UPPSALA
UNIVERSITET

IT kDV 25 034
Degree project 15 credits
March 18, 2026

The GNU Emacs Architecture

Unlocking the Core

Erik J. Karlsson

Bachelor Programme in Computer Science

UPPSALA
UNIVERSITET

The GNU Emacs Architecture

Erik J. Karlsson

Abstract

The GNU Emacs text editor is a part of the GNU Operating System and has been in development since the 1980s. Emacs is known for its user-extensible and self-documenting design that ties back to the free software philosophy that advocates for the users freedoms to run, study, change and distribute computer programs. While there is much documentation on the editors programming language Emacs Lisp, there is a lack of accessible documentation which covers the internal architecture. This significantly complicates effort of introducing new developers the Emacs core, restricting the users freedom to study Emacs. The recent introduction of Lisp threads brought a limited form of concurrency to Emacs, is forced to use a Global Interpreter Lock (GIL) to function with the single-threaded architecture of GNU Emacs. This thesis aims to aid future work in modernizing Emacs by providing a foundation of a comprehensive and accessible documentation covering the internal architecture of GNU Emacs, focusing on components related to concurrency and parallel processing. The documentation is paired with a summarizing analysis covering the architectural limitations of the GNU Emacs core related to concurrent and parallel processing. This work hopes to initiate discussion regarding the single-threaded nature of the core and to act as a foundation for future documentation to expand upon. The documentation covers the GNU Emacs source code and build process, including architectural components such as the Command Loop and Lisp Environment while highlighting features related to concurrency such as variable binding, processes and threads. The analysis addresses the constraints of the cooperative concurrency model, summarizing the workarounds used in contemporary Lisp libraries and possible improvements such as the addition of a preemptive thread scheduler, illustrating the complicated task of removing the GIL. The GNU Emacs core relies heavily on shared state, requiring large changes in its most fundamental systems such as the memory allocator and the Lisp environment.

Faculty of Science and Technology

Uppsala University, Place of publication Uppsala

Supervisor: Aletta Nylén Subject reader: Karl Marklund

Examiner: Johannes Borgström

Contents

1 Introduction

GNU Emacs is a text editor which as of 2025 dates back approximately 40 years. The core of the editor is composed of a Lisp environment for the editors own programming language Emacs Lisp (Elisp), which this thesis will refer to as *Lisp*. The Lisp environment is extended with IO capabilities and Lisp primitives for editing text. A subset of issues frequently discussed within the GNU Emacs community [41] are those emanating from the single-threaded architecture of the Emacs core which prevents Emacs from simultaneously executing OS-level threads. This makes demanding computations inevitably slow down the editor, restricting the maximum achievable performance of Emacs Lisp, limiting the scope of feasible programs.

The GNU Emacs core [3] has used a single threaded, sequentially executed design since (at least) the release of GNU Emacs 16.56 [31]. The GNU Emacs core is primarily made up of a command loop, which allows the user to execute commands within the Lisp environment. One limitation of this single-threaded design is that long-running Lisp tasks will delay the command loop in wait for the command/function to finish executing. This delay postpones the invocation of important auxiliary tasks responsible for maintaining the editors interactivity, rendering the editor unresponsive. This absence of simultaneous execution of OS-level threads creates a bottleneck for the Lisp environment.

A mostly cooperative implementation of threads was introduced into Emacs Lisp in GNU Emacs 26 [4]. In this model, thread switching occurs at specific, well-defined points, such as when waiting for mutex locks, waiting on condition variables, waiting for process output, or through explicit thread yielding. This places a substantial responsibility on programmers to craft code that yields to other threads correctly and efficiently.

A Global Interpreter Lock (GIL) was used to synchronize the threads on the single threaded architecture of GNU Emacs. The GIL prevents Emacs from making progress on multiple threads at once, switching between user-created threads at well defined occasions, processing them one at a time. The sheer size and intricate single-threaded nature of the Emacs core, coupled with its lack of thread safety, renders the Herculean task of unlocking the Emacs core immensely challenging.

The philosophy behind GNU Emacs revolves around encouraging the users freedom to study and change Emacs. This is promoted in many ways, such as through the self-documenting capabilities of Emacs. Emacs Lisp allows the programmer to specify which parts of the code is meant for user customization. A customization interface then allows the user to change these options without writing any code. Documentation strings (or DocStrings) can be provided upon the definition of Lisp objects and allows the user to write documentation for Emacs Lisp commands, functions, variables, packages and key bindings. Emacs includes a hypertext documentation browser for interacting with the GNU Emacs manuals and the documentation of various Lisp packages/libraries. This documentation covers most aspects of GNU Emacs and Emacs Lisp.

GNU Emacs ships with three manuals which covers how to use Emacs and how to modify and extend Emacs using Emacs Lisp. The included manuals are; *the GNU Emacs FAQ* [21], *the GNU Emacs Manual* [4] and *the GNU Emacs Lisp Reference Manual* [22].

Across all manuals only one chapter covers the Emacs core (see *GNU Emacs internals* [22, Appendix E]). The chapter covers how to extend the Emacs core with new Lisp primitives and how to write dynamic modules. It provides some information on the most basic internal data structures, how to use the various helper macros together with some information regarding memory allocation and garbage collection. While the chapter covers how to extend the Emacs core, there is little to no information available regarding the internal implementation or its architecture. Because of this, new core developers are forced to deduce the internal architecture from the higher-level Emacs and Lisp reference manuals or through reading the changelogs.

The Emacs core is composed of 268 C source and header files [3]. The files are on average 2150 lines, with some files even reaching up to around 40000 lines of code. Around 33% of these files use `goto` statements, with the files that utilize `goto` statements, using them on average, 17 times. The `goto` statement is notorious for producing hard to follow code [14]. The scarcity of readily available documentation makes the initial learning curve for the Emacs core steep and time-consuming, posing a significant hurdle for new developers who must essentially start from scratch.

The absence of any accessible documentation that covers the Emacs architecture and core compels new developers to rely on suboptimal resources such as; the git changelogs, the Git commitlogs and the source code comments. The source code comments, in particular, vary widely in quality, frequency, location, and style, making it difficult for those unfamiliar with the codebase to locate information on specific topics.

To uphold its fundamental principles of hackability, extensibility, and customizability [20], GNU Emacs must ensure that these principles extend to all facets of the editor, including the complex internals and core. Promoting and fostering Emacs core development is paramount, since the internals defines the capabilities of the Lisp environment that forms the foundation of the editor. Emacs' power is intrinsically linked to the implementation of its core. Addressing issues such as the lack of a robust concurrency model necessitates modifications to the core.

Accessible documentation covering the architecture of the Emacs core and internals would significantly lower the barrier to entry for new core developers, accelerating Emacs' evolution. In the same way as the Emacs Lisp manuals encourage users to modify Lisp components, a comprehensive documentation of the core would empower a wider audience to understand and contribute to its development.

External information regarding the Emacs core are scarce but do exist beyond the limited manual section. Some notable sources on the GNU Emacs internals include; The *Hackers Guide* on the *Emacs Wiki* [40] which includes a short section on the GNU Emacs core. The book, *The Craft of Text Editing* [17] which features information regarding the design of emacs-type text editors. Commentary present in the comments of the GNU Emacs source code. The Emacs developer mailing list [7] where it is possible to discuss GNU Emacs development and search through old threads. The Git commit log from the GNU Emacs Git repository [3] includes information regarding changes made to the GNU Emacs source.

Due to the single-threaded architecture of Emacs, several workarounds trying to tackle the lack of multi-threading has emerged, e.g `async.el` [39]. Since these workarounds have to run on the single thread of execution within the interpreter and command loop, they cannot be concurrent in the sense of simultaneous execution. Because of this, another less formal definition is needed.

GNU Emacs is constructed with the goal of being a **WYSIWYG** (What You See Is What You Get) editor where data is edited in its final form. One could imagine that the practical purpose of concurrency in Emacs-type text-editors is the minimization of delays present between the cause and effect experienced by the user, as it breaks out of the WYSIWYG abstraction.

It is thus beneficial to talk about a form of pseudo-concurrency defined as the minimization of latency in the interactivity of Emacs, that is, minimizing the temporal distance between user input and output in the UI or techniques used to hide the unwanted delays and latency in the interactivity of Emacs caused by Lisp evaluation. This differs from the purely technical and commonly established definition of concurrency, where concurrency is about executing multiple things at once, often relating to more specific concepts, such as multi-threading or multi-tasking. Something often achieved through context-switching where the task execution context is swapped so fast that it seems like tasks are executing at the same time, or where multiple tasks and their execution context run independently, at the same time on different cores, i.e parallelism.

This thesis will consider concurrency a spectrum between pseudo concurrency and parallelism, where the thesis tries to contribute with an accessible and comprehensive documentation of the Emacs core, while identifying and highlighting issues related to concurrency, paired with suggestions for potential improvements. The objectives ofthis thesis are.

TO1 Providing the foundation for comprehensive and accessible documentation on the internals of GNU Emacs and the architecture of the Emacs core while focusing on components, concepts, and topics relevant to the ability of Emacs to perform concurrent and parallel processing.

TO2 Examining the current capabilities of GNU Emacs to preform concurrent and parallel processing, relating to the documented components of TO1. Finding and summarizing potential issues and limitations that hinder Emacs' concurrent capabilities while suggesting improvements where possible.

The first objective TO1 aims to lower the barrier of entry into Emacs core development, to empower more users to modify and understand the Emacs core. In conjunction, TO1 and TO2 has the

potential to accelerate development on larger issues with concurrency inherent to the Emacs core by making it accessible to more developers.

2 The History and Philosophy of GNU Emacs

Emacs grew out of a collection of editing macros and extensions written for the MIT Lab's text editor TECO (Text Editor and COrrector) [42, ch.6], originating from around 1962 [28].

```
3<svim$-3C3d$Iemacs$>
```

Listing 1: TECO program, replacing the next 3 occurrences of "vim" with "emacs".

TECO was a character-based command dispatcher, acting both as a text editor and a programming language [28] [42, ch.6]. To use TECO, the user entered a "command string" whose individual characters corresponded to specific commands. The command string was then executed after "End of command" character (two ALT characters, written as "\$\$"). listing 1 illustrates a TECO program that replaces the next three occurrences of the word "vim" with "emacs" after the cursor, within the current buffer [1].

A pivotal moment occurred during Richard M. Stallman's (RMS) visit to Stanford, where he encountered the WYSIWYG (What You See Is What You Get) text editor E. This editor, unlike TECO, would dynamically update the display after each keystroke, a feature that left a deep impression on RMS.

Upon returning to the MIT AI lab RMS merged TECO with a hack written by the Carl Mikkelson, making it possible to execute commands using two keystrokes. He continued to implement the ability to save and load macros from files, allowing the macros to be bound to arbitrary two-key combinations. These changes turned TECO into a user-extensible WYSIWYG text-editor [42, ch.6].

The hacker spirit at the MIT AI Lab was built upon knowledge sharing, with programmers moving between each others desks, trading knowledge through modifying and commenting each others source code. Over time, the constant modifications to TECO led to a convoluted codebase, where everyones TECO source code had diverged to such an extent that programmers had to spend hours figuring out their coworkers various TECO macros before they could use their computer. In 1976, Dave Moon, RMS and Guy Steele curated the most useful TECO macros, documented them, finally putting them together as under the name EMACS (Editor MACroS) [21].

Motivated by a strong desire to preserve the hacker doctrine of innovation sharing that had thrived at MIT, RMS embedded terms of use into the source code of Emacs. The terms of use stating that the users were free to share and modify the program, *as long as they shared their modifications back with RMS*. This concept, which RMS termed, the "Emacs Commune" [42, ch.6], was not universally accepted, and multiple Emacs-like programs started to appear throughout the lab. Some notable examples include; Eine (Eine Is Not Emacs) and Zwei (Zwei was Eine Initially) [42, ch.6], written for two of the MIT Labs Lisp machines. Eventually Emacs gained traction outside of MIT, resulting in diverse implementations across different programming languages and operating systems.

RMS later resigned from the AI Lab and started his work on the GNU Project (GNU is Not Unix), aiming to develop a free [2] UNIX-like operating system. After thinking a C compiler would take too long to implement, he decided the first GNU program should be an Emacs. To avoid "reinventing the wheel" RMS started development on GNU Emacs by copying parts of the source code from the editor, Gosling Emacs (Gosmacs). Gosmacs was an Emacs written in C for UNIX which used an extension language called Mocklisp. Mocklisp was syntactically similar to Lisp, but was not a Lisp language since it lacked features such as the list data-structure. After Gosling sold the rights to Gosmacs to UniPress, UniPress threatened legal action. RMS then started phasing out the copied code, reverse engineering the Mocklisp interpreter and replacing it with his own Emacs Lisp interpreter [42, ch.7].

GNU Emacs was released to the public in 1985 [33] followed two months later by the GNU Manifesto, partly a response to the UniPress incident [42, ch.7]. GNU Emacs quickly gained traction among UNIX developers, prompting RMS to create the non-profit organization *Free Software Foundation* (FSF) to handle the

[1] An Emacs package emulating TECO called "teco" can be used execute TECO programs.

[2] Free as in freedom, not as in free beer.

business which arose after the release. In 1985, GNU Emacs earned around 66,700$ (adjusted for inflation) of net-profit from mostly tape sales [42, ch.7].

2.1 The Free Software Philosophy

GNU Emacs distinguishes itself by its close ties to the philosophy and practice of software freedom [33]. This philosophy, now known as free or libre software, originated with RMS' work on GNU Emacs in the 1980s [42, ch.6]. GNU (GNU is Not Unix) is a project created by RMS in order to create a non-proprietary UNIX-like OS, with GNU Emacs being the first GNU program. GNU Emacs holds a unique position of being explicitly created with the intention of being free (as in freedom) and many of its design decisions revolves around promoting user freedom [33]. Many of the ideas that came to be the free software philosophy either evolved alongside GNU Emacs or were directly composed within the Emacs environment itself [33], the release of GNU Emacs leading to the establishment of the Free Software Foundation (FSF) [42, ch.7].

For software to be considered *free* it must adhere to the four essential freedoms. The four essential freedoms as outlined by the FSF [18] are enumerated below, where access to the source code is a precondition for freedoms F1 and F3:

> **F0** "The freedom to run the program as you wish, for any purpose"
>
> **F1** "The freedom to study how the program works, and change it so it does your computing as you wish [...] [a]ccess to the source code is a precondition for this"
>
> **F2** "The freedom to redistribute copies so you can help others"
>
> **F3** "The freedom to distribute copies of your modified versions to others [...] by doing this you can give the whole community a chance to benefit from your changes"

GNU Emacs is inherently designed to facilitate the four essential freedoms. Its rich feature set, emphasis on extensibility, and broad customizability exercises freedom F1, while aiding the user in changing the purpose of Emacs F0. The self-documenting capabilities of Emacs, coupled with its Lisp package system, promotes and simplifies the redistribution of both copies and and user-modified versions of Emacs and Emacs packages (F2, F3). Packages can easily be installed through the package repositories without the need to recompile Emacs, automatically integrating installed packages into the documentation system (F2, F3) significantly simplifying the users ability to study and change and Lisp packages F1.

3 Contemporary GNU Emacs Forks and Concurrency

The inherent ability of GNU Emacs to achieve concurrency is a consequence of the design and architecture used in the Emacs core. The source code for the core can be found in both the GNU Emacs repository on Github [3] and through the GNU Git repository which the Github repository mirrors. In terms of Git, forks are derivations of repositories that copies the original source code and version tracking information so that it is possible to modify it without disturbing the original project.

Forks have historically had a large impact on the evolution of GNU Emacs, one example being XEmacs [32] [34] [16] a GNU Emacs fork which was actively developed from the early 1990s to late 2010s. A schism between the GNU Emacs and XEmacs developers divided the Emacs community and gave rise to a competitive environment which although negatively affecting the GNU Emacs community as a whole, ended up accelerating GNU Emacs development. Much of the features today present in GNU Emacs can be traced back XEmacs [34] [16] [32], such as the tool bar, scroll bars, GTK support, dynamic modules, inline images [1] [34] [16]. This section will summarize two recent GNU Emacs forks that both tackle the problematic concurrency of GNU's Emacs; Emacs NG [8] and Commercial Emacs [2]. This aims to show the current state of Emacs core development relating to improving concurrency in GNU Emacs.

Emacs NG [8] is a fork of the GNU Emacs master branch [3], which rewrites and extends parts of the GNU Emacs core using the Rust programming language. The stated goal of Emacs NG is to explore new development approaches [8].

Emacs NG includes modifications to the dynamic modules of Emacs, where the module system has been extended to allow

modules greater access to the Emacs internals, while remaining fully compatible with GNU Emacs [8], a feature currently marked as unmaintained. Emacs NG also includes the 2D rendering engine WebRender which is capable of leveraging the GPU for content rendering [8], including support for WebAssembly. Furthermore, Emacs NG includes the following experimental features, all of which are marked as outdated and currently disabled [8]: a built-in JavaScript/TypeScript environment, an asynchronous I/O environment, and support for parallel JavaScript execution by using web workers.

Emacs NG introduces a thread-local singleton to the `thread_state` struct, which contains the JavaScript runtime and its associated state. While Lisp globals are shared between threads, consistent with the default behavior in Emacs, JavaScript globals and variables are kept thread-private as each Lisp thread possesses its own JavaScript environment. Web workers utilizing Rust's `std::thread` (analogous to C's `pthread`) are employed to enable parallelism within Emacs NG [9]. The Emacs NG timers and asynchronous I/O leverage the Rust library Tokio for asynchronous task execution, utilizing a thread pool to enqueue tasks [10]. Asynchronous tasks are then polled for completion using a Lisp timer. JavaScript Proxy objects are used to facilitate object translation between JavaScript and Lisp.

The use of JavaScript to extend GNU Emacs has been explicitly denounced by Richard Stallman (RMS) [33], with his statement regarding JavaScript seemingly referring to the Emacs NG project, without explicitly mentioning it. The modifications made to the Emacs core to integrate JavaScript could potentially allow for the integration of other languages, such as Guile Scheme, a language that fit closer to the needs and aesthetic of Emacs [33], being the GNU projects extension language and a Lisp. The `README.md` file [8] in the Emacs NG repository states that the purpose of Emacs NG is not to replace Emacs Lisp, but rather to make GNU Emacs more approachable to a wider audience by employing a more conventional and mainstream programming language. Whereas Emacs Lisp is used to implement most parts of GNU Emacs, whose source code is accessible and modifiable at runtime, JavaScript exists in a separate environment without this property. The use of JavaScript could thus never compare to that of Emacs Lisp unless large parts of GNU Emacs were to be rewritten.

The official repository for Emacs NG remains active [8]. However, due to many significant features being either abandoned, disabled, or unmaintained, the future evolution of Emacs NG is uncertain. Nevertheless, components of the project hold potential, and might be useful if ported to GNU Emacs, especially the extensions made to the dynamic module system.

Commercial Emacs (ComEmacs) [2] is a GNU Emacs fork with major experimental modifications to the Emacs internals, diverging significantly from the GNU Emacs. ComEmacs incorporates numerous modifications to GNU Emacs, including a rewrite of the process management system and garbage collector, as well as modifications to the GNUS newsreader making it non-blocking, as listed in the ComEmacs `README.md` of the Commercial Emacs repository [2]. However, there is limited documentation regarding the project and its various features.

A notable modification not mentioned in the repository's `README.md` file [2] is the reworking of Emacs' threading API. This includes the addition of a preemptive thread scheduler and the removal of the GIL, enabling multiple threads to run simultaneously. The experimental nature of these features requires them to be explicitly enabled by using the `./configure` configuration script. The creator of the fork has provided a demonstration video showcasing the multi-threading implemented in ComEmacs [13] and another video briefly summarizing challenges associated with a multithreaded Emacs [12].

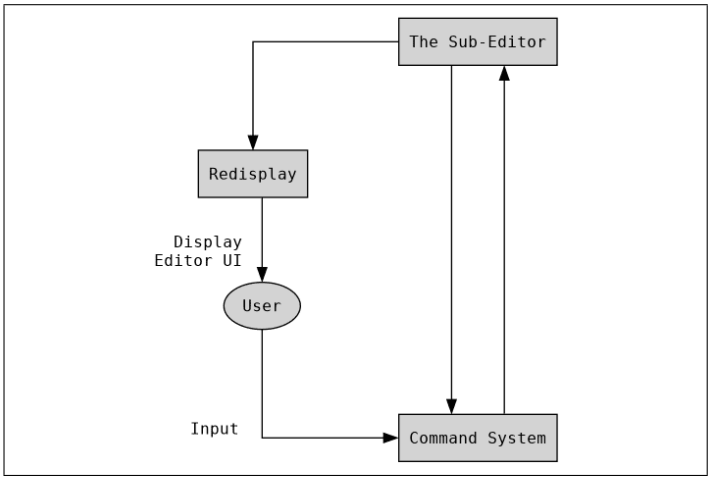

Figure 1: Structure of ACTE.

4 An Emacs-Type Text Editor

In order to introduce concepts present in GNU Emacs, this section will present a simplified, top-down model of an Emacs-type text editor, which we will refer to as **ACTE** (ACTE Conceptual Text Editor).

The structure and components of ACTE is derived from the text editor presented by Finseth [17] where ACTE adds onto summarized parts of Finseths book, tailoring it to better reflect the architecture of GNU Emacs and the sections of this thesis.

ACTE is decomposed into three primary components which can be seen in figure 1; the Sub-Editor (section 4.2), Redisplay (section 4.3) and the Command System (section 4.4).

4.1 Concepts and Definitions

A **buffer** is a data-structure that represents a unit of text [17, ch.6] corresponding to the contents of at most one file, or none at all. Text editing occurs by modifying the contents of buffers. The editor can have multiple buffers open and one buffer being active at a time. This buffer is called "the current buffer" and is the buffer which the editor operates on.

Operations that modify the contents of a buffer are executed at the location of a movable position within the buffer called the **point**. The point is positioned between two characters, and resides after the beginning of the buffer and before its end. Conventionally, the point is located on the left edge of the cursor [17, ch.6].

A **mark**, like the point, designates a position within a buffer. However, unlike the point which is unique per definition, a buffer can contain multiple marks. Marks can be classified into two types; *normal* and *fixed* marks. Normal marks places the point of insertion on the left edge of its position between two characters, where like the point, text is inserted before the mark and will move its position. Fixed marks has their point of insertion on the right edge, where inserting text won't move the marks position [17, ch.6].

The **region** is defined as the continuous span of text between the point and a mark when exactly one mark exists in a buffer [17, ch.6]. Regions enable the editor to operate on contiguous spans of text, facilitating actions like cutting or copying.

User interaction is command-driven, where the user executes **commands** to manipulate the editor. The commands are mapped to by sequences of keys / characters called **keybindings**, which when detected invokes the execution of their command.

A mode is a set of command rebindings [17, ch.8] and editor configurations that are enabled on a per-buffer basis. There are two types of modes; minor and major modes, where only one major mode can be enabled in a buffer, along with zero or more minor modes.

4.2 The Sub-Editor

The internal sub-editor [17, ch.6] is an abstraction serving to encapsulate the non-interactive foundation of ACTE in a operational core of procedures. The sub-editor defines a set of higher-level auxiliary procedures that interface with the editor's core data-structures and objects. This architectural decomposition hides the editor's data-structures through a interface of higher level procedures, reducing complexity and acting as a interface used to implement commands.

The sub-editor interface includes procedures for controlling and manipulating the sub-editor. It includes procedures for manipulating and interacting with buffers, modes, points and regions. The procedures are interconnected into more complex and general procedures for initializing the editor and its components, setting up the editor's initial configuration, terminating the editor,

saving and loading the editor state etc [17, ch.6]. Some sub-editor procedures are implemented with access to the redisplay component (see section 4.3) as many of the sub-editor's procedures change the graphically rendered state of the editor. An example of one such procedure is the "insert-char" procedure, since when we insert a character into a buffer we can assume that the change in editor's state also implies a change in the visual render of the editor.

4.3 The Redisplay Component

The second component of ACTE is called **redisplay**. The redisplay component is responsible for synchronizing the editor's internal state (such as buffers), to the graphical user interface (GUI) rendered by the OS. Redisplay is implemented with direct access to the editor's data-structures without the need to go through the sub-editor interface.

Redisplay is the most complicated and computationally heavy components of the editor. This is because it has to determine exactly which parts of the rendered user interface has changed, in order to only redraw the changes [17, ch.6]. This is especially complicated in GUI Emacsen, as redisplay relies on high-performing incremental rendering algorithms [23].

The UI of Emacs-type editors is often composed of **windows** and **frames**. A **frame** is the outer-most part in the UI that encapsulates all of the editors **windows**. A **window** is the part if the UI that displays buffers. A window displays the content of one buffer at some location and can be split into multiple windows, each displaying the same or different buffers. Redisplaying a frame redisplays all of its windows and redisplaying a window synchronizes the rendered buffer to its contents.

