ChatGPT

Achieving Concurrency in Emacs (Multi-threading and Async Workarounds)

Emacs has traditionally been single-threaded, meaning one execution path can run at a time. This causes long computations or waiting for user prompts (minibuffer input, pop-ups, etc.) to block everything. However, there are ways to introduce concurrency so that one task (e.g. an external **MCP** agent or a long Lisp routine) doesn't freeze the whole editor. Below we outline two main approaches: using Emacs Lisp threads for internal concurrency, and using asynchronous external processes. We also discuss how to handle user prompts and UI updates without blocking other work.

Emacs Lisp-Level Concurrency (Threads & Timers)

Recent versions of Emacs (since 26) include support for Emacs Lisp threads. These threads provide *cooperative concurrency* within a single Emacs process [1]. All threads share the same memory (so they can access the same buffers and variables), and Emacs will switch execution between threads only at well-defined yield points [1]. In practice, this means threads are **mostly cooperative** – one thread must reach a blocking point or explicitly yield before another runs [2]. For example, Emacs will automatically allow thread switches when waiting for keyboard input or process output, or if you call functions like `thread-yield` [2].

Using `(make-thread ...)`, you can offload a long-running Lisp function to a new thread. This can improve responsiveness: while a background thread computes, the main thread (UI) can continue handling user commands. If one thread enters a wait (say it's waiting for user input or sleeping), Emacs can schedule other threads in the interim [2]. In your case, you might run each agent or long computation in its own thread so that, for instance, a prompt waiting for user input in one thread doesn't completely stall other threads' progress. Keep in mind that Emacs threads are not fully parallel on multiple CPU cores (only one thread runs at a time), but they **do** allow interleaving tasks to avoid a single blockage. Also, since threads share state, you should use Emacs's mutexes or other synchronization if threads interact with the same data [3] (to avoid race conditions). Currently the thread support is low-level and some data races are possible in Emacs Lisp, so careful design is needed [4].

Timers and idle callbacks provide another form of cooperative concurrency. If rewriting code for threads is too complex or if you prefer not to use threads directly, you can split heavy tasks into chunks executed via `run-at-time` or idle timers. Libraries like `asyncloop.el` illustrate this approach: they schedule a series of small function calls on the event loop, so that no single chunk blocks the editor for long [5]. The idea is to break a big job into pieces and run each piece when Emacs is idle (or at intervals), allowing Emacs to remain responsive and handle input between chunks. This yields an effect similar to threading (concurrent progress) without true parallel threads. In fact, *asyncloop* was created for the common case where you "have a big slow piece of Emacs Lisp" but need to avoid hanging Emacs, and you've "ruled out async.el or shelling out, because your code needs to manipulate the current Emacs state" [6]. In such cases, cooperative scheduling via timers or explicit yields can keep Emacs interactive.

Using Asynchronous External Processes (Parallelism via OS)

Another powerful strategy is to offload work to **external processes**. Emacs can start subprocesses that run in parallel to the Emacs Lisp interpreter [7]. These are true OS-level processes or threads that can utilize multiple cores and run independently. From Emacs Lisp, you create an asynchronous process (e.g. with `start-process` or `make-process`) and *do not wait* for it to finish. Emacs will continue running and remain responsive while the child process does its work [7]. You can communicate with the subprocess through pipes: send it input, and handle its output via process filters or sentinels. For example, Emacs's compilation mode, or LSP clients, use this mechanism heavily – they spawn compilers/servers and update buffers asynchronously when results arrive, without blocking your editing.

In the context of an **MCP swarm integration**, you could design each agent or heavy computation as an external program (or even a separate Emacs in batch mode) launched by the main Emacs. Emacs would then act as a controller: it starts all the agents and perhaps opens an Emacs buffer for each to capture its output. Each subprocess can run concurrently on its own CPU core, and Emacs can track their status via their process objects [8]. If Emacs itself is not waiting synchronously on them, one agent pausing (or requesting input) won't stall the others – they are truly independent processes. Emacs simply listens for output/events from each. Notably, if you go this route, design the protocol so that **user interaction is handled on the Emacs side** rather than inside the child process. For instance, if an agent needs a user decision, the agent should signal that to Emacs (via output or a status flag); Emacs can then prompt the user and later send the answer back to the agent. This way, the child process doesn't freeze waiting on an input that isn't coming. In fact, Emacs's async library (used for running tasks in a separate Emacs process) explicitly warns that the async subprocess should avoid any user interaction, otherwise you'd end up "stuck with a prompt you will not be able to answer in the child emacs" [9]. Keep child processes headless and non-interactive; funnel all interactive queries through the main Emacs UI.

