid cascading failures or resource leaks in the debugger itself.

Memory-related failures happen when the debugger attempts to read or write memory regions that are not accessible in the target process. This includes unmapped memory regions, memory that has been freed, or memory with restrictive access permissions. The debugger must validate memory access requests and handle failures gracefully without crashing or corrupting its own state.

File descriptor and resource exhaustion can occur when the debugger opens many files for symbol processing or when system limits on ptrace attachments are reached. These failures require resource tracking and cleanup strategies to prevent permanent resource leaks that could affect system stability.

System capability failures involve attempts to use ptrace features that are not available on the current system or kernel version. The debugger should probe for available features and gracefully degrade functionality rather than failing completely when advanced features are unavailable.

⚠ Pitfall: Ignoring Transient ptrace Failures

A common mistake is treating all ptrace failures as permanent errors requiring debugger shutdown. Many ptrace operations can fail transiently due to system load, temporary resource exhaustion, or brief process state changes. Implementing appropriate retry logic with exponential backoff can resolve many seemingly permanent failures.

However, retry logic must be carefully bounded to avoid infinite loops when failures are truly permanent. The debugger should distinguish between retryable errors (like `EAGAIN` or `EINTR`) and permanent failures (like `ESRCH` for non-existent processes) and implement different handling strategies for each category.

Malformed Debug Information

Debug information parsing presents unique challenges because the debugger must handle potentially corrupted, incomplete, or maliciously crafted debug data without crashing or producing incorrect results. Unlike ptrace failures which typically result in clear system error codes, debug information problems often manifest as subtle inconsistencies or impossible values that require careful validation and graceful degradation strategies.

DWARF Structure Validation

DWARF debug information follows a complex hierarchical structure with intricate relationships between different sections and data elements. Malformed DWARF can violate these structural relationships in numerous ways, from simple data corruption to deliberately crafted exploits designed to cause buffer overflows or infinite loops in debug information parsers.

The debugger implements a multi-layered validation strategy that checks structural integrity at multiple levels. Basic validation ensures that section headers point to valid memory ranges and that data structure sizes are reasonable. Intermediate validation checks that cross-references between different DWARF sections are valid and that the hierarchical relationships between debug information entries are consistent. Advanced validation detects subtle inconsistencies that could lead to incorrect symbol resolution or variable location calculation.

Each validation layer operates with different performance and accuracy trade-offs. Basic validation is fast but only catches obvious corruption. Intermediate validation provides good coverage of common problems with modest performance impact. Advanced validation catches subtle issues but requires significant computational resources and is typically performed only when specifically requested or when problems are detected at lower levels.

The validation process produces structured error reports that classify the type and severity of problems found. This information guides the debugger's decision about whether to proceed with degraded functionality, attempt automatic repairs, or refuse to use the debug information entirely.

DWARF Validation Level	Checks Performed	Performance Impact	Error Recovery
Basic Structure	Section boundaries, size limits, alignment	Minimal	Reject entire debug info
Cross-Reference	DIE references, abbreviation table consistency	Moderate	Skip invalid entries
Semantic	Type relationships, address ranges, scope nesting	Significant	Attempt automatic repair
Comprehensive	Value range validation, duplicate detection	High	Manual review flagging

Incomplete Debug Information Handling

Many real-world binaries contain incomplete debug information due to compiler optimizations, partial debug builds, or post-processing tools that strip debug sections. The debugger must handle these situations gracefully by providing whatever functionality is possible with the available information while clearly communicating limitations to the user.

Incomplete debug information can manifest in several ways: missing sections that prevent certain types of analysis, partial symbol tables that cover only some functions, optimization artifacts that make variable locations impossible to determine, or version mismatches between debug information and the actual executable code.

The debugger implements a capability detection system that analyzes available debug information and determines which features can be provided. This analysis happens during symbol loading and produces a capability map that guides subsequent operations. When users request operations that require unavailable debug information, the debugger provides specific feedback about what information is missing and what alternative approaches might work.

Graceful degradation strategies include falling back to assembly-level debugging when source-level information is unavailable, using partial symbol information to provide limited functionality, inferring missing information from available data where safe to do so, and clearly marking uncertain or incomplete information in user displays.

Missing Debug Information	Impact on Functionality	Fallback Strategy	User Communication
.debug_line section	No source-to-address mapping	Assembly-level debugging only	"Source line information unavailable. Debugging at assembly level."
Function symbols	Cannot set breakpoints by name	Address-based breakpoints only	"Function name resolution unavailable. Use addresses: break *0x400000"
Variable location data	Cannot inspect local variables	Register and memory dumps only	"Variable inspection limited. Use 'info registers' and 'x/10x \$sp'"
Type information	No type-aware formatting	Raw memory display only	"Type information missing. Displaying raw bytes: 0x12345678"

Corrupted Symbol Table Recovery

Symbol table corruption can occur due to file system errors, memory corruption during loading, or bugs in the tools that generated the debug information. Unlike missing information, corrupted data can cause the debugger to produce incorrect results, potentially misleading users about program behavior.

The debugger implements corruption detection through consistency checking and data validation. Symbol table entries are checked for reasonable address ranges, valid string table references, and logical consistency between related entries. When corruption is detected, the debugger attempts to isolate the damage and recover usable information from uncorrupted sections.

Recovery strategies depend on the extent and type of corruption detected. Minor corruption in individual symbol entries can often be handled by skipping the affected symbols while preserving the rest of the table. More extensive corruption might require rebuilding portions of the symbol table from other available sources like the dynamic symbol table or export tables.

The debugger maintains detailed logs of corruption detection and recovery attempts to help users understand what information might be unreliable. This logging proves especially valuable when debugging issues that might be caused by corrupted debug information rather than actual program bugs.

Architecture Decision: Fail-Fast vs Graceful Degradation for Corrupted Debug Info

- **Context:** Need to balance reliability (avoiding incorrect results) with usability (providing partial functionality when possible)
- **Options Considered:** Fail completely on any corruption, attempt to repair all corruption automatically, graceful degradation with clear warnings
- **Decision:** Graceful degradation with explicit corruption warnings and conservative fallbacks
- **Rationale:** Complete failure makes debugger unusable for partially corrupted files, automatic repair risks incorrect results, degradation with warnings allows informed user decisions
- **Consequences:** More complex error handling logic, but improved usability and user awareness of potential reliability issues

⚠ Pitfall: Trusting Debug Information Without Validation

Debug information should never be trusted blindly, especially when parsing untrusted binaries. Always validate that addresses fall within expected ranges, that size fields are reasonable, and that cross-references point to valid locations. A corrupted debug entry claiming a variable is located at address 0x00000000 could cause the debugger to crash when attempting to read that location.

Implement sanity checks for all parsed values: addresses should fall within the process memory layout, sizes should be reasonable for the claimed data type, and string references should point to valid string table entries. When validation fails, treat the information as unavailable rather than using potentially incorrect data.

Target Process Crashes

Target process crashes represent a unique category of error because they are often the intended behavior being investigated (the user is debugging a crashing program), but they can also interfere with the debugger's ability to provide useful information. The debugger must distinguish between expected crashes that provide valuable debugging information and unexpected failures that prevent further investigation.

Crash Detection and Classification

The debugger detects target process crashes through multiple channels: signals received via waitpid, changes in process state visible through the proc filesystem, and ptrace operations that suddenly start failing with process-not-found errors. Each detection method provides different types of information about the crash, and combining these sources gives the most complete picture of what occurred.

Signal-based crash detection provides the most immediate and detailed information. Signals like `SIGSEGV`, `SIGBUS`, `SIGFPE`, and `SIGABRT` indicate specific types of program failures and often include additional context like the faulting address or the specific arithmetic error that occurred. The debugger captures this signal information and correlates it with the program state at the time of the crash to provide detailed crash analysis.

Process state monitoring through the proc filesystem provides additional context about resource usage, memory layout, and system interactions at the time of the crash. This information helps distinguish between crashes caused by program bugs and those caused by external factors like resource exhaustion or system configuration issues.

Crash classification involves analyzing the signal type, program counter location, and process state to determine the most likely cause of the crash. This classification guides both the information presented to the user and the debugger's strategy for preserving crash state for analysis.

Crash Signal	Typical Cause	Information Available	Analysis Strategy
SIGSEGV	Invalid memory access	Faulting address, instruction pointer	Check address validity, analyze memory layout
SIGBUS	Bus error, alignment issue	Faulting address, access type	Examine data alignment and hardware requirements
SIGFPE	Arithmetic exception	Operation type, operand values	Analyze arithmetic operation and operands
SIGABRT	Program called abort()	Stack trace, abort reason	Examine call stack and program state
SIGILL	Illegal instruction	Instruction bytes, processor state	Disassemble instruction and check architecture compatibility

Post-Crash State Preservation

When a target process crashes, the debugger has a brief window of opportunity to capture valuable debugging information before the process resources are cleaned up by the operating system. This state preservation is critical for effective crash analysis and must be performed quickly and reliably even when system resources are constrained.

The debugger implements a crash state capture process that prioritizes the most valuable information while working within time and resource constraints. Critical information includes the complete register state at the time of the crash, stack memory contents for stack trace reconstruction, and memory around the crash location for context analysis.

Stack trace capture requires careful handling because stack corruption often contributes to crashes. The debugger uses multiple stack unwinding strategies: frame pointer following when available, DWARF-based unwinding when debug information is present, and heuristic stack scanning as a fallback when other methods fail.

Memory context capture focuses on areas most likely to provide insight into the crash cause. This includes memory around the faulting address for segmentation violations, instruction bytes around the program counter for illegal instruction analysis, and heap structures for memory management related crashes.

The captured state is preserved in a structured format that allows detailed analysis even after the original process has been cleaned up. This crash dump includes not just raw memory contents but also interpreted information like symbol names, source locations, and variable values where available.

State Information	Capture Priority	Storage Format	Analysis Use
CPU registers	Critical	Binary register dump	Program counter analysis, stack pointer validation
Call stack	Critical	Unwound frame list	Function call sequence, crash context
Crash location memory	High	Hexdump with annotations	Instruction analysis, data corruption detection
Local variable values	High	Type-aware formatted dump	Program state reconstruction
Global program state	Medium	Selected memory regions	Environment and context analysis

Crash Recovery and Restart Strategies

After a target process crash, users typically want to restart their debugging session to investigate the crash further or test potential fixes. The debugger implements restart strategies that preserve debugging context while starting fresh with the target program.

Breakpoint preservation across restarts ensures that users don't lose their debugging setup when the target process crashes. The debugger maintains breakpoint information independently of the target process state and automatically restores breakpoints when restarting with the same executable.

Command history and debugging context preservation allows users to quickly recreate their debugging environment. This includes not just the commands they executed, but also the program arguments, environment variables, and working directory used for the crashed session.

Smart restart features can automatically restart the target process after a crash, optionally stopping at the same location where the crash occurred. This capability proves especially valuable when debugging intermittent crashes that require multiple execution attempts to trigger reliably.

The debugger also supports post-mortem debugging modes where users can analyze crash state without restarting the target process. This mode is particularly useful for crashes that are difficult to reproduce or when the crash state contains critical information that would be lost in a restart.

Design Insight: Crash handling is often where debuggers prove their worth. A debugger that gracefully handles crashes, preserves maximum debugging context, and provides clear analysis tools transforms a frustrating debugging experience into an effective problem-solving session.

⚠ Pitfall: Inadequate Crash State Capture Timing

Process cleanup happens quickly after a crash, and debuggers have a limited window to capture useful state information. Don't attempt to capture everything - prioritize the most critical information first. Trying to dump the entire process memory space might cause the capture to be interrupted by system cleanup, losing even the most important register and stack information.

Implement a tiered capture strategy: grab registers and stack first (these are small and critical), then memory around the crash location (medium priority), and finally broader memory context (nice to have). Set reasonable timeouts for each

capture phase to ensure that system cleanup doesn't interrupt the process.

Implementation Guidance

This implementation guidance provides concrete error handling mechanisms that can be integrated throughout the debugger implementation. The focus is on creating robust, informative error handling that helps both users and developers understand and recover from failures.

Technology Recommendations

Error Handling Component	Simple Option	Advanced Option
Error Classification	Simple enum return codes	Structured error objects with context
Error Reporting	Printf-style messages	Structured logging with severity levels
Stack Unwinding	Frame pointer walking	libunwind with DWARF support
Crash Dump Generation	Basic register/memory dump	Core file generation with metadata
Error Recovery	Manual user intervention	Automatic retry with backoff

Recommended File Structure

```
debugger/
├── src/
│   ├── error/
│   │   ├── error.c          ← Core error handling framework
│   │   ├── error.h          ← Error types and macros
│   │   └── error_recovery.c ← Recovery strategy implementations
│   ├── process/
│   │   ├── process_control.c ← Process control with error handling
│   │   └── crash_handler.c   ← Crash detection and state capture
│   ├── symbols/
│   │   ├── dwarf_parser.c    ← DWARF parsing with validation
│   │   └── symbol_validation.c ← Debug information validation
│   └── utils/
│       ├── logging.c         ← Centralized logging system
│       └── diagnostics.c     ← System diagnostic helpers
└── tests/
    ├── error_tests/
    │   ├── ptrace_failure_tests.c
    │   ├── corrupt_dwarf_tests.c
    │   └── crash_handling_tests.c
    └── test_programs/
        ├── crash_test.c        ← Program that crashes predictably
        ├── corrupt_symbols/
        │   └── permission_test.c ← Binaries with malformed debug info
        └── permission_test.c   ← Program requiring elevated privileges
└── docs/
    └── error_codes.md        ← Complete error code reference
```

Error Framework Infrastructure Code

```
// error/error.h - Complete error handling framework

#include <errno.h>

#include <string.h>

#include <stdio.h>

#include <stdint.h>

#include <stdbool.h>

// Core error result enumeration

typedef enum {

    DEBUG_SUCCESS = 0,

    DEBUG_PTRACE_FAILED,

    DEBUG_PROCESS_EXITED,

    DEBUG_INVALID_ADDRESS,

    DEBUG_SYMBOL_NOT_FOUND,

    DEBUG_BREAKPOINT_EXISTS,

    DEBUG_NO_DEBUG_INFO,

    DEBUG_PROCESS_NOT_STOPPED,

    DEBUG_BREAKPOINT_NOT_FOUND,

    DEBUG_MEMORY_ACCESS_FAILED,

    DEBUG_INSTRUCTION_BOUNDARY_ERROR,

    DEBUG_PERMISSION_DENIED,

    DEBUG_CORRUPT_DEBUG_INFO,

    DEBUG_RESOURCE_EXHAUSTED,

    DEBUG_INVALID_DWARF,

    DEBUG_UNSUPPORTED_FEATURE

} DebugResult;

// Structured error object with rich context

typedef struct {
```

C

```
DebugResult code;

char message[256];

const char* function;

int line;

int system_errno;

} DebugError;

// Error creation macro that captures location automatically

#define DEBUG_ERROR(code, msg) debug_error_create(code, msg, __FUNCTION__, __LINE__, errno)

// Complete error handling functions

DebugError debug_error_create(DebugResult code, const char* message,
                             const char* function, int line, int sys_errno) {

    DebugError error = {

        .code = code,
        .function = function,
        .line = line,
        .system_errno = sys_errno
    };

    snprintf(error.message, sizeof(error.message), "%s", message);

    return error;
}

void debug_error_print(const DebugError* error) {

    const char* error_names[] = {
        [DEBUG_SUCCESS] = "SUCCESS",
        [DEBUG_PTRACE_FAILED] = "PTRACE_FAILED",
        [DEBUG_PROCESS_EXITED] = "PROCESS_EXITED",
        [DEBUG_INVALID_ADDRESS] = "INVALID_ADDRESS",
        [DEBUG_SYMBOL_NOT_FOUND] = "SYMBOL_NOT_FOUND",
    }
}
```

```
[DEBUG_BREAKPOINT_EXISTS] = "BREAKPOINT_EXISTS",
[DEBUG_NO_DEBUG_INFO] = "NO_DEBUG_INFO",
[DEBUG_PROCESS_NOT_STOPPED] = "PROCESS_NOT_STOPPED",
[DEBUG_BREAKPOINT_NOT_FOUND] = "BREAKPOINT_NOT_FOUND",
[DEBUG_MEMORY_ACCESS_FAILED] = "MEMORY_ACCESS_FAILED",
[DEBUG_INSTRUCTION_BOUNDARY_ERROR] = "INSTRUCTION_BOUNDARY_ERROR",
[DEBUG_PERMISSION_DENIED] = "PERMISSION_DENIED",
[DEBUG_CORRUPT_DEBUG_INFO] = "CORRUPT_DEBUG_INFO",
[DEBUG_RESOURCE_EXHAUSTED] = "RESOURCE_EXHAUSTED",
[DEBUG_INVALID_DWARF] = "INVALID_DWARF",
[DEBUG_UNSUPPORTED_FEATURE] = "UNSUPPORTED_FEATURE"

};

fprintf(stderr, "ERROR [%s]: %s\n", error_names[error->code], error->message);
fprintf(stderr, " Location: %s:%d\n", error->function, error->line);

if (error->system_errno != 0) {
    fprintf(stderr, " System: %s\n", strerror(error->system_errno));
}

// Error classification helpers

bool debug_error_is_recoverable(DebugResult code) {

    switch (code) {

        case DEBUG_RESOURCE_EXHAUSTED:
        case DEBUG_PROCESS_NOT_STOPPED:
        case DEBUG_MEMORY_ACCESS_FAILED:

            return true;

        default:
            return false;
    }
}
```

```
}

bool debug_error_is_user_fixable(DebugResult code) {

    switch (code) {

        case DEBUG_PERMISSION_DENIED:

        case DEBUG_INVALID_ADDRESS:

        case DEBUG_SYMBOL_NOT_FOUND:

            return true;

        default:

            return false;
    }
}
```