4.4 The Command System

The command system in ACTE refers to all interactive, user and command related components and includes components such as the **command loop**, **commands**, the **dispatch table**, **command execution** and **command sets**.

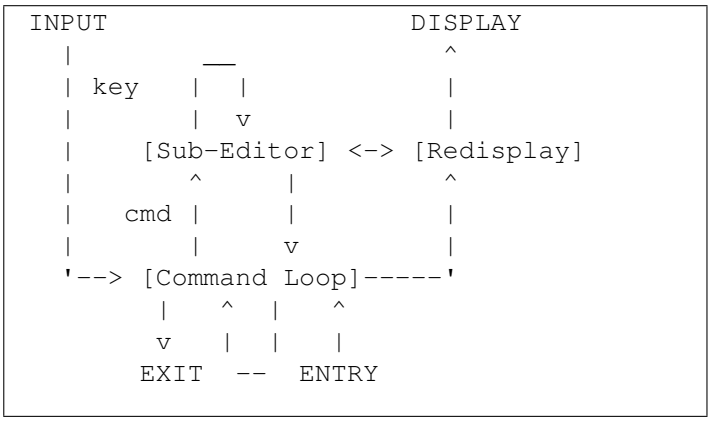

Figure 2: Simplified ASCII-diagram of ACTE.

Figure 2 illustrates a simplified diagram of ACTE' components, where `ENTRY` is the point of entry into ACTE, `EXIT` is the exit out of ACTE, `INPUT` is user input and `DISPLAY` is the rendered text editor / display.

The command loop consists of a loop which reads user input, translates the input into a command and executes it, updating the UI as it changes. The command system includes procedures for waiting on and reading a character / key from user input.

Keybindings are stored using dispatch tables which defines which keys are bound to what commands. Multiple dispatch tables are used for bindings that use sequences of keys. The active bindings and their bound commands are referred to as the command set, with Emacs-type editors typically including the implementation of a specific Emacs-like command set [17, ch.10] which the user typically can modify and extend by defining new commands [17, ch.10].

```
void command_loop()
{
  key_t key;
  bool is_exit;
  do {
    /* Read a key */
    key = read_key();
    /* Evaluate the command */
    is_exit = evaluate(key);
    /* Update display */
    redisplay();
    /* Loop */
  } while (!is_exit);
  /* Exit editor */
}
```

Listing 2: A simple command loop.

The command loop is structured like a Read Eval Print Loop (REPL) and is responsible for the editors command driven user interaction. Listing 2 provides an illustrative implementation of the command loop routine using a C-like language.

The listing illustrates the primary algorithm responsible for user interaction. The algorithm first reads a key from the event queue which events asynchronously are added to. The procedure `read_key` is then used to wait for a input event and returns its key. The key is then passed to the `evaluate` procedure (see listing 3) after which redisplay is called, since the command may have changed rendered parts of the editor. If the command returns `true` the editor will exit the command loop.

```
typedef bool (*command_t) (bool *, key_t);

bool
evaluate(key_t key)
{
  bool is_exit = false;
  /* Lookup command */
  command_t cmd = dispatch(key);

  /* Evaluate command (if any) */
  while (!cmd(&is_exit, key)){
    /* Read additional input for
     * incomplete commands */
    key = read_key();
    redisplay();
  }
  /* Exit to the command loop */
  return is_exit;
}
```

Listing 3: The evaluation procedure.

Listing 3 implements the routine for command evaluation. The listing use a dispatch table, which is indexed using the `dispatch` routine. The dispatch routine is used to find the commands bound to key keys. The argument `is_exit` is set to `true` if the command exits the command loop and the editor. The `key` argument allows the same command to read additional input without exiting to the command loop. This allows the use of keybindings that use multiple keys. The command is incomplete whenever the command procedure `cmd` returns `false`. Incomplete commands are commands that read additional keys to allow key sequences to be bound to commands, or to create commands that doesn't have a static keybinding.

4.5 Model View Controller

The MVC (Model View Controller) architectural model is an architectural model which defines the interaction between and the separation of user interfaces, data structures and user control logic [27]. It consists of three different layers, each with different responsibilities.

I. **The model layer** consists of datastructures and fundamental procedures to interface with them.

II. **The Viewer layer** is composed of everything which interacts with the user, i.e the user interface and displays the datastructures of the model layer.

III. **The controller layer** takes the user input from the view layer and manipulates the model layer or updates the UI of the view layer.

ACTE can be described through the MVC model, where the **view** layer consists of **windows**, **frames** and the **redisplay** component. The **controller** component consists of the **command system**, i.e the command loop, commands, keybindings, together with primitive input procedures, dispatch tables and the evaluation procedure. The **model** layer consists of the **sub-editor** procedures together with the editors most basic data-structure, i.e the buffer.

5 Introducing Emacs Lisp

Since the essence of GNU Emacs is a specialized Lisp environment, studying and documenting the editor and its concurrency / parallelism will eventually transforms into studying Emacs Lisp and its implementation in the core. To learn about Emacs and its architecture it is thus essential to learn about Lisp.

This section features an introduction to reading Lisp, where section 5.1 which contains a general description of the Lisp and Lisp syntax and some of the reasoning behind it. This section primarily targets non-Lisp programmers. And much of this thesis covers the lower level C code and architecture of the Emacs core and its interaction with Lisp, some features of Emacs Lisp that consistently occur throughout this thesis related are introduced in section 5.2. This is meant to give a top-down introduction to some of the important Emacs concepts from the perspective of Emacs Lisp before their implementation reappears later in the thesis. An introductory and more comprehensive Emacs Lisp tutorial is included in section C, this intends to serve as both, a place to understand the basics of Emacs Lisp and as a point of reference to look up parts of the language.

GNU Emacs is designed by the core tenants of Emacs-type text editors, namely extensibility and customizability. To facilitate the ability to customize and extend the editor, Emacs utilizes its own programming language, Emacs Lisp (Lisp). Lisp is a dialect of Lisp (LISt Processing), a family of programming languages where lists serve as both the primary data structure and the syntactic representation of source code. This distinctive characteristic gives Lisp languages a simple and consistent syntax, characterized by the extensive use of parentheses, which is used to denote lists and symbolic expressions.

At its core, Emacs defines a bare Lisp interpreter for Emacs Lisp. This interpreter consists of a set of fundamental Lisp primitives which are defined in C, together with a subsystem for I/O operations [22, Appendix E.1]. The majority of GNU Emacs's functionality, such as its text editing capabilities, are implemented on top of this core using Lisp. Around 75% of the GNU Emacs source code (measured in lines of code) is written in Lisp. This design choice facilitates the portability of Emacs Lisp code across different implementations of the Emacs core. Emacs Lisp code can also be byte-compiled and executed using

the built-in byte-code interpreter. The byte-code can then be further compiled into native machine code that executes directly on the CPU using Emacs' Just-in-Time (JIT) compiler.

Emacs features a self-documenting system that catalogs and interconnects defined Lisp code and any loaded source files with the included Lisp manuals and help system. Emacs Lisp's primarily interpreted nature allows the user to study and change the most of the Emacs source code, while allowing users to modify the code executing within the editor at runtime.

5.1 A Introduction to Lisp Syntax

In Lisp, expressions are written in **polish notation**, which sometimes is referred to as prefix notation. Polish notation puts the operator of mathematical expressions and functions in front of the operands instead of between them.

The infix expression $10+20+\frac{2}{4 \cdot 100}$ is equal to `+ 10 20 (/ 2 (* 4 100))` in prefix notation. In Lisp, expressions are represented as a lists (of expressions) and parentheses are used to denote procedure applications. The leftmost element in a Lisp expression is the operator and the rest of the elements are the operands. For example, the Lisp expression `(f a b c)` applies the operands `a`, `b` and `c` onto the operator `f`.

Lisp variables use the '-' character per convention, and usually not '_'. To differentiate between C and Lisp, this thesis implies that any variable name that contain the '_' character will refers to C, and since the '-' character cannot be used in the name of C variables, symbols names that contain '-' refer to Emacs Lisp, unless explicitly stated otherwise.

5.1.1 Lists and S-Expressions

In Lisp, the fundamental building block is the pair, or the `cons` operator `(cons <car> <cdr>)`. The `cons` cell is a data structure that holds two elements: the first element is called, the `car` and the second element is called, the `cdr`. Linked lists are constructed from a chain of interlinked `cons` cells, where the `car` holds an item, and the `cdr` is either set to `nil` (nothing / empty list) or points to another cons cell, i.e `(cons <item> (cons ... (cons <item> nil)))`. A `cons` pair can be written using a quoted syntax `'(x . y)`, where the left hand side of '`.`' is the car and the right is the cdr. listing 4 illustrates how lists are composed of `cons` cells.

```
(cons x y)
    <=>
'(x . y)

(list 1 2 3)
    <=>
'(p1 . (2 . (3 . nil)))

'(1 2 (3 4) 5)
    <=>
'(1 . (2 . ((3 . (4 . nil))
   . (5 . nil))))
```

Listing 4: Quoted syntax and lists as Nested Pairs.

The tree-like structure that constitute nested lists in Lisp is called symbolic expressions (or S-expressions / S-expr). Lisp is homoiconic, which means that S-expr are used both as the internal representation of code (i.e the syntax of source code) and exists as a native data-structure. Code can be converted to and from the data-structure allowing programs to modify themselves.

It is possible to write Emacs Lisp code in the non-evaluated S-sexpr form using the `quote` construct, with the syntax; `'(...)` or `(quote (...))`, for example; `(quote (+ 1 2))` is equal to `(list '+ 1 2)`. Formatted quotes can be created using the `backquote` (shorthand `` ` ``) macro, where each sub-expression that begins with "`,`" is evaluated and the result quoted, example; `` `(+ 99 ,(* 2 4) ,(if (> 1 2) '(- 2 3) '(+ 2 3))) `` becomes `(+ 99 8 (+ 2 3))`. The `backquote` construct is similar to the `quasiquote` construct used in other variants of Lisp such as scheme.

```
(defun fib (n)
  "The N'th number in the Fibonacci
   sequence."
  (if (= n 0) 0

    (let* ((n-1 0)
           (i   1)
           (tmp 0)
           (n-2 1))

      (while (< i n)
        (setq tmp n-2
              n-2 (+ n-1 n-2)
              n-1 tmp
              i   (1+ i)))
      n-2)))
```

Listing 5: Iterative Fibonacci function written in Emacs Lisp.

```
defun fib(n)
  "The N'th number in the Fibonacci
   sequence."
beg
  if (n == 0) then return 0
  else
  do
    let n-1 = 0
        i   = 1
        tmp = 0
        n-2 = 1
    in
      while (i < n)
      do
        tmp = n-2
        n-2 = n-1 + n-2
        n-1 = tmp
        i   = 1 + i
      end
      return n-2
   end
end
```

Listing 6: Pseudo-code equivilent to the Elisp function `fib` in listing 5.

The high amount of parentheses used in Lisp can sometimes present a challenge to programmers accustomed to other languages. To illustrate this, a Elisp function (`fib`) that iterative computes the `n`'th number in the Fibonacci sequence is provided in listing 5. The same function is given as pseudo-code (in listing 6) using prefix notation, the same function names, without the Lisp parenthesis and with some symbols added for clarification.

5.2 Lisp and Emacs

All concepts in this section will reappear later at a lower level in the architectural parts of this work. Since Emacs is command-driven, where user interaction occurs through commands, due to this a primitive example of a Emacs Lisp command has been given in section 5.2.2. This aims to provide a real-world use case of Emacs Lisp and is not essential to understanding the later parts of this thesis. Ways for the user to configure / control the behavior of Emacs using major / minor-modes and hooks are introduced in section 5.2.1. Finally, variable bindings in Emacs Lisp are introduced in section 5.2.3. These sections are needed to, at a higher level, understand Lisp evaluation, the components of the Emacs core and the concurrency restricting issues relating to variable binding.

5.2.1 Modes and Hooks

Major modes determine how Emacs interacts with a buffer or text. **Minor modes** provide different features that can be enabled while editing [22, ch.24]. Each buffer has at most one Major mode enabled [22, ch.24.2] together with zero or more Minor modes. Both minor and major modes are interactive functions ending in `-mode`. Some examples of major modes are `emacs-lisp-mode` for editing Emacs Lisp code, `c-mode` for editing C source code and `info-mode` which is enabled inside the Emacs info browser.

```
(add-hook 'prog-mode-hook
          #'display-line-numbers-mode)
```

Listing 7: Hook for `c-mode`, enabling Emacs to show line numbers.

Hooks are variables provided for user customization which contains a list of functions which run on during specific occasions [22, ch.24.1], such as before Emacs exits, after a mode is enabled or before a buffer is saved. A **normal hook** is a hook with the `-hook` siffix, whose functions are called without any arguments. Listing 7 shows the function `display-line-numbers-mode` (minor mode) being added to the `prog-mode-hook` hook which runs after any programming related major mode. The function `display-line-numbers-mode` enables Emacs to show line numbers in a buffer.

5.2.2 Creating a Command

When writing a document in the typesetting language LaTeX one might want to encapsulate a set of mathematical equations inside of a `equation` block, i.e between the strings `\begin{equation}` and `\end{equation}`. listing 8 is provided as an example of using Emacs Lisp to extend the text-editing capabilities of Emacs.

```
(defun region-encapsulate (start end)
  "Insert START and END on new lines around
the current region."
  (interactive)

  (save-excursion
    (let ((beg-reg (region-beginning))
          (end-reg (region-end)))

      (goto-char end-reg)
      (insert end)
      (newline)
      (goto-char beg-reg)
      (newline)
      (insert start)
      (newline)
      )))
```

Listing 8: Example of Emacs Lisp function `region-encapsulate`.

In the listing, the function `region-encapsulate` is created and will surround the active region (span of selected of text) with two strings called `start` and `end`, inserted on new lines before and after the beginning and end of the region. The following call will encapsulate the active region in a LaTeXequation block `(region-encapsulate "\\begin{equation}" "\\end{equation}")`.

```
;; Define a key for latex-mode for using
;; region-encapsulate
(define-key LaTeX-mode-map (kbd "C-c M-e")
 '(lambda ()
    (interactive)
    (let ((s (read-string "block name: ")))
      (region-encapsulate
       (format "\\begin{\%s}" s)
       (format "\\end{\%s}"   s)))))
```

Listing 9: Binding `region-encapsulate` to a key in `latex-mode`.

Listing 9 shows how the function can be used when editing. A lambda function is created in the listing and bound to `C-c M-e`

in `latex-mode-map` (the keymap of `latex-mode`). The function prompts for the name of a latex block and then constructs strings for the beginning and end of the latex block, calling `region-encapsulate` with them as arguments.

The `newline` function used in listing 8 is the same function as the function called when return is pressed on the keyboard, `goto-char` also has a keybinding `M-g c`. As `region-encapsulate` moves the point, the special-form function `save-excursion` (see [22, ch.31.3]) is used which saves and restores the identity of the active buffer and the value of its point.

5.2.3 Lexical and Dynamic Bindings

The difference between lexical and dynamic bindings is that; the value of a **lexical bindings** depends on the *syntactic lexical structure* of the program, while the value of a **dynamic binding** depends on the *runtime environment* of the program.

Emacs Lisp is by default a dynamically scoped language, however it supports lexical bindings which can be enabled locally inside of a buffer. A given buffer will use dynamic binding unless the buffer-local lisp variable `lexical-binding` is set to a non-`nil` value [22, ch.12.10.4] on the first line of the buffer. The use of lexical bindings is inherited in such a way that all buffers loaded from a lexically bound buffer will also use lexical bindings.

```
;;; -*- lexical-binding: t; -*-
```

Listing 10: Enabling Lexical Bindings on the first line of in a Buffer.

To enable the use of lexical bindings locally in a buffer, the variable `lexical-binding` has to have a non-`nil` value. This must be set on the first line of the buffer and is usually accomplished by adding a file local variable (see listing 10).

Lexical scoping was introduced to Emacs in version 24.1 partly due to the good compatibility with concurrency [22, ch.12.10.3]. The usage of lexical bindings is expected to increase in the future due to factors such as better compatibility with concurrency such as code optimizations [22, ch.12.10.3], with the Elisp manual stating that the use of dynamic bindings is slowly being be phased out in place of lexical bindings, and its use is expected to decrease in future Emacs releases [22, ch.12.10].

```
#!/usr/bin/sh
LEX_COUNT=0
FILE_COUNT=0

for f in $(find lisp -name '*.el' -type f);
do
  FILE_COUNT=$(\
     calc $FILE_COUNT + 1\
  )
  LINE_1=$(\
     head -n 1 $f\
  )
  HAS_LEX=$(
     echo $LINE_1\
     | grep -I "lexical-binding: *t"\
  )
  if [[ $HAS_LEX  == "" ]]
  then echo $f
  else LEX_COUNT=$(\
    calc $LEX_COUNT + 1\
  )
  fi
done
printf " \%d/\%d files use"\
       "lexical bindings\n"\
       $LEX_COUNT $FILE_COUNT
#  ==> 1558/1558 files use lexical bindings
```

Listing 11: Measuring Amount of files using lexical binding in Emacs.

The amount of Lisp files using lexical bindings was tested using a shell script (see listing 11). The script counts the amount of Lisp files which enable lexical bindings by setting the local variable `lexical-binding` to `t` in a comments on the first line (see listing 10). Running the script listing 11 on the GNU Emacs 29.3.50 source code [3] we found that all Lisp files included in the source code within the `lisp/` [3] sub-directory has lexical bindings enabled.

When lexical bindings is enabled, unnamed (lambda) and named functions are automatically converted into a `closure` [22,

ch.13.10]. A closure is a function object which contains the lexical environment that existed when the function was created. A alist of pairs and symbols is used to represent the lexical environment. Each symbol element declares that symbol to be *"dynamically bound"* [22, ch.12.10.3] within the lexical scope and each pair declares the lexical binding of a symbol (`car`) to a value (`cdr`). Enabling lexical bindings does not affect variables defined using `defvar`, `defcustom` and `defconst`, which will stay dynamically bound.

Listing 12 highlights the difference between using lexical and dynamic binding. The function `foo` creates a new function `bar` using its argument `x`. When `foo` is called using dynamic bindings (see listing 13) `(foo 10)`, the argument `x` is bound to `10` and pushed to the special bindings stack. The function `bar` is then defined and control exits `foo`. This pops `x` from the special bindings stack such that `bar`'s `x` is undefined. Calling `bar` will thus result in an error. Using dynamic bindings it is possible to define `x` outside the scope of `foo`, which solves the issue of `x` being undefined.

```
(defun foo (x)
  (defun bar ()
    (+ x 10)))
```

Listing 12: Emacs Lisp Scoping Demo Function.

```
(foo 10)
(bar)
;==> Error ... x is void
(setq x 100)
(bar)
;==> 110
```

Listing 13: Output of `foo` from listing 12 to demonstrate Dynamic Scoping.

Calling `foo` with lexical bindings (see listing 14) will create a closure for `bar` which has the value of `x` at the time of creation defined. When `bar` later is called, `x` will have been defined. Setting the value of `x` outside of `bar` is not possible since `bar`'s closure shadows any outside definitions of `x`.

```
(foo 10)
(bar)
;==> 20
(setq x 100)
(bar)
;==> 20
```

Listing 14: Output of `foo` from listing 12 to demonstrate Lexical Scoping.

5.2.4 Non-local Exits and Errors

Lisp errors can created using `define-error` and explicitly signaled using the functions `signal` and `error` [22, ch.11.7.3.1]. In Emacs Lisp, error handlers are defined using the special form function `condition-case` which defines a set error handlers that are active during the execution of a specific lisp expression. Listing 15 creates a new error for division by even numbers, the `division-by-even` error is then signaled using `signal` and then handled using `condition-case`. Information regarding the `condition-case`, `signal` and `error` functions can be found in section D.

```
(define-error
    'division-by-even
    "Division by even number")

(condition-case err
    (let* ((i 5) (m 0.5))
      (dotimes (n 10)
        (if (= (mod n 2) 0)
            (signal
              'division-by-even
              `(/ ,i ,n))
          (setq i (/ i n)))))

  (division-by-even
   "A even division was avoided"))
;==> "A even division was avoided"
```

Listing 15: Handling errors in Emacs Lisp using `signal` and `condition-case`.

```
(catch 'return
  (let ((v 0)
        (l '(1 4 8 5 6 9 12 0)))
  (dotimes (n (length l))
    (when (= v (nth n l))
      (throw 'return n)))
  nil))
```

Listing 16: Finding index of element using `catch` and `throw`.

Non-local exits are done using the functions `catch` and `throw` [22, ch.11.7.1]. listing 16 demonstrates a program that use a non-local exit as a substitute for a return statement. The program returns the index `n` of the element `v` in the list `l` using a non-local exit. If no index is found, `nil` is returned.

`(catch tag body ...):` The `catch` function defines a `tag` which is valid outside of the construct until the `catch` statement exits.

`(throw tag value):` The `throw` function looks through all defined catch-tags for a `catch` using `tag`. If found, control transfers back to the `catch` statement which returns `value`.

The function `unwind-protect` can be used to specify cleanup code which always will run, even after experiencing an `error` signal or `throw`. For example, if a program temporarily creates a buffer to do some work and an error or throw transfers control away from the program, the cleanup code which deletes up the buffer might never execute. Unwind protects allows the programmer to specify such "cleanup" code, which always is to be executed, even after errors or throws.

`(unwind-protect body-form cleanup-forms ...):` The `unwind-protect` function will execute `body-form` followed by always executing the list of Lisp expressions `cleanup-forms`, even when a throw or error signal occurs.

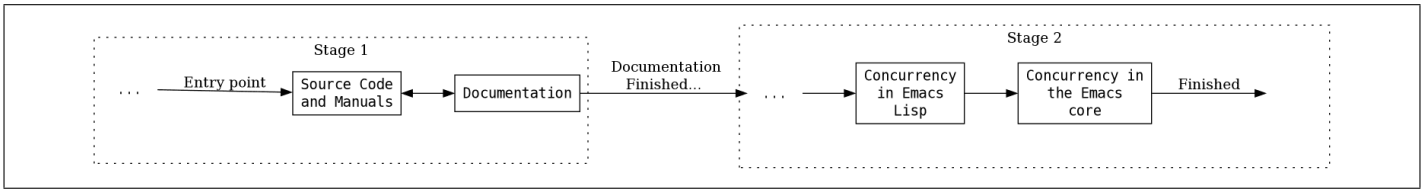

Figure 3: Method workflow for thesis objectives TO1 and TO2.

6 Method

The source code of GNU Emacs version 29.3.5 [3] is used in this thesis, which includes the primary sources regarding the GNU Emacs internals and GNU Emacs Lisp; The GNU Emacs Manual [4], The GNU Emacs FAQ [21] and The GNU Emacs Lisp Reference Manual [22].

The Emacs help system includes features that help locating variable and subroutine definitions in its own C source code. Emacs itself is used as a tool for navigating and studying its own source code. Additional tools and techniques used in the production of this thesis are included in section B. The programs Doxygen and GNU Global are used to generate dependency- and callee-graphs that aid in navigating the source code and understanding the execution flow as well as the connections between subroutines throughout the Emacs core.

The general method used in this thesis is illustrated in figure 3 and is divided into two stages, corresponding to the two thesis objectives (TO1 and TO2).

Stage 1: Documentation (TO1)

Study Source Code Traverse source code from the entry point from the top down. Extend the documentation with important architectural components as they are found. Search the manuals for existing information.

Add Prerequisites Extend the documentation with any prerequisites required to keep the documentation accessible.

The first stage constructs an architectural documentation of the Emacs core. Relevant components / features are identified by studying the source code and the included manuals. The code is traversed from the top down, starting at the entry point to the Emacs command loop and its non-interactive mode and moving through the code according to the control flow and the callee-graphs. The documentation is structured in a similar structure to the MVC (Model View Controller) model and is expanded from the top down, with the components documented as they appear in the source code and the included manuals used to find existing documentation.

When the first stage was finished, a structured architectural documentation of the Emacs core was produced, including topics such as the command loop, Lisp sub-processes and threads. Some of the internal components was left out due to time constraints.

The first stage includes the following selection of topics that are directly or indirectly related to the notion of concurrency.

I Recursive edits, error handling and catch tags.

II Deferring or scheduling Lisp execution using timers and idle timers.

III Running Lisp functions in well-defined locations using hooks.

IV Lisp threads and their interaction with the interpreter.

V Parallelism through external processes.

Stage 2: Concurrency Analysis (TO2).

Concurrency in Emacs Lisp A set of concurrency-related Emacs Lisp packages is studied.

Concurrency in the Emacs core An investigation into the native components used to achieve concurrency including issues with their current implementation along with examples. This is followed by summarizing information regarding the removal of the Global Interpreter Lock, while providing a general view on the thread safety of the Emacs core.