Non-Blocking Prompts and UI Considerations

A key part of making Emacs feel "multithreaded" is ensuring the UI doesn't lock up when something needs user input. By default, functions like `y-or-n-p` or `read-from-minibuffer` *will* block the Emacs thread until answered. To avoid this, you can employ patterns for **non-blocking prompts**. One approach is to invoke prompts via a short timer or idle event, instead of directly in the flow of logic. For example, rather than prompting immediately during a hook or background event (which would halt everything), you can schedule the prompt to appear a moment later. This technique is shown in an Emacs StackExchange solution where the author writes: *"The solution I've come up with is to run the function prompting for user input in a timer using* `run-at-time` *"* [10]. The core idea is that `run-at-time 0` (or a few milliseconds delay) will let Emacs return to its event loop and execute the prompt asynchronously, so other hooks or threads continue running. The prompt still pops up for the user, but it's not halting unrelated Emacs activity in the meantime. You can even arrange a callback or lambda to handle the user's response once obtained, instead of the original function waiting for the return value.

When using Emacs Lisp threads, another tactic is to dedicate one thread to handle UI prompts and user interaction, while other threads perform computations. Because Emacs allows thread switching when waiting for input [2], a thread that is sitting in `read-char` or `read-minibuffer` will yield control while it waits. This means other threads can run in parallel (cooperatively) during that wait. In practice, you might not need to manually create a special UI thread – the main thread (which processes commands and

displays) can serve that role. The important part is to ensure that long-running workers are on separate threads that will yield, so the main thread can always respond promptly to the user.

Finally, to achieve a **"dashboard" view of your MCP swarm**, you can leverage Emacs's ability to display and update multiple buffers. For instance, maintain a *status buffer* that lists all active agents/processes along with their state (running, waiting, needing input, finished, etc.). Each agent could have its own log buffer for detailed output, but the dashboard gives a high-level summary. You can update this dashboard either via a timer (e.g. every second refresh the stats) or event-driven updates – for example, attach a process sentinel that updates the dashboard when a process changes state [8]. If using threads, you could have each thread update some shared status variable, and use `run-at-time` or an idle timer on the main thread to refresh the display from that data. The key is that these updates are non-blocking and happen asynchronously. Emacs is quite capable of handling multiple asynchronous outputs; it was "async before async was cool," as one EmacsConf talk quips, due to its event-driven process IO design [11].

In summary, while Emacs isn't *natively* multi-threaded in the preemptive sense, you can **design concurrency** through cooperative threads and external processes. Use Emacs Lisp threads (and/or chunk your tasks with timers) for work that must be done inside Emacs (e.g. manipulating buffers or calling Emacs APIs) so that one task yields to another and UI stays reactive [2]. For CPU-intensive or independent tasks, spin them out as asynchronous subprocesses – letting the operating system achieve true parallelism [7]. Ensure that any user interaction is decoupled from the core logic: either handle it in the main thread via non-blocking prompts [10] or coordinate via signals so that no background worker sits waiting indefinitely. By combining these techniques, you could have a swarm of MCP agents working concurrently, with Emacs serving as the orchestrator and dashboard for their activities – all without the usual "everything hangs" problem when a prompt appears or a computation runs long.

Sources:

- GNU Emacs Lisp Reference Manual – *Threads* (cooperative concurrency in Emacs Lisp) [1] [2]
- GNU Emacs Lisp Reference Manual – *Processes* (asynchronous subprocesses running in parallel with Emacs) [7]
- StackExchange – *Non-blocking prompt for user input* (using `run-at-time` to avoid blocking other Emacs tasks) [10]
- Edström, N. – **Asyncloop** README (splitting Emacs tasks into non-blocking chunks using timers) [5] [6]
- *Core Dumped* Blog – "A vision of a multi-threaded Emacs" (discussion of Emacs threading and future parallelism) [12] [4]

---

[1] [2] [3] Threads (GNU Emacs Lisp Reference Manual)
https://www.gnu.org/software/emacs/manual/html_node/elisp/Threads.html

[4] [12] A vision of a multi-threaded Emacs • Core Dumped
https://coredumped.dev/2022/05/19/a-vision-of-a-multi-threaded-emacs/

[5] [6] GitHub - meedstrom/asyncloop: Small Emacs-Lisp library for non-blocking threads, if by "thread" you mean "list of functions"
https://github.com/meedstrom/asyncloop

7 8 Processes (GNU Emacs Lisp Reference Manual)
https://www.gnu.org/software/emacs/manual/html_node/elisp/Processes.html

9 emacs-async/README.md at master - GitHub
https://github.com/jwiegley/emacs-async/blob/master/README.md

10 hooks - Non-blocking prompt for user input - Emacs Stack Exchange
https://emacs.stackexchange.com/questions/70969/non-blocking-prompt-for-user-input

11 2022 - talks - Emacs was async before async was cool - EmacsConf
https://emacsconf.org/2022/talks/async/