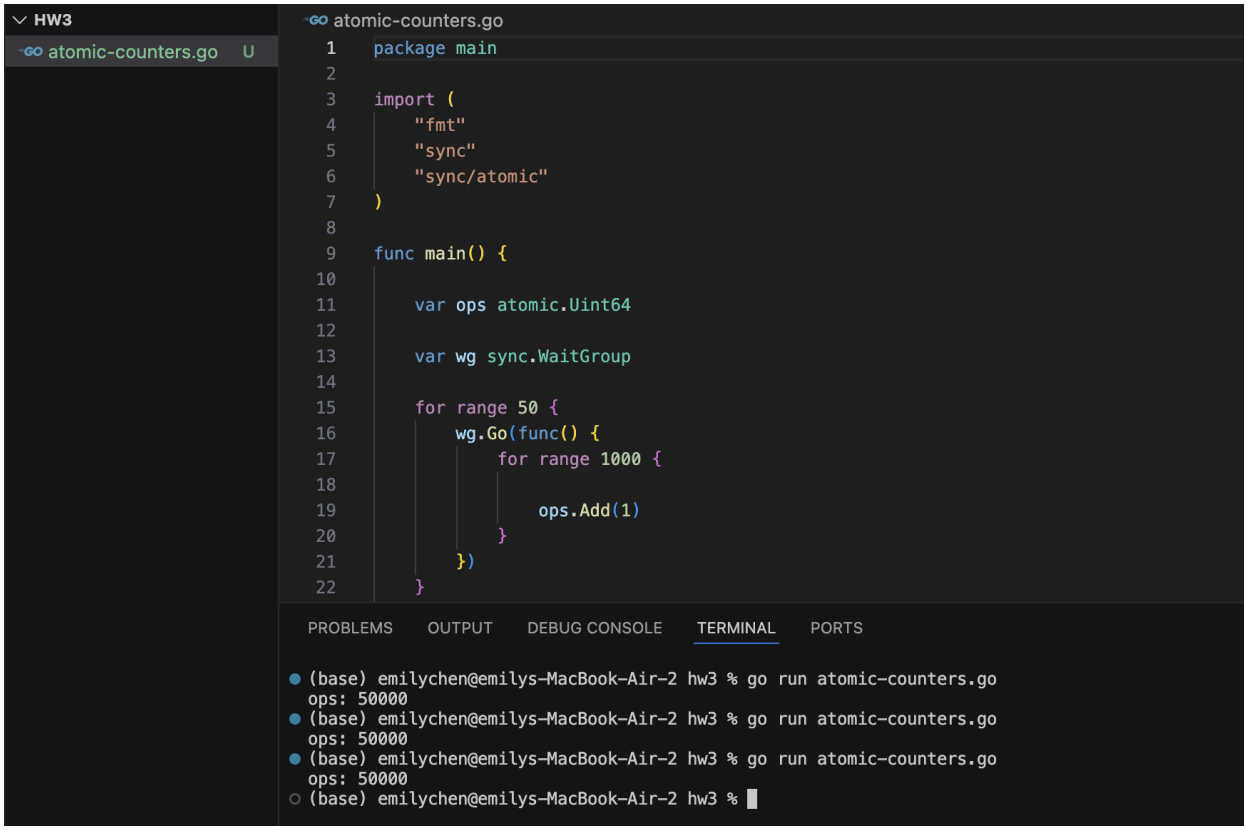
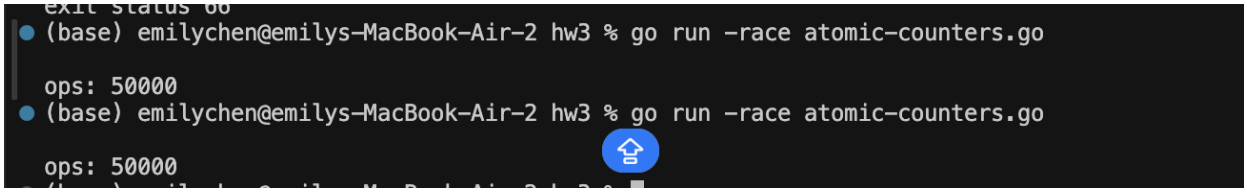
Part II