The second stage starts by examining a set of concurrency-related Emacs libraries, examining the problems they solve, how they are used and how they work. Building upon the architectural documentation produced in the first stage, the Emacs core is then examined in respect to its thread-safety and concurrency. Summarizing some of the issues related to the removal of the Global Interpreter Lock paired with solutions where possible.

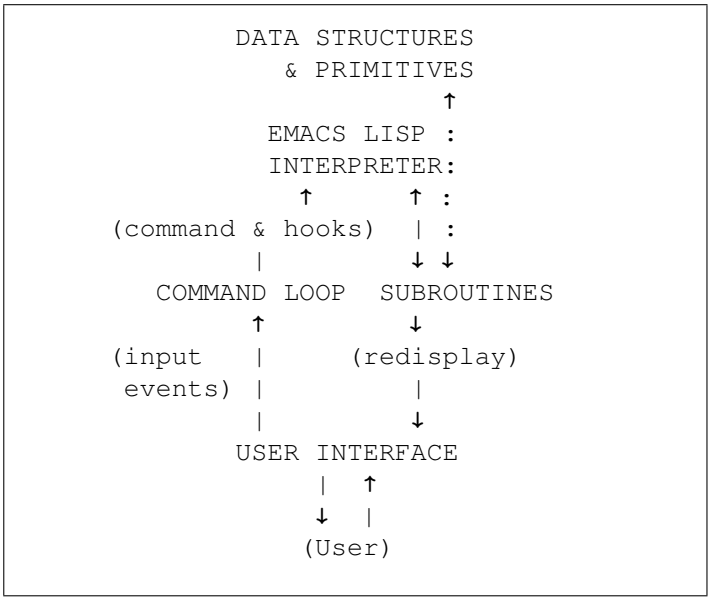

Figure 4: Simplified diagram of Emacs architecture.

7 The GNU Emacs Architecture

This section documents the architecture of the Emacs core, according to the first thesis objective TO1.

GNU Emacs can be described according to the MVC model [30, p.267] where the **model** layer consists of the **Lisp primitives** relating to buffers, text properties, points, markers and overlays, together with the data structures for the Lisp objects, such as the buffer. The **view** layer consists of everything related to displaying and rendering buffers, windows and frames, redisplay and their related Lisp primitives. The **controller** layer consists of everything related to user interaction, commands, reading keyboard input, keymaps, the command loop and the Lisp environment [30, ch.11].

Figure 4 illustrates a simplified model of the interactions between the components of Emacs from the perspective of MVC. They interact with each other in the following way: The user interface (View) reads input events and displays the content of Emacs buffers (Model). The input events are then read into event sequences which the command loop (Control) uses to find bound commands which then are executed in the Lisp environment (Control). The Lisp environment then invokes subroutines and primitives (Model) that directly modify the editor's data structure.

This thesis is structured in a similar way to the MVC model, but without the subroutine category, including subroutines as they appear throughout the text. Its structure follows along the build process, into the starting process, until Emacs enters the command loop. This chapter is divided into four parts; Section 7.1, Section 7.2, Section 7.3 and Section 7.4).

The first part (section 7.1) describes how the source code of GNU Emacs is organized, how to build GNU Emacs from source, what happens on startup and Emacs initialization. The second part (section 7.2) describes the user oriented command-loop, which handles user-input and translates the input into commands that controls the editor. The third section (section 7.3) provides a brief overview of the Emacs UI, covering the Redisplay mechanism and introducing visual Lisp objects such as windows and frames. The final section (section 7.4) is dedicated to the Emacs Lisp environment which is at the heart of the GNU Emacs architecture. This section cover variable bindings (special and lexical bindings), garbage collection and memory allocation, non-local exits and error handling as well as some of the important Lisp objects such as processes (section 7.4.7), threads (section 7.4.12), and timers.

7.1 Starting at the Source

The GNU Emacs build process starts by executing the `autogen.sh` script, generating the configuration script (`configure`). The configuration script is responsible for setting up a build environment according to the arguments given by the user. The configuration script configures the `src/config.h` [3] file together with the makefiles in directories such as `src/`, `lib/`, `lisp/`, `lib-src/` and `leim/` [3]. Emacs is then built using the `make` program, which proceeds to compile `src/emacs.c` into the `temacs` binary,

```
$ temacs -l loadup
```

Listing 17: Running the `temacs` binary directly.

The `temacs` binary is a bare Emacs Lisp interpreter which is temporarily produced in the build process and created exclusively from the C source code of the Emacs core, without any externally loaded Lisp files. The `temacs` binary unable to preform any useful task on its own and contains only some basic input/output routines, giving it the ability to load external Lisp files, excluding any editing commands and the majority of the normal Lisp functions and libraries [4, ch.E.1]. To transform `temacs` into the GNU Emacs text-editor, the file `lisp/loadup.el` is loaded. This initializes the Lisp environment, adding the components necessary to make Emacs useful as a text-editing environment [4, ch.E.1].

At the end of the compilation process, after the `lisp/loadup.el` file has been loaded into `temacs`, `temacs` will be preloaded with Lisp. The `temacs` binary then creates a dump file which contains the preloaded and initialized environment. This dump file is then loaded by the Emacs on startup. Alternatively `temacs` can dump its internal state into a binary executable, disregarding the need for any dump file [4, ch.E.1]. The `temacs` executable is much slower then the final `emacs` binary since it is not preloaded and initialized with any *compiled* Lisp code. It is possible to run the bare `temacs` binary if the correct files required to make Emacs into a usable are loaded. The file `temacs` use to build the dump file `lisp/loadup.el` can be manually loaded by the binary using the `-load` or `-l` argument (see listing 17).

7.1.1 Starting Emacs

Since Emacs ships with native Lisp threads, Emacs initializes a main thread during startup, defined in C as `main_thread` in `src/thread.c` [3]. The global interpreter lock is locked by the main thread and its thread id is set using the C function `pthread_self` through the intermediate C function `sys_thread_self` from `src/systhread.c` [3].

```
// [...]
struct thread_state *current_thread     =
    &main_thread.s;
static struct thread_state *all_threads =
    &main_thread.s;
static sys_mutex_t global_lock;
// [...]
```

Listing 18: Data-structures used for thread management `src/thread.c` [3].

The `main` function is located in `src/emacs.c` [3] is the entry point to the `temacs` binary. The function initializes and loads the various components of Emacs and then enters the **editing command loop**. The `main` function will perform the following tasks in order upon starting GNU Emacs.

I Perform `temacs`-related tasks.

II Perform basic initialization.

III Parse command line arguments. After Emacs has been preloaded with Lisp, the Lisp function `command-line` is used to handle command line arguments. The function is called from the `normal-top-level` function on startup (located in `lisp/startup.el`).

IV Load and initialize a basic Lisp environment.

V Initialize the contents of the C variable `Vtop_level` (accessible in Lisp as `top-level`). The `top-level` variable contains Lisp code which is evaluated whenever Emacs starts.

Its value is set in `main` (defined in `src/emacs.c` [3]) and defaults to `(load "loadup.el")`, and is set to `(normal-top-level)` when `temacs` loads `lisp/startup.el` [3].

VI Enter the top-level command loop through a call to the C function `Frecursive_edit` (defined in `src/keyboard.c` [3]) which never returns when Emacs is started interactively.

7.1.2 Emacs Initialization

To configure the bare Lisp environment and interpreter included in `temacs` into a functional editor, Emacs will load a set of initialization files. This initialization sequence is the sequence where the Lisp engine (implemented in the `src/` [3] sub-directory) loads all of the Lisp components that constitute the user-facing Emacs editor, followed by entering the editor command loop. The Lisp components are located in the `lisp/` sub-directory [3].

I. `lisp/loadup.el` [3]: Loaded by `temacs` during the build process. The `lisp/loadup.el` [3] file loads the Lisp libraries which "sets up the normal Emacs editing environment" [22, ch.E.1] and configures Emacs into an editor. After `temacs` has loaded the file a dump file is created from the internal state of `temacs`, which is loaded by Emacs during startup [3].

II. `lisp/startup.el` [3]: Is one of the files loaded by `lisp/loadup.el` [3] and defines what Emacs does when it first starts. The file preforms startup related configurations and defining functions such as `normal-top-level`, which is executed when Emacs starts.

III. `site-start.el`: Is located in the `site-lisp` directory which stores locally added Lisp libraries and files [4] and will load all files contained in the `site-start.d` sub-directory. The file is loaded on startup unless the `-no-site-lisp` / `-nsl` arguments are given [22, ch.16.3]

IV. `~/.emacs.d/early-init.el`: Is the early initialization file which is loaded before the configuration file (`init.el` / `.emacs`) during the startup process [22, ch.42.1.2].

V. `init.el`: Is the primary user configuration file and is loaded from `~/.emacs.d/init.el` if it exists, otherwise from `~/.emacs` or `~/.emacs.el`.

[3] Alternatively the internal state of `temacs` is dumped into a new binary instead of a dump file [22, ch.E.1].

[4] We found the site-lisp directory in `/usr/share/emacs/site-lisp`.

VI. `default.el`: Is intended for local user customization and is only loaded if found in the `load-path` and is loaded after the user configuration file.

Files (3-6) will load in order after Emacs starts, with `loadup.el` and `startup.el` already loaded by `temacs` during the build process. The files meant for user customization are: `early-init.el`, `init.el` / `.emacs` and `default.el`.

7.2 The Command Loop

When Emacs is started interactively it will enter the editor command loop (defined in `src/keyboard.c`) through a recursive edit. The editor command loop handles user interaction by executing commands in response to user input events. The C function `command_loop` (from `src/keyboard.c` [3]) defines the entry point to the command loop.

The `top-level` variable is then called each time Emacs starts or returns to top-level within the C function `command_loop`. When Emacs starts non-interactively it will quit after returning to top level.

The C variable `command_loop_level` (from `src/keyboard.c` [3]) contains the current recursive edit depth and is used by functions such as `command_loop` to identify if the function is called recursively, where a value of 0 indicates no recursion / top-level.

```
Lisp_Object command_loop (void)
{
  // [...]
  if (command_loop_level > 0 || minibuf_level
      > 0) {
    // In a recursive edit.
    // Return upon catching the exit tag.
    Lisp_Object val;
    val = internal_catch (Qexit,
        command_loop_2, Qerror);
    executing_kbd_macro = Qnil;
    return val;
  }
  else
    while (1) {
      // Setup top-level.
      internal_catch (Qtop_level,
          top_level_1, Qnil);
      // Enter the command loop.
      internal_catch (Qtop_level,
          command_loop_2, Qerror);
      executing_kbd_macro = Qnil;
      // End of file in -batch run causes
          exit here.
      if (noninteractive)
        Fkill_emacs (Qt, Qnil);
      // Otherwise repeat forever
    }
}
```

Listing 19: Parts of the function `command_loop` defined in `src/keyboard.c`.

Upon start, in C, at the end of `main` (from `src/emacs.c` [3]) Emacs enters a long call chain, starting a initial call to `Frecursive_edit` (from `src/keyboard.c` [3], in Lisp called `recursive-edit`). The function then calls `recursive_edit_1` which sets up the default in- and output streams. The function then calls `command_loop` (from `src/keyboard.c` [3], see listing 19). Since the function was called from top-level (not recursively) it will proceed to enter a loop which sets up the top-level catch tags, which Emacs will jump to in case an error occurs evaluating the `top-level` form. This is followed by entering the C function `command_loop_2` which sets up error handlers for command-related errors.

Listing 19 displays these parts of `command_loop`. If Emacs is running in a non-interactive session (that is, in batch mode), then Emacs will quit after returning from `command_loop_2`, otherwise Emacs will execute the Lisp expression contained in `top-level`, followed by reentering `command_loop_2`.

```
do
  val = internal_condition_case (
      command_loop_1,
      handlers,
      cmd_error
  );
 while (!NILP (val));
```

Listing 20: The error handling loop from `command_loop_2` from `src/keyboard.c` [3].

The function `command_loop_2` sets up handlers for errors that may occur during `command_loop_1` (see listing 20) and will only return if invoked by the Lisp function `execute-kbd-macro` and a macro reaches the end of a file or when the macro event stack is empty.

A keyboard macro is composed of a sequence / stack of events and can be executed from Lisp using the function `execute-kbd-macro` (defined in `src/macro.c` [3]). The function iterates through the event stack defining the macro, calling `command_loop_2` each macro iteration. Each events in the sequence is read through the C function `read_char`. When the end of a keyboard macro is reached, the C function `read_key_sequence` returns `0` back to `command_loop_1`, itself returning `nil`, exiting to `command_loop_2`, which also returns `nil` back to `execute-kbd-macro`.

The **editor command loop** is responsible for the command-driven user-interactions within Emacs and is defined internally in the C function `command_loop_1` (see `src/keyboard.c` [3]). The purpose of the command loop is to perform *"command key interpretation"* [22, ch.10.1] allowing the user to execute Lisp commands by translating keyboard input into commands. A **command** is defined as a interactively callable function, function-like object or keyboard macro [22, ch.21.3]

Before entering the command loop in `command_loop_1`, the function will first execute any hooks (in `post_command_hook`) remaining from a previous command, which is done after a command throws to top-level after experiencing an error [5]. This is followed by resizing the echo area if any messages are being displayed, which will redisplay during the call to the C function `resize_echo_area_exactly`. Then if there are any delayed warnings, the Lisp hook `delayed-warning-hook` is executed before entering the loop. The loop within the `command_loop_1` function first checks if the minibuffer is active and if the echo area is displaying a message. If so, then the `sit_for` function is used to delay a redisplay, followed by calling the Lisp hooks `echo-area-clear-hook`.

Then a *complete key sequence* is read using the C function `read_key_sequence`. Once a key sequence has been read, the variable `cmd` is set to the value of either `read_key_sequence_cmd` or `read_key_sequence_cmd_remapped` if any keymaps have remapped the command, both which are global C variables, set to the keymaps command in `read_key_sequence`.

[5] According to comments in the function `command_loop_1` defined in `src/keyboard.c` [3].

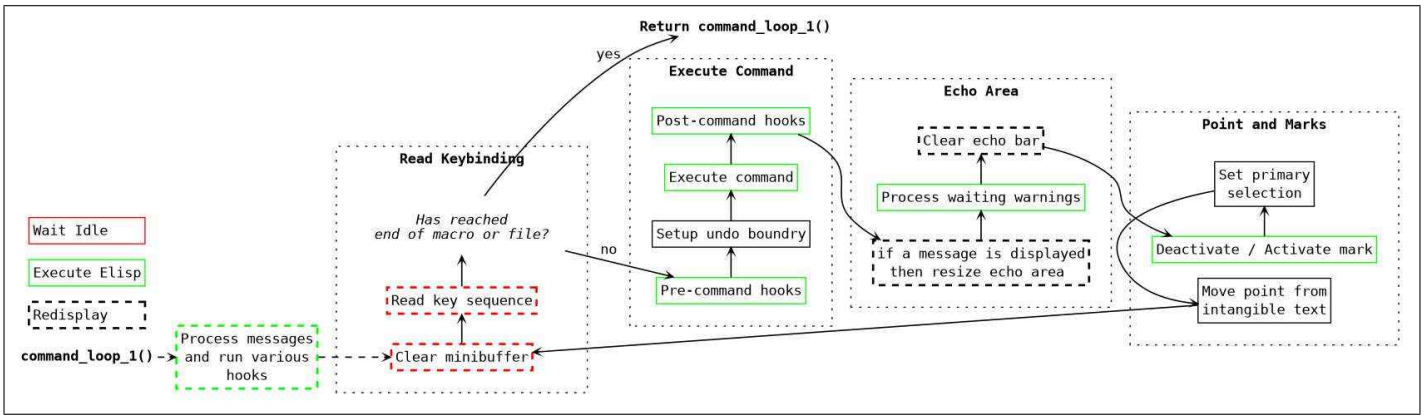

Figure 5: Diagram displaying the editing command loop `command_loop_1` from `src/keyboard.c` [3].

During the wait for user input while reading the key sequence, calls to redisplay might occur. If the command loop has reached the end of a keyboard macro, or the macro has reached the end of a file indicated by `read_key_sequence` returning 0, upon which the function will exit the loop and return. During macro execution, events are read from a "recording" instead of from user input.

Command execution is then prepared by updating various variables, storing `cmd` in the Lisp variable `this-command` and executing the Lisp hook `pre-command-hook`. The undo boundary is setup before executing the `this-command` through the Lisp function `command-execute` by calling the C function `call1`. After the command has finished executing, various hooks such as the Lisp hook `post-command-hook` are executed. If a message is displaying following this, the echo area is resized through the C function `resize_echo_area_exactly` which calls redisplay. If there are delayed warnings, the Lisp hook `delayed-warnings-hook` is executed. The echo bar is then reset and cleared using the C function `echo_now` which in turn calls redisplay.

If the command sets the Lisp variable `deactivate-mark` to a non-`nil` value, then the `deactivate-mark` Lisp function is called. Otherwise, if the command is not in the Lisp list `selection-inhibit-update-commands` and the Lisp variables `tty-select-active-regions` or `select-active-regions` allow for it, then the selection is extracted using the Lisp function `region-extract-function` and the primary selection to the string through `gui-set-selection` and the Lisp hook `post-select-region-hook` is executed with the selected string as argument.

If the command switches back to a buffer with an active mark, the region is reactivated and the Lisp hook `activate-mark-hook` executed. If the point moved into a region of text with intangible properties (`composition`, `display` and `invisible`) then Emacs moves the point to the boundary of that region of text to be able to display the cursor, unless the Lisp variables `global-disable-point-adjustment` or `disable-point-adjustment` is set to a non-`nil` value by the command. Finally, if a keyboard macro is being recorded, then declare the characters which didn't provoke an error as part of the macro. Figure 5 illustrates `command_loop_1`, where red squares denotes all the times Emacs waits idle, and green squares all places where Lisp code is executed and dashes denotes the places where redisplay can occur.

7.3 The User Interface

The user interface of Emacs is the user-accessible part of Emacs. Emacs has special objects that has defined graphical representations, where "Frames" makes up the outermost UI-component. One Emacs session can create multiple frames, each frame containing one or more windows, a minibuffer or the echo area. The minibuffer and echo area are located in the same place, so only one of them can be displayed at once. Frames exist both in the graphical user interface (GUI) and terminal user interface (TUI) of Emacs, while tool bars (and the scroll bars in windows) only exist in GUI frames. Some components of the interface such as the menu bar and tool tips are rendered differently between the interfaces. A frame contains at least one "window", each window able to split into more windows each containing their own buffer [22, ch.29.1]. Windows are tiled such that their combined tiled dimensions always fit within their frame [22, ch.29.1].

A "buffer" is a Lisp object that Emacs use to store and manipulate text. Multiple buffers may exist at the same time, with one buffer designated as being actively used (the "current buffer") [22, ch.28.1]. Buffers are displayed through "windows", where multiple windows may display the same or different buffers. When a file is opened in Emacs, its contents copied into and represented by a buffer, however buffers does not have to be related to a file [22, ch.28].

Some use cases for buffers are representing the input and output streams of subprocesses [22, ch.40.1], temporarily displaying or manipulating text and rendering graphical elements such as UI by using special "propertized" text. If a buffer name starts with a space character the buffer is intended for internal (non-user) usage, where it will be hidden from certain functions, such as when the user lists all buffer. Text properties can be used to dictate how Emacs renders text, such as defining text color, size or font. Special properties can represent graphical objects such as GTK widgets and images [4, ch.41.17] which are rendered along with the text properties.

The "echo area" is a dedicated space on the bottom of frames used for echoing the keystrokes of multi-character commands as they are typed, displaying logs, error messages, or messages sent using the `message` primitive which are logged to a special buffer called `*Messages*`. It is also possible to use the echo area as an output stream by specifying the output stream as `t`. The space occupied by the echo area is also used to contain the "minibuffer", a special buffer used when reading complicated user input such as strings of text [22, ch.41.4] [4, ch.1.2].

7.3.1 Redisplay

The process of redrawing the rendered user interface, a process called redisplay is performed. Redisplay redraws parts of the frames and window according to changes in the internal state [22, ch.41.1]. This usually done automatically in the editor command loop but can also be induced using Lisp primitives such as `sit-for` or `redisplay`. The `redisplay` primitive can manually request (or forcibly invoke) a redisplay where Emacs will determine which parts of its frames to redraw and redisplay them [22, ch.41.2].

The action of updating the display is called `redisplay` (Lisp function defined in `src/xdisp.c` [3]). Running on the Linux X window system, some portions of redisplay might be asynchronously called such as from C functions calling `maybe_quit` or `process_pending_signals`. If X events about mouse movement or exposed frames are reported they might then enter redisplay through `expose_frame` or `note_mouse_highlight` from which Lisp evaluation can occur during the redisplay if the mode-line contains any `:eval` form and `inhibit-eval-during-redisplay` is `nil`. This requires the code to temporary block or unblock input when the global Lisp state shouldn't change. At a high level of abstraction, the redisplay algorithm consists of the following steps [6]:

I. The function `redisplay_internal` (see figure 6) decides which frames has windows which might need to be redrawn.

II. For each window that has a display that might need to be updated, a "glyph matrix" is computed which describes how the windows should look on the display, including the mode lines, header lines and tab lines (done internally through `redisplay_window` called by `redisplay_internal` (see figure 6).

III. The previously rendered glyph matrix is compared with the new one to obtain where and what in the display needs to be updated `update_window` (see figure 6).

[6] Acording to the comments in `src/xdisp.c` [3].

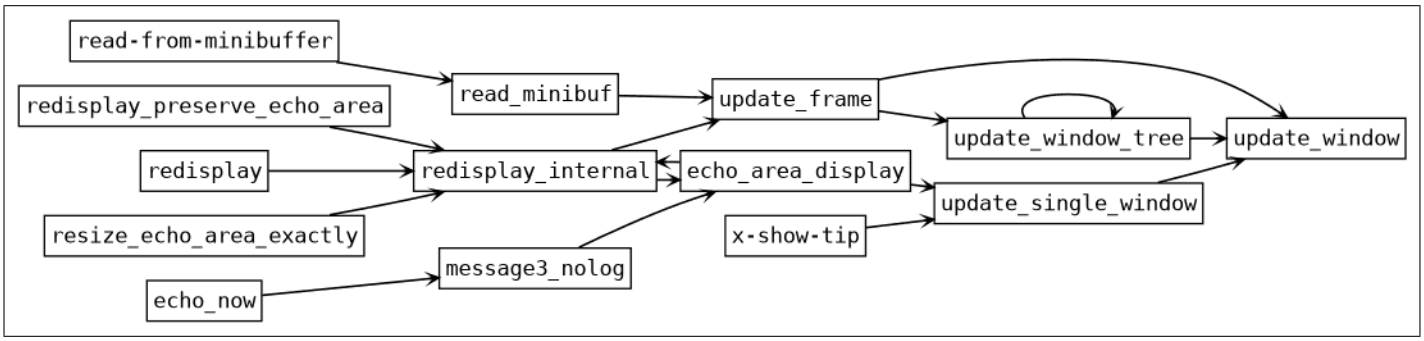

Figure 6: Call-chain around `redisplay_internal` (from `src/xdisp.c` [3]).

Emacs has the Lisp function `redisplay` internally represented as the function `redisplay` (from `src/xdisp.c` [3]) which is used to synchronize the internal state of the editor with the displayed GUI. `redisplay` is a wrapper function to `redisplay_internal` (from `src/xdisp.c` [3]) which actually preform the redisplaying.

Redisplay happens automatically within the command loop `command_loop_1`, where there are five primary ways a redisplay (a call to `redisplay_internal`) can occur. The first four items in the following list (1, 2, 3, 4) represents internal C functions which are called from `command_loop_1`, which are able to produce a call chain ending in `redisplay_internal`. The last item (5) is a collection of Lisp functions which produce call-chains leading either directly or indirectly to a redisplay.

I. `sit_for`; ... [7], `redisplay_preserve_echo_area`, `redisplay_internal`

II. `read_key_sequence`; `read_char`, ..., `redisplay_internal`

III. `echo_now`; `message3_nolog`, `echo_area_display`, `redisplay_internal`

IV. `resize_echo_area_exactly`; `display_internal`

V. Or through one of the following Lisp functions: `recenter`, `xwidget-resize`, `execute-kbd-macro`, `recursive-edit`, `read-from-minibuffer`, `read-key-sequence`, `read-key-sequence-vector`, `make-terminal-frame`, `resume-tty`.

The functions which are defined in `src/keyboard.c` [3] include `sit_for`, `read_key_sequence`, `read_char`, `echo_now` and those defined in `src/xdisp.c` [3] consists of `redisplay_preserve_echo_area`, `message3_nolog`, `echo_area_display`, `resize_echo_area_exactly`, `redisplay_internal`.

[7] Here we use three dots "..." to denote multiple or long call chains.

7.4 The Emacs Lisp Environment

Emacs includes an internal Read-Eval-Loop, defined as the C function `readevalloop` (in `src/lread.c` [3]) which is used to evaluate Emacs Lisp code and files. The Read-Eval-Loop is indirectly invoked from Lisp when the following Lisp functions are called `eval-region`, `eval-buffer` or `load`. This is the primary function which runs when `temacs` loads `lisp/loadup.el`.