Process Control Error Handling Skeleton

```
// process/process_control.c - Core logic skeleton with error handling TODOs
// C

#include "error.h"

#include <sys/ptrace.h>

#include <sys/wait.h>

#include <unistd.h>

// Attach to existing process with comprehensive error handling

DebugResult process_attach(pid_t target_pid, DebuggedProcess* proc) {

    // TODO 1: Validate that target_pid is a valid process ID

    // Check /proc/PID/stat exists and is readable

    // Return DEBUG_INVALID_ADDRESS if process doesn't exist

    // TODO 2: Check permissions before attempting ptrace

    // Compare effective UID with process owner in /proc/PID/status

    // Check YAMA ptrace_scope setting in /proc/sys/kernel/yama/ptrace_scope

    // Return DEBUG_PERMISSION_DENIED with specific guidance if checks fail

    // TODO 3: Attempt ptrace attachment with PTRACE_ATTACH

    // if (ptrace(PTRACE_ATTACH, target_pid, NULL, NULL) == -1) {

    //     switch (errno) {

    //         case EPERM: return DEBUG_ERROR(DEBUG_PERMISSION_DENIED, "Permission denied - check
process ownership and YAMA settings");

    //         case ESRCH: return DEBUG_ERROR(DEBUG_PROCESS_EXITED, "Target process no longer
exists");

    //         case EIO: return DEBUG_ERROR(DEBUG_PTRACE_FAILED, "I/O error during ptrace - process
may be in invalid state");

    //         default: return DEBUG_ERROR(DEBUG_PTRACE_FAILED, "Unknown ptrace failure");

    //     }

    // }

}
```

```
// TODO 4: Wait for attachment confirmation with timeout

// Use waitpid with WNOHANG and implement 5-second timeout

// Return DEBUG_PTRACE_FAILED if attachment doesn't complete


// TODO 5: Initialize DebuggedProcess structure

// Set proc->pid = target_pid, proc->state = STOPPED

// Load executable path from /proc/PID/exe

// Initialize empty breakpoint list and symbol table


// TODO 6: Verify attachment by reading initial register state

// Use ptrace(PTRACE_GETREGS) to confirm we have control

// Return DEBUG_PTRACE_FAILED if register read fails


return DEBUG_SUCCESS;

}

// Continue process execution with signal handling

DebugResult process_continue(DebuggedProcess* proc) {

    // TODO 1: Validate process is in correct state for continuation

    // Check proc->state == STOPPED, return DEBUG_PROCESS_NOT_STOPPED if not


    // TODO 2: Handle pending breakpoint if one exists

    // If proc->breakpoints->single_step_restore != NULL, need to:
    //     - Restore original instruction byte
    //     - Single step past the breakpoint
    //     - Re-patch the breakpoint
    //     - Then continue normally


    // TODO 3: Execute ptrace continue operation

    // Use ptrace(PTRACE_CONT, proc->pid, NULL, NULL)
```

```
// Handle errno values: ESRCH (process died), EIO (invalid state)

// TODO 4: Wait for next stop event with signal analysis

// Use waitpid to wait for SIGTRAP, SIGSEGV, or other signals

// Classify the stop reason and update proc->state accordingly

// TODO 5: Update process state based on wait result

// If WIFEXITED(status): proc->state = EXITED, return DEBUG_PROCESS_EXITED

// If WIFSTOPPED(status) with SIGTRAP: check for breakpoint hit

// If WIFSTOPPED(status) with other signals: handle crash detection

return DEBUG_SUCCESS;

}
```

DWARF Validation Skeleton

```
// symbols/symbol_validation.c - Debug information validation C

#include "error.h"

#include <elf.h>

// Validate dwarf debug information structure

DebugResult dwarf_validate_debug_info(const uint8_t* debug_info_data,
                                      size_t data_size,
                                      const uint8_t* debug_abbrev_data,
                                      size_t abbrev_size) {

    // TODO 1: Validate compilation unit header structure

    // Check that unit_length field is reasonable (< data_size)

    // Verify version number is supported (2, 3, 4, or 5)

    // Ensure debug_abbrev_offset points within abbrev_size

    // TODO 2: Parse and validate abbreviation table

    // Check that all abbreviation codes are unique

    // Verify attribute specifications are well-formed

    // Detect circular references in abbreviation definitions

    // TODO 3: Walk DIE tree and validate structure

    // Ensure all attribute references point to valid data

    // Check that parent-child relationships are consistent

    // Validate that address ranges are non-overlapping and within bounds

    // TODO 4: Cross-reference validation

    // Verify that DW_AT_specification and DW_AT_abstract_origin references are valid

    // Check that type references (DW_AT_type) point to actual type DIEs

    // Ensure location expressions reference valid registers/memory
```

```
// TODO 5: Semantic consistency checks

// Verify that function address ranges don't overlap

// Check that variable scopes are properly nested

// Validate that line number sequences are monotonically increasing


// Validation should return specific error codes:

// DEBUG_INVALID_DWARF for structural problems

// DEBUG_CORRUPT_DEBUG_INFO for consistency violations

// DEBUG_UNSUPPORTED_FEATURE for valid but unhandled DWARF features


return DEBUG_SUCCESS;

}

// Graceful degradation for missing debug sections

DebugResult symbols_load_with_fallbacks(const char* executable_path,
                                         SymbolTable* symbols) {

    // TODO 1: Attempt to load full DWARF debug information

    // Try to parse .debug_info, .debug_line, .debug_abbrev sections

    // If successful, create full symbol table with source mapping


    // TODO 2: Fallback to symbol table only if DWARF unavailable

    // Parse .symtab section for function names and addresses

    // Create limited symbol table without source line information

    // Mark capabilities as "assembly debugging only"


    // TODO 3: Final fallback to dynamic symbol table

    // Use .dynsym section if .symtab is stripped

    // Provide minimal symbol resolution for library functions

    // Warn user about limited debugging capabilities
```

```
// TODO 4: Generate capability report for user

// Create structured report of available debugging features

// Include specific guidance on what operations will/won't work

// Suggest ways to get better debug information if possible

return DEBUG_SUCCESS;

}
```

Crash Handling Implementation

```
// process/crash_handler.c - Target process crash detection and analysis C

#include "error.h"

#include <signal.h>

#include <sys/wait.h>

// Comprehensive crash analysis structure

typedef struct {

    int signal_number;

    void* fault_address;

    struct user_regs_struct registers;

    uint8_t stack_dump[4096];

    uint8_t instruction_context[64];

    char crash_summary[512];

} CrashAnalysis;

// Analyze and handle target process crash

DebugResult crash_handle_and_analyze(DebuggedProcess* proc,
                                      int wait_status,
                                      CrashAnalysis* analysis) {

    // TODO 1: Extract crash signal and context from wait_status

    // Use WTERMSIG(wait_status) to get terminating signal

    // For SIGSEGV/SIGBUS, extract fault address from siginfo if available

    // Record signal number and fault context in analysis structure

    // TODO 2: Capture CPU register state at crash

    // Use ptrace(PTRACE_GETREGS) to read all general-purpose registers

    // Store register state in analysis->registers

    // Calculate instruction pointer and stack pointer values
```

```
// TODO 3: Dump stack memory for backtrace analysis

// Read memory from stack pointer backwards (up to 4KB)

// Use process_read_memory with error handling for unmapped regions

// Store valid stack contents in analysis->stack_dump


// TODO 4: Capture instruction context around crash location

// Read 32 bytes before and after the instruction pointer

// Handle cases where IP points to unmapped or unreadable memory

// Store instruction bytes in analysis->instruction_context


// TODO 5: Generate human-readable crash summary

// Analyze signal type and provide likely cause explanation

// Include register values, fault address, and instruction analysis

// Format summary in analysis->crash_summary for user display


// TODO 6: Attempt to generate stack backtrace

// Use frame pointer walking or DWARF unwinding if available

// Handle corrupted stacks gracefully with partial backtraces

// Include function names and source locations where possible


// TODO 7: Preserve crash state for post-mortem analysis

// Save complete crash analysis to temporary file

// Maintain process memory image for continued inspection

// Allow user to explore crash state before cleanup


return DEBUG_SUCCESS;

}
```

Milestone Checkpoints

Milestone 1 Error Handling Verification:

- Test ptrace attachment to processes with different ownership - should get clear permission error messages
- Attempt to debug non-existent process - should detect and report process not found
- Kill target process during debugging - should detect process exit and clean up properly
- Expected output: "ERROR [PERMISSION_DENIED]: Permission denied - check process ownership and YAMA settings"

Milestone 2 Error Handling Verification:

- Set breakpoint at invalid address - should validate address and return appropriate error
- Set multiple breakpoints at same location - should detect duplicate and handle appropriately
- Target process crashes on breakpoint - should preserve breakpoint state and allow restart
- Expected behavior: Graceful error messages with specific guidance on how to fix issues

Milestone 3 Error Handling Verification:

- Load executable with stripped debug symbols - should fall back to basic symbol table
- Parse deliberately corrupted DWARF data - should detect corruption and degrade gracefully
- Handle executables with unsupported DWARF versions - should provide clear version mismatch message
- Expected output: "WARNING: Debug information incomplete. Source-level debugging unavailable. Use assembly mode."

Milestone 4 Error Handling Verification:

- Attempt to inspect optimized-out variables - should report variable not available with explanation
- Access variables after they go out of scope - should detect scope issues and warn appropriately
- Handle variables with complex DWARF location expressions - should fall back to raw memory display when needed
- Expected behavior: Clear explanations of what information is/isn't available and why

Debugging the Debugger Tips

Symptom	Likely Cause	Diagnosis Method	Fix
ptrace always fails with EPERM	YAMA ptrace restrictions enabled	Check <code>/proc/sys/kernel/yama/ptrace_scope</code> value	Set <code>ptrace_scope</code> to 0 or use <code>sudo</code>
Process appears to hang after attach	Debugger not handling SIGSTOP correctly	Use <code>strace -f ./debugger</code> to see system calls	Implement proper signal handling in <code>waitpid</code> loop
Breakpoints don't trigger	INT3 instruction not being written	Examine target memory with <code>gdb -p PID</code>	Verify memory write permissions and instruction patching
Symbol resolution fails silently	Debug sections missing or corrupted	Use <code>readelf -S binary</code> and <code>objdump -h binary</code>	Check for <code>.debug_*</code> sections and validate structure
Crash analysis produces garbage	Reading unmapped memory regions	Check <code>/proc/PID/maps</code> for valid address ranges	Validate memory addresses before ptrace reads
Variable inspection shows wrong values	Incorrect DWARF location interpretation	Use <code>objdump --dwarf=info binary</code> to examine location expressions	Implement proper DWARF expression evaluation

Testing Strategy

Milestone(s): All milestones — this section provides comprehensive testing approaches and verification criteria for each milestone, along with integration testing strategies and test program design patterns

The Scientific Method Analogy

Think of testing a debugger like conducting controlled scientific experiments. Just as a scientist designs precise experiments to isolate variables and verify hypotheses, debugger testing requires carefully designed test scenarios that isolate specific functionality and verify expected behaviors. Each test acts as a controlled experiment where we know the exact inputs, expected outputs, and environmental conditions. The test programs serve as laboratory specimens — simple, predictable subjects that we can examine under controlled conditions to verify our debugger's "instruments" (components) are working correctly.

Unlike testing typical applications where you might mock dependencies, debugger testing requires real processes, real memory, and real system interactions. This makes it more like field testing scientific equipment — you need actual specimens to verify your instruments work correctly in real conditions.

Milestone Verification Checkpoints

Each milestone represents a major capability that must be thoroughly verified before proceeding to the next level of functionality. The verification checkpoints provide concrete tests with specific expected behaviors, along with diagnostic

techniques when results don't match expectations.

Milestone 1: Process Control Verification

Process control forms the foundation of all debugging capabilities, so verification must be extremely thorough. The tests focus on the fundamental ptrace operations and signal handling that enable debugger control.

Basic Process Attachment Test

Test Scenario	Expected Behavior	Verification Method	Common Failure Modes
Attach to sleeping process	<code>process_attach</code> returns <code>DEBUG_SUCCESS</code> , process state becomes <code>STOPPED</code>	Check process state with <code>ps</code> command shows T (traced)	Permission denied (EPERM), process already being traced
Attach to non-existent PID	<code>process_attach</code> returns <code>DEBUG_PROCESS_EXITED</code> or <code>DEBUG_PTRACE_FAILED</code>	Error code matches expected value	Incorrect error handling, segfault on invalid PID
Attach without permissions	<code>process_attach</code> returns <code>DEBUG_PTRACE_FAILED</code> with <code>EPERM</code> errno	Check <code>system_errno</code> field contains <code>EPERM</code>	Missing permission checks, unclear error messages

The basic attachment test uses a simple helper program that sleeps for 30 seconds, giving the debugger time to attach. The test should fork this helper, get its PID, and attempt attachment while the child is running.

Process Creation and Control Test

Test Operation	Test Input	Expected Result	Verification Steps
Start process under debugger	Simple "hello world" executable path	Process starts in stopped state before main()	Read <code>/proc/PID/stat</code> , status field should be 'T'
Single-step execution	Call <code>process_single_step</code> on stopped process	Exactly one instruction executes, process stops again	Read instruction pointer before/after, verify increment
Continue execution	Call <code>process_continue</code> on stopped process	Process runs until exit, debugger receives exit status	<code>waitpid</code> returns with <code>WIFEXITED</code> true
Signal forwarding	Send SIGUSR1 to traced process	Signal delivered to tracee, debugger notified	Process signal handler executes, debugger gets notification

The single-step verification requires checking the instruction pointer advancement. On x86-64, instructions vary in length from 1-15 bytes, so the test should verify that the IP advanced by exactly the length of the instruction at the previous location.

Signal Handling Verification

Signal handling represents one of the most complex aspects of process control, requiring careful coordination between the debugger and operating system.

Signal Scenario	Setup Steps	Expected Debugger Behavior	Verification Method
SIGTRAP from breakpoint	Set software breakpoint, continue process	Debugger receives SIGTRAP, identifies breakpoint hit	Check <code>waitpid</code> status with <code>WIFSTOPPED</code> and <code>WSTOPSIG</code>
SIGSEGV from target	Target dereferences NULL pointer	Debugger receives SIGSEGV, can inspect crash location	Verify signal number and fault address from wait status
SIGINT forwarding	Send SIGINT to debugger while target running	Signal forwarded to target, not handled by debugger	Target receives signal, debugger continues tracing
Process exit	Target calls exit(42)	Debugger receives exit notification with correct code	<code>WIFEXITED</code> true, <code>WEXITSTATUS</code> returns 42

Critical Testing Insight: Signal handling bugs often manifest as race conditions or timing-dependent failures. Tests should include delays, rapid signal sequences, and concurrent operations to expose these issues.

Process Control State Machine Test

The process control component maintains a state machine that tracks the debugged process state. This requires systematic testing of all valid state transitions.

Current State	Trigger Event	Expected Next State	Required Actions	Failure Conditions
RUNNING	SIGTRAP received	STOPPED	Save signal info, notify user	State not updated, signal lost
STOPPED	process_continue called	RUNNING	Issue PTTRACE_CONT, start waiting	ptrace fails, state inconsistent
STOPPED	process_single_step called	STOPPED	Execute one instruction, update registers	IP not advanced, infinite loop
RUNNING	Process exits	EXITED	Clean up resources, save exit status	Zombie process, memory leak

Milestone 2: Breakpoint Management Verification

Breakpoint functionality builds on process control but adds the complexity of memory modification and instruction restoration. Testing must verify both the mechanical aspects (instruction patching) and the coordination aspects (hit detection and restoration).

Software Breakpoint Implementation Test

Breakpoint Operation	Test Setup	Expected Memory Changes	Verification Steps
Set breakpoint	Target address contains valid instruction	Original byte saved, 0xCC written to address	Read memory at address, confirm INT3 present
Hit detection	Execute to breakpoint address	SIGTRAP generated, IP points to breakpoint	Check signal type and instruction pointer value
Restoration	Handle breakpoint hit	Original byte restored, IP decremented by 1	Verify original instruction back in memory
Re-enable	Continue past breakpoint	INT3 re-installed after single-step	Confirm breakpoint active for next hit

The breakpoint implementation test requires a target program with known instruction sequences. A simple function that performs arithmetic operations provides predictable instruction boundaries for breakpoint placement.

Multiple Breakpoint Coordination Test

Managing multiple breakpoints simultaneously presents challenges in hit detection, restoration, and coordination. The test must verify that breakpoints don't interfere with each other.

Test Scenario	Setup Configuration	Expected Behavior	Verification Criteria
Two breakpoints in sequence	Set breakpoints at consecutive function calls	Each breakpoint hits independently	Both hit counts increment correctly
Overlapping breakpoint attempts	Try to set breakpoint at same address twice	Second attempt returns <code>DEBUG_BREAKPOINT_EXISTS</code>	No memory corruption, original preserved
Breakpoint removal	Remove one of multiple active breakpoints	Other breakpoints unaffected, memory restored	Verify other breakpoints still trigger
Rapid hit sequence	Loop containing breakpoint	Multiple hits detected correctly	Hit count accurate, no missed triggers

Breakpoint Lifecycle Management Test

Lifecycle Stage	Operation	Memory State Verification	Data Structure Verification
Creation	<code>breakpoint_set(addr, "test")</code>	Original byte saved, INT3 written	Breakpoint added to list, enabled=true
Hit	Execute to breakpoint	IP at breakpoint address	hit_count incremented, is_enabled checked
Disable	Toggle breakpoint off	Original byte restored	is_enabled=false, still in list
Re-enable	Toggle breakpoint on	INT3 re-written	is_enabled=true, original_byte preserved
Deletion	<code>breakpoint_remove(addr)</code>	Memory fully restored	Breakpoint removed from list

Milestone 3: Symbol Resolution Verification

Symbol resolution testing must verify both the parsing of debug information and the accuracy of address-to-source mappings. This requires test programs compiled with debug information and predictable symbol layouts.

ELF and DWARF Parsing Test

Debug Information Element	Test File Requirement	Expected Parse Result	Verification Method
ELF section headers	File with .debug_info, .debug_line sections	Sections located correctly	Compare parsed offsets with <code>readelf -S</code> output
DWARF compilation unit	Single-file C program	CU DIE parsed with correct attributes	Verify compilation directory and source file name
Function symbols	Program with multiple functions	Each function has name and address mapping	Compare with <code>objdump -t</code> symbol table
Line number mapping	Source with known line numbers	Address maps to correct source location	Cross-reference with <code>addr2line</code> utility

The test requires a carefully constructed C program with known symbol layout:

```
// test_symbols.c - compile with -g -O0 for debug info
C

#include <stdio.h>

int global_var = 42; // Global symbol at known address

int test_function(int param) { // Function symbol with parameters

    int local_var = param * 2; // Local variable with DWARF location info

    printf("Value: %d\n", local_var); // Line number mapping point

    return local_var; // Return statement at specific line
}

int main() { // Main function entry point

    return test_function(global_var); // Function call with known arguments
}
```

Address-to-Source Resolution Test

Address Type	Test Input	Expected Output	Accuracy Verification
Function entry point	Address of <code>test_function</code>	File: test_symbols.c, Line: 5, Function: test_function	Compare with <code>addr2line</code> tool output
Mid-function address	Address of printf call	File: test_symbols.c, Line: 7, Function: test_function	Verify line number accuracy
Global variable address	Address of <code>global_var</code>	Symbol: global_var, Type: int, Address: known value	Cross-check with symbol table
Invalid address	Address not in program	Error: <code>DEBUG_SYMBOL_NOT_FOUND</code>	Proper error handling

Name-to-Address Resolution Test

Symbol Name	Expected Resolution	Verification Method	Error Cases
"test_function"	Function entry address	Set breakpoint by name, verify hit	Function name not found
"main"	Main function address	Compare with ELF entry point calculation	Multiple symbols with same name
"global_var"	Variable address in data segment	Read value from resolved address	Variable optimized away
"nonexistent"	<code>DEBUG_SYMBOL_NOT_FOUND</code> error	Error handling correctness	Partial name matches

Milestone 4: Variable Inspection Verification

Variable inspection testing must verify the complete chain from DWARF location expressions through memory access to type-aware formatting. This requires test programs with various variable types and storage locations.