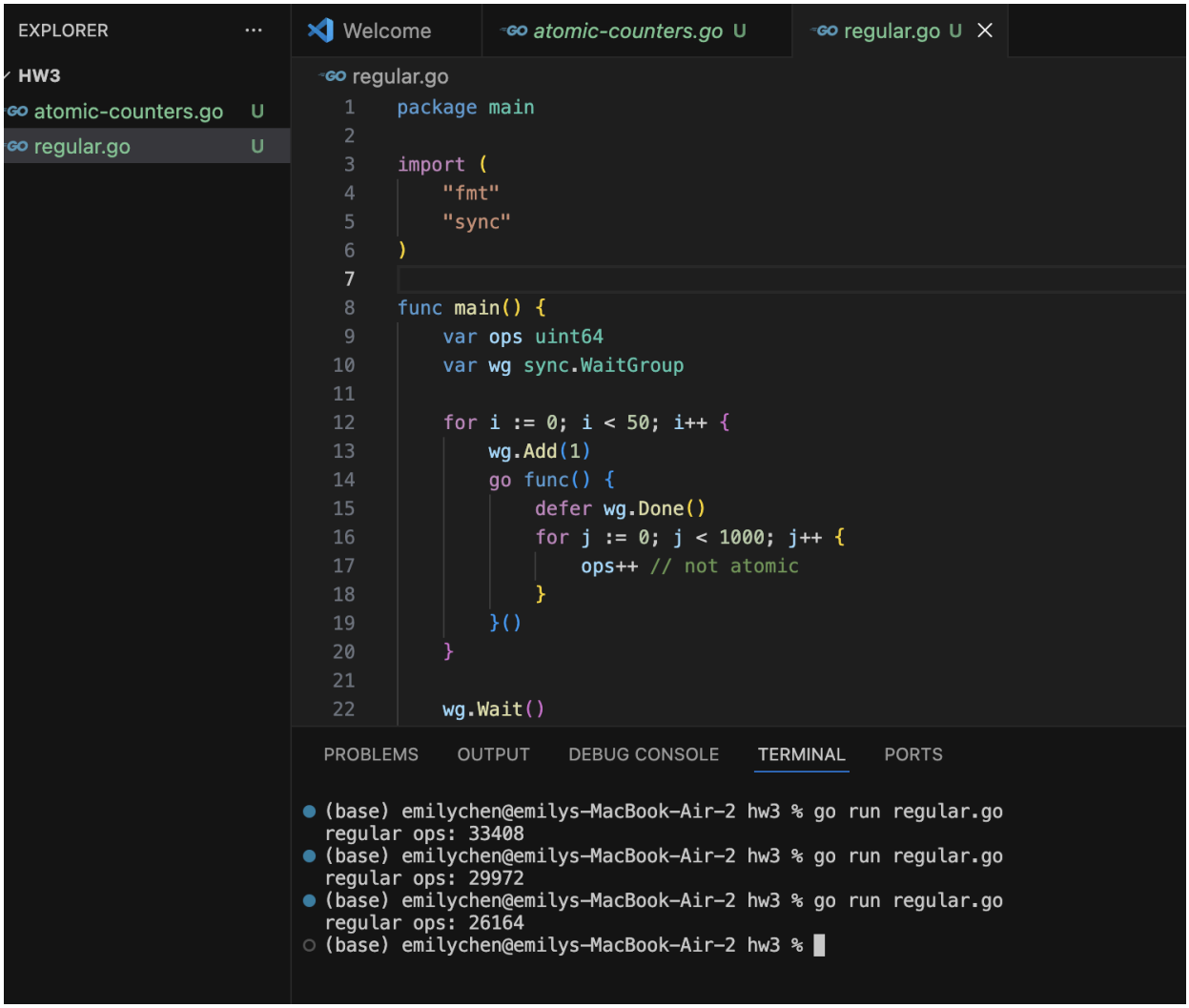
Atomicity



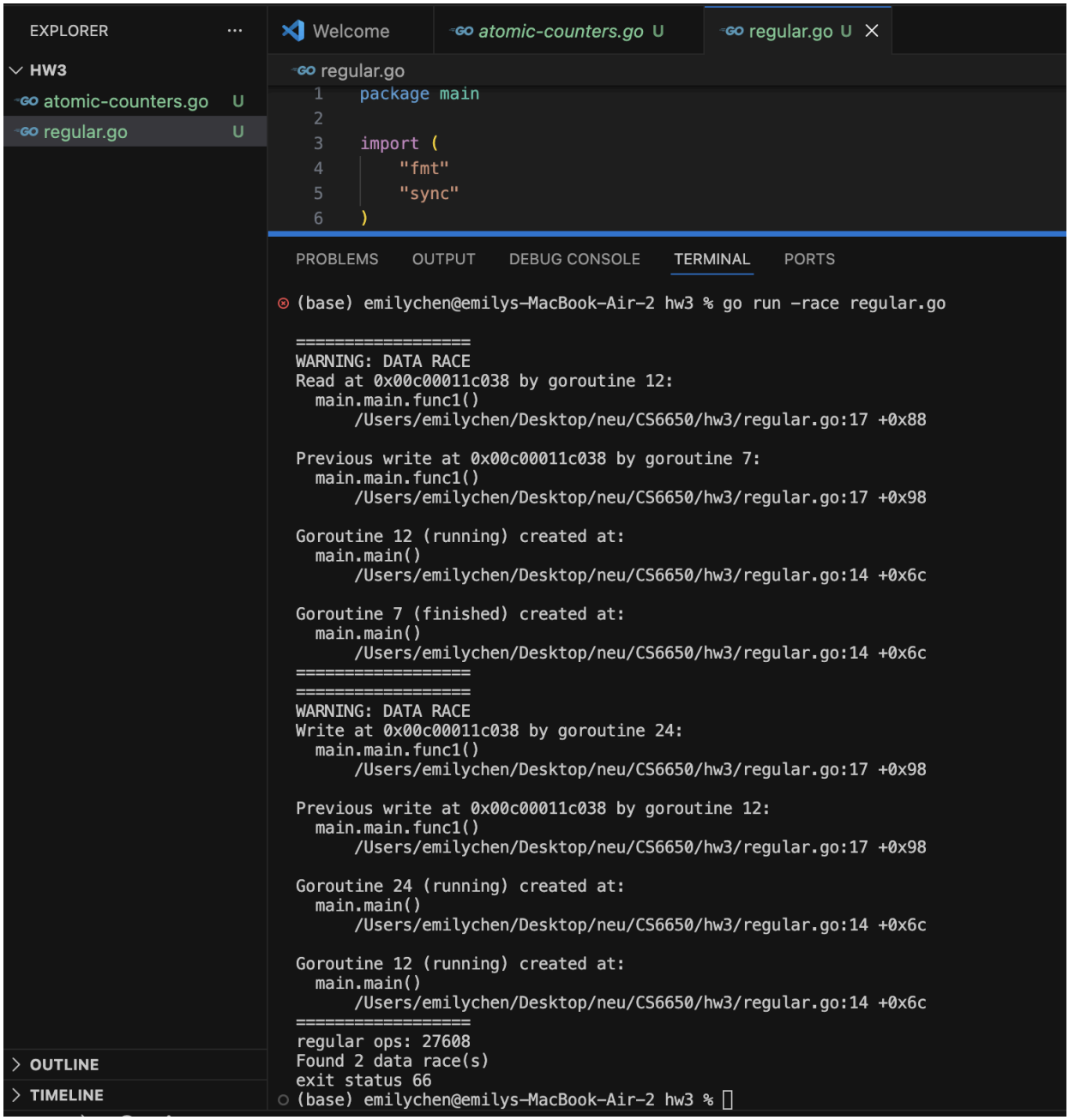
Atomic operation: no data race



Ops++ : lost update

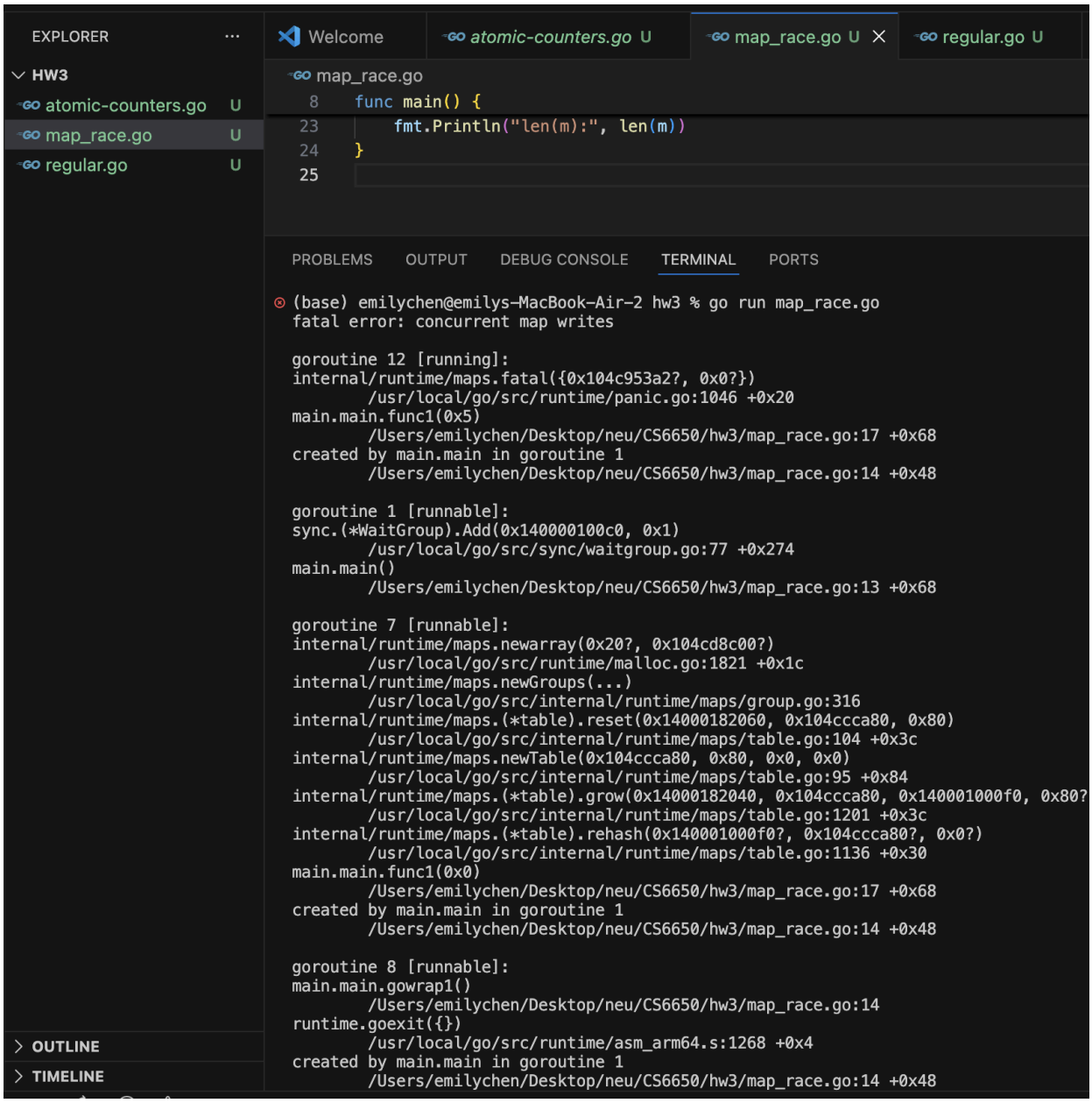

EXPLORER
HW3
atomic-counters.go U
regular.go U
Welcome
atomic-counters.go U
regular.go U
regular.go
1 package main
2
3 import (
4 "fmt"
5 "sync"
6)
7
8 func main() {
9 var ops uint64
10 var wg sync.WaitGroup
11
12 for i := 0; i < 50; i++ {
13 wg.Add(1)
14 go func() {
15 defer wg.Done()
16 for j := 0; j < 1000; j++ {
17 ops++ // not atomic
18 }
19 }()
20 }
21
22 wg.Wait()
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
(base) emilychen@emilys-MacBook-Air-2 hw3 % go run regular.go
regular ops: 33408
(base) emilychen@emilys-MacBook-Air-2 hw3 % go run regular.go
regular ops: 29972
(base) emilychen@emilys-MacBook-Air-2 hw3 % go run regular.go
regular ops: 26164
(base) emilychen@emilys-MacBook-Air-2 hw3 %