The `readevalloop` function works by iterating a buffer from start to end, iterating over each character. When a lisp object has been parsed, its string is **read** into a Lisp Object using the function `read0` (defined in `src/lread.c` [3]). This new Lisp object is then **eval**uated using the C function `eval_sub` (from `src/eval.c` [3]) or if its a macro, it is expanded. Lastly the lisp object is printed (given `printFlag` is true) and the loop repeated until the end of the buffer is reached.

GNU Emacs has three different stacks (see comment on line 3537 in `src/lisp.h` [3]): The C stack, the special bindings stack (`specpdl`) and the handler stack `handlerlist`. The handler stack is implemented as a manually managed, doubly-linked list and keeps track of active `catch` tags and `condition-case` handlers. The special bindings stack is used for many purposes such as debugging and for keeping track of changes to dynamically bound variables.

7.4.1 The Special Bindings Stack

The special bindings stack is sometimes is referred to as the `unwind-protect` stack and is used for dynamic variable scoping, the tracking of `unwind-protect`'s, and local-variable bindings allowing for the temporary binding of variables within a scope, ensuring that the values can be restored after exiting the scope. Its used in `condition-case` to ensure that values which might have changed inside of the body are restored after the error.

```
union specbinding
{
  // [...]
  struct {
    ENUM_BF (specbind_tag) kind : CHAR_BIT;

    Lisp_Object symbol;
    Lisp_Object old_value;
    Lisp_Object where;
  } let;
  // [...]
}
```

Listing 21: specbinding for `let` (modified for readability).

One of the purposes the special bindings stack is used for is to keep track of the changes in `let`-defined values, which is internally stored in the `specbinding` structure which can be seen in listing 21.

The `specpdl` array is the special bindings stack, with `specpdl_ptr` pointing to the current position on the stack. Pushing a variable which is dynamically bound to the `specpdl` stack saves the current value and sets the new value. This is acheived through pushing a `specbinidng` onto the stack using the internal function `specbind`. When exiting a scope which has defined some variables, the stack is popped, restoring the old values. This is done through functions such as `unbind_to`.

The following variables and functions are used to operate on the special bindings stack.

- `specpdl` (variable, array): The special bindings array
- `specpdl_size` (variable): Total number of `specbinding` slots in the `specpdl` array.
- `max_specpdl_size` (variable): Points the current position in the stack (first unused `specbinding` slot in the array), with the end given by `specpdl_ptr`.
- `unbind_to` (function): Used to execute and pop values from the special bindings stack.
- `specpdl_unrewind` (function): Unwind or rewind the last elements of the `specpdl` stack. Change the value of bindings on the special bindings stack to their old value. Used when switching threads.
- `specbind` (function): Bind a symbol to a value as a binding on the `specpdl` stack.
- `record_unwind_protect` (function): Used for unwind protects and available in Lisp through `unwind-protect`. Unwind-protects ensures that certain specified (usually cleanup) code always are executed after some Lisp expression, even if it exits non-locally through an error or signal. This is done through storing some of the execution context on the special bindings stack.

7.4.2 Keyboard Input

In Emacs, key sequences are composed of sequences of single input events (keys), such as from the keyboard, touch screen, mouse or a menu-bar [22, ch.23.2]. Key sequences are bound to commands using keymaps with the function `read_key_sequence` (from `src/keyboard.c` [3]) reading a sequence of keys, looking for a bound command in the active set of keymaps.

A keymap is a Lisp data-structure which contains a mapping of single events to a keymap or command. Keymaps that map events to keymaps are called *prefix keys* [22, ch.23.2]. A keymap has the form `(keymap <keymap> <keymap> ...)`, where the `car` is the `keymap` symbol and the rest is a list of keymaps.

```
;; C-x C-c -- Exit Emacs
'(keymap
   ;; ^X = 24
   ;; ^C = 3
  (?^X keymap
    (?^C . save-buffers-kill-terminal)))
```

Listing 22: Keybinding for exiting Emacs, `C-x C-c`.

The keymap in listing 22 defines the default keybinding for exiting Emacs used when typing `C-x C-c` (control and 'x' then control and 'c'). We define a keybinding containing one keymap which maps control-x ASCII "CANCEL" character X̂ (with the ascii code 24) to a keymap mapping the control-c ASCII character Ĉ (with ascii code 3) to the Lisp function `save-buffers-kill-terminal` which prompts the user to save all unsaved buffers and then exits Emacs.

Emacs uses the keybinding `C-g` bound to the Lisp function `keyboard-quit` to signal a `quit` condition which causes control to return to the most recent handler for `quit` errors. When the Lisp function / command `keyboard-quit` (from `lisp/simple.el` [3]) is invoked, it signals the `quit` signal (see `keyboard-quit` in `lisp/simple.el` [3]) which causes control to return the most recent handler. Since the function is a command, it will only handle the `C-g` keybinding when Emacs is able to read user input and execute a command. If there is Lisp that is evaluating and Emacs cant read user input the C function `maybe_quit` (from `lisp/lisp.h` [3]) is used.

> "Check quit-flag and quit if it is non-nil. Typing C-g [...] sets Vquit_flag [but] does not directly cause a quit [...] [so] the program needs to call maybe_quit at times when it is safe to quit [...] [which is especially important in] loop[s] that might run for a long time [...] [when the] [quit-flag] is set to kill-emacs the SIGINT handler has received a request to exit Emacs when it is safe to do. When not quitting, [Emacs will] process any pending signals"

Any non-nil (symbol) value of throw-on-input determines a catch tag which Emacs will throw to on keyboard input (see the DocString for throw-on-input in src/keyboard.c [3]). The value of throw-on-input is checked at the end of kbd_buffer_store_buffered_event from src/keyboard.c [3] which is invoked at an interrupt level to store events. At the end it checks if throw-on-input is non-nil. If so, set the quit-flag from src/eval.c [3] to the catch tag stored in throw-on-input (defined in src/keyboard.c) [3]. The next time maybe_quit is called, it will throw to the value of quit_flag.

When Emacs is unable to read user input, and C-g is inputted, it is delivered as a SIGINT error signal, handled by handle_interrupt_signal from src/keyboard.c [3]. This interrupt signal handler is invoked when (1) outside signals are received or (2) the keyboard-quit Lisp function is called by the C-g keybinding. When C-g is invoked, the top frame is stored in internal_last_event_frame from src/keyboard.c [3] and handle_interrupt (from src/keyboard.c [3]) is called. If the signal is not generated by C-g, it will cause Emacs to quit.

The function with-local-quit lisp/subr.el [3] binds inhibit-quit to nil inside of a handler / condition-case that catch the quit error signal. This ensures that any call to maybe_quit or incoming quit signal to be caught by the handler so as to return control to the with-local-quit.

while-no-input lisp/subr.el [3] executes a Lisp expression until user input is detected. This is done through creating a catch statement in which a Lisp expression is executed with inhibit-quit set to nil and throw-on-input set to the new catch tag. When input is detected, the quit-flag is set to the value of throw-on-input. When the C function maybe_quit then is invoked, it will throw to the value of the quit-flag.

```
(defun probably-quit nil
  ;; if (!NILP (Vquit_flag) && NILP
      (Vinhibit_quit))
  (if (and quit-flag (null inhibit-quit))
      ;; process_quit_flag ();
      (process_quit_flag)
    ;; else if (pending_signals)
    ;;   process_pending_signals ();
    (if ...
        (process_pending_signals))))
```

Listing 23: A Lisp expression which represents the body of the C Function probably_quit from src/eval.c [3].

Listing 23 implements a function similar to the C function probably_quit from src/eval.c [3] in Lisp as to better visualize how quit-flag and throw-on-input is treated in terms of Lisp.

```
(defun process-quit-flag nil
  ;; Lisp_Object flag = Vquit_flag
  (let ((flag quit-flag))
    ;; Vquit_flag = Qnil
    (setq quit-flag nil)
    ;; if (EQ ( flag, Qkill_emacs ))
    (if (eq flag kill-emacs)
        ;; Fkill_emacs ( Qnil, Qnil )
        (kill-emacs nil nil))
    ;; if (EQ ( Vthrow_on_input, flag ))
    (if (eq throw-on-input flag)
        ;; Fthrow( Vthrow-on-input, Qt )
        (throw throw-on-input t))
    ;; quit()
    (signal 'quit nil)
    ))
```

Listing 24: A Lisp expression equivalent to the body of the C function `probably_quit_flag` from `src/eval.c` [3].

Listing 24 contains a Lisp implementation of the function `probably_quit_flag` which has been reproduced as close as possible to the body of the C function. This function is relevant since it is called by `probably_quit`, which was called from `maybe_quit`. Everything except the C variable declarations [8] have Lisp expressions that can express equivalent or similar expressions. The new Lisp function `process-quit-flag` defined in listing 24 is invoked from the new Lisp function `probably-quit` from listing 23. The comments above each Lisp expression contains the equivalent C expressions.

We use the term equivalence here as `signal` (Lisp) is a wrapper around `signal_or_quit` (C), `eq` (Lisp) is a wrapper around `EQ`, `(if cond then [else])` (Lisp) evaluates the `cond` expression, which for `throw-on-input` or `quit-flag` is the value of the symbols and then the if-body if non-`nil`. The functions `Fkill_emacs` (defined in `src/emacs.c` [3]) and `Fthrow` (defined in`src/eval.c` [3]) are references to the C functions that implements the Lisp functions `kill-emacs` and `throw` respectively.

[8] Unlike C, Emacs Lisp stores modifications to variables in the special bindings stack and / or the lexical interpreter environment.

7.4.3 Handlers and Non-Local Exits

Handlers for `catch` tags and `error` signals are both defined in C using the C `struct handler` (see listing 25 from `src/lisp.h` [3]). Handlers are kept in a doubly linked list, where each handler contains information regarding changes to the interpreter state, which can be used to restore the execution context when a non-local jump occurs (comment in `src/lisp.h` [3]).

```
struct handler
{
  enum handlertype type;
  Lisp_Object tag_or_ch;
  enum nonlocal_exit nonlocal_exit;
  Lisp_Object val;
  struct handler *next;
  struct handler *nextfree;
//[...]
  sys_jmp_buf jmp;
  EMACS_INT f_lisp_eval_depth;
  specpdl_ref pdlcount;
//[...]
};
```

Listing 25: `struct handler` with some members and comments removed.

Description	type	val	tag_or_ch
Used for catch	CATCHER	longjmp return value	The catch tag
Wildcard that catches all throws	CATCHER_ALL	longjmp return value	Unused
Used for condition-case	CONDITION_CASE	longjmp return value	List of conditions
Used for handler-bind	HANDLER_BIND	Handler function	List of conditions
Hides N condition handlers, used for handler-bind	SKIP_CONDITIONS	Unused	N

Table 1: Values for val and tag_or_ch given struct handler's type.

7.4.4 struct handler

Members of struct handler include (with some removed):

- type: The type of the handler, defined by the enum handlertype (from src/lisp.h [3]), see table 1.
- val and tag_or_ch: The stored values of these members differ depending on the type of handler.
- nonlocal_exit: Type of non-local exit defined by the enum nonlocal_exit (from src/lisp.h [3])
- next: A pointer to the next outer catchtag (comment in src/lisp.h [3])
- nextfree: Pointer to the previous inner element (Opposite direction to next)
- jmp: Execution context which is to be reset. This is done using the functions setjmp and longjmp.
- f_lisp_eval_depth: Previous evaluation depth
- pdlcount: How much to unwind the specpdl stack.

7.4.5 catch and throw

A catch implies a non-local exit from the executing code, where control "jumps" from executing in one location to another. When a (throw <tag> <value>) is executed, the interpreter looks for the a handler whose value of tag_or_ch is equal to <tag>. When a handler is found, its val listing 25 member is set to <value>, the specpdl stack is unwound, and the value of val us returned from the (catch <tag> ...) form. The nonlocal_exit member contains the type of the handler, either a signal or throw.

Non-local jumps are done using the function setjmp(env) (included from setjmp.h) which stores the execution context in env at the location where it is invoked, later in the code when an error occurs longjmp(env) will jump to the location where setjmp(env) was executed and restore the execution context.

Internally the function internal_catch (from src/eval.c [3]) is used to define a catch statement. It works by calling sys_setjmp in a if statement with the handlers jmp member as argument. This stores the current execution context in the handler, where sys_setjmp returns a non-zero value if a non-local jump was performed. The function is called in a if-statement (if (! sys_setjmp (...->jmp))) which executes the true-block during the initial call. When a non-local exit is performed, control will jump back to that if statement and the return value will be non-zero (= false) and the false-block will execute which removes the handler from the handler-list and returns the value val.

7.4.6 Memory Allocation

Each block is a structure containing an array of a predefined size containing (one type of) lisp objects. An array of bits, where each bits corresponds to an element in the Lisp object array. If the bit is set (= 1) the corresponding Lisp object is marked and wont be garbage collected.

A referenced or in-use Lisp object is called **marked** and a non-referenced Lisp object is called **unmarked**. Unmarked objects will be freed when the garbage collector runs with their memory locations being reused to store new Lisp objects [22, ch.E.3]. The garbage collector reclaims (frees up) marked cells, storing them in a "free list" which is the first place Emacs tries to allocate new Lisp objects from.

Lisp objects are allocated using these blocks. Each Lisp object has its own linked list of blocks, each block being an array of the Lisp object. A free-list stores pointers to garbage-collected cells so they can be reused in future allocations. Together the blocks are stored in a red-black-tree (see `struct mem_node` in `src/alloc.c` [3]).

The memory location of a new Lisp object is determined by the following process: When the free-list is non-empty, one of its elements is popped and its address used. If the free-list is empty, then the first unused slot in the last block is used, if the Lisp object block is full a new block will be created and its first location used.

Emacs stores preloaded Lisp objects in **pure storage**. The pure space is allocated at compile-time and only used by `temacs`. These objects never change during the normal runtime of Emacs [22, E.2], and are stored in the `DATA` memory section rather then on the heap (See Appendix section A).

An array called `spare_memory` (defined in `src/alloc.c` [3]) contains pointers to *spare blocks*. Spare blocks are blocks that can be freed if Emacs were to run out of memory. These blocks include one large int block, two string blocks and four cons blocks.

When the allocator is used and `malloc` fails, the resulting pointer from calling `malloc` doesn't fit in a `EMACS_INT` (which is 8 bytes) or if an overflow prevented `malloc` from being called, the function `memory_full` in `src/alloc.c` [3] is used. If the amount of memory allocated exceeds the spare memory size, then `malloc` is attempted until successful, otherwise the `spare_memory` is freed.

```
struct cons_block
{
  /* Place `conses' at the beginning, to ease
     up CONS_INDEX's job.  */
  struct Lisp_Cons conses[CONS_BLOCK_SIZE];
  bits_word gcmarkbits[1 + CONS_BLOCK_SIZE /
      BITS_PER_BITS_WORD];
  struct cons_block *next;
};
```

Listing 26: Structure for the cons allocation block from `src/alloc.c` [3].

As an example, the structure `cons_block` (seen in listing 26 from `src/alloc.c` [3]) stores the `cons` lisp object. Cons cells are the foundation of Lisp languages as they make up the smallest abstractions of lists. All `cons` blocks are kept in the global variable `cons_block` and its "free list" in `cons_free_list`.

7.4.7 Lisp Processes

Emacs has the ability to create asynchronous and synchronous child processes [22, ch.40]. When using synchronous child processes the Emacs parent process will wait for child processes to terminate before it execution continues.

Asynchronous child processes run in parallel to the parent processes (see figure 7) and are represented as the `process` lisp object. The parent process is able to communicate with and control asynchronous child processes through reading in- or output, sending signals to and reading the child process status. Emacs supports connecting to child processes using TCP, UTP, Serial port connections, Pipeing and transaction queues.

Input to child processes can be sent as a string to the child process standard input. When the parent process receives the standard output or standard error stream of a child process, a *filter* function is invoked. A process is usually associated with a buffer determined upon the creation of the child process [22, ch.40.9.1] where killing the process buffer kills the process. The filter function also use the buffer to manage the input and output streams. A

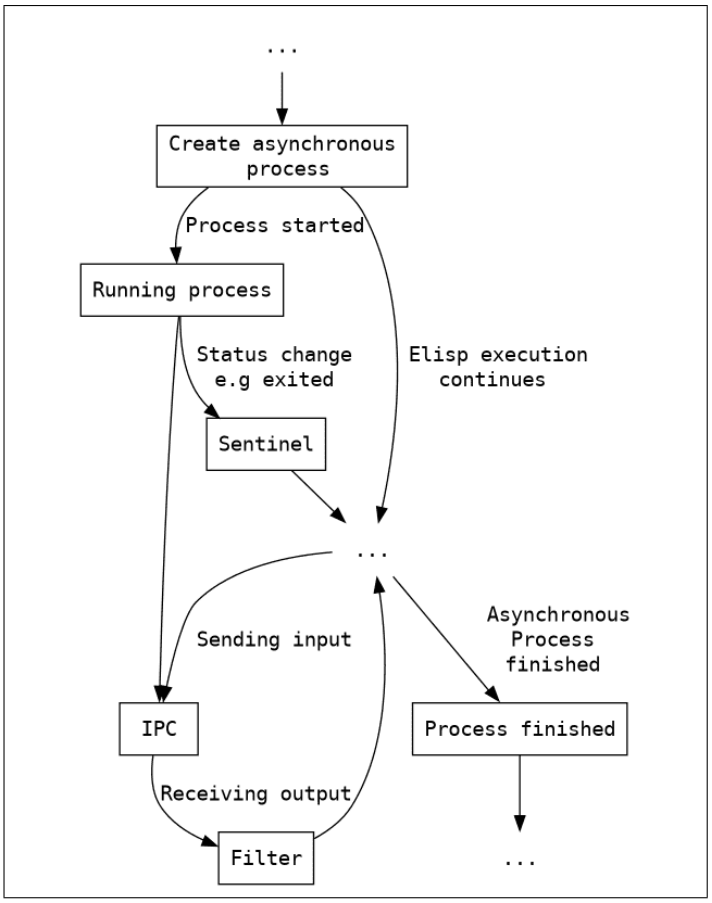

Figure 7: Execution of a asynchronous child process.

function called a *sentinel* is used to handle changes in the status of the child process which is communicated by the child process such as if the child process has stopped exited or various errors that may have occurred [22, ch.40.10].

To create asynchronous subprocesses, Emacs use the internal function `emacs_spawn` (see listing 66), from `src/callproc.c` [3]). The `newpid` argument to `emacs_spawn` is set to the subprocess id by the function. The standard input, output and error streams, `std_in`, `std_out` and `std_error` can be specified, together with a argument vector `argv` (`argv[0]` is the subprocess binary), environment block `envp` and the working subprocess directory `cwd`.

The function `emacs_spawn` uses `vfork` (from `unistd.h`) to clone the Emacs parent process. Unlike `fork`, the `vfork` function won't clone the whole address space, which is more efficient. After a call to `vfork`, the parent process will be suspended until the new child process exits or calls `execve` – replaces the forked Emacs subprocess with a new process. `vfork` returns the process Pid to the parent process or `-1` on a error. `vfork` function returns `0` in the child process.

The `child_setup` function is invoked by `emacs_spawn` as the final step in the creation of the synchronous and asynchronous child processes (see listing 67, from `src/callproc.c`). The function redirects the file descriptors using `dup2` and calls `emacs_exec_file`. The C function `emacs_exec_file` executes a program from a file using `execve` (see listing 68 from `src/sysdep.c`). The `execve` function replaces the parent process with that of a program.

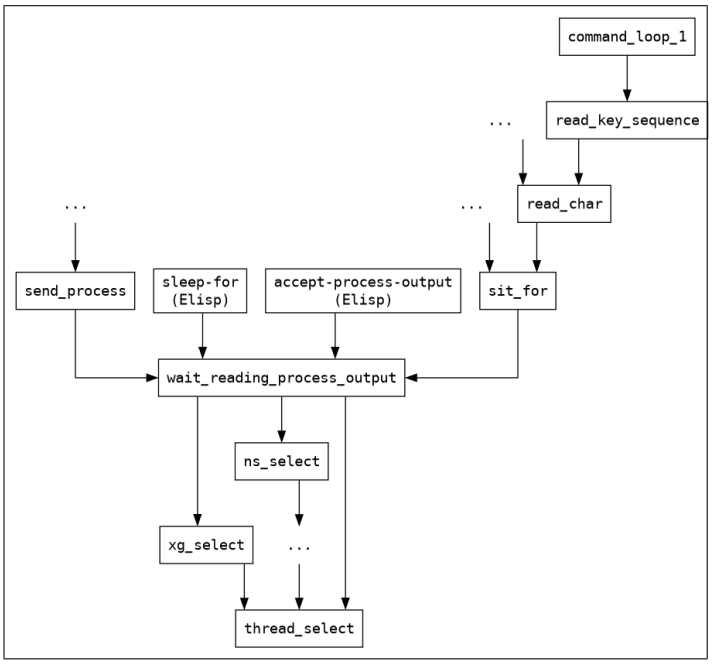

Figure 8: `thread_select` caller graph.

7.4.8 Reading Subprocess I/O

The function `thread_select` (from `src/thread.c` [3]) is when compiled for Linux-based systems a wrapper around the `pselect` function (see listing 27 for function synopsis) [9], which allows for "synchronous I/O multiplexing" (see the manual for "`pselect(2)`" [36]). The `pselect` function is used to monitor file-descriptors and determine which ones are ready for read or write operations, to handle pending input events and to monitor the death of a child process. The `thread_select` function calls `really_call_select` (from `src/thread.c` [3]).

The function `really_call_select` first blocks the `SIGINT` signal while releasing the GIL in-turn calling `pselect` which waits for child process file descriptors to become available for input / output optionally during a set amount of time. After the call to `pselect` the global lock is reacquired – if it is not already held such as when `C-g` (`keyboard-quit`) signals the `quit` condition which causes the signal handler to call `maybe_reacquire_global_lock`. A simplified caller graph of `thread_select` is given in figure 8 [10].

[9] According to a comment in `xgselect` function defined `src/xgselect.c` [3].

[10] No consideration has been taken for the conditions invoking the calls.

```
int pselect(int nfds, fd_set *readfds, fd_set
    *writefds,
            fd_set *exceptfds, const struct
                timespec *timeout,
            const sigset_t *sigmask);
```

Listing 27: `pselect` from `sys/select.h`.

The primary use of `thread_select` is found in `wait_reading_process_output` (defined in `src/process.c` [3]) where `thread_select` is used to detect the `SIGCHLD` signal – signaled from a child process when it exits. It is also used to find the file-descriptors of a child process which are ready for input or output operations. This is done through setting the sets of file-descriptors, `Available` and `Writeok` (through `pselect` [11]). If the Lisp variable `process-prioritize-lower-fds` is set to `nil` the file-descriptors will be read using Round-Robin (`wait_reading_process_output` in `src/process.c` [3]).

7.4.9 Internal Representation of Child processes

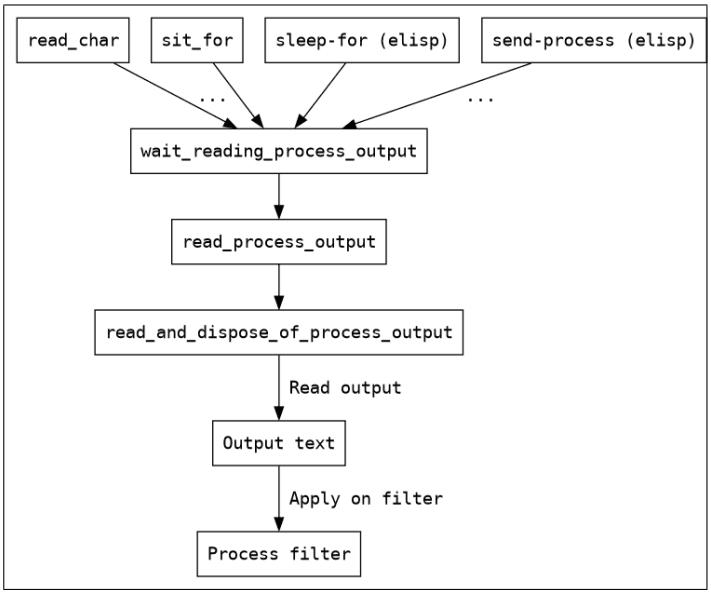

Figure 9: Reading child process output.

[11] According to the `pselect` Linux page of the System Calls Manual.