Variable Location Resolution Test

Variable Location Type	Test Variable	DWARF Expression	Expected Resolution
Stack-relative	Local int variable	DW_OP_fbreg + offset	Frame base + offset calculation
Register-stored	Function parameter in register	DW_OP_reg0 (or appropriate register)	Register number identification
Global memory	Static variable	Direct address	Memory address from symbol table
Complex expression	Struct member access	Multi-operation expression	Correct final address calculation

The test program should include variables stored in different locations:

```
// test_variables.c - compile with -g -O0
C

struct Point {
    int x, y;
};

static struct Point global_point = {10, 20}; // Global variable

int inspect_variables(int param) { // Parameter potentially in register
    int stack_var = param + 5; // Stack-allocated local variable
    struct Point local_point = {param, param * 2}; // Struct on stack
    int *pointer_var = &stack_var; // Pointer to local variable

    // Breakpoint location for inspection

    return stack_var + local_point.x; // Line with multiple variables in scope
}
```

Memory and Register Access Test

Access Type	Test Scenario	Expected Behavior	Error Handling
Stack memory read	Read local variable value	Correct value retrieved via ptrace	Invalid address detection
Register read	Read parameter from register	Correct register value returned	Register not available
Pointer dereferencing	Follow pointer to target	Target value read correctly	Null pointer detection
Array element access	Access array[index]	Correct element address calculation	Bounds checking

Type-Aware Value Formatting Test

Data Type	Raw Memory Bytes	Expected Display Format	Special Cases
int (32-bit)	0x2A 0x00 0x00 0x00	"42" (decimal)	Negative values, endianness
float	IEEE 754 bytes	"3.14159" (decimal notation)	NaN, infinity, very small values
char*	Address bytes	"hello world" (string content)	Null pointer, non-printable chars
struct Point	Member bytes concatenated	"Point { x: 10, y: 20 }"	Nested structs, padding bytes

Integration Testing Approach

Integration testing verifies that the four components work together correctly during realistic debugging scenarios. Unlike unit tests that isolate individual components, integration tests exercise complete workflows that span multiple components.

End-to-End Debugging Session Test

The comprehensive integration test simulates a complete debugging session from process start to variable inspection. This test exercises all four milestones in sequence and verifies their interactions.

Test Scenario: Debug Simple Calculator Program

Session Phase	User Action	Expected Component Interactions	Verification Points
Session start	debugger ./calculator	Process Controller starts target, Symbol Resolver loads debug info	Process in stopped state, symbols loaded
Set breakpoint	break add_numbers	Symbol Resolver finds function address, Breakpoint Manager sets INT3	Breakpoint confirmed at correct address
Run program	continue	Process Controller resumes execution	Program runs until breakpoint hit
Breakpoint hit	INT3 triggers	Process stops, Breakpoint Manager handles hit, restores instruction	User notified of breakpoint hit
Variable inspection	print param1	Variable Inspector locates variable, reads memory, formats output	Correct parameter value displayed
Step through code	step	Process Controller single-steps, Symbol Resolver provides source info	Next source line displayed
Continue execution	continue	Breakpoint Manager re-enables breakpoint, Process Controller resumes	Program completes successfully

This integration test requires a calculator program with predictable behavior:

```
// calculator.c - integration test target

#include <stdio.h>

int add_numbers(int a, int b) {
    int result = a + b; // Breakpoint location
    return result; // Step target location
}

int main() {
    int x = 10, y = 20;
    int sum = add_numbers(x, y);
    printf("Sum: %d\n", sum);
    return 0;
}
```

Component Interaction Verification

Integration testing must verify the correct flow of information between components, ensuring that data structures are properly shared and updated across component boundaries.

Breakpoint-Symbol Integration Test

Interaction Flow	Component A Action	Component B Response	Data Exchange Verification
Set breakpoint by name	User requests <code>break main</code>	Symbol Resolver finds main address → Breakpoint Manager sets INT3	Address correctly passed between components
Breakpoint hit notification	Breakpoint Manager detects hit	Symbol Resolver provides source location for hit address	Source info accurately displayed to user
Function stepping	Process Controller advances execution	Symbol Resolver tracks source line changes	Line numbers update correctly

Variable-Symbol Integration Test

Variable Access Pattern	Symbol Component Role	Variable Component Role	Memory Component Role	Success Criteria
Local variable by name	Resolve name to DWARF DIE	Extract location expression	Read from calculated address	Correct value displayed
Struct member access	Find struct type definition	Calculate member offset	Read member bytes	Individual fields shown
Pointer dereferencing	Resolve pointer type info	Calculate target address	Read target memory	Target value displayed

Error Propagation Testing

Integration testing must verify that errors are properly detected, classified, and handled across component boundaries. Error conditions often involve multiple components and require coordinated responses.

Cross-Component Error Scenarios

Error Condition	Origin Component	Affected Components	Expected Error Handling	Recovery Verification
Process exits unexpectedly	Process Controller detects exit	All components must handle invalid process	Graceful shutdown, resources cleaned	Debugger returns to command prompt
Breakpoint at invalid address	Symbol Resolver provides bad address	Breakpoint Manager fails to set	Error reported to user, no memory corruption	System remains stable
Memory access violation	Variable Inspector reads protected memory	Process Controller receives SIGSEGV	Error caught, debuggee state preserved	Debugging session continues
Corrupted debug information	Symbol Resolver encounters invalid DWARF	Variable Inspector can't resolve locations	Fallback to address-only mode	Basic debugging still functional

Test Program Design

Effective debugger testing requires carefully designed target programs that provide predictable, observable behavior. These programs serve as controlled test subjects that exercise specific debugger functionality while maintaining simplicity for verification.

Layered Test Program Architecture

Test programs should be designed in layers, from minimal functionality tests to complex integration scenarios. Each layer builds on the previous one while adding specific challenges for the debugger.

Layer 1: Minimal Test Programs

Program Type	Purpose	Key Features	Usage
Hello World	Basic process control	Simple main function, predictable execution	Process start, attach, continue, exit
Sleep Loop	Signal handling	Infinite loop with sleep calls	SIGINT handling, process interruption
Crash Generator	Error handling	Intentional segfaults, null dereferences	Signal delivery, crash analysis
Exit Codes	Process termination	Various exit codes and methods	Exit status handling, cleanup verification

Layer 2: Breakpoint Test Programs

Program Type	Breakpoint Challenges	Implementation Features	Verification Points
Linear Execution	Sequential breakpoints	Series of function calls	Multiple breakpoint hits
Loop with Breakpoint	Repeated breakpoint hits	Loop containing debugger trigger	Hit count accuracy
Recursive Function	Stack-aware breakpoints	Function that calls itself	Breakpoint in recursive context
Conditional Paths	Breakpoints in branches	if/else with different execution paths	Selective breakpoint triggering

Example recursive test program:

```
// recursive_test.c

int factorial(int n) {

    if (n <= 1) {

        return 1; // Base case breakpoint
    }

    return n * factorial(n - 1); // Recursive call breakpoint
}

int main() {

    int result = factorial(5); // Entry breakpoint

    return result;
}
```

Layer 3: Symbol Resolution Test Programs

Symbol resolution testing requires programs with rich debug information and predictable symbol layouts. The programs should include various symbol types and scoping scenarios.

Symbol Complexity	Program Elements	DWARF Features Tested	Verification Approach
Basic symbols	Global variables, simple functions	Function DIs, variable DIs	Name-to-address resolution
Scoped variables	Nested blocks, local variables	Lexical block DIs	Variable visibility by PC
Complex types	Structs, arrays, pointers	Type DIs, member DIs	Type information accuracy
Inline functions	Compiler-inlined code	Inlined subroutine DIs	Address range mapping

Layer 4: Variable Inspection Test Programs

Variable inspection testing requires programs that exercise different variable storage locations and type complexities. The programs should be compiled with minimal optimization to preserve variable locations.

Variable Category	Test Scenarios	Storage Locations	Type Complexities
Primitive types	int, float, char variables	Stack, register, global memory	Signed/unsigned, different sizes
Composite types	Structs, unions, arrays	Stack allocation, heap allocation	Nested structures, pointer chains
Dynamic data	Malloc'd memory, linked lists	Heap addresses, pointer dereferencing	Memory management, null pointers
Optimized variables	Compiler-optimized storage	Register promotion, dead code elimination	Optimized-away detection

Test Data Management

Test programs should include predictable data patterns that enable verification of debugger accuracy. The data should be designed for easy verification while exercising various debugger capabilities.

Predictable Data Patterns

Data Type	Test Pattern	Verification Method	Edge Cases Covered
Integer sequences	Powers of 2, prime numbers	Mathematical verification	Overflow, negative values
String patterns	Known text, character sequences	String comparison	Null terminators, Unicode
Pointer chains	Linked data structures	Traversal verification	Null pointers, circular references
Array structures	Multi-dimensional arrays	Index calculation verification	Bounds checking, stride calculation

Test Program Compilation Guidelines

Proper compilation of test programs is crucial for debugger testing. The compilation options must preserve debug information while maintaining predictable code generation.

Compilation Aspect	Recommended Setting	Rationale	Alternative Options
Debug information	<code>-g -gdwarf-4</code>	Full debug info in standard format	<code>-g3</code> for macro info
Optimization level	<code>-O0</code>	Preserve variable locations, prevent inlining	<code>-O1</code> for optimization testing
Symbol table	<code>-rdynamic</code>	Export symbols for dynamic loading	Strip symbols for minimal testing
Stack protection	<code>-fno-stack-protector</code>	Predictable stack layout	Enable for security testing

Test Environment Setup

The testing environment must provide isolated, reproducible conditions for debugger verification. This includes process isolation, resource management, and cleanup procedures.

Process Isolation Strategy

Isolation Aspect	Implementation Approach	Benefits	Verification Method
PID namespace	Use containers or chroot	Prevent interference with system processes	Process list isolation
File system	Temporary directories per test	Clean slate for each test	Directory cleanup verification
Signal handling	Block signals during setup	Prevent race conditions	Signal mask verification
Resource limits	ulimit constraints	Prevent resource exhaustion	Resource usage monitoring

Test Execution Framework

A robust test execution framework provides consistent test running, result verification, and failure diagnosis. The framework should handle both automated verification and manual inspection scenarios.

Framework Component	Responsibility	Implementation Requirements	Success Criteria
Test runner	Execute test sequences	Parallel execution, timeout handling	All tests complete within time limit
Result verification	Compare expected vs actual	Flexible comparison methods	Clear pass/fail determination
Failure diagnosis	Analyze test failures	Debug output capture, state dumps	Actionable failure information
Resource cleanup	Clean up after tests	Process termination, file removal	No resource leaks

Implementation Guidance

Testing a debugger requires a systematic approach that combines automated verification with manual inspection techniques. The testing infrastructure must handle real processes, memory access, and system interactions while providing reliable, repeatable results.

Technology Recommendations

Testing Component	Simple Option	Advanced Option	Recommended Choice
Test runner	Shell scripts with basic assertions	Custom C test framework	Shell scripts for milestone tests, C framework for integration
Target compilation	Manual gcc commands	Makefile with multiple targets	Makefile for consistent builds
Memory verification	Manual ptrace calls	Valgrind integration	ptrace for correctness, Valgrind for leaks
Process management	Basic fork/exec	Process groups and namespaces	Process groups for isolation

Recommended File Structure

```

project-root/
  tests/
    unit/           ← Component-specific tests
      process_control_test.c ← Milestone 1 verification
      breakpoint_test.c   ← Milestone 2 verification
      symbol_test.c     ← Milestone 3 verification
      variable_test.c   ← Milestone 4 verification
    integration/    ← Cross-component tests
      end_to_end_test.c ← Complete debugging session
      error_handling_test.c ← Error propagation tests
    targets/
      simple/          ← Test program sources
        hello.c, loop.c, crash.c
        breakpoint/     ← Breakpoint test programs
          linear.c, recursive.c, conditional.c
        symbols/         ← Symbol resolution test programs
          complex_types.c, scoped_vars.c
        variables/       ← Variable inspection test programs
          all_types.c, optimized.c
    helpers/          ← Test infrastructure
      test_framework.h ← Common test utilities
      process_utils.c  ← Process management helpers
      memory_verify.c ← Memory state verification
    scripts/
      run_milestone_tests.sh ← Milestone verification scripts
      build_targets.sh   ← Test program compilation
      verify_debugger.sh ← End-to-end verification

```

Test Framework Infrastructure

Complete test framework code for managing test processes and verifying results:

```
// tests/helpers/test_framework.h
```

C

```
#ifndef TEST_FRAMEWORK_H
```

```
#define TEST_FRAMEWORK_H
```

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
```

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <signal.h>
```

```
// Test result tracking
```

```
typedef struct {
```

```
    int total_tests;
```

```
    int passed_tests;
```

```
    int failed_tests;
```

```
    char current_test[256];
```

```
} TestResults;
```

```
// Test process management
```

```
typedef struct {
```

```
    pid_t target_pid;
```

```
    pid_t debugger_pid;
```

```
    int status;
```

```
    char executable_path[1024];
```

```
} TestProcess;
```

```
// Test assertion macros
```

```
#define TEST_ASSERT(condition, message) \
```

```
do { \
```

```
    if (!(condition)) { \
```

```
    printf("FAIL: %s - %s (line %d)\n", test_results.current_test, message, __LINE__); \
    test_results.failed_tests++; \
    return 0; \
} \
test_results.passed_tests++; \
return 1; \
} while(0)

#define TEST_ASSERT_EQUAL(expected, actual, message) \
TEST_ASSERT((expected) == (actual), message)

#define TEST_ASSERT_NOT_NULL(pointer, message) \
TEST_ASSERT((pointer) != NULL, message)

// Global test state

extern TestResults test_results;

// Test framework functions

void test_framework_init();
void test_framework_cleanup();
void test_run(const char* test_name, int (*test_func)());
void test_print_summary();

// Process management utilities

int test_process_create(const char* executable, char* const argv[], TestProcess* proc);
int test_process_attach_debugger(TestProcess* proc, const char* debugger_path);
int test_process_wait_for_signal(TestProcess* proc, int expected_signal, int timeout_ms);
void test_process_cleanup(TestProcess* proc);

// Memory verification utilities

int test_memory_read_byte(pid_t pid, uintptr_t address, uint8_t* value);
int test_memory_verify_pattern(pid_t pid, uintptr_t address, uint8_t* pattern, size_t length);
```

```
int test_memory_compare_before_after(pid_t pid, uintptr_t address, uint8_t* before, uint8_t* after,
size_t length);

#endif // TEST_FRAMEWORK_H
```

```
// tests/helpers/test_framework.c

#include "test_framework.h"

#include <errno.h>

#include <sys/ptrace.h>

TestResults test_results = {0};

void test_framework_init() {

    memset(&test_results, 0, sizeof(test_results));

    printf("Starting debugger test suite...\n");

}

void test_framework_cleanup() {

    // Clean up any remaining processes

    system("pkill -f test_target");

    printf("Test cleanup completed.\n");

}

void test_run(const char* test_name, int (*test_func)()) {

    strncpy(test_results.current_test, test_name, sizeof(test_results.current_test) - 1);

    test_results.total_tests++;




    printf("Running: %s... ", test_name);

    fflush(stdout);




    if (test_func()) {

        printf("PASS\n");

    } else {

        printf("FAIL\n");

    }

}
```

C

```
void test_print_summary() {

    printf("\n==== Test Summary ====\n");

    printf("Total tests: %d\n", test_results.total_tests);

    printf("Passed: %d\n", test_results.passed_tests);

    printf("Failed: %d\n", test_results.failed_tests);

    printf("Success rate: %.1f%%\n",
        (float)test_results.passed_tests / test_results.total_tests * 100);

}

int test_process_create(const char* executable, char* const argv[], TestProcess* proc) {

    proc->target_pid = fork();

    if (proc->target_pid == 0) {

        // Child process - become traceable and exec target

        if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) == -1) {

            perror("ptrace TRACEME");

            exit(1);

        }

        raise(SIGSTOP); // Stop for debugger attachment

        execvp(executable, argv);

        perror("exec");

        exit(1);

    } else if (proc->target_pid > 0) {

        // Parent process - wait for child to stop

        if (waitpid(proc->target_pid, &proc->status, 0) == -1) {

            perror("waitpid");

            return 0;

        }

        strncpy(proc->executable_path, executable, sizeof(proc->executable_path) - 1);

        return 1;

    } else {


```

```

    perror("fork");

    return 0;
}

}

int test_memory_read_byte(pid_t pid, uintptr_t address, uint8_t* value) {
    errno = 0;

    long data = ptrace(PTRACE_PEEKDATA, pid, (void*)address, NULL);

    if (data == -1 && errno != 0) {

        return 0; // Read failed
    }

    *value = (uint8_t)(data & 0xFF);

    return 1; // Success
}

int test_memory_verify_pattern(pid_t pid, uintptr_t address, uint8_t* pattern, size_t length) {
    for (size_t i = 0; i < length; i++) {

        uint8_t actual;

        if (!test_memory_read_byte(pid, address + i, &actual)) {

            return 0; // Read failed
        }

        if (actual != pattern[i]) {

            return 0; // Pattern mismatch
        }
    }

    return 1; // Pattern matches
}