Operations ++: data race



Collections

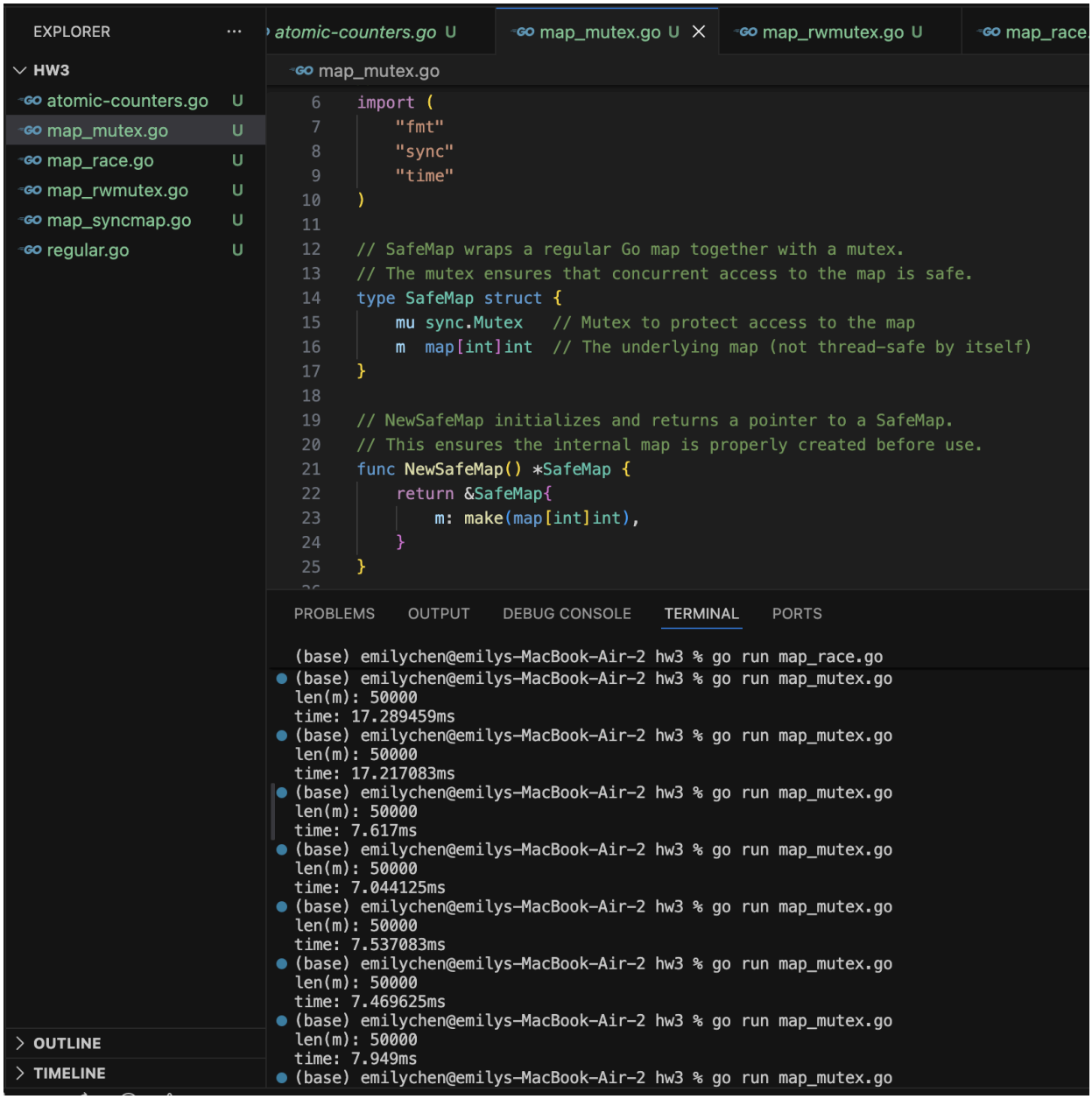
Map_race.go crash for 3 times



The program crashes because Go's built-in map is not safe for concurrent writes. Multiple goroutines write to the same map without synchronization, which can corrupt the map's internal hash table structure, especially during growth/rehash. To prevent silent memory corruption, the Go runtime detects this and terminates the program with fatal error: concurrent map writes.

Mutex

map_mutex:



A Go map is not safe for concurrent access. If multiple goroutines read/write the same map at the same time, you can get race conditions or even a runtime panic (“concurrent map writes”). Wrapping the map in a struct with a sync.Mutex makes every access safe by forcing goroutines to take turns.

Correctness improves, but performance can drop. After adding the mutex, the map length becomes stable and correct across runs, and -race warnings go away. However, total runtime often increases because all goroutines serialize on the same lock. If the critical section is frequent or long, the mutex becomes a bottleneck.

The key tradeoff is safety vs throughput. A mutex is a simple, reliable way to protect shared state, but it reduces parallelism when many goroutines contend for the same lock. More