A child process or network connection is defined by the `Lisp_Process` `struct` (from `src/process.h` [3]) with some of its most important Lisp members being;

- `tty_name`: The name of the subprocess terminal `tty_name`.
- `name`: The process name `name`.
- `command`: A list of the arguments `command` applied to the subprocess program.
- `filter`: A Lisp function `filter` that receives the standard output of the running process.
- `sentinel`: A Lisp function `sentinel` that is called whenever the process state changes, such as when exiting or receiving signals.
- `log`: A lisp function `log` used for logging.
- `buffer`: The associated process `buffer`.
- `childp`: When the process is a network or serial connection `childp` is their arguments, otherwise `childp` is set to `t`.
- `plist`: A property list `plist` with child processes state information.
- `type`: A symbol `type` describing the process. Value is either `real`, `network` or `serial`.
- `mark`: The point of the last insertion `mark`.
- `status`: A symbol `status` indicating the status of the process. Examples being `run`, `open`, `closed`, `listen`, `failed` or a `cons` pair such as `(connect . ADDRINFOS)` or `(failed ERR)`.
- `decode_coding_system`: The coding system `decode_coding_system` used for the process output.
- `write_queue`: The process write queue `write_queue`.
- `stderrproc`: A pipe process `stderrproc` attatched to the subprocess standard error output.
- `thread`: The process `thread`, if set to a non-`nil` value then the child process is bound to a thread, which is the only thread allowed to wait on it. When a associated thread dies, `thread` is set to `nil` permitting other threads to wait for it (see `src/process.c` on line 964 [3]).

The non-Lisp members include;

- `pid`: The OS Pid (Process Id).
- `open_fd`: Associated file descriptors.
- `read_output_delay`: The delay for reading the process output (in ms).
- `alive`: A flag describing if the process is alive.
- `port`: The process port (*for network processes*).
- `socktype`: Socket type (*for network processes*).

The global variable `Vprocess_alist` (from `src/process.c` [3]) is an lisp `alist` containing the names and process lisp objects.

The internal function `wait_reading_process_output` (from `src/process.c` [3]) is responsible for reading asynchronous process output. The function can run under a time limit (if `time_limit` > 0) where it will try and wait for the process output for a maximum of `timeout` seconds. The function will unless specified using `read_kbd` or `do_display` perform re displays and listen for keyboard input. The function will stop waiting for process output and return if input is detected. The process output is read using `read_process_output` (in `src/process.c` [3]) fetching the process output in turn calling `read_and_dispose_of_process_output` which applies the process object and output text to the filter function figure 9.

7.4.10 The Process Write Queue

A write queue is utilized when sending data to a subprocess. Data is sent to a subprocess from Lisp using `process-send-string`, `process-send-eof` and `process-send-region` (defined in `src/process.c` [3]) which may block on some network processes. The function calls

`send_process` (from `src/process.c` [3]) checks if the write queue has data. If it has data it will push the string to the selected processes write queue using `write_queue_push` (in `src/process.c` [3]), followed by writing data from the queue until the queue is empty. If the process is unable to process input, then the data popped from the queue is requeued and `wait_reading_process_output` called. Data is sent either through the process file descriptor or a network socket.

The write queue ensures that Emacs does not block while waiting for a subprocess to be ready to receive input. Each process has a private write queue located in the process struct as the member `Lisp_Process.write_queue`. The write queue is manipulated using the functions `pset_write_queue` which sets the write queue of a process and `write_queue_push` and `write_queue_pop` which push and pops to the queue.

The queue is then handled in `send_process` through the function `emacs_write` which calls `emacs_full_write` (from `src/sysdep.c` [3]) which in turn calls `write` from libc (included in the `unistd.h` header) which writes data to a file descriptor. If a write were to block, then `wait_reading_process_output` is called to make the task of sending data non-blocking. After the function returns the write is retried until the write queue is empty.

```
(defun proc-filter (proc output)
  (message "%s"  proc (split-string output
     "\n")))

(defun proc-sentinel (proc event)
  (message "%s - %s" proc event))

(defun make-proc ()
  "List files in home directory."

  (make-process :name "ls"
                :command `("ls")
                :filter 'proc-filter
                :sentinel 'proc-sentinel
                :connection-type 'pipe))

(make-proc)
```

Listing 28: Lisp Example Process.

Listing 28 demonstrates the creation of a asynchronous child process from Lisp that execute the `ls` program which will print a list of the files in the current directory. The filter function `proc-filter` prints a message containing the standard output of the command when it is received. A sentinel `proc-sentinel` sends a message with the process name and status once invoked. Running it will yield messages similar to those in listing 29, where the directory filenames have been replaced with `<directory files> ...`.

```
( <directory files> ...)
ls - finished
```

Listing 29: Example output of elisp process example.

7.4.11 Emacs Batch Mode

Emacs has the ability to start in its *batch mode* by supplying the `-batch` command line flag. This starts a non-interactive Emacs session that doesn't define any user interaction, only executing a Lisp program followed by quitting when finished. The following arguments can be used to specify what Emacs is to execute.

- `-l <file>`: Execute the file `<file>`.
- `-f <func>`: Execute the Lisp function `<func>` (without arguments).
- `-eval <expr>`: Executes the Lisp expression `<expr>`.

This can be used together with the `-Q` flag which starts Emacs without any user configuration [22, ch.42.17]. Emacs Lisp programs often create external child processes to offload heavier compilations from the primary Emacs process. This is used in packages such as `async` [39] which utilize Batch Mode to create a asynchronous Emacs child process that executes Lisp in parallel to the parent process, with the processes communicating back and forward through Inter-Process Communication (IPC) such as a pipe. The child Emacs process would have its own state and memory space.

Futures is a common concurrency abstraction typically applied to threads, that talks about ongoing asynchronous tasks that finish sometime in the future. Here the future would be a object representing the child Emacs process, providing access to states such as "failed", "completed" and "pending", together with the result of commands or Lisp expressions sent to execute on the child process.

7.4.12 Lisp Threads

Threads were introduced in Emacs version 26 [4, ch.4.6] which included primitives for threads, mutexes and condition variables. All Lisp threads share the memory of the parent Emacs process. Emacs threads are "mostly" cooperative where the switch between threads only occur at well-defined occasions, either explicitly when a thread yields, while waiting for keyboard input, asynchronous process output, or during thread blocking operations (such as joining threads or operating on mutex locks).

When `src/thread.c` [3] is loaded Emacs creates a main thread, and sets itself to `current_thread`. The thread subsystem is then further initialized at the end of `main` in `src/emacs.c` [3] through calling `init_threads` (from `src/thread.c` [3]) which will lock the GIL to the main thread and initialize its byte compiler thread.

Emacs keeps a pointer to the state of the currently active thread in the global scope of `src/thread.c` [3] as the `current_thread` C variable. The state of all running threads is kept in a linked list `all_threads` and the mutex `global_lock` is used to as the Global Interpreter Lock (GIL) and is always held the currently running thread, giving it exclusive access to the Emacs Lisp Interpreter.

Each thread is defined by the structure `thread_state` (from `src/thread.h` [3]). The most relevant members of `thread_state` include the following.

- `name`: The thread name `name` given at creation when calling `make-thread`.
- `function`: The Lisp function `function` provided to `make-thread`.
- `result`: The return value `result` obtained from `function`.
- `error_data`, `error_symbol`: Error symbol `error_symbol` and data `error_data` used to store thread signals.
- `event_object`: A event `event_object` that the thread might be waiting on.
- `m_handlerlist`, `m_handlerlist_sentinel`: The threads active condition handlers `m_handlerlist`, `m_handlerlist_sentinel` used to handle error signals.
- `handlerlist`: A list of tags `handlerlist` from `catch` constructs.
- `m_specpdl`: A pointer to the address at the beginning of the threads special bindings stack `m_specpdl`. The pointer is set when the thread upon creation and cleared when the thread dies.
- `m_specpdl_end`: A Pointer to exactly after the threads last element in the special bindings stack `m_specpdl_end`.
- `m_specpdl_ptr`: A pointer to the threads first unused element in the special bindings stack `m_specpdl_ptr`.
- `m_lisp_eval_depth`: A value representing the current recursion depth of the thread `m_lisp_eval_depth`.
- `m_current_buffer`: A pointer to the threads current buffer `m_current_buffer` which is inherited from the caller when creating the thread.
- `thread_condvar`: A condition variable `thread_condvar` which the thread broadcasts to when upon exit, used for `thread-join`.
- `wait_condvar`: The condition variable the thread is waiting for `wait_condvar`, if one exists.
- `next_thread`: The next thread state `next_thread` in the linked list of threads.

7.4.13 Switching Threads

Thread switching only occurs in Emacs when:

I A thread explicitly yields using `thread-yield` (from `src/thread.c` [3]).

II Emacs is waiting for keyboard input

III Waiting for asynchronous process output using `accept-process-output` [22, ch.38.9.4].

IV Blocking operations on objects such as mutex locks, condition variables or operations like `thread-join` (defined in `src/thread.c` [3])

V When a interrupt / signal causes the main thread to "forcibly" reacquire the GIL (see `maybe_reacquire_global_lock` in `src/thread.c` [3]).

The cooperative nature of threads implies that threads must either explicitly yield or enter one of the previous conditions to switch threads. If Emacs will appear frozen if prevented from doing so for a longer amount of time as progress on other threads is blocked.

7.4.14 Yielding Threads

A Lisp thread yields using the Lisp function `thread-yield`, which (internally) is implemented as a call to `yield_callback` (from `src/thread.c` [3]) through `flush_stack_call_func` (from `src/alloc.c` [3]) (see figure 10) . The `flush_stack_call_func` function is called before Emacs releases the GIL helping the GC to identify roots in the registers of threads that are not currently executing any Lisp and flushes the threads registers onto the stack (see comment in function `flush_stack_call_func` in `src/alloc.c` [3]). Roots serve as entry points for finding reachable lisp objects, which are not to be garbage collected.

`yield_callback` saves a pointer to the currently running threads state, releases the GIL (using `release_global_lock` from `src/thread.c` [3]) and then yields (see figure 10) using `sys_thread_yield` from `src/systhread.c` [3]. If Emacs is compiled for a Linux system `sys_thread_yield` will have been defined using `sched_yield` (from the C library *sched*, see Man-page `sched_yield(7)`) where the OS scheduler will cause the running thread to relinquish the CPU, move it to the end of the OS priority queue and run the next thread. When the OS returns the CPU to the thread it will continue by acquiring the GIL (with `acquire_global_lock` from `src/thread.c` [3]) using the threads previously saved stat.

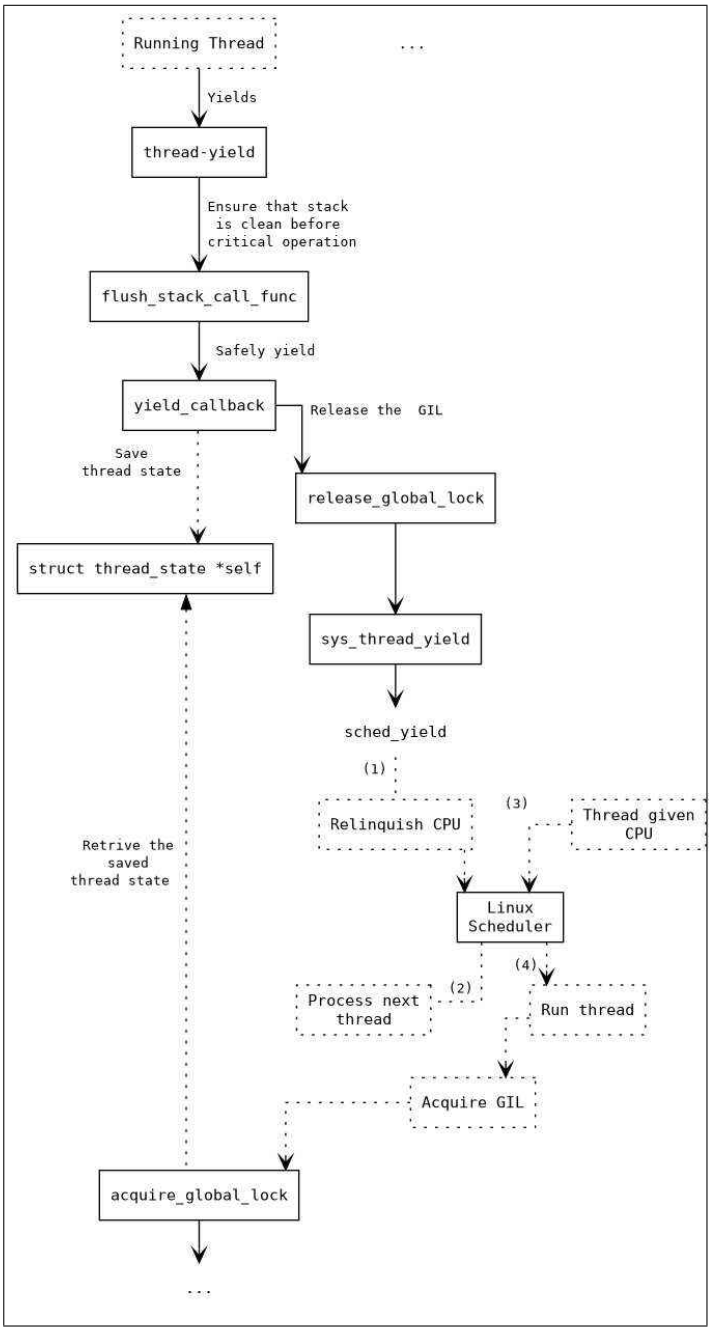

Figure 10: Flow diagram of a yielding Lisp thread.

7.4.15 Creating and Running a Thread

The lisp function `make-thread` creates and runs a lisp thread. The function first allocates `thread_state` and sets arguments provided upon the call to `make-thread` (`function` and `name`) in the thread state. The currently running threads buffer is then inherited by the new thread, the special bindings stack and `thread_condvar` is set up. Finally the new threads `thread_state` is prepended to the linked list `all_threads` which contains all threads. An OS thread is then created using `sys_thread_create` which executes the function `run_thread` in the os thread using the new thread state as argument. This creates and runs the new thread. While the new threads function has been called within the new OS thread the previously running thread creates a lisp object for the new thread and returns it.

The internal function `run_thread` runs on the new OS thread. It first initializes the thread and aquires the GIL acquired after which the Lisp function given upon thread creation is called using `invoke_thread_function` (from `src/thread.c` [3]) within a condition case (to handle thread errors). The result is then stored in the thread structure. When the thread function returns, the thread is removed from any processes locked to it and its special bindings and handler-stack are freed. Then the thread will broadcast to its `thread_condvar` to signal all threads waiting for it. Finally the thread is removed from the `all_threads` linked list and releases the GIL.

7.4.16 The Global Interpreter Lock

The Global Interpreter Lock (GIL) is a mutex called `global_lock` (from `src/thread.c` [3]) that ensures that only one thread has access to the internal state of Emacs at once. It is used in Emacs to synchronize the execution of threads, preventing race-conditions by restricting Emacs to only run one thread at a time where all but one thread stay blocked waiting on the mutex lock. The GIL prevents Emacs from any form of simultaneously occurring Lisp execution, which the interpreter is not built to handle. This acts as the bottleneck on the Emacs' Lisp threads, and their further abilities for concurrent processing.

Upon startup Emacs while in the `main` function, the internal function `init_threads` is called (defined in `src/thread.c` [3]) . The function initializes the GIL, and locks it to the main thread. When a thread acquires the GIL, the internal function `acquire_global_lock` (from `src/thread.c` [3]) is called. The function use `sys_mutex_lock` to lock the GIL implemented using `pthread` (from `src/systhread.c` [3]) when Emacs is compiled on a Linux system with `pthread` support. This is followed by unbinding the previously running threads special bindings and rebinding the to-be-running threads bindings, then the current buffer will be updated to reflect the new thread and if any pending signals exists they are processed.

There are three different types of functions which calls `acquire_global_lock` (see figure 11) the first are Lisp functions which directly acquires the GIL, namely `thread-yield` and `make-thread`.

The second type is when the main thread tries to handle a incoming signal. The `SIGINT` signal usually terminates programs on UNIX systems, however Emacs only terminates if its in Batch Mode when receiving the signal. The `SIGINT` handler always run on the main thread. When a `SIGINT` has been delivered to the main thread and has prevented the main thread from acquiring the GIL during `thread_switch`, then the main thread will call `maybe_acquire_global_lock` to ensure that it holds the GIL.

The third type is a tree of functions which go through `wait_reading_process_output` (to the right in figure 11). The function `wait_reading_process_output` will call `thread_select`, and as such all of its callers will indirectly call `thread_select`.

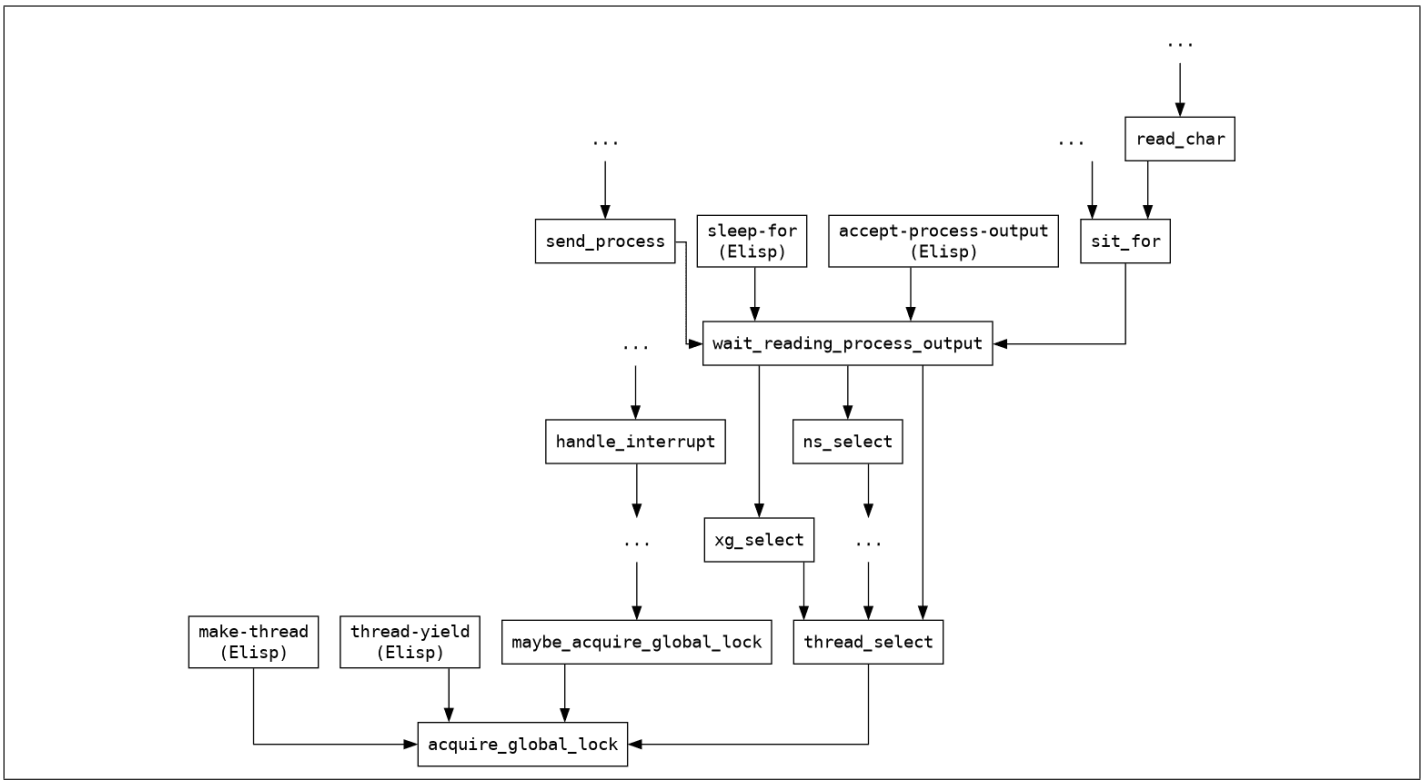

Figure 11: Caller graph for `acquire_global_lock`.

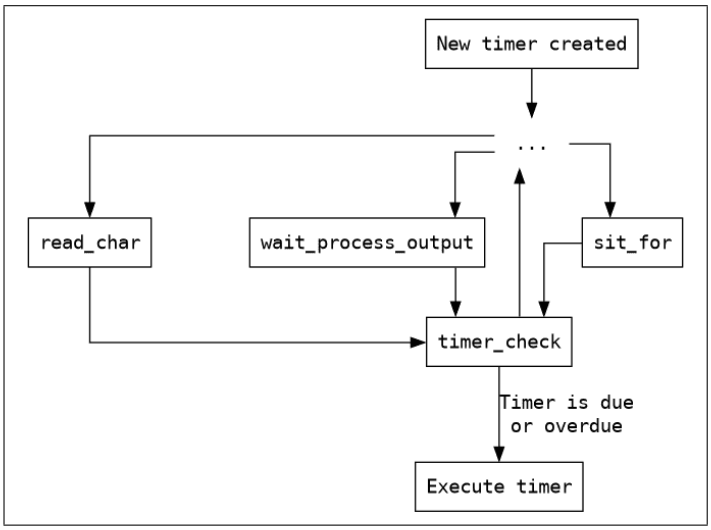

Figure 12: Internal invocation of timers.

The function `flush_stack_call_func` is a "trampoline function" (comment in `src/alloc.c` on line `5502`) [3]) used to flush the registers to the stack whenever Emacs is about to release the GIL. This helps the garbage collector to find roots in the registers of threads not actively executing lisp. The function is used in the implementation of `thread-join` (from `src/thread.c` [3]) when calling the threads callback function.

7.4.17 Lisp Timers

Emacs has a feature called timers. A timer is a special object containing information about a function and a planned future invocation time (or times). Timers are used to make periodic or single call to a function at a specified future time or after a certain length of idleness [22, ch.42.11]. The invocation of a timer can only occur while Emacs is waiting to accept output from a child process or is inside certain primitive functions such as `sit-for` or `read-event`. The implementation of the Lisp function `read-event` calls `read_filtered_event` which in turn calls `read_char`.

Since timers only can execute while Emacs is waiting for process output, user input or explicitly waiting using `sit-for`. A timer set to execute at a time when Emacs is busy will then be executed at the next possible opportunity. Due to timers being used to execute code "out-of-sight" they are usually written to be lightweight to avoid any noticeable slowdown from delaying the command loop for too long. When a timer does a lot of work, the computation-heavy expression should be wrapped in a `with-local-quit` block, which allows the keybinding `C-g` (bound to `keyboard-quit`) to work. Otherwise the UI of Emacs could freeze without any way for the user to cancel the running execution [22, ch.42.11].

Some internal functions that directly or indirectly make calls `timer_check` are given below [12].

- From `command_loop_1` to `read_key_sequence` to `read_char` ... `timer_check`.
- From `wait_reading_process_output` to `timer_check`.
- From `sit_for` ... `timer_check`.

[12] All functions are defined in `src/keyboard.c` [3].

```
(defvar-local timer (timer-create))
(timer-set-time timer 10)
(timer-set-function timer '(lambda ()
    (message "Hello from timer")))
(timer-activate timer)
```

Listing 30: Creating a timer in Emacs Lisp.

We provide a example (see listing 30) which creates a new timer, which is bound to after 10 seconds execute a anonymous lambda function which sends the message `"Hello from timer"`.

8 Emacs and Concurrency

This section fulfills the second thesis objective TO2 and describes Emacs concurrency in two levels of abstraction, starting with concurrency from the perspective of Emacs Lisp in section 8.1 followed by a lower-level description of the concurrency of the Emacs core in section 8.2

8.1 Concurrency in Emacs Lisp

The native implementation of threads in Emacs Lisp relies on the GIL to lock down the interpreter, preventing Lisp from executing in two or more threads at once. As the Lisp interpreter cannot run simultaneously, it may be preferred to use abstractions that achieve concurrency using different, more tested, native features of Emacs Lisp [13]. This section summarizes a set of Emacs packages that provide different forms of concurrency in Emacs Lisp.

One way to achieve parallel Lisp execution is through the asynchronous subprocesses, where worker Emacs processes are created to evaluate Lisp in the background. This is currently leveraged in multiple places, such as the native Emacs Lisp compiler which spawns Emacs workers processes to compile source files in parallel, running in the background (see the files `lisp/emacs-lisp/comp-run.el` and `lisp/emacs-lisp/comp.el` [3]).

Deferred execution can be modeled using **futures** and **promises**, objects that relate to a not (yet) known result of some computation. [26] [37].

Many Emacs libraries that implement concurrency do so by applying concepts similar to futures and promises, often borrowed from existing implementations in other languages. This section will provide a rough introduction to these concepts in preparation following sections.

The definition of promises / futures differs between implementations and languages, but can be summarized as a object representing the result of a computation whose value will become known in the future, given to a consumer and set by a producer

[13] More on this in section 8.2.

Some examples of such Emacs Lisp libraries are; `pfuture.el` [25] (see section 8.1.3), `aio.el` (see section 8.1.2), `deferred.el` [29] (see section 8.1.1), `async-await.el` [5] and `promise.el` [6].