```

Milestone Test Skeletons

Skeleton test code for each milestone with detailed TODO comments mapping to verification requirements:

```
// tests/unit/process_control_test.c

#include "../helpers/test_framework.h"
#include "../../src/process_control.h"

int test_process_attachment() {

    TestProcess proc;
    DebuggedProcess debug_proc;

    // TODO 1: Create simple test target that sleeps for 10 seconds
    char* argv[] = {"./targets/simple/sleep", "10", NULL};
    TEST_ASSERT(test_process_create("./targets/simple/sleep", argv, &proc),
               "Failed to create test target process");

    // TODO 2: Attach debugger to the sleeping process
    DebugResult result = process_attach(proc.target_pid, &debug_proc);
    TEST_ASSERT_EQUAL(DEBUG_SUCCESS, result, "Process attachment failed");

    // TODO 3: Verify process is in STOPPED state
    TEST_ASSERT_EQUAL(STOPPED, debug_proc.state, "Process not in stopped state");

    // TODO 4: Verify process shows as traced in /proc/PID/stat
    // Read /proc/<pid>/stat and check status field is 'T'

    // TODO 5: Clean up attached process
    test_process_cleanup(&proc);

    return 1;
}

int test_single_step_execution() {
    TestProcess proc;
    DebuggedProcess debug_proc;
```

```
struct user_regs_struct regs_before, regs_after;

// TODO 1: Create test target with known instruction sequence

char* argv[] = {"./targets/simple/arithmetic", NULL};

TEST_ASSERT(test_process_create("./targets/simple/arithmetic", argv, &proc),
            "Failed to create arithmetic test target");

// TODO 2: Attach and get initial register state

process_attach(proc.target_pid, &debug_proc);

register_reader_get_all(debug_proc.pid, &regs_before);

// TODO 3: Execute single step

DebugResult result = process_single_step(&debug_proc);

TEST_ASSERT_EQUAL(DEBUG_SUCCESS, result, "Single step failed");

// TODO 4: Get registers after step and verify IP advanced

register_reader_get_all(debug_proc.pid, &regs_after);

TEST_ASSERT(regs_after.rip > regs_before.rip, "Instruction pointer did not advance");

// TODO 5: Verify exactly one instruction executed (not more, not less)

// This requires disassembling instruction at regs_before.rip to get length

test_process_cleanup(&proc);

return 1;
}
```

```
// tests/unit/breakpoint_test.c

#include "../helpers/test_framework.h"

#include "../../src/breakpoint_manager.h"

int test_software_breakpoint_setting() {

    TestProcess proc;

    DebuggedProcess debug_proc;

    BreakpointList bp_list;

    uint8_t original_byte, current_byte;

    // TODO 1: Create test target and attach debugger

    char* argv[] = {"./targets/breakpoint/linear", NULL};

    test_process_create("./targets/breakpoint/linear", argv, &proc);

    process_attach(proc.target_pid, &debug_proc);

    // TODO 2: Read original instruction byte at target address

    uintptr_t target_addr = 0x401000; // Known function address from test program

    TEST_ASSERT(test_memory_read_byte(debug_proc.pid, target_addr, &original_byte),

               "Failed to read original instruction byte");

    // TODO 3: Set breakpoint at target address

    DebugResult result = breakpoint_set(debug_proc.pid, &bp_list, target_addr, "test breakpoint");

    TEST_ASSERT_EQUAL(DEBUG_SUCCESS, result, "Breakpoint setting failed");

    // TODO 4: Verify INT3 (0xCC) was written to memory

    TEST_ASSERT(test_memory_read_byte(debug_proc.pid, target_addr, &current_byte),

               "Failed to read modified instruction byte");

    TEST_ASSERT_EQUAL(0xCC, current_byte, "INT3 instruction not written");

    // TODO 5: Verify original byte was saved in breakpoint structure
```

```
Breakpoint* bp = breakpoint_find(&bp_list, target_addr);

TEST_ASSERT_NOT_NULL(bp, "Breakpoint not found in list");

TEST_ASSERT_EQUAL(original_byte, bp->original_byte, "Original byte not saved correctly");


test_process_cleanup(&proc);

return 1;

}

int test_breakpoint_hit_detection() {

TestProcess proc;

DebuggedProcess debug_proc;

BreakpointList bp_list;

// TODO 1: Set up test target with breakpoint at main function

char* argv[] = {"./targets/breakpoint/linear", NULL};

test_process_create("./targets/breakpoint/linear", argv, &proc);

process_attach(proc.target_pid, &debug_proc);

// TODO 2: Set breakpoint at main function entry

uintptr_t main_addr = 0x401020; // Address from symbol table

breakpoint_set(debug_proc.pid, &bp_list, main_addr, "main entry");

// TODO 3: Continue execution until breakpoint hit

DebugResult result = process_continue(&debug_proc);

TEST_ASSERT_EQUAL(DEBUG_SUCCESS, result, "Continue execution failed");

// TODO 4: Wait for SIGTRAP signal indicating breakpoint hit

int status;

waitpid(debug_proc.pid, &status, 0);

TEST_ASSERT(WIFSTOPPED(status), "Process not stopped");
```

```
TEST_ASSERT_EQUAL(SIGTRAP, WSTOPSIG(status), "Expected SIGTRAP signal");

// TODO 5: Handle breakpoint hit and verify hit count incremented

result = breakpoint_handle_hit(debug_proc.pid, &bp_list, main_addr);

TEST_ASSERT_EQUAL(DEBUG_SUCCESS, result, "Breakpoint hit handling failed");

Breakpoint* bp = breakpoint_find(&bp_list, main_addr);

TEST_ASSERT_EQUAL(1, bp->hit_count, "Hit count not incremented");

test_process_cleanup(&proc);

return 1;

}
```

Milestone Checkpoint Scripts

```
#!/bin/bash                                BASH

# tests/scripts/run_milestone_tests.sh

set -e # Exit on any error

MILESTONE=${1:-"all"}

TEST_DIR=$(dirname "$0")/..
BUILD_DIR="$TEST_DIR/..build"

echo "==== Debugger Milestone Testing ===="

echo "Testing milestone: $MILESTONE"
echo "Build directory: $BUILD_DIR"

# Ensure test targets are built

echo "Building test targets..."

cd "$TEST_DIR/targets"

make clean && make all

# Build debugger if not already built

if [ ! -f "$BUILD_DIR/debugger" ]; then
    echo "Building debugger..."
    cd "$TEST_DIR/.."
    make clean && make all
fi

# Function to run milestone tests

run_milestone_test() {

    local milestone=$1

    local test_program="$BUILD_DIR/test_milestone_$milestone"

    echo "--- Running Milestone $milestone Tests ---"
```

```
if [ ! -f "$test_program" ]; then
    echo "ERROR: Test program $test_program not found"
    return 1
fi

# Run the test with timeout

if timeout 30 "$test_program"; then
    echo "✓ Milestone $milestone tests PASSED"
    return 0
else
    echo "✗ Milestone $milestone tests FAILED"
    return 1
fi
}

# Run requested milestone tests

case $MILESTONE in
"1"|"process")
    run_milestone_test 1
    ;;
"2"|"breakpoint")
    run_milestone_test 2
    ;;
"3"|"symbol")
    run_milestone_test 3
    ;;
"4"|"variable")
    run_milestone_test 4
    ;;

```

```

"all")

    for i in 1 2 3 4; do

        run_milestone_test $i || exit 1

    done

    echo "🎉 All milestone tests PASSED!"

    ;;

*)

    echo "Usage: $0 [1|2|3|4|process|breakpoint|symbol|variable|all]"

    exit 1

    ;;

esac

```

Debugging Tips for Test Failures

Test Failure Symptom	Likely Cause	Diagnostic Steps	Fix Approach
Process attachment fails with EPERM	Permission/security restrictions	Check if ptrace is disabled: <code>cat /proc/sys/kernel/yama/ptrace_scope</code>	Set <code>ptrace_scope</code> to 0 or run as root
Breakpoint not hit, process continues	INT3 not written or wrong address	Use <code>objdump -d</code> to verify target address, check memory with <code>gdb</code>	Verify symbol resolution and address calculation
Single step advances too far	PTRACE_SINGLESTEP skipping instructions	Check for signal handling issues, verify wait status	Ensure proper signal forwarding
Memory reads return garbage	Address calculation wrong or process dead	Verify process still alive, check address alignment	Fix address calculation, handle process death
Tests hang indefinitely	Deadlock in ptrace operations	Use <code>strace -p <test_pid></code> to see system calls	Add timeouts, fix signal handling

Debugging Guide

Milestone(s): All milestones — this section provides comprehensive debugging strategies and troubleshooting techniques for issues encountered while building the debugger itself

The Diagnostic Laboratory Analogy

Think of debugging a debugger as running a medical diagnostic laboratory. When a patient (your debugger) shows symptoms (crashes, incorrect behavior, or unexpected results), you need multiple diagnostic tools and techniques to identify the root cause. Just as a doctor uses blood tests, X-rays, and MRIs to examine different aspects of patient health, you'll use tools like `strace`, `objdump`, and `gdb` to examine different layers of your debugger's operation. The key insight is that debugging a debugger requires meta-debugging — you're simultaneously operating at multiple abstraction levels, examining both the debugger's interaction with the operating system and its interpretation of binary formats.

Building a debugger presents unique debugging challenges because you're working with multiple interacting systems: your debugger code, the target process, the operating system's process control mechanisms, and complex binary formats. When something goes wrong, the failure could originate from any of these layers, and symptoms often manifest far from their root causes. This section provides systematic approaches for diagnosing problems at each layer, with specific attention to the most common failure modes encountered during debugger development.

The debugging process follows a structured diagnostic approach: symptom identification, hypothesis formation, targeted investigation using appropriate tools, root cause confirmation, and systematic fix implementation. Each subsection provides symptom-cause-fix tables that map observable behaviors to likely underlying problems, along with specific diagnostic procedures and remediation steps.

Process Control Debugging

The Puppet Show Analogy

Process control debugging is like diagnosing problems in a puppet show where the puppeteer (your debugger) controls marionettes (target processes) through strings (ptrace system calls). When the puppet doesn't respond correctly, the problem could be with the puppeteer's commands, broken or tangled strings, or issues with the puppet itself. The complexity arises because you're debugging across process boundaries, dealing with timing-sensitive operations, and managing shared state between multiple processes.

The most challenging aspect of process control debugging is that failures often involve race conditions, signal timing, and process state synchronization issues that can be difficult to reproduce. Your debugger must coordinate with the kernel's process scheduler while handling asynchronous signals, making many bugs timing-dependent and environment-sensitive.

Common Process Control Symptoms and Solutions

Symptom	Likely Cause	How to Diagnose	Fix
<code>ptrace(PTRACE_ATTACH, pid, 0, 0)</code> returns -1 with EPERM	Insufficient permissions or target process not traceable	Check process ownership with <code>ps -o pid,uid,gid,cmd -p <pid></code> . Verify debugger runs as same user or root	Run debugger with appropriate privileges or check if target process has <code>PR_SET_DUMPABLE</code> disabled
<code>ptrace(PTRACE_ATTACH, pid, 0, 0)</code> returns -1 with ESRCH	Target process doesn't exist or has already exited	Verify process exists with <code>kill -0 <pid></code> . Check <code>/proc/<pid></code> directory existence	Add process existence check before ptrace calls. Handle process exit gracefully
Debugger hangs indefinitely after <code>ptrace(PTRACE_CONT, pid, 0, 0)</code>	Target process hit breakpoint or signal but debugger missed the stop notification	Use <code>strace -p <debugger_pid></code> to see if debugger is blocked in <code>waitpid()</code> . Check if <code>SIGCHLD</code> was delivered	Implement proper signal handling with <code>sigaction()</code> . Use <code>waitpid()</code> with <code>WNOHANG</code> in signal handler
<code>waitpid()</code> returns -1 with EINTR repeatedly	Signal interruption during wait, common in interactive debuggers	Check if <code>SIGINT</code> or other signals are being delivered to debugger process	Use <code>waitpid()</code> in loop, retrying on <code>EINTR</code> , or block signals during wait with <code>sigprocmask()</code>
Single-step execution skips multiple instructions	<code>PTRACE_SINGLESTEP</code> not supported on architecture or kernel issue	Check architecture support. Use <code>strace</code> to verify <code>PTRACE_SINGLESTEP</code> call succeeds	Fall back to breakpoint-based single-stepping by setting temporary breakpoint at next instruction
Target process becomes zombie after debugger exit	Debugger exits without detaching from target process	Check process state with <code>ps aux grep <pid></code> . Look for <code>Z</code> in state column	Always call <code>ptrace(PTRACE_DETACH, pid, 0, 0)</code> in cleanup code and signal handlers
<code>PTRACE_PEEKDATA</code> returns valid data but <code>PTRACE_POKEDATA</code> fails with EFAULT	Target address not mapped in target process memory space	Check <code>/proc/<pid>/maps</code> to verify address is within valid memory region	Validate addresses against memory map before write attempts. Handle memory allocation in target process
Debugger receives <code>SIGTRAP</code> but target process wasn't single-stepping	Target process hit software interrupt (INT3) from another source	Examine instruction at current IP with <code>ptrace(PTRACE_PEEKDATA)</code> to see if it's 0xCC	Check if instruction is your breakpoint or external INT3. Handle accordingly

Process Attachment and Lifecycle Issues

Process attachment failures represent the most fundamental category of process control problems. When `process_attach()` fails, the issue typically stems from permission problems, process state conflicts, or resource

limitations. The diagnostic approach involves examining the target process state, verifying permissions, and checking for conflicting tracers.

Permission-related attachment failures occur when the debugger lacks sufficient privileges to trace the target process. This commonly happens when attempting to debug processes owned by different users, system processes, or processes that have explicitly disabled tracing through security mechanisms. The diagnostic procedure involves checking the effective user ID of both processes and examining security policies that might prevent attachment.

Detailed Diagnostic Procedure for Attachment Failures:

1. Verify target process existence using `kill -0 <pid>` system call
2. Check process ownership and permissions using `stat /proc/<pid>`
3. Examine process security context with `cat /proc/<pid>/status | grep -E "(Uid|Gid|TracerPid)"`
4. Test basic ptrace functionality with `ptrace(PTRACE_ATTACH)` followed by immediate `ptrace(PTRACE_DETACH)`
5. Check for existing tracers by examining `TracerPid` field in `/proc/<pid>/status`
6. Verify system-level tracing permissions in `/proc/sys/kernel/yama/ptrace_scope`

Key Insight: Process attachment failures often cascade into subsequent debugging problems. Always verify successful attachment before attempting any other ptrace operations, and implement robust error handling that distinguishes between transient and permanent attachment failures.

Signal Handling Synchronization Problems

Signal handling represents one of the most complex aspects of process control debugging because it involves coordinating between your debugger process, the target process, and the kernel's signal delivery mechanisms. The fundamental challenge is that signals are asynchronous events that can arrive at any time, potentially disrupting the debugger's control flow and leaving processes in inconsistent states.

The most common signal handling problems involve missed `SIGCHLD` notifications, incorrect signal forwarding, and race conditions between signal delivery and `waitpid()` calls. These issues manifest as hangs, missed breakpoints, or incorrect process state transitions.

Signal Handling Diagnostic Framework:

Signal Type	Expected Behavior	Diagnostic Command	Common Problems
<code>SIGCHLD</code>	Debugger receives notification when target process stops	<code>strace -e signal=chld -p <debugger_pid></code>	Signal ignored or handler not installed
<code>SIGTRAP</code>	Delivered when breakpoint hit or single-step completes	Check <code>WSTOPSIG(status)</code> in <code>waitpid</code> status	Confused with other trap causes
<code>SIGSTOP / SIGCONT</code>	Process control signals for pause/resume	<code>kill -STOP <pid></code> followed by status check	Incorrectly forwarded or blocked
<code>SIGSEGV / SIGBUS</code>	Target process memory access violations	Examine fault address with <code>siginfo_t</code>	Debugger causes fault through invalid memory access

The race condition between signal delivery and process state checking represents a particularly subtle class of bugs. Consider this scenario: the target process hits a breakpoint and sends `SIGCHLD` to the debugger, but before the debugger calls `waitpid()`, another signal arrives that changes the process state. The debugger's `waitpid()` call may return status information that doesn't match the actual current state, leading to incorrect state transitions.

Critical Pattern: Always use `waitpid()` with `WNOHANG` in signal handlers to avoid blocking, and implement a main event loop that regularly polls for process state changes independent of signal delivery. This provides resilience against missed signals and race conditions.

Memory Access and Address Space Issues

Memory access problems in process control occur when the debugger attempts to read or write target process memory through `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` operations. These failures typically manifest as `EFAULT` errors, incorrect data reads, or crashes in either the debugger or target process.

The root causes of memory access problems include invalid addresses, unmapped memory regions, permission violations, and alignment issues. Diagnostic procedures must examine both the debugger's address calculations and the target process's memory layout to identify the source of problems.

Memory Access Diagnostic Checklist:

1. **Address Validation:** Verify addresses fall within mapped regions using `/proc/<pid>/maps`
2. **Alignment Verification:** Check that addresses meet architecture alignment requirements
3. **Permission Analysis:** Examine memory region permissions (read/write/execute) from memory maps
4. **Cache Consistency:** Verify memory cache invalidation after writes that modify executable code
5. **Endianness Handling:** Confirm byte order conversion for multi-byte values on cross-architecture debugging

The memory mapping verification process involves parsing `/proc/<pid>/maps` to understand the target process's address space layout. Each line in this file represents a memory mapping with start address, end address, permissions, and backing file information. Debugger memory access should only target addresses within these mapped regions with appropriate permissions.

Address space layout randomization (ASLR) introduces additional complexity for memory access debugging. When ASLR is enabled, the same program will have different memory layouts on each execution, making address calculations more complex and potentially invalidating cached address information.

Breakpoint Issues

The Tripwire Laboratory Analogy

Debugging breakpoint problems is like managing a laboratory full of delicate tripwires that must trigger reliably without interfering with each other. Each breakpoint is a precisely placed sensor (INT3 instruction) that must detect when execution passes through a specific location, record the event, temporarily disable itself to allow continued execution, then re-enable itself for future detection. The complexity arises because you're modifying the target program's executable code in real-time while it's running, requiring perfect coordination between instruction patching, execution control, and state restoration.

Breakpoint debugging involves multiple interacting systems: instruction encoding, memory management, execution control, and signal handling. When breakpoints fail, the symptoms often appear far from the root cause because instruction modification can affect program behavior in subtle ways, and timing issues can create race conditions between breakpoint operations.

Breakpoint Setting and Management Problems

Breakpoint setting failures represent the most fundamental class of breakpoint problems. When `breakpoint_set()` fails to establish a working breakpoint, the issue typically involves address validation, instruction boundary detection, memory access permissions, or existing breakpoint conflicts.

Symptom	Root Cause	Diagnostic Steps	Resolution Strategy
<code>breakpoint_set()</code> succeeds but breakpoint never triggers	Breakpoint set at wrong address or in non-executable code	Use <code>objdump -d <executable></code> to verify instruction at target address. Check <code>/proc/<pid>/maps</code> for executable permissions	Recalculate address using symbol table. Verify target address contains executable instructions
Target process crashes with <code>SIGILL</code> after setting breakpoint	Breakpoint set in middle of multi-byte instruction	Disassemble surrounding instructions with <code>objdump -d .</code> . Check instruction boundaries	Use disassembler to find instruction start addresses. Never set breakpoints at non-instruction boundaries
Breakpoint triggers but target process doesn't stop	<code>SIGTRAP</code> delivered but debugger doesn't handle signal correctly	Use <code>strace -e signal=trap -p <debugger_pid></code> to verify signal delivery. Check signal handler registration	Implement proper <code>SIGTRAP</code> handler with <code>sigaction()</code> . Verify <code>waitpid()</code> status checking
Multiple breakpoints interfere with each other	Breakpoints set too close together or overlapping instruction sequences	Examine memory around breakpoint addresses with <code>ptrace(PTRACE_PEEKDATA)</code> . Check for 0xCC bytes	Maintain minimum distance between breakpoints. Track original instruction bytes per breakpoint
Breakpoint works once then stops triggering	Original instruction not properly restored after first hit	Check memory at breakpoint address after hit. Should show original instruction, not 0xCC	Implement restore-step-repatch sequence correctly. Store original byte per breakpoint
<code>PTRACE_POKEDATA</code> fails when setting breakpoint	Insufficient permissions or target address not writable	Check memory region permissions in <code>/proc/<pid>/maps</code> . Verify address is in writable region	Target read-only code sections. May need to change memory protection first

The instruction boundary detection problem deserves special attention because x86 and x86_64 architectures use variable-length instructions. Setting a breakpoint in the middle of a multi-byte instruction corrupts the instruction stream and typically results in `SIGILL` (illegal instruction) when the target process attempts to execute the partially overwritten instruction.

Instruction Boundary Diagnostic Process:

1. **Disassemble Target Region:** Use `objdump -d <executable>` to examine instructions around target address
2. **Verify Instruction Start:** Confirm target address corresponds to first byte of complete instruction
3. **Check Instruction Length:** Determine how many bytes the instruction occupies
4. **Validate No Overlap:** Ensure no existing breakpoints within instruction length of target
5. **Examine Machine Code:** Use `ptrace(PTRACE_PEEKDATA)` to read actual instruction bytes from process memory

The restore-step-repatch sequence represents the core breakpoint management algorithm and is a frequent source of bugs. This sequence must execute atomically from the target process's perspective while handling potential interruptions from signals or other events.

Critical Sequence: When breakpoint hits: 1) Restore original instruction byte, 2) Adjust instruction pointer back by one, 3) Single-step exactly one instruction, 4) Re-patch with INT3, 5) Continue execution. Any deviation from this sequence can cause missed breakpoints or process corruption.

Breakpoint Hit Detection and State Management

Breakpoint hit detection involves correctly interpreting `SIGTRAP` signals and determining which breakpoint (if any) caused the trap. This process is complicated by the fact that `SIGTRAP` can be generated by sources other than breakpoints, including single-step execution, hardware debug registers, and explicit INT3 instructions in the target program.