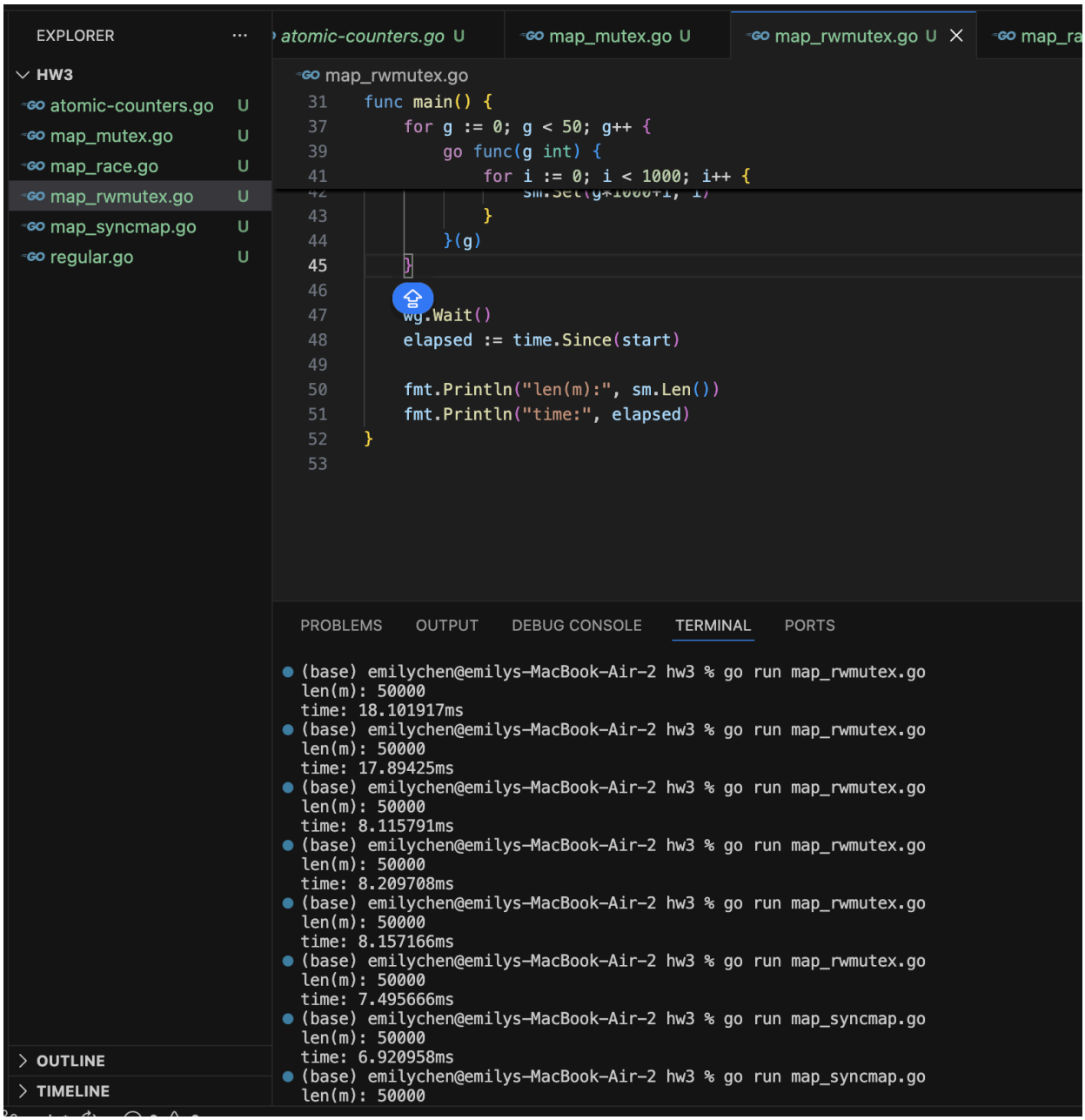
contention → more waiting → less speedup.

Practical takeaway: Use a mutex when you truly need shared mutable state, but try to reduce lock contention by:

- limiting how much work happens while holding the lock,
- using sharding (multiple maps/locks),
- or avoiding shared state via channels / per-worker local maps and merging results at the end.