A future is composed of a value and a status, where the status signifies to the consumer if the value has been computed. The status will be either **pending**, **cancelled** or **completed** at any point in time. When a future is created its object is returned directly, the consumer can block waiting on the value or check the status of the value which is available when the producer has finished its computation, updated the status and returned / set the value. A future may also be composed of a callback function which is called on the value given by a producer when it has finished its computation [37].

The concept of **Async / Await** is similar to futures and promises, where a **async** operation allows for asynchronous computation which the consumer can choose to synchronously wait on through applying a **await** operation.

8.1.1 Package `deferred.el`

The Emacs package `deferred.el` [29] is an implementation of the JavaScript library `JSDeferred` (see JSDeferred) which adds promises to JavaScript [29]. The Promise object in the Lisp library has the following parameters (see the struct `deferred` in `deferred.el` [29]):

`callback` Callback function.

`errorback` Error callback function.

`cancel` Canceling function

`next` A chained Promise.

`status` Is set to the symbol `ok` when callback has been called `ng` if the error callback was called, or `nil` if neither.

`value` The promise return value or `nil`.

Deferred revolves around chains of promises, where each promise is executed concurrently using timers and idle timers. A concurrent Lisp function implemented using `deferred.el` in section F.

8.1.2 Package `aio.el`

The `aio.el` (Async IO) Emacs package implements a Lisp library inspired by the `asyncio` Python library and async/await JavaScript feature [38, 37]. The `aio.el` library uses timers / idle timers in order to achieve asynchronous and delayed Lisp execution. Generators (see [22, ch.11.6]) are used to allow functions to yield and resume execution.

The `aio.el` package is built around the concept of promises, represented by the promise object. A promise object is a composition of the future **result** of a asynchronous computation and a list of callback functions called *subscribers* waiting on the result. Once the result is available, each subscriber is called with the result [37].

```
;;; async-insert.el -*- lexical-binding: t;
    -*-
(require 'aio)

(defconst small-delay 0.00001
  "Default delay for \\[async-insert].")

(aio-defun async-insert
   (pos text &optional delay)
  "Asynchronously insert `TEXT' at position
   `POS' in the current buffer. Wait `DELAY'
    after inserting each character."
  (interactive "d\ns")
  (undo-boundary)

  (let ((e (point-marker)))
    (cl-loop for c in (string-to-list text)
             when (aio-await (aio-sleep
                      (or delay small-delay)
                         t))
             do (with-current-buffer
                  (marker-buffer e)
                  (save-mark-and-excursion
                    (goto-char
                       (marker-position e))
                    (insert-before-markers
                      (string c)))))))
```

Listing 31: Example of using `aio.el` [38].

Listing 31 demonstrates the implementation of a asynchronous and non-blocking `insert` wrapper using `aio.el`. The function `async-insert` from listing 31 asynchronously inserts text into a buffer one character at a time, waiting some `delay` between each insertion.

8.1.3 Package `pfuture.el`

The Emacs package `pfuture.el` [25] provides a minimal implementation of futures for the purpose of *running external programs*. The package implements the ability to run external processes as futures, which can be synchronously waited on and whose IO streams can be handled, with the ability to attach callback functions to the futures.

```
;; -*- lexical-binding: t; -*-
(require 'pfuture)
(defun test nil
  (let ((start   (float-time))
        (future0 (pfuture-new "cat"
                   "invalid"))
        (future1 (pfuture-new "echo" "done")))

    ;; Wait for each future, at most 1 sec
    (pfuture-await future0 :timeout 1
       :just-this-one nil)
    (pfuture-await future1 :timeout 1
       :just-this-one nil)

    (message "f0: %s\nf1: %s"
             (pfuture-stderr future0)
             (pfuture-result future1))

    ;; Return total execution time
    (- (float-time) start)))
```

Listing 32: Lisp Program using `pfuture.el`.

Listing 32 illustrates a Lisp program using the `pfuture.el` library [25]. The program creates two futures where *future 1* tries to output a non-existing file to `STDOUT` and the *future 2* prints "done". The program then waits for each future *at most 1 second* and then prints the `STDERR` stream of the first future followed by the `STDOUT` stream of the second, returning the time difference from before and after the execution.

```
f0: /usr/bin/cat: invalid: No such file or
    directory
f1: done
0.0147552490234375
```

Listing 33: Output of `test`.

Executing the function `test` interactively would result in a message output similar to listing 33.

8.1.4 Package `asyncloop.el`

The GNU Emacs package `asyncloop` [15] tries to be easier to use than `aio.el` [38] and `deffered.el` [29] by not relying on *"too sophisticated"* concepts [15] such as callback functions, deferred execution. The difference between `asyncloop` and `async.el` [39] is that `asyncloop` allows the execution of functions with direct access to the internal state of the primary Emacs process [15]. The primary use-case for `asyncloop` is to execute a list of functions without the latency of having to wait for each one to finish [15].

```
(cl-defstruct (asyncloop (:constructor
    asyncloop-create)
                         (:copier nil))
  starttime
  log-buffer
  immediate-break-on-user-activity
  (timer (timer-create))
  (paused nil)
  (remainder nil)
  (scheduled nil)
  (just-launched nil))
```

Listing 34: The `asyncloop` Data structure from `asyncloop.el`.

The package works by introducing the concept of **asyncloop**s as a data structure (see its definition in listing 34 taken from `asyncloop.el` [15]).

The value of `immediate-break-on-user-activity` in the struct will cause any input to immediately stop the execution of the `asyncloop` if set to a non-`nil` value. A queue of functions waiting to execute is stored in `remainder` where the current function is popped from at the end of its execution. An idle timer is stored in `timer` which is used to pause execution and allow user input during the execution of the loop. The variable `scheduled` is set to a non-`nil` value when the asyncloop timer has been started and `paused` to a non-`nil` value whenever the asyncloop is not scheduled to run in the future [15].

The loop can be configured to cancel the execution of Lisp after any new input events are queued, or when `C-g` / `keyboard-quit` (defined in `lisp/simple.el` [3]) is invoked. If so, it delays the reinvocation of the currently executing function to the next loop using a idle timer [15]. This is possible since the loop uses a queue of Lisp functions which is only popped when the function has finished executing. If input is detected, the next loop iteration is delayed and the same function will execute.

The package remaps `keyboard-quit` to a new Lisp function `asyncloop-keyboard-quit` [15] which iterates through all active `asyncloop` objects and cancels them using `asyncloop-cancel`, which stops and resets the queued up `asyncloop`s, configuring them for a *fresh* restart with a subsequent call to `asyncloop-run`.

The primary function of `asyncloop` [15] is `asyncloop-run` which queues a list of functions, delaying any function whenever user input is pending. Each function should take the `asyncloop` object as argument, allowing it to be manipulated by the function.

The specific permutation of Lisp functions is hashed to detect attempts of creating new `asyncloop` objects using the same permutation of functions, in which case it will use the already existing `asyncloop`. A running `asyncloop` can be manipulated by the executing functions with the following functions; `asyncloop-pause`, `asyncloop-resume`, `asyncloop-cancel`, `asyncloop-remainder` and `asyncloop-log`.

The function `asyncloop-eat` resumes an `asyncloop`, calling the queued function inside a `while-no-input` statement to ensure that Emacs will quit out of any running code when user input is received followed by rescheduling the future continuation of the loop whenever Emacs has been idle for 1 second. If the `asyncloop` was created with the `immediate-break-on-user-activity` flag set to `nil` it won't stop upon user input, but will instead ensure that the loop can be stopped by `keyboard-quit` through running the Lisp function inside of `with-local-quit` ensuring that `C-g` works as expected.

The function `asyncloop-schedule` sets a `asyncloop` object to resume at the next possible time by creating an idle timer set to run when Emacs is idle for a specific amount of seconds / immediately upon becoming idle.

The function `asyncloop-remainder` accesses the list of functions which are queued up to execute, allowing values to be

pushed during the execution of the loop, making it possible to queue the current function to be executed in the next loop.

The `asyncloop-chomp` function sequentially executes the queued up functions of a `asyncloop`, calling itself recursively in order to execute the next function. The function checks if any user input is pending between each invocation. If input is pending then `asyncloop-schedule` is used to cancel the running loop, postponing its execution 1 second of idle time. If the loop has recursively called itself more then 100 times, then it is rescheduled to run at the next possible idle moment. This prevents the case where there are a lot of scheduled functions or modifications of the queue from making Emacs hit the maximum recursion limit which would signal an error (see `max-lisp-eval-depth` defined in `src/eval.c` [3]).

8.1.5 Package `async.el`

The Emacs package `async.el` [39] is a package that provides a library for parallel Lisp execution, including functions that are loaded by and execute on the child Emacs process. This keeps the state separated between the processes, allowing for *"true parallelism"*.

The package adds the ability to call Lisp asynchronously through child Emacs worker processes, providing auxiliary functions for data / message passing, managing state, futures, non-blocking reads and asynchronous callback. Once a asynchronous Lisp function created using `async.el` returns, the child process exits [39].

The function `async-start` (see `async.el` in [39]) is used to start a child Emacs process. The function works by spawning a non-interactive Emacs process using the `-batch` flag, set to load the `async` library and evaluate `async-batch-invoke` without any user configuration.

The package allows for two-way communication between the processes. This is achieved through printing Lisp-expressions (*after removing any text properties*) into UTF-8 strings, sending the base64 encoded strings between the processes, applying the process in reverse to decode received messages (see the functions `async-insert-sexp`, `async-receive-sexp` and `async-transmit-sexp` in `async.el` [39]).

Auxillary macros are included to ease with the separation of state between the processes, such as `async-let`, `async-sandbox` (from `async.el` [39]). The package relies on callback functions for the return value of the asynchronous function and for handling messages (see the DocString of `async-start` in `async.el` [39]).

The Emacs package `async-job-queue.el` builds upon `async.el` and adds the ability to queue tasks in a FIFO queue. This intends to limit the number of sub-processes created by the `async.el` library.

8.1.6 Computing `async.el` Overhead

To roughly determine the delays of using a child Emacs process to execute Lisp in the `async.el` library, this section splits up the process into variables which each represents a step in the process of; setting up the child process, sending an encoded Lisp expression as a string, for the child process to decode and evaluate the Lisp expression, the time it takes to encode the result and send it back to the parent process, which then decodes the received message and reads it into a Lisp expression. The time it takes to decode / encode the printed Lisp expression is:

$$t_{dec} = t_{d64} + t_{read} \tag{1}$$

The time it takes for the child process to decode a received base 64 encoded string is t_{dec} and the time it takes to decode the received string from base64 into a UTF-8 string is t_{d64} and t_{read} is the time it takes to parse / read string into a S-Expression.

$$t_{enc} = t_{print} + t_{utf-8} + t_{e64} \tag{2}$$

The variable t_{enc} is the time it takes to create a base 64 encoded string from a S-Expression, where t_{print} is the time it takes to print a S-Expression into a string, t_{utf-8} is the time it takes to encode the printed string into UTF-8 and t_{e64} is the time it takes to encode the UTF-8 encoded string into base 64, which is the final string that is sent to the child Emacs process.

$$t_{new} = t_{exe} + t_{load} + t_{eval} \tag{3}$$

The time to start Emacs and enter a state where it is ready to process received Lisp code and listen to messages from the parent process t_{new} is the total delay it takes for `async.el` [39] to

setup the child process. The time to start the Emacs child process is t_{exe} , i.e the time to start the Emacs binary, the time it takes for the started child process to load a Emacs Lisp library is t_{load} which in this case is the time it takes to load `async.el`, after which Emacs invokes `async-batch-invoke` from `async.el` with a delay of t_{eval}.

$$t_{init} = t_{conf} + t_{new} \tag{4}$$

Time it takes for the parent process to setup a asynchronous environment is t_{init}, i.e time it takes and for the child process to enter a listening state, ready to receive commands from the parent process. Where the initialization file executed by the child process on startup, used instead of the default Emacs file used when normally Emacs can be given in the variable `async-child-init` when `async.el` [39] is used, which would add a additional delay, this delay is included in t_{conf}.

$$t_{send} = t_{enc} + t_{trans} \tag{5}$$

$$t_{recv} = t_{buf} + t_{dec} \tag{6}$$

The time it takes to send a encoded S-Expression t_{send} and the time it takes for the parent process to be ready to evaluate the answer t_{recv} , it is assumed that t_{send} and t_{recv} its equal in both directions, which in real world conditions likely wouldn't be the case as the Emacs instances would have different delaying factors such as timers. The t_{buf} variable is the time it takes for the parent process to buffer the received base 64 encoded string until it is ready to decode. It is assumed that there is no buffering when messages are sent. The variable t_{trans} is the transmission time from the encoded string is sent until it is received and added to the buffer.

$$t_{async} = t^c_{init} + t^p_{send} + t^c_{recv} + t^c_{eval} + t^c_{send} + t^p_{recv} + t^p_{eval} \tag{7}$$

Roughly, the total time it takes to call `async-start` (from `async.el`) and for a result to be returned and read by the parent is t_{async}, i.e asynchronous Lisp evaluation. We assume that this only includes the initialization child process t_{init} . The parent process is t^p and child process is t^c. The whole process includes the parent sending a Lisp expression t^p_{send} , the child process receiving it t^c_{recv}, evaluating it t^c_{eval}, sending the result back t^c_{send}, the parent process receiving the result t^p_{recv} and finally interpreting represented by the t^p_{eval} delay. All delays vary, however t_{new} will stay close to constant since it is not dependent on any runtime environment. This assumes that the child process doesn't reply after initialization with a *"readyness"* message. Computing the time for a child process that stays alive over multiple asynchronous evaluations is harder the environment could change, and as such also the delays.

```
;;; -*- lexical-binding: t -*-

(require 'async)

(defun t-async (val)
  (let ((start (current-time))
        (f (lambda (val)
             (expt 999912311 val))))

    (let* ((start2 (current-time))
           (time2 (progn
                    (funcall f val)
                    (float-time (time-since
                        start2))))
           (start (current-time))
           (result
            (async-get (async-start
                        `(lambda ()
                           (funcall ,f
                               ,val))))))


      (abs (- (float-time (time-since
          start))
              time2)))))

(t-async 1000) ;; ==> ~ 0.154...
```

Listing 35: A program estimating t_{async}.

Listing 35 computes a estimate of t_{async} using the function `t-async`. This is done through executing a simple function that computes a large exponent two times, first normally in the parent process, then synchronously in the child Emacs process.

Assuming that the execution-time for the function `f`, t_f is equal on the child and parent process $0 = t_f - t_f$ leaves only the delays relating to the use of `async.el`. The `t-async` function then returns the time difference in seconds, which *roughly* should correspond to t_{async}, which in our case t_{async} was around $0.11 - 0.15$ seconds. The result will be a rough estimate as it is sensitive to factors such as hardware, system configuration, state of Emacs etc.

8.1.7 Package `session-async.el`

The `async.el` package creates a new sub-process each time it runs the function `async-start`, which due to the overhead of creating new Emacs processes can be quite resource intensive. The package `session-async.el` attempts to build upon `async.el` and add functionality that allows it to reuse child-processes instead of creating a new one for each task.

The package makes use of iterators to wait on messages from the Emacs child-processes and provides the function `session-async-new` for creating a new session, which sets up the Emacs subprocess and directs it to load its library components. When a new asynchronous session is created, two child-processes are created, one "session" process and one network "listener" process. The `session-async-shutdown` function terminates the session, killing the subprocess.

When a new session process is created, it is given the path to the `session-async.el` library and set to execute the function `session-async-eval-loop`. The eval loop starts by creating a connection back to the Emacs parent process using the Emacs package `jsonrpc`, which handles requests using a dispatcher, translating the received strings to S-expressions, evaluating them, translating the result back into a string and sending it back.

There are two ways to asynchronously execute Lisp; The first way is through the `session-async-start` function, which executes a function in a session process, returning a session Object and passing the return value to a callback function. The second way is through `session-async-future`, which creates a future that executes in the session process, which can be waited on and accessed using `iter-next`.

```
;;; -*- lexical-binding: t -*-
(require 'session-async)

;; Session
(defvar semacs (session-async-new))

(defun compute-primes-under (range)
  "Compute all prime numbers under `RANGE'
  as a vector.
Uses the Sieve of Eratosthenes algorithm."

  (session-async-future
   `(lambda ()
     ;; Returns vector of primes < range.
   ))
   semacs)
```

Listing 36: Example program using `session-async.el`.

```
;; Compute primes for 10, 99, 14 in subprocess
(let ((p1 (compute-primes-under 10))
      (p2 (compute-primes-under 99))
      (p3 (compute-primes-under 14)))

  (message "1: %s\n2: %s\n3: %s"
           (iter-next p1)
           (iter-next p2)
           (iter-next p3)))
;; ==> 1: (7 5 3 2 1)
;;     2: (97 89 83 79 73 71 67 [...]
;;     3: (13 11 7 5 3 2 1)
(session-async-shutdown semacs)
```

Listing 37: Example of using function from listing 36 `session-async.el`.

Listing 36, shows an example of implementing a function using the library, where the function `compute-primes-under` computes a vector of prime numbers, listing 37 then uses the function to compute three different vectors of primes and prints them, finally shutting down the session.

8.2 Concurrency: The User Experience

This section contains examples of issues that we found while working on this thesis, it will attempt to highlight some of the issues with the current implementation, implement smaller solutions in Lisp and explain the background required to understand them.

The cooperative threading model used in GNU Emacs currently lacks any proper system for preemption and leaves the responsibility to the programmer to write code that doesn't give rise to concurrency related bugs, such as deadlocks. Emacs currently has a tendency to temporarily or permanently freeze when certain bugs occur using the `thread` library (internally defined in `src/thread.h` [3] and `src/thread.c` [3]).

It was found that bugs that softbrick or freeze Emacs often occur when errors and / or local-exits (especially the `quit` or `top-level` catch tags) are handled in a way that prevents Emacs from returning to the command loop (or top level), making it hard to regain interactivity after something goes wrong in Lisp threads.

A Lisp condition variable (condvar) is a object that can be used to block threads until notified or signaled by another thread using `thread-signal` or `condition-notify` (see [22, ch.39.3]). When another thread has signaled or notified the thread or condvar, all waiting threads are unblocked.

```
(let* ((m  (make-mutex "mutex"))
       (cv (make-condition-variable m)))
  (mutex-lock m)
  (condition-wait cv))
;; ==> Emacs will freeze
```

Listing 38: Softbricking the Emacs main thread.

Emacs doesn't prevent threads from waiting on condition variables when there are no other threads able to unlock the condition variable, as demonstrated in listing 38. We tried unsuccessfully signaling `SIGUSR2` from an outside process to try to unfreeze Emacs by forcing the main thread to handle the interrupt, so we had to resort to the `SIGKILL` or `SIGTERM` signals to kill / terminate the process.

```
// [...]
mutex = XMUTEX (cvar->mutex);
if (!lisp_mutex_owned_p (&mutex->mutex))
  error ("Condition variable's mutex is not
      held by current thread");
flush_stack_call_func
    (condition_wait_callback, cvar);
// [...]
```

Listing 39: From the C definition of `condition-wait` in `src/thread.c` [3].

```
// [...]
saved_count = lisp_mutex_unlock_for_wait
    (&mutex->mutex);
if (NILP (self->error_symbol))
  {
    self->wait_condvar = &cvar->cond;
    sys_cond_wait (&cvar->cond, &global_lock);
  }
// [...]
```

Listing 40: Parts of the function `condition_wait_callback` from `src/thread.c` [3].

Parts of the C definition of `condition-wait` in `src/thread.c` [3] can be seen in listing 39. We can see that it does nothing to confirm that it is possible for other threads to notify or signal the condition variable using `thread-signal` or `condition-wait` (see [22, ch.39.3]).

Implementing a safeguard to prevent the code from listing 38 from freezing Emacs wouldn't require too much work and can be implemented in Lisp. To implement such a safeguard we must first identify conditions which cause Emacs to freeze. The problem found in listing 38 was found to occur in the following two situations.

I. If **only one thread exists** and it blocks waiting on a condition variable then no threads will be able to signal or notify the condition variable, permanently freezing the thread.

II. A thread should not be able to wait on a condition variable when *all other threads* are *waiting on a mutex / condition* **variable** as this would deny the possibility for any thread to unblock the waiting thread, in this case creating a **deadlock**.

```
(define-error
 'infinite-condition-wait
 "Can't wait on a condition variable with one
     thread")

(defun thread-alone-p (thread)
  "Return t if only the `THREAD' is running"
  (let ((all-thr (all-threads)))
    (and (= (length all-thr) 1)
         (eq thread (car all-thr)))))

(defun safe-condition-wait (condvar)
  "Safe version of \\[condition-wait], make
     the \\[current-thread] wait for
     condition variable `CON\
  DVAR'."
  (when (thread-alone-p (current-thread))
    (signal 'infinite-condition-wait
            `(thread-alone-p
                (current-thread))))

  ;; [ More tests can be added here ...]

  condvar)

(advice-add 'condition-wait
            :filter-args
                #'safe-condition-wait)
```

Listing 41: Macro for protecting `condition-wait` from executing when only one thread exists.

```
(apply #'condition-wait
       (apply #'safe-condition-wait ARGS))
```

Listing 42: How `safe-condition-wait` is applied on the arguments of `condition-wait`.

Listing 41 mitigates the problem from listing 38 by filtering the arguments for `condition-wait` with a advice [22, ch.13.11]. This allows us to call the function `safe-condition-wait` (from listing 41) on the arguments to `condition-wait` before it is called. This is illustrated in listing 42. We have implemented the predicate `thread-alone-p` (in listing 41) to check if a `thread` is the only live thread. If so `safe-condition-wait` will signal a custom error symbol called `infinite-condition-wait` to ensure that the error can be handled using `condition-case`.

```
(let* ((m (make-mutex "mutex"))
       (cv (make-condition-variable m)))
  (mutex-lock m)
  (condition-wait cv))
;; ==> Can't wait on a condition variable
    with one thread ...
```

Listing 43: Testing the `safe-condition-wait` macro from listing 41.

The same code from listing 38 is executed again in listing 43, but with the new function we added as a advice in listing 41. As intended, executing the code signals a error instead of freezing Emacs. The function can be expanded to check for more situations that softbrick or freeze Emacs.

```
kill -s SIGUSR2 <pid>
```

Listing 44: Forcing switch to main thread using signals.

Emacs will only switch threads at well defined times [22, ch.39]. The conditions for Emacs to switch threads are stated in section 7.4.13. If Emacs fails to reach one of these conditions during the execution of a thread, Emacs won't switch threads and long running Lisp code will make Emacs appear frozen or unresponsive. The only workaround is to wait until the computation is

finished, press or hold down C-g, restart Emacs, or signal either (depending on configuration) SIGUSR1 or SIGUSR2 from an outside process (see [22, ch.22.7.12, ch.19.1.1]).

The SIGUSR2 signal will by default make Emacs enter the debugger. Both SIGUSR1 and SIGUSR2 can be bound to a arbitrary command in the special-event-map (defined in src/keyboard.c [3]).

8.3 In The Emacs Core

This section will introduce ways to improve the way the Emacs core supports concurrency, with much of it focused on the removal of the GIL. Together with improvements discovered during the writing of section 7, this chapter will summarize various improvements published or mentioned online.

8.3.1 Data Races and Shared State

The first step in improving the concurrency of GNU Emacs is to identify the shared state throughout the Emacs core, followed by determining critical sections where the shared state is accessed. Critical sections are sections of code which only one thread of execution should be able to exist in at once. These sections should generally be as small as possible and should together cover all access to the shared state in the core, such as certain global and static variables [24, p.105]. The critical sections can then be protected using mutex locks. Certain variables that differ between threads (such as the lexical interpreter state) should be moved to the thread state. Automated tools (such as Doxygen) can be used to generate dependency, caller and callee graphs, something which is useful for visualizing the internal shared state.

One solution to the shared state of C code used in components is to give threads their own memory space for Lisp object allocations that occur during the runtime of the thread. This is currently being worked on in preparation for greater future multi-threading support through indirection hidden using macros. When a Lisp function or variable is created using the DEFUN and DEFVAR_... macros, they are added to the struct emacs_globals, which encapsulates them (see comment for DEFVAR_LISP in src/lisp.h [3].

It is currently possible for data races to occur when different OS threads use functions that directly or indirectly interact with the allocator and garbage collector. This is because many of the internal functions are dependent on BSS or DATA allocated global and static variables (see section A), which all OS threads share access to.

One example of this is the Lisp Object blocks, which are used whenever Lisp Objects are allocated (such as calling cons). Currently blocks has no mechanism for preventing serious memory and allocation-related data races that may happen whenever a threads tries to allocate a Lisp object at the same time as another concurrent OS-level thread modifies the same block before the first thread has finished allocating the object. Any interaction with the block allocator should thus be contained in a critical section and protected by a mutex lock.