The state management challenge involves tracking multiple breakpoints simultaneously while maintaining consistency between the debugger's internal state and the target process's actual memory contents. Race conditions can occur when multiple threads access breakpoint data structures or when signals arrive during breakpoint operations.

Breakpoint State Consistency Verification:

State Component	Verification Method	Expected Behavior	Inconsistency Symptoms
Internal breakpoint list	Iterate through <code>BreakpointList</code> and check each entry	Every enabled breakpoint has valid address and saved original byte	Breakpoints in list don't correspond to INT3 in memory
Target process memory	Read bytes at each breakpoint address with <code>ptrace(PTRACE_PEEKDATA)</code>	Enabled breakpoints show 0xCC, disabled show original instruction	Memory contains unexpected byte values
Hit detection state	Check <code>hit_breakpoint</code> field after <code>SIGTRAP</code>	Points to breakpoint that caused trap, or NULL for non-breakpoint traps	Wrong breakpoint identified or NULL when breakpoint actually hit
Instruction pointer	Read IP register with <code>ptrace(PTRACE_GETREGS)</code>	Points to byte after INT3 instruction when breakpoint hits	IP points to unexpected location

The instruction pointer adjustment represents a subtle but critical aspect of breakpoint hit handling. When the processor executes an INT3 instruction, it increments the instruction pointer past the INT3 before delivering `SIGTRAP`. To continue

execution from the original instruction, the debugger must decrement the instruction pointer by one byte before restoring the original instruction.

Hit Detection State Machine:

Current State	Event	Next State	Actions Required
RUNNING	SIGTRAP received	STOPPED	Check IP, identify breakpoint, adjust IP back by 1
STOPPED	Continue command	RUNNING	Restore original instruction, single-step, re-patch
SINGLE_STEPPING	SIGTRAP received	STOPPED	Re-patch breakpoint, continue normal execution
STOPPED	Remove breakpoint	STOPPED	Remove from list, restore original byte permanently

Multi-Breakpoint Coordination Issues

Managing multiple breakpoints simultaneously introduces coordination challenges that don't exist with single breakpoints. The primary issues involve breakpoint overlap, shared instruction sequences, and consistent state updates across the entire breakpoint collection.

Breakpoint overlap occurs when two breakpoints are set close enough to interfere with each other's operation. This can happen when breakpoints target adjacent instructions, overlapping instruction sequences in loops, or when function calls create temporary breakpoints near existing ones.

Multi-Breakpoint Conflict Resolution:

- Proximity Detection:** Check distance between new and existing breakpoint addresses
- Instruction Overlap Analysis:** Verify no breakpoint falls within instruction boundaries of another
- Execution Path Mapping:** Consider control flow that might cause rapid breakpoint transitions
- State Serialization:** Ensure breakpoint enable/disable operations are atomic across all breakpoints
- Hit Priority Resolution:** Define behavior when multiple breakpoints could trigger simultaneously

The shared instruction sequence problem occurs in programs with computed jumps, function pointers, or dynamically generated code where the same instruction sequence might be reached through different execution paths. Each path might have associated breakpoints that need independent management despite targeting the same memory locations.

Design Insight: Implement breakpoint reference counting for shared instruction locations. Multiple logical breakpoints can target the same address, but only one physical INT3 patch is needed. The breakpoint triggers when any associated logical condition is met.

Symbol Resolution Problems

The Archaeological Excavation Analogy

Debugging symbol resolution problems is like troubleshooting archaeological excavation equipment that must decode ancient texts in multiple languages and formats. Your symbol resolver acts as a combination translator and artifact analyzer, attempting to decode DWARF debug information and ELF symbol tables that may be incomplete, corrupted, or

encoded in unexpected variants. When symbol resolution fails, you're often dealing with format specification ambiguities, compiler variations, or incomplete debug information that requires forensic analysis to understand.

The complexity of symbol resolution debugging stems from the layered nature of the problem: ELF file format parsing, DWARF debug information interpretation, symbol table construction, and address-to-source mapping. Each layer can introduce its own class of failures, and symptoms at one level often indicate problems at a different level entirely.

ELF File Format and Section Parsing Issues

ELF file parsing problems represent the foundation layer of symbol resolution issues. When the debugger cannot correctly parse ELF headers or locate debug sections, all subsequent symbol resolution operations will fail. These problems often manifest as segmentation faults, incorrect symbol addresses, or missing debug information.

Problem Category	Specific Symptom	Diagnostic Approach	Resolution Method
File access	<code>elf_open()</code> returns file open error	Check file existence with <code>ls -l</code> . Verify read permissions with <code>stat</code>	Ensure executable exists and is readable. Handle missing files gracefully
Header validation	Invalid ELF magic number or corrupted header	Use <code>readelf -h <executable></code> to verify ELF header. Check file integrity with <code>file</code> command	Validate ELF magic bytes (0x7f, 'E', 'L', 'F'). Handle non-ELF files gracefully
Section enumeration	Cannot find <code>.debug_info</code> or <code>.syms</code> sections	Use <code>readelf -S <executable></code> to list all sections. Check for debug info with <code>objdump --debugging</code>	Verify executable compiled with debug information (-g flag). Handle missing debug sections
Section parsing	Incorrect section size or offset calculations	Use <code>readelf -x .debug_info <executable></code> to dump section contents in hex	Validate section header fields. Check for 32-bit vs 64-bit ELF format differences
String table access	Corrupted or missing string table references	Use <code>readelf -p .shstrtab <executable></code> to examine string table contents	Validate string table offsets before dereferencing. Handle truncated string tables

The ELF format variant detection represents a common source of parsing errors. The debugger must correctly identify whether the target executable uses 32-bit or 64-bit ELF format, little-endian or big-endian byte ordering, and which version of the ELF specification. Incorrect format detection leads to misinterpreted header fields and invalid memory access attempts.

ELF Format Detection Procedure:

- 1. Read ELF Identity Block:** First 16 bytes of file contain format identification
- 2. Validate Magic Number:** Bytes 0-3 must be [0x7f, 'E', 'L', 'F']
- 3. Check Architecture Class:** Byte 4 indicates 32-bit (1) or 64-bit (2) format
- 4. Determine Byte Order:** Byte 5 indicates little-endian (1) or big-endian (2)
- 5. Verify Version:** Byte 6 should be 1 for current ELF version
- 6. Select Structure Sizes:** Use appropriate `Elf32_*` or `Elf64_*` structure definitions

The section header parsing process must account for the fact that section headers are not necessarily stored in file offset order, and the section header string table might not be the first string table encountered. The section header string table index is specified in the ELF header's `e_shstrndx` field.

Common Mistake: Assuming section headers appear in a specific order or that the first string table encountered is the section header string table. Always use the `e_shstrndx` index from the ELF header to locate the correct string table.

DWARF Debug Information Parsing Failures

DWARF debug information parsing represents the most complex aspect of symbol resolution debugging. DWARF is a sophisticated format with multiple versions, complex encoding schemes, and intricate relationships between different debug information sections. Parsing failures can result from version mismatches, encoding errors, or incomplete debug information.

The DWARF parsing process involves several interdependent stages: compilation unit enumeration, debug information entry (DIE) parsing, abbreviation table interpretation, and attribute value extraction. Each stage depends on successful completion of the previous stages, making error diagnosis particularly challenging.

DWARF Parsing Diagnostic Framework:

Parsing Stage	Validation Checks	Error Indicators	Recovery Strategies
Compilation unit header	Version number, unit length, abbreviation offset	Invalid version (not 2, 3, 4, or 5), length exceeds section size	Skip invalid compilation units, continue with next unit
Abbreviation table	Valid abbreviation codes, attribute specifications	Duplicate abbreviation codes, invalid attribute forms	Build partial table, mark invalid entries as unusable
DIE tree structure	Parent-child relationships, sibling chains	Circular references, invalid nesting	Detect cycles, prune invalid subtrees
Attribute values	Form-specific value encoding, reference validity	Invalid forms, out-of-bounds references	Use default values, mark attributes as unavailable

The DWARF version compatibility issue requires special attention because different compilers produce different DWARF versions, and each version introduces new features while potentially changing existing structures. Your debugger should detect the DWARF version from each compilation unit header and adapt its parsing strategy accordingly.

DWARF Version Compatibility Matrix:

DWARF Version	GCC Support	Clang Support	Key Differences	Parsing Strategy
DWARF 2	GCC 3.x+	Limited	Basic DIE structure, simple location expressions	Full support required
DWARF 3	GCC 4.x+	Clang 3.x+	Namespace support, improved location lists	Extend DWARF 2 parser
DWARF 4	GCC 4.5+	Clang 3.x+	Type units, improved line number program	Add type unit handling
DWARF 5	GCC 5+	Clang 8+	Split debug info, enhanced location lists	Optional advanced features

The abbreviation table parsing represents a critical dependency for all subsequent DIE parsing. The abbreviation table defines templates that specify the structure of debug information entries, including which attributes are present and how they are encoded. Corruption or misinterpretation of the abbreviation table causes failures throughout the DIE parsing process.

Critical Dependency: Always validate abbreviation table entries before using them to parse DIEs. Invalid abbreviation entries can cause buffer overruns, infinite loops, or incorrect attribute interpretation that corrupts the entire symbol table.

Address-to-Source and Name-to-Address Mapping Issues

The mapping between memory addresses and source code locations involves interpreting the DWARF line number program, which is a compact bytecode representation of the correspondence between machine instructions and source lines. Line number program interpretation failures result in incorrect source location displays and inability to set breakpoints by line number.

Name-to-address resolution involves building hash tables or search trees from function and variable symbols extracted from DWARF subprogram and variable DIEs. The challenge lies in handling scope resolution, function overloading, and namespace disambiguation while providing efficient lookup performance.

Address Mapping Diagnostic Procedures:

- Line Number Program Validation:** Verify line number program opcodes are valid and state machine transitions are consistent
- Address Range Verification:** Confirm mapped addresses fall within executable memory regions
- Source File Resolution:** Validate source file paths and handle relative vs absolute path resolution
- Duplicate Address Handling:** Manage multiple source locations that map to same address (inlined functions, optimized code)
- Gap Analysis:** Identify address ranges with no corresponding source information

The line number program state machine maintains current values for address, file, line, and column as it processes opcodes. State inconsistencies can arise from corrupted opcodes, incorrect initial state, or implementation bugs in the state machine logic.

Line Number Program State Tracking:

State Variable	Valid Range	Update Rules	Error Conditions
address	Within text section bounds	Monotonically increasing within sequence	Address decreases or exceeds section bounds
file	1 to file table size	Set by DW_LNS_set_file opcode	File number 0 or exceeds table size
line	1 to reasonable maximum	Incremented by line opcodes	Line number 0 or exceeds source file length
column	0 to line length	Set by column opcodes	Column exceeds reasonable line length

The symbol hash table construction process must handle name collisions, scope qualification, and memory management for dynamically allocated symbol entries. Hash table performance directly affects debugger responsiveness during symbol lookup operations.

Performance Consideration: Use separate hash tables for different symbol types (functions, variables, types) to reduce collision rates and improve lookup performance. Consider symbol name prefixing to handle scope resolution efficiently.

Debugging Tools and Techniques

The Multi-Instrument Observatory Analogy

Debugging a debugger requires assembling a sophisticated observatory with multiple specialized instruments, each designed to observe different aspects of the system's behavior. Just as astronomers use radio telescopes, optical telescopes, and X-ray detectors to study celestial objects from different perspectives, debugger developers must use system call tracers, binary analysis tools, and meta-debuggers to examine their debugger's operation across multiple system layers simultaneously.

The key insight is that no single tool provides complete visibility into debugger behavior. System call tracing reveals interaction with the kernel, binary analysis tools expose file format interpretation issues, disassemblers show instruction-level effects, and meta-debugging provides high-level control flow analysis. Effective debugger debugging requires orchestrating these tools to build a complete picture of system behavior.

System Call Tracing with strace

System call tracing represents the most fundamental debugging technique for process control issues because it provides complete visibility into your debugger's interaction with the kernel. Since ptrace-based debugging relies entirely on system calls, `strace` output reveals the exact sequence of kernel interactions and their success or failure.

The `strace` tool captures every system call made by the debugger process, including parameters, return values, and error conditions. This information is invaluable for diagnosing permission problems, signal handling issues, and process synchronization failures that are difficult to observe through other means.

Essential strace Command Patterns:

Debugging Scenario	strace Command	Key Information Revealed
Process attachment issues	<code>strace -e ptrace,wait4 ./debugger target</code>	ptrace call success/failure, wait behavior
Signal handling problems	<code>strace -e signal=all -p <debugger_pid></code>	Signal delivery timing and handler execution
Memory access failures	<code>strace -e ptrace ./debugger followed by filtering PTRACE_PEEKDATA / PTRACE_POKEDATA</code>	Memory access patterns and failure points
File parsing issues	<code>strace -e open,read,lseek,mmap ./debugger</code>	ELF file access patterns and read operations
Performance analysis	<code>strace -c ./debugger target</code>	System call frequency and timing statistics
Multi-process debugging	<code>strace -f -e ptrace,wait4 ./debugger</code>	Parent-child process coordination

The process attachment diagnostic sequence using `strace` involves monitoring the complete attachment lifecycle from initial `ptrace(PTRACE_ATTACH)` through signal handling and process state synchronization. This sequence reveals timing issues, permission problems, and signal delivery failures that cause attachment to fail.

strace-based Attachment Diagnosis:

1. **Monitor Initial Attachment:** Look for `ptrace(PTRACE_ATTACH, <pid>, 0, 0)` and its return value
2. **Verify Signal Delivery:** Check for `--- SIGCHLD` delivery after attachment
3. **Examine Wait Behavior:** Analyze `wait4()` calls and their return values
4. **Track State Transitions:** Follow `ptrace(PTRACE_CONT)` and subsequent wait operations
5. **Identify Error Patterns:** Look for repeated system call failures or unexpected return values

The signal handling analysis using `strace` provides crucial insights into the asynchronous aspects of debugger operation. Signal delivery timing, handler execution, and signal mask manipulation are all visible through system call tracing.

Diagnostic Pattern: Use `strace -e signal=all` to capture all signal-related activity, then correlate signal delivery times with debugger state changes. Look for missing `SIGCHLD` signals or signals delivered to the wrong process.

Binary Analysis with objdump and readelf

Binary analysis tools provide essential capabilities for diagnosing symbol resolution and breakpoint placement issues. These tools allow you to examine the target executable's structure independently of your debugger's parsing logic, providing ground truth for comparison with your debugger's interpretation.

The `objdump` tool excels at disassembly and section analysis, while `readelf` provides detailed ELF structure examination. Together, they enable comprehensive analysis of executable format issues and symbol resolution problems.

Binary Analysis Diagnostic Workflows:

Analysis Target	Primary Tool	Command Pattern	Expected Output Validation
ELF header validation	<code>readelf</code>	<code>readelf -h <executable></code>	Magic number, architecture, entry point
Section enumeration	<code>readelf</code>	<code>readelf -S <executable></code>	Section names, sizes, file offsets
Symbol table analysis	<code>readelf</code>	<code>readelf -s <executable></code>	Function names, addresses, binding types
Debug section contents	<code>readelf</code>	<code>readelf --debug-dump=info <executable></code>	DWARF compilation units and DIEs
Instruction disassembly	<code>objdump</code>	<code>objdump -d <executable></code>	Assembly instructions at specific addresses
String table examination	<code>readelf</code>	<code>readelf -p .strtab <executable></code>	Symbol name strings and their offsets

The instruction boundary verification process using `objdump` is critical for breakpoint debugging. When breakpoints fail to trigger or cause crashes, examining the disassembly around the breakpoint address reveals whether the breakpoint is correctly aligned with instruction boundaries.

Breakpoint Placement Verification Process:

- Disassemble Target Region:** Use `objdump -d --start-address=<addr> --stop-address=<addr+20>` to examine instructions around breakpoint
- Identify Instruction Boundaries:** Confirm target address corresponds to first byte of complete instruction
- Check Instruction Encoding:** Verify instruction opcode matches expected values
- Analyze Control Flow:** Examine branch targets and call destinations for correctness
- Compare with Symbol Information:** Cross-reference disassembly with symbol table entries

The DWARF debug information analysis using `readelf` provides detailed examination of debug information structure and content. This analysis is essential for diagnosing symbol resolution failures and understanding why certain functions or variables cannot be located.

Validation Strategy: Always compare your debugger's symbol resolution results with `readelf --debug-dump` output. Discrepancies indicate parsing errors or interpretation problems in your DWARF processing logic.

Meta-Debugging with GDB

Meta-debugging involves using GDB to debug your debugger while it debugs target processes. This creates a three-layer debugging scenario: GDB controlling your debugger, your debugger controlling the target process, and complex interactions between all three processes.

Meta-debugging is particularly valuable for diagnosing race conditions, signal handling bugs, and complex state management issues that are difficult to observe through other means. However, it requires careful setup to avoid

interference between the multiple debugging layers.

Meta-Debugging Setup Requirements:

Configuration Aspect	Requirement	Implementation	Potential Issues
Signal isolation	Prevent GDB signals from interfering with debugger	Use <code>handle SIGCHLD nostop noprint pass</code> in GDB	GDB may catch signals intended for debugger
Process tracking	Enable GDB to follow fork/exec operations	Set <code>set follow-fork-mode parent</code>	GDB may lose track of child processes
Breakpoint coordination	Avoid breakpoint conflicts between layers	Use different breakpoint mechanisms	Hardware vs software breakpoint interference
I/O redirection	Separate input/output streams	Redirect debugger I/O to separate terminal	Output confusion between debugging layers

The signal handling coordination in meta-debugging requires careful configuration because both GDB and your debugger need to handle `SIGCHLD` and `SIGTRAP` signals. The default GDB behavior of stopping on all signals can interfere with your debugger's signal handling logic.

GDB Configuration for Meta-Debugging:

- Configure Signal Handling:** `handle SIGCHLD nostop noprint pass` to avoid interfering with debugger's child process management
- Set Fork Following:** `set follow-fork-mode parent` to stay with debugger process rather than following target
- Enable Multi-Process:** `set detach-on-fork off` if you need to observe both debugger and target
- Configure Breakpoints:** Use software breakpoints in debugger, let debugger use hardware breakpoints for target
- Separate I/O:** `set inferior-tty /dev/pts/X` to redirect target process I/O

The race condition diagnosis using meta-debugging involves setting breakpoints at critical synchronization points and examining the state of data structures during signal handling. This technique is particularly effective for diagnosing timing-dependent bugs that are difficult to reproduce consistently.

Meta-Debugging Strategy: Focus on state consistency at synchronization boundaries. Set breakpoints before and after critical operations like breakpoint setting, signal handling, and process state transitions to verify state remains consistent.

Memory and Register Analysis Techniques

Memory and register analysis provides low-level visibility into debugger operation and target process state. These techniques are essential for diagnosing memory corruption, register handling errors, and data structure consistency issues.

The memory analysis process involves examining both the debugger's internal data structures and the target process's memory contents to verify consistency and detect corruption. Register analysis focuses on CPU state management and the correctness of register read/write operations.