RWMutex

map_rwmutex



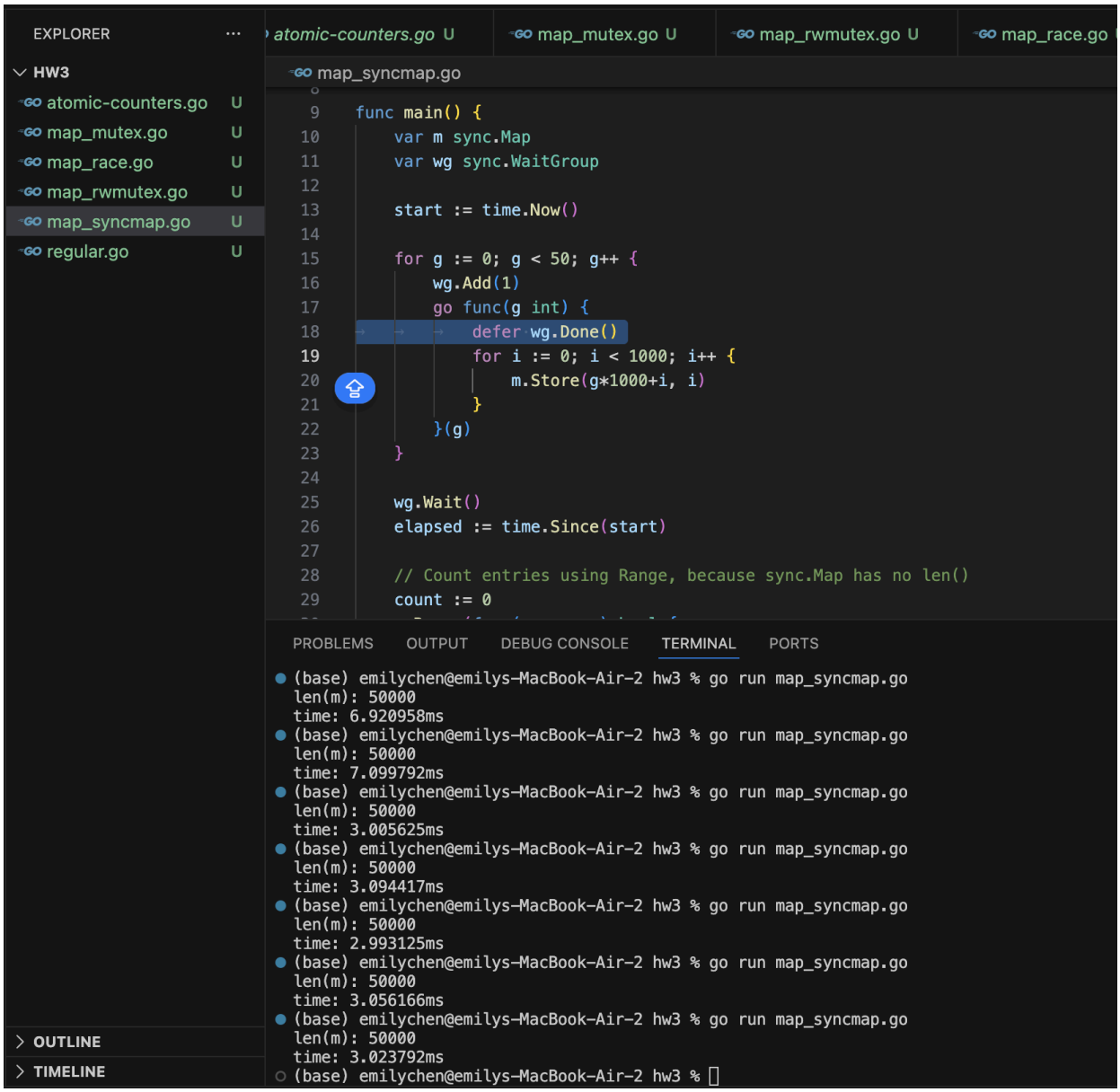
No meaningful improvement. RWMutex did not consistently reduce runtime and was sometimes slower than Mutex.

Because the workload is dominated by writes. RWMutex only improves performance when there are many concurrent reads and few writes. In this program, almost all operations require exclusive write locks, so RWMutex offers no advantage and adds overhead.

The lesson is that synchronization primitives must match the access pattern. RWMutex is beneficial in read-heavy workloads with many concurrent readers. In write-heavy workloads like this one, a regular mutex is simpler, more predictable, and often faster.

Sync.Map

map_syncmap:



Mutex + map: simplest and predictable. Great for write-heavy workloads, but serializes both reads and writes.

RWMutex + map: allows concurrent reads, so it can be faster in read-heavy workloads, but adds overhead and does not help much when writes dominate.

sync.Map: specialized concurrent map. Often good for read-mostly or write-once-read-many patterns. API is different (no len), and performance may be worse for write-heavy workloads.’

Approach	Typical Time (ms)	Correctness	Notes
Mutex + map	~8.1 ms	✔	Stable, simple
RWMutex + map	~8.0 ms (with outliers up to ~18 ms)	✔	No benefit here
sync.Map	**~3.0 ms**	✔	Fastest after warm-up

Why `sync.Map` was fastest

`sync.Map` is a specialized concurrent map implementation designed to reduce contention in common concurrent patterns. In the measurements, once warmed up, it reduced overhead enough to run around ~3 ms. It also has a different internal strategy than “one global lock for all writes”.

The warm-up effect (first two runs slower) is normal and can come from allocations, map growth, and runtime/cache effects.’

Why plain `map` crashes without locks

Go’s built-in `map` is not safe for concurrent writes. Multiple goroutines writing can corrupt internal hash table state, so the runtime stops the program with `fatal error: concurrent map writes`.

Why `RWMutex` did not help here

This workload is almost entirely writes:

- 50 goroutines × 1000 stores = **50,000 writes**
- Only one read at the end (count length)

RWMutex only improves throughput when there are many concurrent reads. Writes still require the exclusive lock, so RWMutex cannot enable parallelism here. It can also add bookkeeping overhead, which helps explain the outliers.

If reads dominate, what changes?