Data races are currently possible if one thread allocates a Lisp object at the same time as another thread affects the free-list, block index or cons block. More specifically, when multiple Lisp Objects of the same type is either allocated or freed. When allocating cons objects, the variable consing_until_gc (from src/alloc.c [3]) is decremented, which can cause an invocation of the GC through when maybe_gc src/lisp.h [3] is called, see C definition for the Lisp subroutine cons in src/alloc.c [3]).

8.3.2 Shared State, The Lisp Reader

The Emacs Lisp reader (or parser) (defined in src/lread.c [3]) is an example of a component which relies on a lot of shared state.

The reader is used whenever source code needs to be read from text into Lisp and contains many of the most crucial and used auxiliary C functions and Lisp subroutines. This includes Lisp subroutines for tasks such as reading input (see read-char and read-event src/lread.c [3]), evaluating buffers / regions (see eval-buffer and eval-region src/lread.c [3]), loading files (see load), interning symbols (see intern), parsing / reading Lisp expressions (see read and read-from-string src/lread.c [3]), defvar src/lread.c [3]. These functions are integral to Lisp evaluation, parsing and loading, making up much of the commonly used parts of the language. This makes the reader

component especially important to harden were the GIL ever to be removed.

8.3.3 Buffers and Concurrency

Buffers are especially important in Emacs as much of text editing occurs through them. There can only be one active (or current) buffer at one time, with threads having their own current buffer, inherited from their parent thread upon creation [22, p. 28.2]. The macro `current_buffer` (from `src/thread.h` [3]) is used in the core to refer to the current buffer and points to `m_current_buffer` of the current threads struct. Whenever a thread acquires the GIL, it calls `post_acquire_global_lock` (in `src/tread.c` [3]), which calls `set_buffer_internal_2` (from `src/buffer.c` [3]) to setup the buffers different attributes, such as local variables, the undo list and markers.

```
static struct print_buffer print_buffer;
```

Listing 45: Shared variable `print_buffer` from `src/print.c` [3].

A print buffer `print_buffer` (see listing 45 defined in `src/print.c` [3]) is a buffer used to store the intermediary text created during the process of printing, such as to a buffer. The variable `print_buffer` is statically allocated in the `BSS` segment during compile time and is thus shared between the threads.

A future removal of the GIL would need to move the print buffer to thread-local storage or adding some form of synchronization to avoid races during IO. Mutex locks could be used around the critical sections of `src/print.c` [3] where the print buffer is used.

8.3.4 Atomicity

Currently `block_input` (from `src/blockinput.h` [3]) is used internally to create a critical sections that prevents the signal handler from being invoked. This is achieved with the help of the semaphore `interrupt_input_blocked` (defined in `src/keyboard.c` [3]) and is used to ensure that the signal handler won't be called whenever any non-reentrant functions (that can't be interrupted) are used, such as `malloc`.

As *non-reentrant* functions are thread-unsafe, a non-blocked signal handler also implies the possibility that Emacs might switch to the main thread and execute Lisp [22, ch.54.4]. Because of this, any thread-level critical section must also implicitly block the signal handler. We propose a unified synchronization primitive such as a mutex lock that expands upon `block_input` to allow for the creation of more generalized critical sections where the signal handler is blocked.

8.3.5 Atomic Change Groups

Emacs Lisp has a feature called **Atomic Change Groups** [4, ch.33.33] that is similar to database transactions and is used to implement the Emacs undo system. These groups are used to group set of Lisp forms that must occur together. The groups can then be activated, applying the changes onto the buffer, or canceled, applying none of the changes - such as when any non-local exit or error occurs [4, ch.33.33].

As Emacs was written under the assumption of single-threaded execution, the atomic change groups does not provide any isolation between threads. Other Lisp threads can access the intermediate buffer states while a atomic change group is being constructed or activated. As a result making the execution of the groups non-linearizable and thus non-atomic in a multi-threaded environment.

We suggest protecting the internal buffer structures using synchronization primitives and adding the ability for threads to lock write access to buffers. Then the atomic change groups can be modified such that they execute on a thread private copy of the buffer, which is swapped out for the original upon successful completion. This would make the groups appear linearizable (to the other threads), preserving their atomicity.

Moreover, we propose the need for atomic arithmetic operations made available to Lisp so that integers can be shared between threads without being locked by a mutex lock. Similarly to how `src/systhread.c/h` [3] ensures that Emacs can use threads on different operating systems through a common API that abstracts the OS differences. We propose a new component `src/sysatomic.c/h` [3] that implements atomic arithmetic and atomic load and store. This component could then be used to implement atomic subroutines for Lisp.

8.3.6 The Interpreter Environment

The Emacs interpreter uses the internal argument list `Vinternal_interpreter_environment` to store its lexical environment. Since the variable is shared between threads it has to be pushed to the special bindings stack and then be unwound every time Emacs switches threads, a blocking process which depends on the GIL. Moving the interpreter environment to TLS (Thread Local Storage) could be done by giving each thread its own lexical environment.

This wouldn't need much changes to the architecture of the Lisp interpretator and could be as simple as to extend the `thread_state` to include a new `Lisp_Object` such as `current_thread->lexical_environment`. All occurrences of `Vinternal_interpreter_environment` and `Qinternal_interpreter_environment` would then be substituted with a reference to the new thread-local Lisp Object. The function `make-thread` would then be modified such that it inherits a copy of the parent-threads environment, preventing changes to the new threads environment from propagating to the parent.

```
DEFUN ("make-thread", Fmake_thread,
    Smake_thread, 1, 3, 0,
// ...
  new_thread->lexical_environment =
    Fcopy_sequence(
      current_thread->lexical_environment
  );
// ...
}
```

Listing 46: Inheriting thread local lexical environment in Commercial Emacs.

The bindings can then be garbage collected after the thread dies. This method would prevent the threads from sharing variables using the lexical state, which then must be dynamically bound. Another solution is to give the inheriting thread read-only access to the active lexical environment, only adding the variables to the local lexical environment whenever the current threads tries to change the values. This would require that the current thread searches both the thread local lexical environment followed by the parent-threads lexical environment, only copying variables from the parent environment into the local environment when a value has changed.

```
DEFUN ("defvar", Fdefvar, Sdefvar, 1,
    UNEVALLED, 0,
...

// Commercial Emacs

  if (!NILP
      (current_thread->lexical_environment)
...
    current_thread->lexical_environment =
      Fcons (sym,
          current_thread->lexical_environment);

// GNU Emacs

  if (!NILP
      (Vinternal_interpreter_environment)
...
    Vinternal_interpreter_environment =
      Fcons (sym,
          Vinternal_interpreter_environment);
}
```

Listing 47: Comparison between interpreter environment in Commercial Emacs and GNU Emacs.

This is similar to the GNU Emacs fork Commercial Emacs, where the lexical environment has been added to TLS and is inherited upon creation, see listing 46. The difference between using the interpreter environment in GNU Emacs [3] and in Commercial Emacs [2] can be seen in listing 47 which illustrates the difference between how Commercial Emacs and GNU Emacs access the interpreter environment, where the code is taken from the definition of the special-form Lisp Subroutine `defvar` in `src/eval.c` [2, 3].

8.3.7 Using OBArrays

An ObArray is a data type used by the Lisp reader to map the names of symbols to their values. They function similarly to hash tables and consists of a fixed-size vector of buckets, where each bucked points to a linked list of `Lisp_Symbol`'s whose name hash to that index. *Interning* a symbol involves hashing the symbols name and scanning the bucket's list. If no matches are found a new `Lisp_Symbol` is allocated and inserted as the buckets `next` pointer (see `struct Lisp_Symbol` in `src/lisp.h`[2]).

The Emacs fork *Commercial Emacs* [2] uses a combination of a thread-local lexical environment and obarray to handle thread local let bindings. Commercial Emacs extends the native Lisp threads `src/thread.c/h`[2] by adding a thread-local lexical environment `thread_state->lexical_environment` that is inherited upon thread creation and the ability to run threads outside of the GIL when `thread_state->cooperative` is false. The fork also gives threads their own ObArray `thread_state->obarray` [2] which threads use to create thread-private, interned symbols [22, ch.9.3].

Commercial Emacs has changed much of the GNU Emacs codebase, including a reworked memory allocator built on a Red Black Tree (RBT), see (`src/mem_note.c` [2]) to store blocks of `Lisp_Object`'s. A global RBT contains all allocated blocks `mem_nil`[2] with each thread state given their own branch of the RBT to allocate memory from called `thread_state->m_mem_root`[2]. This allows threads to merge their memory allocations back into the main thread upon completion, see `reap_thread_allocations` defined in `src/alloc.c`. Each thread allocates to their thread-local lexical environment, which is used in place of `Vinternal_interpreter_environment`.

Commercial Emacs further gives each thread its own private *exception stack* composed of an array of `jmp_buf` that is used instead of the global `handlerlist`. Moreover, looking up a symbol value now always tries the current threads ObArray before looking in the initial ObArray, which can be seen in `find_symbol_value` from `src/data.c` or in `thread_symbol` from `src/data.c` [2]. This allows threads to shadow the value of global symbols by defining them in the thread-local ObArray.

In vanilla GNU Emacs, function calls, variable lookup and symbol interning all share `Vinternal_interpreter_environment` and one single global ObArray `Vobarray`. This is not a problem since the GIL prevents data races. By giving threads their own ObArray and private memory-space, Commercial Emacs allows the removal of the GIL while preventing collisions whenever threads concurrently intern symbols that share the same name. This is due to each thread binding the new symbol in their local ObArray, thus allowing both threads to bind the same name to different Lisp Objects. When a thread then dies, a semaphore is used to protect the allocator while the allocated memory is merged back into the main thread. Since threads allocate to TLS, they can be garbage collected without a GIL locking down the interpreter.

8.3.8 Thread Implementation

The variable for tracking the currently active thread (`current_thread` from `src/thread.h` [3]) used to index the current threads state is shared between threads. As the `current_thread` variable only has one value at a time, it will be unable to track multiple simultaneous threads. If the GIL is removed, its value would be the same on each concurrent OS-thread, leading to each OS thread referring to the same Lisp / Emacs thread. Commercial Emacs solves this in `src/thread.h` [2] by defining `current_thread` using the `__thread` keyword, which specifies that it should be stored in TLS.

```c
// thread.h
#include <assert.h>

// [...]
static sys_mutex_t thread_list_lock;
// Return thread-state of active thread
thread_state *current_thread_state()
{
  thread_state *cur;
  int tid;

  // Current OS Thread ID
  tid = sys_thread_self();

  sys_mutex_lock (&thread_list_lock);

  // Find thread (if exists)
  for (cur = all_threads;
       cur->next_thread;
       cur = cur->next_thread)

    // Check if thread matches
    if (cur->thread_id == tid) {
        sys_mutex_unlock (&thread_list_lock);
        return cur;
    }

  // Should not get here
  assert (1);
}
// Define macro to make it look like a
    variable
#define current_thread current_thread_state()
```

Listing 48: Indexing Threads by ID.

We have provided one inefficient solution to this problem where we search for the id of the currently running OS thread in the linked list of active threads, allow the current thread to refer to itself and its state without using the shared `current_thread` variable. This can then be abstracted using macros and indirection. We have provided the function `current_thread_state` defined in listing 48 attempts to provide a substitute for the variable `current_thread` while creating a abstraction from the function through the

`current_thread` macro.

The function gets the ID of the current OS thread using `sys_thread_self` followed by iterating through the linked list of threads looking for the thread state with a matching thread ID (`thread_id` of `thread_state` in `src/thread.h` [3]). Once a `thread_id` matches the OS thread ID, the thread is returned. A mutex lock `thread_list_lock` is used during the iteration since `thread_list` could have been modified during the iteration. This mutex lock should be added around each access to `thread_all`, primarily to protect against concurrent thread creation and deletion. An assertion is raised if the thread id is not found in the linked list. This solution is inefficient and much slower then the current solution which use a pointer as each access needs to iterate through the linked list.

8.3.9 Preemptive Scheduling

One of the problems with the current cooperative threading model is that the responsibility is on the programmer to write responsive and bug-free code, which easily leads to serious bugs that may freeze or slow down Emacs. One solution to this is to use a preemptive thread scheduler, which would handle the task of switching threads, automatically preempting running threads according to a scheduling algorithm. To minimize latency, a fair scheduling algorithm such as Round-Robin (RR) could be used, which would ensure that each thread gets a fair (equal) execution time.

A preemptive RR scheduler would fit the requirements of a real-time text editor, ensuring each thread gets a fair (equal) share of CPU-time, preventing Emacs from freezing on long running threads. However, a preemptive scheduler would slow Emacs down in proportion to the amount of running thread as Emacs needs to wait on every other thread after each time-slice.

The Erlang virtual machine, BEAM VM [19] uses a work-slice instead of a time-slice. In Emacs Lisp we can use a commonly invoked internal mechanism such as function applications to measure work. Two of the most commonly used internal functions for function applications are `funcall_subr`, which is used to call subroutines and `funcall_lambda`, which is used to call Lisp functions (see `src/eval.c` [3]).

A work-sharing RR scheduler would use work-slots w instead of time-slots. The Work-slots w represents the maximum amount of continuous work allocated to each thread. A statement is then put in the beginning of the `funcall_lambda` and `funcall_subr` (from `src/eval.c` [3]) that increments the work counter and checks if the thread has finished its work-quantum. This can be determined by testing if the work counter c modulo the work-slot size w is zero.

$$c \equiv 0 (\text{mod } w)$$

When the condition holds true the thread has reached the end of the work-quantum and the scheduler will preemptively switch the next in the thread in the RR queue. Before switching threads, the execution context of the interpreter is stored to allow the thread to later resume its execution at $c + 1$.

The C function `maybe_quit` allows Emacs to respond to the user pressing `C-g` to allowing the user to stop Lisp execution. The function is used in places where long executions may take place [22, Ch E.7]. Since the scheduler would try to minimize threads running for too long, the scheduler could context switch close to `maybe_quit` instead of directly after incrementing the work counter in `funcall_lambda` and `funcall_subr`. Priority queues could also be introduced to optimize I/O-bound threads.

```
funcall_subr (...)
{
  eassume (numargs >= 0);
  thread_state *curthr = current_thread;

  // Check if thread work slice is out
  if (++curthr->counter %
      scheduler->work_slot == 0)
    {
      // Store current execution context
      store_execution_state   (curthr);
      // Switch to another thread
      schedule_thread_rr      (curthr);
      // Restore back to the this threads
          execution context
      restore_execution_state (curthr);

    }
  // [...]
}
```

Listing 49: The internal `funcall_subr` function (from `src/eval.c` [3]) modified for preemptive scheduling.

We provide theoretical modifications to `funcall_subr` for a preemptive scheduler in listing 49. The listing illustrates how we first increment the work counter followed by testing if the current threads work-slice has been exhausted. If it has, we save the current threads execution context and schedule the next thread, once control then is handed back the previous execution context is restored. For example, suppose thread A is given a work-slot of $w = 10k$ operations, but its task requires total of $15k$ operations to fully complete. The scheduler will preemptively switch to the next queued thread after the initial work-slice of $10k$ has elapsed ($10k \equiv 0 \pmod{10k}$). When the scheduler eventually hands back control to thread A, it will complete its remaining $15k - 10k = 5k$ operations without triggering another preemption.

9 Future Work

This thesis establishes the foundation for a basic architectural documentation of the Emacs Core (TO1) as well as an investigation into the state of concurrency in Emacs Lisp, introducing a set of concurrency related Lisp libraries followed by a summary of the architectural restrictions present in the Emacs core paired with a summary of improvements (TO2).

The first objective in this thesis (TO1) constitutes the foundation for a primitive documentation of the Emacs core, including the prerequisites required to remain accessible to new developers. The depth and scope of the documentation should be expanded upon in the future to provide a more detailed and faithful representation of the Emacs Core at an architectural level of abstraction.

Some topics that can be expanded upon in future work include documenting the differences between Emacs running in the interactive and non-interactive modes, multi-threaded bytecode interpretation, customizing the `temacs` build process, providing greater insight into the Lisp Environment and the inner workings of the interpreter and its state as well as adding more in depth description of the other subsystems that were left out together with their relation to the other internals, a good example being `lisp/xdisp.c` [3].

The second objective of this thesis (TO2) introduces the idea of using a preemptive thread scheduler instead of the cooperative threading model. A preemptive scheduler would offload the responsibility of synchronizing threads from the programmer, reducing the amount of bugs and performance issues stemming from badly written cooperatively-threaded code. A simple, theoretical preemptive thread scheduler is proposed that uses the Round Robin scheduling algorithm to give each thread a fair amount of work, with work measured in function applications. A preemptive scheduler works together with the GIL and would prevent the issue where badly used cooperative threads freezing Emacs by never giving up control. Priority queues could further improve the performance of I/O bound threads.

Thread safety in the Emacs Core requires revised synchronization across several critical subsystems; execution context management in `src/eval.c`, buffer synchronization in `src/buffer.c/h`, atomic filesystem operations in `src/fileio.c`, timer data race prevention `atimer.c/h` and

`src/keyboard.c/h` and atomicity in sequence operations `src/fns.c` [3].

The Emacs NG fork includes modifications that extends the dynamic modules API to allow developers greater access to the Emacs core, increasing the potential of whats possible using dynamic modules. The modifications primarily give modules greater access to the internal state of the Emacs core. Emacs NG improves the performance of modules that directly access the buffer text contents by providing direct access to the text contents of buffers, something which previously required the module copy the whole buffer [11]. Further research into increasing the capabilities of the module API could have massive implications on the performance of Emacs Lisp libraries. Moreover, research that quantifies the overhead of extending Emacs Lisp through dynamic modules versus by directly extending core would be interesting, especially in relation to different models for sandboxing the modules.

While discussion of the Lisp threads remains inactive in the GNU Emacs developer mailing list (as of Q1 2025) [7], it is reasonable to expect further development on the Lisp threads in the long term future, since the manual mentions the move away from a cooperative threading model [22, ch.39].

Development on Commercial Emacs [2] has made significant progress on the topics of preemptive scheduling and multi-threading and remains (as of April 2025) active on Github, however, development on multi-threading has slowed down with the last commits mentioning multi-threading being submitted in Jan 2025 (as of Mar 2025) in the file `src/thread.c` [2].

Development on the Emacs fork **Guile Emacs** has restarted (see [35]). Guile Emacs is a fork of GNU Emacs that integrates Guile Scheme with the intention of replacing the Lisp interpreter with one written in Scheme. Using Scheme Lisp to extend Emacs is an idea that explicitly has been embraced by RMS and the Emacs development community. The end goal of the fork being to eventually rewrite the Emacs Lisp engine fully in Scheme, allowing scheme to customize and extend the editor (in addition to by using Emacs Lisp). Looking through the GIT repository highlights some recent activity, but the project itself is still far from finished.

10 Discussion

This thesis is divided in two parts according to the thesis objectives: The **first part** (TO1) contains the architectural documentation, which primarily focuses on parts deemed relevant to concurrency such as threads, subprocesses, timers, non-local exits and stacks. The **second part** (TO2) describes the concurrency enabled in the Emacs core.

An introduction to the Free Software Philosophy and the history of Emacs is included before the documentation as the development and design of GNU Emacs is directly tied to this philosophical framework. To keep the documentation accessible, this thesis includes introductory prerequisites that describe both a generalized description of Emacs-type editors (see section 4) and a short introduction to the Emacs Lisp language. An improvement would be to make the description of Emacs-type editors less generalized and more similar to that of GNU Emacs, which would improve the transition into the primary documentation.

A documentation according to TO1 is given, centering around the Emacs command loop and the Emacs Lisp environment together with a lower level description of internal components and related topics, such as the handling of user input and redisplay.

The Emacs Lisp environment that defines the native Lisp fundamentals, the execution of Emacs Lisp and the internal state is especially important to concurrency. Therefore the thesis includes documentation regarding the implementation of various fundamental libraries and features. These were chosen according to their relevance to the concurrent capabilities of Emacs and includes *processes, threads, timers, non-local exits, state and memory handling*, together with the *interpreter environment*.

The documentation given in this thesis provides a skeleton for future documentation to built upon and which can be expanded both in coverage and detail. Many parts of the documentation explores the internal state of Emacs and calls to blocking components in order to illustrate why the GIL is needed.

According to second thesis objective (TO2) a selection of Lisp packages is then explained and described, going into more detail regarding their implementation of various forms of concurrency. This is followed by a summary of various concurrency related issues, both found online and discovered while researching the architecture. The issues are paired with solutions where possible.

The Lisp interpreter is then investigated along with the components that limit the possibilities for greater forms of Lisp-level concurrency, such as multi-threading, including big limiting concepts such as the GIL.

It is demonstrated how the GIL allows for the current implementation of Lisp-threads, and compensates for the lack of thread safety in the architecture of Emacs by locking the interpreter to the current thread.

However, the implementation of the threads lacks any practical real world usage since the GIL prevents multiple Lisp threads from executing at the same time while the single threaded architecture and the global state of the editor makes using the threads exceptionally hard.

This thesis examines a selection of concurrency related Lisp libraries highlighting the theoretical concepts used to achieve concurrency, which in our case mostly are related to the concepts of futures / promises.

Several Lisp packages implement ways to achieve concurrency without the native Lisp threads, such as by spawning external Emacs processes, using timers and with generators.

The implementation of the Lisp libraries is analyzed and we conclude that there are three primary Emacs Lisp features that commonly is used to achieve concurrency.

I **Idle Timers**: *"Run concurrent tasks when the user is idle"*

II **Subprocesses**: *"Run parallel tasks in Emacs worker processes"*

III **Generators**: *"Yield and resume execution of concurrent task"*

Idle Timers are used to schedule the execution of a Lisp function to a future moment when the user is idle, thus minimizing the users awareness of the Lisp evaluation. We demonstrated through our examination of the set of Lisp libraries that idle timers is one of the more common features used.

Since idle timers are executed when Emacs has waited for user input a set amount of time, they allow the programmer to defer execution of Lisp to the future. Using a idle timer set to execute after 0 seconds of time idle will effectively schedule it to execute as soon as possible. In the selection of Lisp libraries examined in this thesis, idle timers was one of the most common methods abstracted upon.

Another observation was that another way to implement concurrency is by using non-interactive Emacs processes as workers. We saw that these worker processes often were used to either, execute a Lisp program and die when the result is produced, or start a event loop that repeatedly listened and executed code from the parent process. This allows the libraries to bypass the single threaded design of Emacs and achieve parallel Lisp execution as to the workers run in different, isolated memory spaces and communicate through IPC (such as file descriptors).

This thesis demonstrated a faulty program where a condition variables could be used to indefinitely freeze Emacs, requiring the user to kill and restart the program. We include a simple solution that prevents programmers from making this mistake.

The thesis demonstrates that many of the issues related to Lisp threads are directly related to the *cooperative threading model* used, leaving responsibility to the programmer to synchronize threads. We found that invalid or buggy code can result in Emacs permanently freezing, and the user any unsaved work. The thesis introduces the idea of using a simple preemptive scheduler to remove responsibility from the programmer and phase out the cooperative threading model. The thesis further builds on top of this idea by providing the implementation of a scheduler.

The analysis demonstrates how the removal of the Global Interpreter Lock requires large fundamental changes on throughout the whole core and most of its internal components. The next steps towards a thread-safe core would include a complete redesign of the core or the step-wise addition of *thread synchronization* in components such as the garbage collector, I/O. Furthermore, the introduction of *atomic operations* in the most fundamental data structures and functions of Emacs Lisp such as the `cons` construct is necessary, as well as redesigning components to rely on thread-private instead of global state. There are contemporary forks of GNU Emacs that experiment with introducing greater forms of concurrency, such as **Emacs NG** and **Commercial Emacs**, introducing experimental multi-threading and thread schedulers.

References

[1] Ben Wing et al. *XEmacs 21.5 Internals*. https://www.xemacs.org/Documentation/21.5/html/internals_toc.html (accessed October 2024) (cit. on p. 4).

[2] dickmao et al. *Commercial Emacs (Source Code)*. https://github.com/commercial-emacs/commercial-emacs (accessed in May 2024). Github repository. 2024 (cit. on pp. 4, 5, 53, 54, 57).

[3] Richard M. Stallman et al. *GNU Emacs 29.3 source code*. Github Repository, Commit ae8f815. 2024. URL: https://github.com/emacs-mirror/emacs (cit. on pp. 1, 2, 4, 13, 16, 18–38, 40, 41, 44, 45, 48, 50–57, 61, 62, 66, 67).

[4] Richard M. Stallman et al. *GNU Emacs Manual*. 29.3. Free Software Foundation, Inc. 2024 (cit. on pp. 1, 16, 18, 23, 35, 51).

[5] chuntaro. *Async/Await 1.1 (Emacs package)*. https://github.com/chuntaro/emacs-async-await (accessed October 2024. 2024 (cit. on p. 42).

[6] chuntaro. *Promises/A+ (Emacs package)*. https://github.com/then/promise commit cec51feb5f957e8febe6325335cf57dc2db6be30. 2021 (cit. on p. 42).