Memory Analysis Diagnostic Framework:

Analysis Type	Tools Required	Information Gathered	Common Problem Indicators
Debugger heap analysis	Valgrind, AddressSanitizer	Memory leaks, buffer overruns, use-after-free	Invalid pointer dereferences, corrupted data structures
Target process memory	<code>ptrace(PTRACE_PEEKDATA)</code> , <code>/proc/<pid>/maps</code>	Memory layout, breakpoint patches, variable values	Unexpected byte values, unmapped regions
Register state consistency	<code>ptrace(PTRACE_GETREGS)</code> , GDB register dumps	CPU register contents across operations	Incorrect register values, missing updates
Data structure validation	Custom validation functions, assertions	Internal consistency of debugger state	Corrupted linked lists, invalid hash tables

The breakpoint memory consistency verification involves comparing the debugger's internal breakpoint state with the actual memory contents of the target process. Inconsistencies indicate bugs in breakpoint management logic or memory access errors.

Breakpoint Memory Verification Procedure:

- 1. Enumerate Internal Breakpoints:** Iterate through `BreakpointList` and collect all active breakpoints
- 2. Read Target Memory:** Use `ptrace(PTRACE_PEEKDATA)` to read bytes at each breakpoint address
- 3. Verify Patch State:** Enabled breakpoints should show 0xCC, disabled should show original instruction
- 4. Check Original Byte Storage:** Verify stored original bytes match actual instruction bytes when breakpoint disabled
- 5. Detect Memory Corruption:** Look for unexpected byte patterns or invalid memory contents

The register handling verification process ensures that register read and write operations correctly preserve CPU state across debugging operations. Register corruption can cause target process crashes or incorrect variable value displays.

Validation Technique: Implement register state checkpoints before and after critical operations. Compare register dumps to detect unexpected modifications and verify that saved/restored register state maintains target process execution integrity.

Implementation Guidance

The debugging process for a debugger implementation requires a systematic approach using multiple complementary tools and techniques. This section provides practical guidance for applying the diagnostic strategies described above, with emphasis on C language implementation specifics and Unix system programming considerations.

Technology Recommendations

Debugging Category	Basic Tools	Advanced Tools	Integration Strategy
System call tracing	<code>strace</code> basic usage	<code>strace</code> with filtering and timestamps	Integrate into test scripts for automated diagnosis
Binary analysis	<code>readelf</code> , <code>objdump</code>	<code>eu-readelf</code> , <code>llvm-objdump</code>	Build verification scripts that compare tool outputs
Meta-debugging	GDB command line	GDB Python scripting	Create custom GDB commands for debugger-specific analysis
Memory analysis	Manual <code>ptrace</code> calls	Valgrind, AddressSanitizer	Compile debugger with sanitizers during development

Debugging Infrastructure Implementation

The debugger should include built-in diagnostic capabilities that can be enabled during development and testing. This infrastructure should provide visibility into internal state and operation without requiring external tools for basic diagnosis.

```
// Error reporting infrastructure with context capture
```

C

```
typedef struct {

    DebugResult code;

    char message[256];

    const char *function;

    int line;

    int system_errno;

    uint64_t timestamp;

    pid_t debugger_pid;

    pid_t target_pid;

    uintptr_t context_address;

} DiagnosticError;
```

```
// Macro for creating diagnostic errors with full context
```

```
#define DIAGNOSTIC_ERROR(code, msg, addr) \
    diagnostic_error_create(code, msg, __FUNCTION__, __LINE__, errno, addr)
```

```
DiagnosticError diagnostic_error_create(DebugResult code, const char *message,
                                       const char *function, int line,
                                       int sys_errno, uintptr_t address) {

    // TODO 1: Initialize DiagnosticError structure with all parameters

    // TODO 2: Capture current timestamp using clock_gettime()

    // TODO 3: Record current process IDs for debugger and target

    // TODO 4: Store context address for memory-related errors

    // TODO 5: Preserve system errno value for system call failures

    // TODO 6: Return fully populated diagnostic error structure

}
```

```
// State validation functions for internal consistency checking
```

```
DebugResult validate_breakpoint_consistency(BreakpointList *list, pid_t target_pid) {

    // TODO 1: Iterate through all breakpoints in list
```

```
// TODO 2: For each enabled breakpoint, read memory at address using ptrace

// TODO 3: Verify memory contains INT3 (0xCC) if breakpoint enabled

// TODO 4: Verify memory contains original byte if breakpoint disabled

// TODO 5: Check that hit_count values are reasonable and non-negative

// TODO 6: Validate linked list structure for loops or corruption

// TODO 7: Return DEBUG_SUCCESS if all checks pass, appropriate error otherwise

}

// Memory access validation with detailed error reporting

DebugResult validate_memory_access(pid_t target_pid, uintptr_t address,
                                    size_t size, int required_perms) {

    // TODO 1: Parse /proc/<pid>/maps to get memory region information

    // TODO 2: Find memory region containing target address

    // TODO 3: Verify entire access range [address, address+size) is within region

    // TODO 4: Check region permissions match required_perms (read/write/execute)

    // TODO 5: Handle special cases like vsyscall or vdso regions

    // TODO 6: Return detailed error with specific failure reason

}
```

Process Control Debugging Implementation

```
// Enhanced ptrace wrapper with comprehensive error reporting C

DebugResult process_ptrace_with_retry(int request, pid_t pid,
                                      void *addr, void *data,
                                      int max_retries) {

    // TODO 1: Validate ptrace parameters before making system call

    // TODO 2: Check target process existence with kill(pid, 0)

    // TODO 3: Attempt ptrace operation with error capture

    // TODO 4: On EINTR, retry operation up to max_retries times

    // TODO 5: On ESRCH, verify process didn't exit unexpectedly

    // TODO 6: On EPERM, provide detailed permission diagnosis

    // TODO 7: Log all ptrace operations for debugging trace

    // TODO 8: Return success or detailed error with context

}

// Signal handling state machine with validation

typedef struct {

    pid_t target_pid;

    ProcessState expected_state;

    ProcessState actual_state;

    int last_signal;

    uint64_t state_change_count;

    struct timespec last_state_change;

} ProcessStateTracker;

DebugResult process_wait_with_validation(ProcessStateTracker *tracker,
                                         int *status, int options) {

    // TODO 1: Record timestamp before waitpid call

    // TODO 2: Call waitpid with specified options

    // TODO 3: Validate wait return value and status
```

```
// TODO 4: Update process state tracker with new information  
  
// TODO 5: Check state transition is valid (e.g., RUNNING->STOPPED)  
  
// TODO 6: Detect and report unexpected state transitions  
  
// TODO 7: Handle special cases like process exit or signal delivery  
  
}
```

Symbol Resolution Debugging Infrastructure

```
// DWARF parsing validation with detailed error context

C

typedef struct {

    uint32_t compilation_units_parsed;

    uint32_t dies_parsed;

    uint32_t parsing_errors;

    uint32_t version_mismatches;

    char last_error_context[512];

} DwarfParsingStats;

DebugResult dwarf_parse_with_validation(const uint8_t *debug_info_data,
                                         size_t data_size,
                                         const uint8_t *abbrev_data,
                                         size_t abbrev_size,
                                         DwarfParsingStats *stats) {

    // TODO 1: Initialize parsing statistics structure

    // TODO 2: Validate debug_info section header and version

    // TODO 3: Parse abbreviation table with error checking

    // TODO 4: Iterate through compilation units with bounds checking

    // TODO 5: For each CU, parse DIE tree with cycle detection

    // TODO 6: Record parsing statistics and error contexts

    // TODO 7: Return aggregated results with detailed error information

}

// Symbol table consistency verification

DebugResult symbols_verify_integrity(SymbolTable *symbols) {

    // TODO 1: Verify hash table structure is not corrupted

    // TODO 2: Check that all symbol addresses are within reasonable ranges

    // TODO 3: Validate symbol name strings are null-terminated

    // TODO 4: Verify address mappings are sorted and non-overlapping
```

```
// TODO 5: Check for duplicate symbols with conflicting information  
  
// TODO 6: Validate cross-references between symbols and debug information  
  
}
```

Milestone Verification Checkpoints

Milestone 1 Verification (Process Control):

- Expected behavior: `./debugger target_program` should attach successfully and display prompt
- Test command: Set up basic process attachment and verify with `ps` that both processes exist
- Diagnostic check: Run `strace -e ptrace ./debugger` and verify `PTRACE_ATTACH` succeeds
- Common failure: Permission denied - check process ownership and ptrace permissions

Milestone 2 Verification (Breakpoints):

- Expected behavior: Setting breakpoint at function entry should halt execution when function called
- Test command: `break main; continue` should stop at main function entry
- Diagnostic check: Use `objdump -d` to verify breakpoint address corresponds to instruction boundary
- Common failure: Breakpoint set in wrong location - verify symbol resolution accuracy

Milestone 3 Verification (Symbol Tables):

- Expected behavior: `info functions` should list all functions from debug information
- Test command: Resolve function name to address and verify with `nm` or `objdump`
- Diagnostic check: Compare debugger symbol output with `readelf -s` symbol table
- Common failure: Missing debug information - verify executable compiled with `-g` flag

Milestone 4 Verification (Variable Inspection):

- Expected behavior: `print variable_name` should display current value with correct type
- Test command: Set breakpoint in function with local variables, examine variable values
- Diagnostic check: Compare displayed values with GDB output for same program state
- Common failure: Variable optimized out - test with unoptimized build (`-O0`)

Future Extensions

Milestone(s): All milestones — this section explores potential enhancements and demonstrates how the current architecture accommodates future features

The Evolution Analogy

Think of our debugger architecture as a well-designed house foundation. We've built the essential rooms (process control, breakpoints, symbols, variables) with strong structural support. Now we're planning additions that can connect to the existing foundation without requiring major demolition. Just as a good house foundation includes utility hookups and

structural points for future expansion, our component-based architecture provides extension points where new capabilities can plug in naturally.

The key insight is that architectural extensibility isn't about predicting every future need — it's about creating clean interfaces and separation of concerns that make unanticipated changes possible. Our four-component design with well-defined data structures and communication patterns provides exactly this flexibility.

Multi-Threading Support

Modern applications are predominantly multi-threaded, making thread-aware debugging essential for real-world debugger usage. Our current single-process architecture provides the foundation, but extending to multi-threading requires careful consideration of process relationships, shared state, and debugging complexity.

The Orchestra Conductor Analogy

Think of debugging a multi-threaded application like conducting an orchestra. Instead of following one musician (process), the conductor (debugger) must track dozens of musicians (threads) simultaneously, each playing their own part but contributing to a shared performance. The conductor needs to stop individual sections for corrections while keeping the overall piece coherent, and must understand how each part relates to the whole composition.

In our debugger context, this means tracking multiple execution contexts within a single process, managing breakpoints that affect all threads, and presenting a coherent view of the program state despite concurrent execution.

Thread Representation and Management

The first architectural challenge is extending our `DebuggedProcess` structure to represent multiple threads of execution. Each thread maintains its own register state and stack, but shares the same memory space and symbol table.

Data Structure Extension	Purpose	Relationship
<code>ThreadInfo</code>	Individual thread state and registers	One per thread in process
<code>ThreadList</code>	Collection of threads in process	Embedded in <code>DebuggedProcess</code>
<code>ThreadEvent</code>	Thread-specific debug events	Generated by ptrace for each thread
<code>GlobalBreakpoint</code>	Process-wide breakpoint	Shared across all threads
<code>ThreadLocalBreakpoint</code>	Thread-specific breakpoint	Affects only target thread

The `ThreadInfo` structure extends our existing register management to handle multiple execution contexts:

Field Name	Type	Purpose
thread_id	pid_t	Linux thread ID (LWP)
thread_state	ThreadState	Current thread execution state
registers	struct user_regs_struct	Thread-specific CPU registers
stack_pointer	uintptr_t	Current stack location
program_counter	uintptr_t	Current instruction address
signal_pending	int	Thread-specific pending signal
single_step_active	bool	Whether thread is single-stepping
next	struct ThreadInfo*	Linked list pointer

Thread State Synchronization

Multi-threaded debugging introduces complex synchronization challenges. When one thread hits a breakpoint, the debugger must decide whether to stop all threads or allow others to continue executing. This decision affects both correctness and usability.

Decision: All-Stop vs Non-Stop Threading Model

- **Context:** When one thread hits a breakpoint, other threads can continue running (non-stop) or be forced to stop (all-stop)
- **Options Considered:**
 - All-stop model stops all threads when any thread hits breakpoint
 - Non-stop model allows other threads to continue execution
 - Hybrid model with user-configurable thread groups
- **Decision:** Implement all-stop model initially, with architecture support for non-stop extension
- **Rationale:** All-stop model provides more predictable debugging experience and simpler state management, matching GDB's default behavior. Non-stop requires complex thread scheduling and state coordination.
- **Consequences:** Simpler initial implementation but may not match all debugging workflows. Extension point designed for future non-stop support.

Threading Model	Advantages	Disadvantages
All-Stop	Predictable state, simpler implementation, matches user expectations	May halt time-sensitive threads unnecessarily
Non-Stop	Realistic execution environment, better for real-time debugging	Complex state management, race conditions
Hybrid	Flexible per-scenario configuration	Increased complexity, user confusion

The thread synchronization mechanism requires extending our process control interface:

Method Name	Parameters	Returns	Description
<code>threads_enumerate</code>	<code>proc DebuggedProcess*</code>	<code>ThreadList*</code>	List all threads in process
<code>thread_attach</code>	<code>proc DebuggedProcess*, tid pid_t</code>	<code>DebugResult</code>	Attach to specific thread
<code>thread_stop_all</code>	<code>proc DebuggedProcess*</code>	<code>DebugResult</code>	Stop all threads in process
<code>thread_continue_all</code>	<code>proc DebuggedProcess*</code>	<code>DebugResult</code>	Resume all stopped threads
<code>thread_single_step</code>	<code>proc DebuggedProcess*, tid pid_t</code>	<code>DebugResult</code>	Step specific thread
<code>thread_get_state</code>	<code>proc DebuggedProcess*, tid pid_t</code>	<code>ThreadInfo*</code>	Get thread-specific state

Breakpoint Coordination Across Threads

Multi-threading significantly complicates breakpoint management. A software breakpoint set in shared code affects all threads that execute that instruction. The debugger must track which thread triggered the breakpoint and coordinate the restore-step-repatch sequence without interfering with other threads.

The challenge is particularly acute when multiple threads hit the same breakpoint simultaneously or in rapid succession. The original instruction byte can only be restored once, and the single-step operation must be carefully coordinated.

Critical insight: Thread-aware breakpoints require atomic restoration and repatching operations to prevent race conditions where one thread restores the original instruction while another thread expects to find the INT3 patch.

Breakpoint Challenge	Problem	Solution Approach
Simultaneous hits	Multiple threads hit same breakpoint	Queue threads, process hits sequentially
Restoration timing	When to restore original instruction	Restore only when last thread completes
Repatch coordination	When to re-insert INT3	Repatch after all threads complete sequence
Thread-specific breakpoints	Some breakpoints should affect specific threads	Thread ID filtering in hit detection

Architecture Extension Points

Our existing component architecture accommodates multi-threading through well-defined extension points:

Process Control Component Extensions: The process controller gains thread enumeration and control capabilities. The existing `process_single_step` and `process_continue` methods are extended with thread-specific variants, while maintaining backward compatibility for single-threaded debugging.

Breakpoint Manager Extensions: The breakpoint manager evolves to track which threads are involved in breakpoint hit processing. The existing `BreakpointList` structure gains thread-awareness without breaking existing single-threaded

functionality.

Symbol Resolver Extensions: Thread-local storage and thread-specific variables require additional DWARF parsing capabilities. The symbol resolver extends to handle thread-local storage (TLS) variable locations and per-thread variable instances.

Variable Inspector Extensions: Variable inspection must handle thread-local variables and show per-thread variable values. The existing variable location resolution extends to include thread context in location expressions.

Remote Debugging Protocol

Remote debugging enables debugging processes running on different machines, containers, or embedded systems. This capability transforms our local ptrace-based debugger into a distributed system with network communication, protocol design, and fault tolerance challenges.

The Remote Control Analogy

Think of remote debugging like operating a sophisticated robotic rover on Mars. The operator (local debugger) sits on Earth (local machine) and sends commands through space (network) to control the rover (remote target process). The rover has local sensors and actuators (remote debugger agent) that execute commands and send back telemetry data. The communication delay and potential for signal loss require careful protocol design and error handling.

Just as the Mars rover must operate semi-autonomously due to communication delays, the remote debugging agent must handle some operations locally while staying synchronized with the controlling debugger.

Protocol Architecture

Remote debugging requires splitting our monolithic debugger into two cooperating processes: the debugger client (user interface and command processing) and the debugger server (process control and memory access). This client-server architecture introduces network communication, serialization, and distributed state management.

Component	Location	Responsibility
DebuggerClient	Local machine	User interface, command processing, symbol resolution
DebuggerServer	Remote machine	Process control, memory access, breakpoint management
ProtocolHandler	Both sides	Message serialization, network communication
StateSync	Both sides	Distributed state consistency

The remote protocol must serialize all our existing operations and data structures for network transmission:

Operation Category	Local Method	Remote Message Type
Process control	<code>process_start</code> , <code>process_attach</code>	<code>MSG_PROCESS_CONTROL</code>
Execution control	<code>process_continue</code> , <code>process_single_step</code>	<code>MSG_EXECUTION</code>
Memory operations	<code>process_read_memory</code> , <code>process_write_memory</code>	<code>MSG_MEMORY_ACCESS</code>
Breakpoint management	<code>breakpoint_set</code> , <code>breakpoint_remove</code>	<code>MSG_BREAKPOINT</code>
Register access	<code>register_reader_get_all</code>	<code>MSG_REGISTER</code>
Symbol queries	<code>symbols_resolve_address</code>	<code>MSG_SYMBOL_LOOKUP</code>

Message Format Design

The protocol requires a binary message format that efficiently serializes our data structures while maintaining version compatibility and extensibility. The message format must handle variable-length data like symbol names and memory contents.

Decision: Binary Protocol vs Text Protocol

- **Context:** Network messages need efficient serialization while maintaining debuggability and extensibility
- **Options Considered:**
 - JSON-based text protocol for human readability
 - Protocol Buffers for version compatibility
 - Custom binary format for minimal overhead
- **Decision:** Custom binary format with length prefixes and type tags
- **Rationale:** Debugging generates high-frequency small messages (memory reads, register access). Binary format minimizes bandwidth and latency. Custom format avoids external dependencies.
- **Consequences:** Efficient network usage but requires careful version compatibility design.

Message Component	Size	Purpose
Magic header	4 bytes	Protocol identification
Protocol version	2 bytes	Version compatibility checking
Message type	2 bytes	Operation identification
Sequence number	4 bytes	Request/response matching
Payload length	4 bytes	Variable payload size
Payload data	Variable	Serialized operation data
Checksum	4 bytes	Message integrity verification

The message payload serialization uses a type-length-value (TLV) encoding that handles our existing data structures:

Data Type	Encoding	Example
<code>DebugResult</code>	1 byte enum	<code>DEBUG_SUCCESS</code> = 0x00
<code>uintptr_t</code>	8 bytes little-endian	Address values
<code>pid_t</code>	4 bytes little-endian	Process identifiers
String	Length prefix + UTF-8	Symbol names, filenames
Register set	Fixed 512-byte structure	CPU register dump
Memory block	Length + address + data	Memory read results

Connection Management and Fault Tolerance

Remote debugging must handle network failures, connection losses, and partial message delivery. The protocol requires connection establishment, heartbeat monitoring, and graceful degradation when connectivity problems occur.

The connection lifecycle involves authentication, capability negotiation, and state synchronization:

1. **Connection establishment:** Client connects to server, exchanges protocol versions
2. **Authentication:** Server verifies client permissions for process debugging
3. **Capability negotiation:** Exchange supported features (threading, watchpoints, etc.)
4. **Initial state sync:** Server sends current process list and state
5. **Command processing:** Normal request/response message exchange
6. **Heartbeat monitoring:** Periodic keep-alive messages detect connection issues
7. **Graceful shutdown:** Proper cleanup of debug sessions and process state

Fault Condition	Detection	Recovery Strategy
Connection timeout	Send timeout on socket	Reconnect with state resynchronization
Partial message	Incomplete payload	Buffer partial data, request retransmission
Server crash	Heartbeat failure	Graceful degradation, inform user of lost session
Process termination	Server notification	Clean up local state, update UI
Network partition	Extended timeout	Queue commands, reconnect when possible

State Synchronization Challenges

Remote debugging creates distributed state consistency challenges. The client maintains symbol tables and user interface state, while the server holds the actual process state and memory contents. Changes on either side must be synchronized to maintain consistency.

The critical challenge is maintaining the illusion of local debugging while operating over an unreliable network with latency and potential failures.

State Category	Owner	Synchronization Strategy
Symbol tables	Client	Send to server during attachment
Breakpoints	Both	Client tracks user requests, server tracks actual patches
Process state	Server	Server sends state changes to client
Memory contents	Server	Client requests on-demand, caching for performance
Register values	Server	Client requests when needed, no caching
User preferences	Client	Local storage, not synchronized

Architecture Integration

Remote debugging capability integrates into our existing architecture through abstraction layers that hide the local/remote distinction from higher-level components.

The `ProcessController` interface gains network-aware implementations that serialize operations into protocol messages:

Interface Method	Local Implementation	Remote Implementation
<code>process_start</code>	Direct ptrace and fork	Serialize to <code>MSG_PROCESS_CONTROL</code>
<code>process_read_memory</code>	Direct <code>PTRACE_PEEKDATA</code>	Send <code>MSG_MEMORY_ACCESS</code> request
<code>breakpoint_set</code>	Direct memory patching	Send <code>MSG_BREAKPOINT</code> message

The symbol resolution component requires special handling since debug information files may exist on either the client or server machine:

Decision: Symbol File Location Strategy

- **Context:** Debug information may be available on client machine, server machine, or both
- **Options Considered:**
 - Always transfer debug files to client for local processing
 - Always process symbols on server and send results to client
 - Hybrid approach with capability detection
- **Decision:** Process symbols on client when files are available locally, otherwise request server-side processing
- **Rationale:** Client-side symbol processing reduces server load and network traffic for large debug files. Fallback to server processing handles cases where files are only available remotely.
- **Consequences:** More complex but flexible symbol resolution, better performance in common cases.

Advanced Debugging Features

Beyond our core debugging primitives, several advanced features significantly enhance debugging capability. These extensions demonstrate how our architecture accommodates sophisticated debugging techniques while building on the foundational components.

Watchpoints and Memory Monitoring

Watchpoints monitor memory locations for read, write, or access operations, triggering debugger stops when watched memory is accessed. Unlike breakpoints that monitor code execution, watchpoints monitor data access patterns.

The Security Camera Analogy

Think of watchpoints as security cameras monitoring valuable areas in a building. While breakpoints are like doorway sensors that trigger when someone passes through a specific entrance, watchpoints are area monitors that trigger when someone approaches or touches watched objects, regardless of which path they took to get there.

This distinction is crucial: breakpoints monitor control flow (where the program goes), while watchpoints monitor data flow (what the program touches).

Hardware vs Software Watchpoints

Modern processors provide hardware debugging registers that can monitor memory addresses without performance overhead. However, these registers are limited in number (typically 4-8 per processor), requiring careful resource management.

Decision: Hardware Watchpoints with Software Fallback

- **Context:** Need to monitor memory access with varying performance and capability requirements
- **Options Considered:**
 - Hardware watchpoints only (limited quantity, high performance)
 - Software watchpoints only (unlimited quantity, severe performance impact)
 - Hybrid system with automatic fallback
- **Decision:** Prioritize hardware watchpoints with software simulation for overflow cases
- **Rationale:** Hardware watchpoints provide zero-overhead monitoring for common cases. Software fallback ensures functionality when hardware resources are exhausted.
- **Consequences:** Complex implementation but optimal performance in typical usage patterns.

Watchpoint Type	Implementation	Performance	Limitations
Hardware read	CPU debug register	Zero overhead	4-8 total watchpoints
Hardware write	CPU debug register	Zero overhead	Address alignment restrictions
Software read	Page protection + trap	1000x slower	No alignment restrictions
Software write	Page protection + trap	100x slower	Page-granularity only

Watchpoint Data Structures

Watchpoint implementation requires extending our breakpoint management with memory monitoring capabilities:

Field Name	Type	Purpose
address	uintptr_t	Watched memory address
size	size_t	Number of bytes to monitor
watch_type	WatchType	Read, write, or access monitoring
hardware_register	int	Which debug register is used (-1 for software)
old_value	uint8_t*	Previous memory contents for change detection
hit_count	uint32_t	Number of times watchpoint triggered
condition	char*	Optional condition expression

The `WatchType` enumeration defines monitoring behavior:

Constant	Value	Meaning
WATCH_READ	1	Trigger on memory read access
WATCH_WRITE	2	Trigger on memory write access
WATCH_ACCESS	3	Trigger on read or write access

Memory Change Detection

Software watchpoints require detecting when watched memory changes between debugging operations. This involves periodic memory scanning and change detection:

1. **Checkpoint creation:** Save watched memory contents when watchpoint is set
2. **Periodic scanning:** Check watched memory during process stops
3. **Change detection:** Compare current contents with checkpoint data
4. **Attribution:** Determine which instruction caused the change
5. **Notification:** Report change to user with before/after values

Method Name	Parameters	Returns	Description
watchpoint_set	proc, address, size, type	DebugResult	Create new memory watchpoint
watchpoint_check_changes	proc, watchpoint_list	WatchpointHit*	Scan for memory changes
watchpoint_hardware_allocate	address, size, type	int	Allocate hardware debug register
watchpoint_software_enable	proc, watchpoint	DebugResult	Enable software memory monitoring

Conditional Breakpoints

Conditional breakpoints extend basic breakpoints with expression evaluation, triggering only when specified conditions are met. This capability reduces debugging noise by filtering breakpoint hits based on program state.

The Smart Alarm Analogy

Think of conditional breakpoints as smart home security systems. A basic motion detector (regular breakpoint) triggers every time anything moves past a doorway. A smart system (conditional breakpoint) only triggers when specific conditions are met: "Alert me when someone enters, but only if it's after midnight and the security system is armed."

This filtering happens automatically without user intervention, reducing false alarms while ensuring important events are never missed.

Expression Evaluation Engine

Conditional breakpoints require evaluating C-like expressions in the context of the stopped program. The expression evaluator must access variables, perform arithmetic, and handle type conversions.

Expression Component	Example	Implementation
Variable access	<code>x > 10</code>	Symbol lookup + memory reading
Arithmetic	<code>count * 2</code>	Standard operator evaluation
Comparisons	<code>ptr != NULL</code>	Pointer and value comparisons
Function calls	<code>strlen(str)</code>	Limited function evaluation
Type casts	<code>(int)floatvar</code>	Type-aware conversions

The expression evaluator integrates with our existing variable inspection system:

Method Name	Parameters	Returns	Description
<code>expression_evaluate</code>	<code>proc, expression, result</code>	<code>DebugResult</code>	Evaluate condition expression
<code>expression_parse</code>	<code>expression_text, ast</code>	<code>DebugResult</code>	Parse expression into AST
<code>expression_resolve_variables</code>	<code>proc, ast</code>	<code>DebugResult</code>	Resolve variable references
<code>expression_execute</code>	<code>proc, ast, result</code>	<code>DebugResult</code>	Execute parsed expression

Performance Optimization

Conditional breakpoints can significantly impact debugging performance if expressions are complex or frequently evaluated. Optimization strategies include expression caching, fast-path evaluation, and condition compilation.

The performance challenge: Complex conditional expressions evaluated at high-frequency breakpoints can slow debugging by orders of magnitude.

Optimization Strategy	Benefit	Implementation Cost
Expression caching	Avoid re-parsing expressions	Low - hash table lookup
Variable value caching	Avoid repeated memory reads	Medium - cache invalidation
Condition compilation	Generate native code for expressions	High - requires code generator
Fast-path detection	Optimize simple expressions	Low - pattern matching

Reverse Debugging Capabilities

Reverse debugging allows stepping backward through program execution, undoing the effects of instructions and returning to previous program states. This powerful capability helps understand how programs reached problematic states.

The Time Machine Analogy

Think of reverse debugging as a time machine for program execution. Just as a time traveler can go back to earlier points in history and observe how events unfolded, reverse debugging lets you step backward through program execution to see how variables changed and control flow developed.

The key insight is that reverse debugging isn't just running the program backward — it's about reconstructing previous program states from recorded execution history.

Implementation Strategies

Reverse debugging can be implemented through several approaches, each with different trade-offs between performance overhead and functionality:

Decision: Checkpoint-Based Reverse Debugging

- **Context:** Need to provide reverse execution with acceptable performance overhead
- **Options Considered:**
 - Full instruction recording (complete but massive overhead)
 - Periodic checkpointing with forward replay (balanced approach)
 - Copy-on-write process snapshots (simple but memory intensive)
- **Decision:** Periodic checkpointing with deterministic replay
- **Rationale:** Balances functionality with performance. Forward replay is fast, checkpoints limit replay time. Deterministic replay ensures consistency.
- **Consequences:** Moderate memory usage, predictable performance, but cannot reverse through non-deterministic operations.

Approach	Memory Overhead	Time Overhead	Reverse Capability
Full recording	Very high (10-100x)	High (2-10x)	Perfect reverse execution
Checkpointing	Medium (2-5x)	Low (10-20%)	Reverse to checkpoint boundaries
Process snapshots	High (5-20x)	Very low (<5%)	Snapshot-to-snapshot jumps

Checkpoint Management

The checkpoint system must balance memory usage with reverse debugging granularity. More frequent checkpoints enable finer-grained reverse stepping but consume more memory and processing time.

Checkpoint Component	Purpose	Storage Requirements
Process memory image	Complete address space	10MB - 1GB per checkpoint
CPU register state	All registers and flags	512 bytes per checkpoint
File descriptor state	Open files and positions	Variable per checkpoint
System call log	Non-deterministic operations	Variable per checkpoint
Breakpoint state	Active breakpoints	Small per checkpoint

The checkpoint management system requires careful resource management:

Method Name	Parameters	Returns	Description
checkpoint_create	proc, checkpoint	DebugResult	Create full process checkpoint
checkpoint_restore	proc, checkpoint	DebugResult	Restore process to checkpoint state
checkpoint_cleanup	checkpoint_list	DebugResult	Free old checkpoint memory
replay_forward	proc, checkpoint, target_pc	DebugResult	Replay from checkpoint to target

Architecture Extension Points

These advanced debugging features integrate into our existing architecture through well-defined extension points that maintain compatibility with existing functionality:

Process Control Extensions: The process controller gains state management capabilities for checkpointing and replay. Hardware debug register management extends the existing ptrace interface.

Breakpoint Manager Extensions: Conditional breakpoints extend the existing breakpoint system with expression evaluation. Watchpoints add a parallel monitoring system with similar lifecycle management.

Symbol Resolver Extensions: Expression evaluation requires enhanced symbol lookup capabilities and type information access. The existing symbol resolution extends to support expression context.

Variable Inspector Extensions: Advanced features rely heavily on variable inspection for expression evaluation and memory change detection. The existing variable access infrastructure extends to support these use cases.

Common Pitfalls

⚠ Pitfall: Thread Race Conditions in Breakpoint Management Multi-threaded debugging introduces race conditions where multiple threads can hit the same breakpoint simultaneously. The debugger may attempt to restore the original instruction multiple times or lose track of which thread is in the restore-step-repatch sequence. This leads to corrupted breakpoints and unpredictable debugging behavior.

Why it's wrong: The breakpoint restoration assumes single-threaded access to the instruction patching mechanism. Multiple threads violate this assumption and can corrupt the target process memory.

How to fix: Implement atomic breakpoint state transitions with thread queuing. When a breakpoint is hit, queue subsequent threads until the first thread completes the full restore-step-repatch cycle.

 **Pitfall: Network Protocol Version Incompatibility** Remote debugging protocols often evolve over time, but failing to handle version mismatches gracefully causes connection failures and debugging session loss. Hardcoded protocol assumptions break when client and server versions differ.

Why it's wrong: Network protocols require careful versioning to maintain compatibility across software updates. Version mismatches without graceful degradation cause complete functionality loss.

How to fix: Implement protocol version negotiation during connection establishment. Support backward compatibility for at least one previous version, with clear error messages for unsupported versions.

 **Pitfall: Watchpoint Resource Exhaustion** Setting too many watchpoints can exhaust hardware debug registers, but failing to handle this gracefully causes watchpoint requests to silently fail or crash the debugger. Users may not realize their watchpoints are inactive.

Why it's wrong: Hardware debug registers are limited resources (typically 4-8 per processor). Assuming unlimited watchpoints leads to resource exhaustion and undefined behavior.

How to fix: Track hardware register allocation explicitly. When hardware resources are exhausted, either fall back to software watchpoints or provide clear error messages about resource limits.

 **Pitfall: Conditional Expression Side Effects** Complex conditional expressions may contain function calls or operations with side effects. Evaluating these expressions during breakpoint checking can modify program state and change program behavior.

Why it's wrong: Debugging should observe program behavior without changing it. Expressions with side effects violate this fundamental principle and create unpredictable debugging artifacts.

How to fix: Restrict conditional expressions to side-effect-free operations. Prohibit function calls, assignment operators, and operations that modify memory or register state.

Implementation Guidance

The future extensions demonstrate how our current architecture provides extension points for advanced debugging features. While these features are beyond the scope of the initial project, understanding their architectural requirements helps design a robust foundation.

Technology Recommendations for Extensions

Extension Type	Simple Approach	Advanced Approach
Multi-threading	Extend existing ptrace with thread enumeration	Full thread-aware debugging with GDB-compatible interface
Remote protocol	JSON over TCP with simple request/response	Binary protocol with compression and multiplexing
Watchpoints	Software-only with periodic memory scanning	Hardware debug registers with software fallback
Conditional breakpoints	Simple expression parser with basic operators	Full C expression evaluator with type system
Reverse debugging	Manual checkpointing with user control	Automatic checkpointing with deterministic replay

Extension Architecture Patterns

Each extension follows similar architectural patterns that maintain compatibility with the existing four-component design:

Interface Extension Pattern: Existing interfaces gain additional methods without breaking existing functionality. The `ProcessController` interface adds thread-aware methods alongside existing single-threaded methods.

Component Delegation Pattern: New functionality is implemented in separate components that delegate to existing components for core operations. The `WatchpointManager` delegates to `MemoryReader` for memory access.

State Machine Extension Pattern: Existing state machines gain new states and transitions while preserving existing behavior. The process state machine adds thread-specific states while maintaining backward compatibility.

Data Structure Composition Pattern: New data structures compose existing structures rather than replacing them. `ThreadInfo` contains a `ProcessState` rather than reimplementing process state tracking.

Recommended File Structure for Extensions

```
src/
  core/           ← existing core components
    process.c
    breakpoint.c
    symbol.c
    variable.c
  extensions/
    threading/
      thread_manager.c   ← multi-threading support
      thread_state.c
    remote/
      protocol.c        ← remote debugging protocol
      network.c
      message.c
    advanced/
      watchpoint.c       ← watchpoint implementation
      conditional.c     ← conditional breakpoints
      reverse.c          ← reverse debugging
  include/
    extensions/
      threading.h
      remote.h
      advanced.h
```

This structure allows building the debugger with or without extensions through conditional compilation, maintaining a clean separation between core functionality and advanced features.

Extension Implementation Skeleton

Each extension follows a common initialization and integration pattern:

```
// Threading extension initialization

C

DebugResult threading_init(DebuggedProcess* proc) {

    // TODO 1: Enumerate existing threads in process

    // TODO 2: Attach ptrace to each thread individually

    // TODO 3: Initialize thread list data structure

    // TODO 4: Set up all-stop coordination mechanism

    // TODO 5: Register thread-aware command handlers

}

// Remote protocol message handler skeleton

DebugResult protocol_handle_message(NetworkConnection* conn,
                                    const uint8_t* message,
                                    size_t length) {

    // TODO 1: Validate message header and checksum

    // TODO 2: Extract message type and sequence number

    // TODO 3: Deserialize payload according to message type

    // TODO 4: Delegate to appropriate local operation handler

    // TODO 5: Serialize response and send back to client

}

// Watchpoint condition checking

DebugResult watchpoint_evaluate_condition(DebuggedProcess* proc,
                                           Watchpoint* watch,
                                           bool* should_trigger) {

    // TODO 1: Read current memory contents at watched address

    // TODO 2: Compare with previous checkpoint value

    // TODO 3: If changed and condition exists, evaluate expression

    // TODO 4: Set should_trigger based on condition result

    // TODO 5: Update checkpoint value for next comparison

}
```

Extension Integration Points

The existing architecture provides specific integration points where extensions connect to core functionality:

Command Processing Integration: Extensions register additional command handlers with the existing command processor. Thread commands (`thread list`, `thread select`) integrate alongside existing commands (`break`, `continue`).

Event Handling Integration: Extensions extend the existing signal handling and event processing to handle thread-specific events, network messages, and watchpoint triggers.

State Management Integration: Extensions add state tracking that coordinates with existing process state management. Thread states synchronize with overall process state through well-defined state transitions.

Error Handling Integration: Extensions use the existing error handling infrastructure (`DebugResult` codes, structured error messages) to maintain consistent error reporting across all functionality.

The architectural foundation established in the four core components provides a robust platform for these advanced debugging capabilities while maintaining the simplicity and clarity essential for learning the fundamentals of debugger implementation.

Glossary

Milestone(s): All milestones — this section provides definitions for technical terms, acronyms, and domain-specific vocabulary used throughout the debugger design document

The Language Learning Analogy

Think of this glossary as a specialized dictionary for a technical language that combines three dialects: debugging terminology, systems programming concepts, and binary format specifications. Just as learning a foreign language requires understanding both common vocabulary and specialized technical terms, building a debugger requires fluency in debugging concepts (breakpoints, stepping), systems programming primitives (processes, signals), and binary format structures (ELF sections, DWARF entries). This glossary serves as your translation guide between these interconnected technical domains, helping you navigate conversations about debugger implementation with precision and clarity.

The terminology is organized into three major categories that reflect the layered nature of debugger architecture. Debugging terminology covers the high-level concepts that users interact with. System programming terms describe the low-level Unix mechanisms that make debugging possible. Binary format terms explain the file format structures that contain the metadata necessary for source-level debugging.