[7] Emacs Developers. *The GNU Emacs Developer Mailing List*. mailto:emacs-devel@gnu.org (email address). 2024 (cit. on pp. 2, 57).

[8] Emacs NG Devs. *Emacs NG (Source code)*. https://github.com/emacs-ng/emacs-ng (Github repository). Release version 0.0.2b. 2024 (cit. on pp. 4, 5).

[9] Emacs NG Devs. *Emacs NG: A new approach to Emacs - Advanced Features*. https://emacs-ng.github.io/emacs-ng/js/adv-features/ (accessed July 2024). Official Emacs NG homepage. 2024 (cit. on p. 5).

[10] Emacs NG Devs. *Emacs NG: A new approach to Emacs - Architecture*. https://emacs-ng.github.io/emacs-ng/js/architecture/ (accessed July 2024). Official Emacs NG homepage. 2024 (cit. on p. 5).

[11] Emacs NG Devs. *Emacs NG: A new approach to Emacs - Dynamic Modules*. https://emacs-ng.github.io/emacs-ng/ng-module/ (accessed Mars 2025). Official Emacs NG homepage. 2025 (cit. on p. 57).

[12] dickmao. *Emacs Multithreading: How Hard Can It Be?* https://www.youtube.com/watch?v=7H1Pe9HkJ7I (accessed in May 2024). Video. 2024 (cit. on p. 5).

[13] dickmao. *Multithreaded Emacs*. https://www.youtube.com/watch?v=Ne6ZpeEop_4 (accessed in May 2024). Video. 2024 (cit. on p. 5).

[14] Edsger W. Dijkstra. "Letters to the editor: go to statement considered harmful". In: *Commun. ACM* 11.3 (Mar. 1968), pp. 147–148. ISSN: 0001-0782. DOI: 10.1145/362929.362947. URL: https://doi.org/10.1145/362929.362947 (cit. on p. 1).

[15] Martin Edström. *asyncloop (Emacs package)*. https://github.com/meedstrom/asyncloop (version 1.1). 2024 (cit. on p. 44).

[16] Ben Wing email:ben@xemacs.org. *A Tour of XEmacs*. https://www.xemacs.org/Architecting-XEmacs/xemacs-tour.html (accessed October 2024). 2017 (cit. on p. 4).

[17] Craig A. Finseth. *The Craft of Text Editing –or– Emacs for the Modern World*. en. Springer-Verlag and Co., 1999. ISBN: 978-1411682979 (cit. on pp. 2, 6, 7).

[18] Inc Free Software Foundation. "What is Free Software?" In: *GNU Philosophy* (2024). http://www.gnu.org/philosophy/free-sw.en.htmll (accessed May 2024) (cit. on p. 4).

[19] Javier Garcia. *Elixir: Understanding the concurrency model*. https://manzanit0.github.io/elixir/2019/09/29/elixir-concurrency.html (accessed in 2024). 2019 (cit. on p. 55).

[20] GNU. *GNU Emacs (Homepage)*. http://www.gnu.org/software/emacs/ (accessed May 2024). 2024 (cit. on p. 2).

[21] GNU. *The GNU Emacs FAQ*. 29.3. Free Software Foundation, Inc. 2024 (cit. on pp. 1, 3, 16).

[22] GNU. *The GNU Emacs Lisp Reference Manual*. 29.3. Free Software Foundation, Inc. GNU, 2024 (cit. on pp. 1, 9, 11–16, 19, 21, 23, 26, 30, 31, 35, 36, 40, 42, 48–51, 53, 55, 57).

[23] James Gosling. "A redisplay algorithm". In: *ACM SIGOA Newsletter* 2.1–2 (Apr. 1981), pp. 123–129. ISSN: 0737-819X. DOI: 10.1145/1159890.806463. URL: https://doi.org/10.1145/1159890.806463 (cit. on p. 7).

[24] Bil Lewis and Daniel J. Berg. *Threads primer: a guide to multithreaded programming*. Prentice Hall Press, 1995. ISBN: 0134436989 (cit. on p. 50).

[25] Alexander Miller. *pfuture.el (Emacs package)*. https://github.com/Alexander-Miller/pfuture (version 1.10.3). 2022 (cit. on pp. 42, 43).

[26] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. "Concurrency Among Strangers". In: *Trustworthy Global Computing*. Ed. by Rocco De Nicola and Davide Sangiorgi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 195–229. ISBN: 978-3-540-31483-7 (cit. on p. 41).

[27] Mozilla. *MVC*. https://developer.mozilla.org/en-US/docs/Glossary/MVC (accessed October 2024). 2024 (cit. on p. 9).

[28] Dan Murphy. "The Beginnings of TECO". In: *IEEE Annals of the History of Computing* 31.4 (2009), pp. 110–115. DOI: 10.1109/MAHC.2009.127 (cit. on p. 3).

[29] Masashi Sakurai. *deferred.el (Emacs package)*. https://github.com/kiwanami/emacs-deferred (version 0.5.1). 2017 (cit. on pp. 42, 44).

[30] D. Spinellis and G. Gousios. *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*. O'Reilly Media, 2009. ISBN: 9780596554392. URL: https://books.google.se/books?id=h34pwy005nYC (cit. on p. 17).

[31] Richard M. Stallman. *GNU Emacs 16.56 source code*. Archived source code. 1985 (cit. on p. 1).

[32] Richard M. Stallman. *The Origin of XEmacs*. https://stallman.org/articles/xemacs.origin (accessed October 2024). 2024 (cit. on p. 4).

[33] Richard M. Stallman. *What I'd like to see in Emacs (talk)*. https://emacsconf.org/2022/talks/rms/ (accessed May 2024). Delivered on EmacsConf 2022. 2022 (cit. on pp. 3–5).

[34] Stephen J. Turnbull. *XEmacs vs. GNU Emacs*. https://www.xemacs.org/About/XEmacsVsGNUemacs.html (accessed October 2024). 2017 (cit. on p. 4).

[35] Various. *Guile Emacs (Emacs Fork)*. https://codeberg.org/guile-emacs/guile-emacs. Mar. 2022 (cit. on p. 57).

[36] Colin Watson. *Linux Manual Pages*. Fedora Linux 39, Version 6.05. 2023 (cit. on p. 32).

[37] Chris Wellons. *An Async / Await Library for Emacs Lisp*. https://nullprogram.com/blog/2019/03/10/ (blog). 2019 (cit. on pp. 41, 42).

[38] Chris Wellons. *emacs-aio (Emacs package)*. https://github.com/skeeto/emacs-aio (version 1.0. 2019 (cit. on pp. 42–44).

[39] John Wiegley. *emacs-async (Emacs package)*. https://github.com/jwiegley/emacs-async commit d604187c3c8a3290e78c00b4532b83d8d908b5c3. 2023 (cit. on pp. 2, 35, 44–46).

[40] Emacs Wiki. *Hackers Guide*. https://www.emacswiki.org/emacs/HackerGuide (accessed June 2024). 2024 (cit. on p. 2).

[41] Emacs Wiki. *No Threading*. https://www.emacswiki.org/emacs/NoThreading (accessed May 2024). 2024 (cit. on p. 1).

[42] Sam Williams, ed. *Free as in Freedom*. en. Sebastopol, CA: O'Reilly Media, 2002. ISBN: 978-0596002879 (cit. on pp. 3, 4).

A Memory Allocation in C

The memory layout for C processes on Linux is used since the Emacs core when compiled for Linux is written in C. The primary sections of the process memory layout, are the `STACK`, `HEAP` and the `DEC` segments. If an uninitialized pointer is created it is stored in the `BSS` section of `DEC`, if a initialized pointer is used its stored in the `DATA` section of `DEC`, constants and machine code is stored in the `TEXT` section of `DEC`. Memory allocated using `malloc` or `alloc` are allocated on the heap and function parameters and local variables are stored on the `STACK`,

Threads can all access the pre-allocated `DEC` and the dynamically runtime allocated `HEAP` section, with the Stack being the only thread-private section in the process memory layout. The sections of the `DEC` are allocated during compile time and loaded from the binary.

The `DATA` section contains global and initialized static variables, such as (in C), `int x = 10;` (global scope), `static y = 30;`, `global z = 10;`. The `BSS` section contains uninitialized global and static variables. where no data was initialized to the variable by the programmer. Each variable is initialized to 0 at compile-time. Examples: `static int x;`, `int x;` (global scope). Finally the `TEXT` section stores the program (Machine Code) and its constants.

B Notes on Studying the Emacs Core

GNU Global is a useful tool which can aid with finding references to and the definition of symbols, with the programs `gtags` for generating tag files and `global` for looking up symbols. A program called `ctags` can also be used.

```
global --result=grep --color=always\
--path-style=shorter \
--from-here=218:keyboard.c\
-- quit_char
```

Listing 50: References to `quit_char` defined on `src/line 218 in` keyboard.c [3].

Listing 50 illustrates using GNU Global for finding all references to `quit_char` which is defined on line `218` in `src/keyboard.c` [3]. The output includes the file name, line number and line contents for each reference separated by a colon.

```
global --result=grep --color=always\
--path-style=shorter\
command_loop_1
```

Listing 51: Finding location where the function `command_loop_1` is defined.

Listing 51 illustrates using GNU Global for finding the place of definition for the function `command_loop_1`.

A Emacs package called `ggtags` [14] can be used as a frontend to GNU Global. The package allows the programmer to look up and jump to the definition of symbols and list references where the symbol is used and jump between them. This is very convinient when studying the Emacs source code since it correctly detects definitions of Lisp subroutines and symbols which are defined in the code using C macros such as `DEFUN`.

[14] The package `ggtags` is Written and maintained by Leo Liu and available in the "GNU" Emacs repository.

B.1 Studying The GNU Emacs Source Code

The extensive documentation system of Emacs provides a good resource for learning about the implementation of GNU Emacs and makes it possible to find and jump to the exact location of function and variable definitions. Emacs is per default unable to find the location of Lisp functions and variables which are defined in C as GNU Emacs often is distributed as a binary without the C source code. This is fixed by retrieving the GNU Emacs source code and setting the Lisp variable `find-function-C-source-directory` to path of the `src/` [3] sub-directory of the source code (see listing 52). The help command will include references to the C source code, once this has been set.

```
(setq find-function-C-source-directory
  "path/to/emacs/src")
```

Listing 52: Allowing `help-mode` to lookup definitions from the C source code.

B.2 Introduction to Doxygen

Doxygen is a documentation system which is available for C, allowing the user to generate HTML documentation for source code. Doxygen can be configured to generate dependency graphs and caller and callee graphs. This can aid in understanding the relationship between different functions and files. Some modifications to the doxygen configuration file are needed to make Doxygen generate graphs. listing 53 displays some flags in the Doxygen configuration that needs to be enabled to enable graph generation.

```
ENABLE_PREPROCESSING = YES
HAVE_DOT             = YES
INCLUDE_GRAPH        = YES
CALL_GRAPH           = YES
CALLER_GRAPH         = YES
COLLABORATION_GRAPH  = YES
```

Listing 53: Generating caller and callee graphs in `doxygen`.

```
DOT_GRAPH_MAX_NODES  = 64
MAX_DOT_GRAPH_DEPTH  = 256
```

Listing 54: Specifying depth of Doxygen graphs.

Because the graphs are stored as images, it may be appropriate to limit the graph size and depth if caller and callee graphs are enabled due to the large size of the Emacs C source code. This can be achieved through changing the options seen in listing 54 to smaller values. While testing different Doxygen configurations without any limit for the graph depth combined with a high amount of nodes (max nodes), our documentation was able to reach a size of 80 Gb. However, limiting the graph depth and max nodes to values close to those in listing 54 our documentation size was around 30-40 Gb.

```
DOT_CLEANUP = NO
```

Listing 55: Configure Doxygen to keep intermediate graphviz source files.

The intermediary graphviz files generated by Doxygen when "dot" is enabled can be kept through disabling the cleanup of dot files (as seen in Listing 55).

B.3 Coding Conventions and Macros

This section includes information regarding coding conventions and internal macros needed to understand the internal code.

Forwarded Lisp variables are defined in C using the macros, `DEFVAR_LISP`, `DEFVAR_KBOARD`, `DEFVAR_LISP_NOPRO`, `DEFVAR_BOOL` and `DEFVAR_INT` and variables bound to buffers defined using; `DEFVAR_PER_BUFFER`. The internal (C) variable name of Lisp variables is prefixed by "V" except integer, keymap and boolean variables. Symbols are defined using `DEFSYM`, with the internal name prefixed by "Q". Functions are defined using `DEFUN`, with the internal function name prefixed by "F".

Components usually have a function prefixed by the `syms_of_` which defines all of the Lisp symbols and variables. The function called `staticpro` is used to protect Lisp objects from garbage collection. Only C variables or function-names includes underscores, and only Lisp functions and variables include a dash since C will interpret it as the subtraction operation.

C A Quick Introduction to Elisp

Readers who are unfamiliar with Lisp can use the following short examples of Emacs Lisp as a reference or brief introduction. To start the Emacs Lisp interpreter (REPL) included in GNU Emacs, press `M-x` (`<Alt>` + `x`) inside of Emacs, then type `ielm` and press `<Enter>`.

The following subjects are covered in this quick introduction.

I Defining variables (see listing 56)

II Defining functions (see listing 57)

III Defining commands (see listing 58)

IV Control structures (see listing 59)

V Booleans (see listing 60)

VI Loop constructs (see listing 61)

VII Scoped local variables (see listing 62)

VIII Lists and pairs (see listings 63 and 64)

IX Buffers (see listing 65)

C.1 Defining Variables

Listing 56 demonstrates the most common way variables are set in Elisp. It first defines a global dynamically bound variable `Y` as the value `10`, then it sets the symbol `X` to list `'(1 2 3)`. The function `defvar` is used to define a global (dynamic) variable together with a initial value and optional documentation, while `setq` can be used to define new variables, it is usually used to set or update the values of variables. The function `defvar-local` defines a buffer local variable which only is bound inside of a certain buffer. The function `set` takes a symbol in quoted form as its first argument, `setf` is used to set the value of a *location*, such as the 0'th value of the list `x` to `-1`.

```
(defvar-local a 123) ;;     a := 123
(defvar Y 10)        ;;     Y := 10
(setq x '(1 2 3))    ;;     x := (1 2 3)
(set 'z 20)          ;;     z := 20
(setf (nth 0 x) -1)  ;; x[0] := -1
;; ==> x = '(-1 2 3)
```

Listing 56: Defining variables in Elisp.

C.2 Functions

Listing 57 shows how to define functions in Emacs Lisp where the `defun` function is used to define named functions, the `lambda` function creates a anonymous $\lambda -$ function. If the a lambda function is bound to a variable it is called through functions such as `funcall` or `apply`. Lambda functions can also be called through lambda applications, which would look something similar to $((\lambda x.[x \cdot x])12)$ in Lambda Calculus.

```
;; Define function `SQR' as the square its
    argument, `X'
(defun sqr (x)  (* x x))

;; Lambda expression equivilent to `SQR'
(setq f (lambda (x) (* x x)))

;; Calling the defined functions
(sqr 10)                ;; ==> 100
(funcall f 11)          ;; ==> 121

;; Direct application of a lambda expression
((lambda (x) (* x x)) 12) ;; ==> 144
```

Listing 57: Defining functions in Elisp.

C.3 Defining Commands

Listing 58 describes the definition of a interactive function or command. The `interactive` keyword allows a function to be called from the minibuffer (using `M-x`). Interactive Emacs sessions are sessions where the user interacts with Emacs, an example of a non-interactive Emacs session is when Emacs is called using the `-batch` argument where Emacs doesn't enter the command event loop. A function has to be a commands to be able to be bound to a keybinding.

```
;; Define command `MY-COMMAND' to display a
    message
(defun my-command ()
(interactive)
(message "You have pressed the special
    keys!"))

;; Bind `MY-COMMAND' to a keybinding
(global-set-key (kbd "C-c m") 'my-command)
```

Listing 58: Interactive functions in Elisp.

C.4 Control Structures

Listing 59 illustrates the basic control structure `if` in Emacs Lisp and shows how it is evaluated. Something important in Elisp is that the only defined boolean is "false" or `nil` all other values act as "truth". This is commonly used to check if optional arguments has been given when a function was called (they have a `non`-nil value).

```
(if t 1 2)                  ;; ==> 1
(if nil 1 2 )               ;; ==> 2
(if (if 10 nil t) 12 21)    ;; ==> 21
```

Listing 59: Control structures in Elisp.

C.5 Booleans and Predicate Functions

Listing 60 demonstrates basic boolean operations and predicate functions. The name of predicate functions usually end with the character "p" (for predicate), a function which checks if a number is a integer is thus called `integerp`. The way truth is defined in Elisp allows for non-standard usage of logical operations such as `or` and `and`, where the second argument to the `and` operation is only evaluated if the first was true, or non-`nil`. And the arguments to the `or` operation only continues to be evaluated if the previous arguments has had a `nil` value. This makes it possible to define the `if` control structure only in terms of `and` and `or` operations: `(if A B C)` is equal to `(and (or A C) B)`. Due to this it is common in Elisp to say `non`-nil instead of true.

```
(floatp 0.4)    ;; ==> t
(integerp 0.4) ;; ==> nil
(or t nil)      ;; ==> t
(> 10 2)        ;; ==> t

(and (or (> 2 (sqrt 1.4))
"no")
"yes")
;; ==> "yes"
```

Listing 60: Booleans and predicate functions in Elisp.

C.6 Loops

In listing 61 we demonstrate how to create some loops in Elisp using the functions `dolist` for iterating over a list, and `while` for looping until a condition is true.

```
;; For each element of list
(let* ((sum 0))
(dolist (e '(1 2 3))
(setq sum (+ sum e)))
sum)
;; ==> 6

;; While condition is true
(let* ((n 10))
(while (> n 5)
(setq n (- n 1)))
n)
;; ==> 5
```

Listing 61: Loops and iteration in Elisp.

C.7 Scoped Variables

Listing 62 illustrates the use of scoped local variables using the `let` and `let*` constructs, these constructs allow you to create variables which are only valid within the constructs body and to shadow variables defined outside of the construct. The `let*` function allows its values to be changed or the value definitions to refer to the other definitions, while `let` only allows variables with a const value.

```
;; Bind constants in a scope (body)
(let ((a 1) (b 2))
(+ a b))
;; ==> 3

;; Define variables within a scope
(let* ((a 1) (b 2))
(setq a 2
b 4)
(+ a b))
;; ==> 6
```

Listing 62: Elisp `let` statements.

C.8 Lists and Pairs

Listing 63 demonstrates lists in Elisp, and some utility functions which can be applied to them. These functions include `reverse`, `nth`, `append` and `mapcar` which are used to create and manipulate lists. Pairs, which lists are made of

```
(setq a (list 1 2 3) ;; ==> a = (1 2 3)
      b '(4 5 6))    ;; ==> b = (4 5 6)

(append a b)            ;; ==> (1 2 3 4 5 6)
(reverse (append a b)) ;; ==> (6 5 4 3 2 1)
(nth 2 c)         ;; ==> 3
(mapcar #'sqr a) ;; ==> (1 4 9)
```

Listing 63: Lists in Elisp.

```
(setq c (cons 1 (cons 2 (cons 3 nil))))
;; ==> (1 2 3)
(car (cons 1 0))         ;; ==> 1
(cdr (cons 1 0))         ;; ==> 0

(car (cdr c))            ;; ==> 2
(setf (car (cdr c)) -1) ;; c == (1 -1 3)
```

Listing 64: Pairs in Elisp.

Listing 64 demonstrates the basic use of pairs through creating a list `(1 2 3)` using only `cons` pairs, as well as demonstrating `car` and `cdr` which returns the head and tail of the pair. The function `setf` is used to set the value stored in the location at `(car (cdr ...))` "the head of the tail" of the list

C.9 Manipulating Buffers

Listing 65 shows simple buffer manipulation, switching to the scratch buffer, inserting a string and then returning the contents of the buffer as a string.

```
;; 1. Switch to buffer named *scratch*
;; 2. Insert text "Hello world"
;; 3. Return buffer contents as string
(with-current-buffer "*scratch*"
(insert "Hello world")
(buffer-string))
;; ==> "Hello world"
```

Listing 65: Buffer manipulation in Elisp.

D Error Handling in Emacs Lisp

`(signal error-symbol data)`: Signal an error using an `error-symbol` and `data`. The `data` should be associated to the `error-symbol`. For example `(signal 'arith-error '(/ 1 0))` to signal a division by zero error. The `error-symbol` must be a valid error symbol which means that it previously have been declared using `define-error`. The `signal` function never returns, instead passing on the flow of execution to any active error handlers.

`(error format-string arguments ...)`: The `error` function takes a format control string `format-string` together with the strings arguments. The format string is formatted using `format-message` and wrapped in a list before signaling `error`: `(signal 'error (list <error message>))`.

`(condition-case var body-form handlers ...)`: `condition-case` defines a list of `handlers`. Each handler has the form `(conditions body ...)` where `conditions` is a `error-symbol`, `t` (matches with all error symbols) or list of `error-symbol`'s together with a `body` the Lisp expression which will execute when a signal signals any of the specified. When control returns to the condition case following a error signal, the variable `var` assumes the value `(error-symbol . error-data)`, and executing the body of the first handler with the error symbol `(car var)` in its `conditions`. The last value from the handler `body` is then returned.

E Creating an Asynchronous Subprocess

The listing 66 shows the function `emacs_spawn` (from `src/callproc.c` [3]) which starts a new asynchronous child process by forking the parent Emacs process using `vfork`. The function `child_setup` from `src/callproc.c` [3] is then called on the new child process and which sets up the file descriptors of the child process (see listing 67), finally calling `emacs_exec_file` (from `src/sysdep.c` [3]) which executes the process program using `execve` (see Listing 68).

```
int
emacs_spawn (pid_t *newpid, int std_in, int
    std_out, int std_err,
char **argv, char **envp, const char *cwd,
const char *pty_name, bool pty_in, bool
    pty_out,
const sigset_t *oldset)
{
  // [...]
  pid = vfork ();
  // [...]
  if (pid == 0)
  pid = child_setup (std_in, std_out,
      std_err, argv, envp, cwd);
  // [...]
}
```

Listing 66: The function `emacs_spawn` (from `src/callproc.c` [3]) modified for clarity.

```
/* ... */ child_setup (int in, int out, int
    err, char **new_argv, char **env,
const char *current_dir)
{
  // [...]
  set_process_dir (current_dir);
  // [...]
  dup2 (in, STDIN_FILENO);
  dup2 (out, STDOUT_FILENO);
  dup2 (err, STDERR_FILENO);
  // [...]
  int errnum = emacs_exec_file (new_argv[0],
      new_argv, env);
}
```

Listing 67: The function `child_setup` (from `src/callproc.c` [3]) modified for clarity.

```
int
emacs_exec_file (char const *file, char
    *const *argv, char *const *envp)
{
  // [...]
  execve (file, argv, envp);
  return errno;
}
```

Listing 68: The function `emacs_exec_file` (from `src/sysdep.c` [3]) modified for clarity.

```
(defun async-grep (dir regexp)
 (interactive "fWhere:\nsRegexp:")
  (condition-case err
   (deferred:$
    ;; Call grep as a subprocess
    (deferred:process "grep"
        "--with-filename" "-r" "-n" regexp
        dir)
    (deferred:error
     it
     (lambda (err)
       (error "Grep failed")))
    ;; Split resulting string into lines
     (deferred:nextc it
     (lambda (res)
       (if (string= res "")
           (error "Empty result")
         (split-string res "\n")
         )
       ))
    ;; Extract line, file and string of the
        matches
     (deferred:nextc it #'format-lines)
    ;; Insert matching lines into buffer
     (deferred:nextc it #'insert-matches ))

   (t
   (error "Promise Failed!"))))
```

Listing 69: Example of concurrency using `deferred.el`

F Example: `deferred.el`

Listings 69 to 71 shows a command that use `deferred.el` to recursively search for a string in all files in a directory, using the `deferred:$` function to chain promises. The process of searching is divided into different stages, represented by lambda functions, where the output of each stage is passed into a following stage upon completion.

```
(require 'deferred)
 (defun format-lines (lines)
   (cdr (nreverse
     (mapcar
       (lambda (line)
         (let ((tmp  (split-string line ":")))
           `( :file ,(nth 0 tmp)
              :line ,(nth 1 tmp)
              :match ,(apply #'concat (cddr
                  tmp)))))
  lines))))
```

Listing 70: Auxillary function for listing 69.

```
(defun insert-matches (matches)
 ;; Clear existing buffer
 (when (buffer-live-p (get-buffer "*output
     grep*"))
   (kill-buffer "*output grep*"))

 (switch-to-buffer-other-window "*output
     grep*")
 ;; Insert all matching lines
 (insert (string-join
    (mapcar (lambda (m) (plist-get m :match))
         matches)
    "\n\n")))
```

Listing 71: Auxillary function for listing 69.