Debugging Terminology

Debugging terminology encompasses the conceptual vocabulary that describes what debuggers do and how users interact with them. These terms represent the fundamental operations and concepts that make debugging possible, from the basic ability to stop program execution to the sophisticated analysis of program state.

Term	Definition	Context
breakpoint	Execution stop point set by debugger using INT3 patching that halts program execution when reached	Software breakpoints replace instruction bytes with 0xCC interrupt
software breakpoint	Breakpoint implemented by instruction patching with INT3 interrupt instruction	Contrasts with hardware breakpoints that use CPU debug registers
hardware breakpoint	Breakpoint implemented using CPU debug registers without modifying program memory	Limited in number but doesn't alter target program instructions
watchpoint	Memory monitoring point that triggers when specified memory location is accessed	Can monitor read access, write access, or both types of access
single-step execution	Advancing exactly one machine instruction at a time under debugger control	Implemented using PTRACE_SINGLESTEP to execute one instruction
process control	Starting, stopping, and stepping through process execution under debugger supervision	Foundation capability that enables all other debugging operations
process attachment	Gaining debugger control over target process via ptrace system call	Allows debugging existing processes or newly spawned processes
instruction patching	Temporarily replacing instruction bytes with breakpoint interrupt (INT3)	Core mechanism for implementing software breakpoints
restore-step-repatch	Sequence for continuing past breakpoint: restore original instruction, single-step, re-patch INT3	Standard algorithm for breakpoint continuation without losing breakpoint
breakpoint lifecycle	States a breakpoint goes through from creation to deletion (set, enabled, hit, disabled, removed)	Manages breakpoint state transitions and memory consistency
hit detection	Determining when and which breakpoint was triggered by SIGTRAP signal	Correlates signal address with known breakpoint locations
instruction pointer adjustment	Correcting IP after INT3 execution to point back to breakpoint location	INT3 advances IP by one byte, requiring correction for proper continuation
symbol resolution	Converting between addresses and source code locations using debug information	Bidirectional mapping between memory addresses and source file locations
address-to-source mapping	Translation from memory address to source location (file, line, column)	Uses DWARF line number program to map program counter values
name-to-address resolution	Translation from symbol name to memory address for function or variable	Enables setting breakpoints by function name rather than address
variable inspection	Reading and displaying variable values from process memory using type information	Combines memory access, location resolution, and type interpretation
type-aware interpretation	Converting raw bytes to typed values using DWARF type information	Formats memory contents as int, float, struct, array, etc.

Term	Definition	Context
composite type handling	Navigating structs, arrays, and pointers with field access and bounds checking	Handles complex data structures with multiple levels of indirection
variable location resolution	Interpreting DWARF expressions to find variables in memory/registers	Evaluates DWARF location expressions to determine storage location
location expression evaluator	DWARF bytecode interpreter for variable location calculation	Executes DWARF opcodes to compute variable addresses
memory access caching	Optimization strategy for reducing expensive ptrace memory reads	Caches recently read memory regions to improve performance
stack unwinding	Reconstructing function call sequence from stack memory	Traces function calls from current frame back to main
crash analysis	Examination of process state at time of failure	Analyzes registers, memory, and stack to determine crash cause
post-mortem debugging	Analyzing program state after crash without restart	Debugging using core dumps or crash state snapshots
conditional breakpoint	Breakpoint that triggers only when specified conditions are met	Evaluates expression at breakpoint location before stopping
expression evaluation	Calculating values from C-like expressions in program context	Supports arithmetic, variable access, and function calls
reverse debugging	Stepping backward through program execution to previous states	Requires state checkpointing or deterministic replay
checkpoint-based debugging	Saving program state snapshots for reverse execution	Creates full process state snapshots at specific points
deterministic replay	Reproducing exact execution sequence from recorded state	Replays execution from checkpoint to specific target location

System Programming Terms

System programming terminology covers the Unix and Linux kernel interfaces that debuggers use to control and inspect processes. These low-level mechanisms provide the foundation for all debugger functionality, from basic process control to memory access and signal handling.

Term	Definition	Context
ptrace	Unix system call for process tracing and control that enables debugging	Primary interface for debugger to control target process
process	Independent program execution context with private memory space	Unit of execution that debugger attaches to and controls
signal	Asynchronous notification mechanism for inter-process communication	Used for breakpoint notification and process control
signal management	Handling and forwarding signals between debugger and debuggee	Debugger must intercept and optionally forward signals
SIGTRAP	Signal generated by breakpoint or single-step instruction execution	Standard signal used to notify debugger of breakpoint hits
SIGSTOP	Signal that unconditionally stops process execution	Used by debugger to pause target process
SIGCONT	Signal that resumes stopped process execution	Used by debugger to continue target process
SIGSEGV	Segmentation violation signal indicating invalid memory access	Common crash signal that debugger must handle
SIGBUS	Bus error signal indicating invalid memory alignment or access	Hardware-specific memory access error
SIGFPE	Floating point exception signal for arithmetic errors	Division by zero, overflow, or invalid operations
SIGABRT	Abort signal generated by abort() function call	Programmatic termination signal
SIGILL	Illegal instruction signal for invalid machine code	Execution of invalid or privileged instructions
SIGINT	Interrupt signal typically generated by Ctrl+C	User request to interrupt program execution
SIGCHLD	Child process state change signal sent to parent	Notifies debugger of target process state changes
waitpid	System call to wait for process state changes and retrieve exit status	Debugger uses to monitor target process state
fork	System call to create child process as copy of parent	Used to spawn target process under debugger control
exec	System call to replace process image with new program	Loads target executable into child process
process state synchronization	Coordinating debugger expectations with actual target process state transitions	Ensures debugger state matches target process state

Term	Definition	Context
race condition	Timing-dependent bug where outcome depends on execution order	Common problem in signal handling and process control
zombie process	Terminated child process whose parent hasn't called waitpid	Must be cleaned up to prevent resource leaks
thread	Lightweight execution context within process sharing memory space	Unit of execution in multi-threaded programs
thread synchronization	Coordinating execution state across multiple threads	Required for debugging multi-threaded applications
all-stop model	Debugging model that stops all threads when any thread hits breakpoint	Simplifies debugging by providing consistent global state
non-stop model	Debugging model that allows other threads to continue when one stops	More complex but allows continued execution of unrelated threads
thread-local storage	Per-thread memory regions with separate instances per thread	Storage mechanism that requires special handling in debugger
permission validation	Checking access rights before attempting operations	Prevents permission-related failures in ptrace operations
system call tracing	Monitoring debugger's kernel interactions using strace	Meta-debugging technique for diagnosing ptrace issues
endianness conversion	Byte order transformation between target and host architectures	Required when debugging across different CPU architectures
memory protection	Kernel mechanism controlling read/write/execute permissions for memory pages	Affects debugger's ability to read and modify target memory
virtual memory	Abstraction that provides each process with private address space	Debugger must work within target's virtual address space
memory mapping	Association between virtual addresses and physical memory or files	Affects how debugger interprets memory addresses

Binary Format Terms

Binary format terminology describes the file formats and data structures that store executable code and debugging information. Understanding these formats is essential for implementing symbol resolution and variable inspection capabilities.

Term	Definition	Context
ELF	Executable and Linkable Format for Unix binaries containing code and metadata	Standard binary format on Linux systems
DWARF	Debug information format embedded in ELF binaries for source-level debugging	Industry standard for encoding debug information
symbol table	Mapping between function/variable names and memory addresses	Enables name-based access to program elements
debug information	Compiler-generated metadata for source-level debugging embedded in executable	Contains source-to-binary mappings and type information
section	Named region within ELF file containing specific type of data	Examples include .text (code), .debug_info (debug data)
Debug Information Entry	DWARF record describing program element like function or variable	Basic unit of DWARF debug information
abbreviation table	DWARF template definitions for DIE structure to reduce storage space	Compression mechanism for DWARF data
compilation unit	DWARF tree representing one source file and its debug information	Top-level container for source file debug data
location expression	DWARF bytecode program that calculates variable storage location	Executable instructions for finding variables
line number program	DWARF bytecode for mapping addresses to source line numbers	Executable instructions for address-to-source mapping
type descriptor	Unified representation of DWARF type information	Internal structure for representing data types
DIE	Debug Information Entry - DWARF record describing program elements	Abbreviation commonly used in DWARF documentation
attribute	Named property of DWARF DIE providing specific information	Examples: name, type, location, size
tag	DWARF identifier specifying the kind of program element in DIE	Examples: DW_TAG_subprogram, DW_TAG_variable
compilation directory	Base directory path for resolving relative source file names	Stored in DWARF compilation unit header
address range	Contiguous memory region associated with program element	Used for mapping addresses to functions or compilation units
location list	DWARF structure describing variable location changes over address ranges	Handles variables that move between registers and memory
type encoding	DWARF specification of how basic types are represented	Examples: signed integer, floating point, boolean

Term	Definition	Context
composite type	Complex data type built from simpler types (struct, array, union)	Requires recursive navigation for member access
member offset	Byte offset of structure field from beginning of structure	Used to calculate field addresses within composite types
array stride	Byte distance between consecutive array elements	Used to calculate element addresses in multi-dimensional arrays
pointer indirection	Following memory address to access referenced data	Requires additional memory read to access pointed-to value
symbol binding	Visibility and linkage properties of symbol (local, global, weak)	Affects symbol resolution and name lookup
symbol type	Classification of symbol (function, object, file, section)	Determines how symbol should be interpreted
relocation	Adjustment of addresses when loading executable at runtime	May affect symbol addresses during debugging
program header	ELF structure describing memory segments for program loading	Defines how executable is loaded into memory
section header	ELF structure describing file sections and their properties	Used to locate debug information sections
string table	ELF section containing null-terminated strings referenced by index	Storage mechanism for symbol names and section names
hash table	Data structure for fast symbol lookup by name	Optimization for symbol resolution performance
DWARF version	Version of DWARF specification used in debug information	Different versions have incompatible formats
compilation flags	Compiler options that affect debug information generation	Influences availability and format of debug data
optimization level	Compiler optimization setting that affects debug information accuracy	Higher optimization may make variables unavailable
inlined function	Function call replaced with function body at compile time	Complicates address-to-source mapping
macro information	DWARF section containing preprocessor macro definitions	Used for accurate source code representation

Acronyms and Abbreviations

This section provides definitions for technical acronyms and abbreviations commonly encountered in debugger implementation and documentation.

Acronym	Full Form	Definition
GDB	GNU Debugger	Popular open-source debugger for Unix systems
LLDB	LLVM Debugger	Debugger component of LLVM compiler infrastructure
PIE	Position Independent Executable	Executable that can be loaded at any memory address
ASLR	Address Space Layout Randomization	Security feature that randomizes memory layout
TLS	Thread Local Storage	Per-thread storage mechanism
CPU	Central Processing Unit	Main processor executing program instructions
IP	Instruction Pointer	CPU register pointing to next instruction
PC	Program Counter	Alternative name for instruction pointer
SP	Stack Pointer	CPU register pointing to current stack location
BP	Base Pointer	CPU register pointing to current stack frame
API	Application Programming Interface	Defined interface for software components
ABI	Application Binary Interface	Low-level interface for compiled code
ISA	Instruction Set Architecture	CPU instruction format specification
MMU	Memory Management Unit	Hardware component managing virtual memory
TLB	Translation Lookaside Buffer	Cache for virtual-to-physical address translation
GOT	Global Offset Table	Data structure for position-independent code
PLT	Procedure Linkage Table	Data structure for dynamic function linking
BSS	Block Started by Symbol	Uninitialized data section in executable
ROData	Read-Only Data	Section containing constant data

DWARF-Specific Constants and Opcodes

DWARF debug information uses numerous constants and opcodes that debugger implementations must recognize and handle. These constants define the structure and interpretation of debug information.

Constant	Value/Meaning	Usage Context
DW_TAG_compile_unit	DWARF compilation unit DIE tag	Identifies top-level compilation unit entries
DW_TAG_subprogram	DWARF function DIE tag	Identifies function definition entries
DW_TAG_variable	DWARF variable DIE tag	Identifies variable declaration entries
DW_TAG_formal_parameter	DWARF function parameter DIE tag	Identifies function parameter entries
DW_TAG_base_type	DWARF primitive type DIE tag	Identifies basic types like int, float, char
DW_TAG_pointer_type	DWARF pointer type DIE tag	Identifies pointer type definitions
DW_TAG_structure_type	DWARF struct type DIE tag	Identifies structure type definitions
DW_TAG_member	DWARF struct member DIE tag	Identifies structure member entries
DW_AT_name	Name attribute for DWARF entries	Contains symbol or type name
DW_AT_type	Type reference attribute	Points to type definition DIE
DW_AT_location	Location attribute for variables	Contains location expression
DW_AT_ranges	Address ranges attribute	Specifies valid address ranges
DW_AT_comp_dir	Compilation directory attribute	Base directory for relative paths
DW_OP_reg0	DWARF register 0 location operator	Variable stored in CPU register 0
DW_OP_fbreg	Frame base register operator	Variable offset from frame pointer
DW_OP_piece	Piece operator for composite locations	Variable split across multiple locations
DW_OP_deref	Dereference operator	Follow pointer to access value
DW_OP_plus	Addition operator	Add offset to address calculation
DW_OP_breg0	Base register 0 operator	Address relative to register 0
DW_ATE_signed	Signed integer encoding	Type interpretation for signed values
DW_ATE_unsigned	Unsigned integer encoding	Type interpretation for unsigned values
DW_ATE_float	Floating point encoding	Type interpretation for float values
DW_ATE_boolean	Boolean encoding	Type interpretation for boolean values
DW_ATE_address	Address/pointer encoding	Type interpretation for pointer values
DW_ATE_signed_char	Signed character encoding	Type interpretation for char values
DW_LNS_advance_pc	Line number program PC advance	Increment program counter in line table
DW_LNS_advance_line	Line number program line advance	Increment line number in line table
DW_LNS_copy	Line number program copy operation	Emit line table entry

Error Codes and System Constants

Debugger implementations must handle various error conditions and system constants. Understanding these codes is essential for proper error handling and recovery.

Constant	Value/Meaning	Context
DEBUG_SUCCESS	0	Successful operation result
DEBUG_PTRACE_FAILED	-1	ptrace operation failed
DEBUG_PROCESS_EXITED	-2	Target process has exited
DEBUG_INVALID_ADDRESS	-3	Invalid memory address
DEBUG_SYMBOL_NOT_FOUND	-4	Symbol not found in debug information
DEBUG_BREAKPOINT_EXISTS	-5	Breakpoint already exists at address
DEBUG_PERMISSION_DENIED	-6	Insufficient permissions for operation
DEBUG_CORRUPT_DEBUG_INFO	-7	Debug information is corrupted
DEBUG_RESOURCE_EXHAUSTED	-8	System resources exhausted
DEBUG_INVALID_DWARF	-9	Invalid DWARF format
DEBUG_UNSUPPORTED_FEATURE	-10	Feature not supported
EPERM	1	Operation not permitted (errno value)
ESRCH	3	No such process (errno value)
EINTR	4	Interrupted system call (errno value)
EAGAIN	11	Resource temporarily unavailable (errno value)
EFAULT	14	Bad memory address (errno value)
PTRACE_TRACEME	0	Enable tracing of calling process
PTRACE_ATTACH	16	Attach to existing process
PTRACE_DETACH	17	Detach from traced process
PTRACE_CONT	7	Continue execution
PTRACE_SINGLESTEP	9	Single instruction execution
PTRACE_PEEKDATA	2	Read process memory word
PTRACE_POKEDATA	5	Write process memory word
PTRACE_GETREGS	12	Get CPU registers
PTRACE_SETREGS	13	Set CPU registers
INT3	0xCC	x86 breakpoint instruction opcode
WIFSTOPPED	macro	Check if process stopped by signal
WSTOPSIG	macro	Get signal that stopped process
WIFEXITED	macro	Check if process exited normally

Constant	Value/Meaning	Context
WEXITSTATUS	macro	Get process exit status
WNOHANG	1	waitpid option for non-blocking wait

Process and Thread States

Understanding process and thread states is crucial for implementing proper debugger state management and synchronization.

State	Meaning	Transitions
RUNNING	Process executing normally	Can transition to STOPPED or EXITED
STOPPED	Process stopped by debugger or signal	Can transition to RUNNING or EXITED
EXITED	Process has terminated	Terminal state, no further transitions
THREAD_RUNNING	Thread executing normally	Can transition to THREAD_STOPPED or THREAD_EXITED
THREAD_STOPPED	Thread stopped by debugger	Can transition to THREAD_RUNNING or THREAD_EXITED
THREAD_EXITED	Thread has terminated	Terminal state for individual thread

Watchpoint and Advanced Features

Advanced debugging features require additional terminology for implementation and usage.

Term	Definition	Implementation Notes
software watchpoint	Memory monitoring using page protection and trap handling	Uses mprotect() to catch memory access
hardware watchpoint	Memory monitoring using CPU debug registers	Limited number available, faster execution
WATCH_READ	Trigger watchpoint on memory read access	Monitor load instructions
WATCH_WRITE	Trigger watchpoint on memory write access	Monitor store instructions
WATCH_ACCESS	Trigger watchpoint on read or write access	Monitor any memory access
remote debugging protocol	Network communication format for distributed debugging	Enables debugging across network connections
client-server architecture	Split debugger into UI and process control components	Separates user interface from debugging engine
state synchronization	Maintaining consistency between distributed components	Required for remote debugging reliability

Implementation Quality and Testing Terms

Testing and quality assurance require specialized terminology for verification and validation activities.

Term	Definition	Testing Context
integration testing	Testing component interactions with realistic scenarios	Verifies components work together correctly
milestone verification	Specific tests to verify correct implementation of each milestone	Checkpoints for incremental development progress
test framework	Infrastructure for running and verifying debugger tests	Provides consistent testing environment
process isolation	Preventing test interference through process separation	Ensures tests don't affect each other
memory verification	Checking memory state changes during debugging operations	Validates memory reads and writes
corruption detection	Identifying invalid data structures or memory contents	Prevents crashes from bad data
graceful degradation	Reducing functionality when resources unavailable while maintaining core operation	Continues working with limited capabilities
fail-fast	Immediate failure when problems detected	Prevents cascading failures
error propagation	How errors flow between components during testing	Ensures errors reach appropriate handlers

Meta-Debugging and Development Tools

Building a debugger requires using additional tools to debug the debugger itself, creating a meta-debugging environment.

Term	Definition	Usage
meta-debugging	Using GDB to debug the debugger while it debugs target processes	Three-level debugging: GDB → your debugger → target
system call tracing	Monitoring debugger's kernel interactions using strace	Diagnoses ptrace and signal handling issues
binary analysis	Examining executable structure with objdump and readelf	Validates symbol resolution and DWARF parsing
instruction boundary verification	Confirming breakpoints align with instruction starts	Prevents corrupting multi-byte instructions
signal handling coordination	Managing signals between debugger, target, and meta-debugger	Complex interaction requiring careful design
memory consistency validation	Comparing debugger state with target process memory	Ensures debugger accurately reflects target state
DWARF parsing validation	Verifying debug information structure and version compatibility	Prevents crashes from malformed debug data
diagnostic error context	Comprehensive error information with timestamps and process IDs	Enables thorough error analysis
race condition diagnosis	Using timing analysis to identify timing-dependent bugs	Requires careful instrumentation and logging
error propagation tracking	Following errors through component interactions	Maps error flow through complex systems