If the workload becomes read-heavy, for example many goroutines repeatedly doing `Get/Load` on existing keys and few writes:

- `map + Mutex`: gets worse relative to others, because every read waits in line behind a single lock.
- `map + RWMutex`: usually improves a lot, because many goroutines can hold `RLock()` at the same time.
- `sync.Map`: often performs very well in read-mostly scenarios, especially when keys are stable and reads are frequent.

So the ranking often shifts in read-heavy workloads: RWMutex and sync.Map tend to pull ahead, while Mutex becomes the bottleneck.

Tradeoffs summary

map + Mutex

- Pros: simplest, type-safe, predictable.
- Cons: serializes all reads and writes.

map + RWMutex

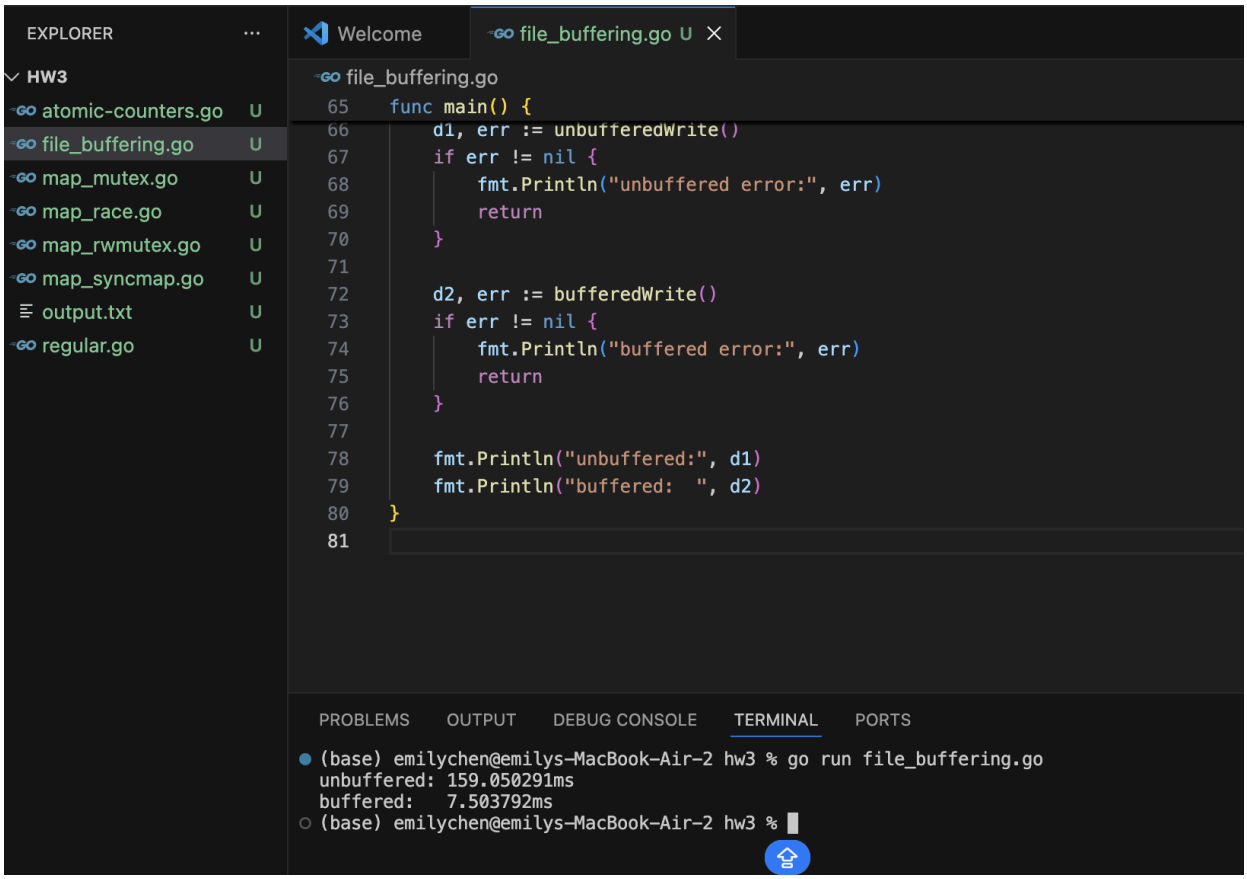
- Pros: best when reads dominate, concurrent reads can proceed.
- Cons: little or no benefit when writes dominate, extra overhead, can show more variance.

sync.Map

- Pros: can be fastest under certain concurrent patterns, good for high-concurrency access and often read-heavy use cases.
- Cons: different API (no `len`, must `Range`), uses `any` (less type safety), performance depends on workload pattern.

File Access

File_buffering.go

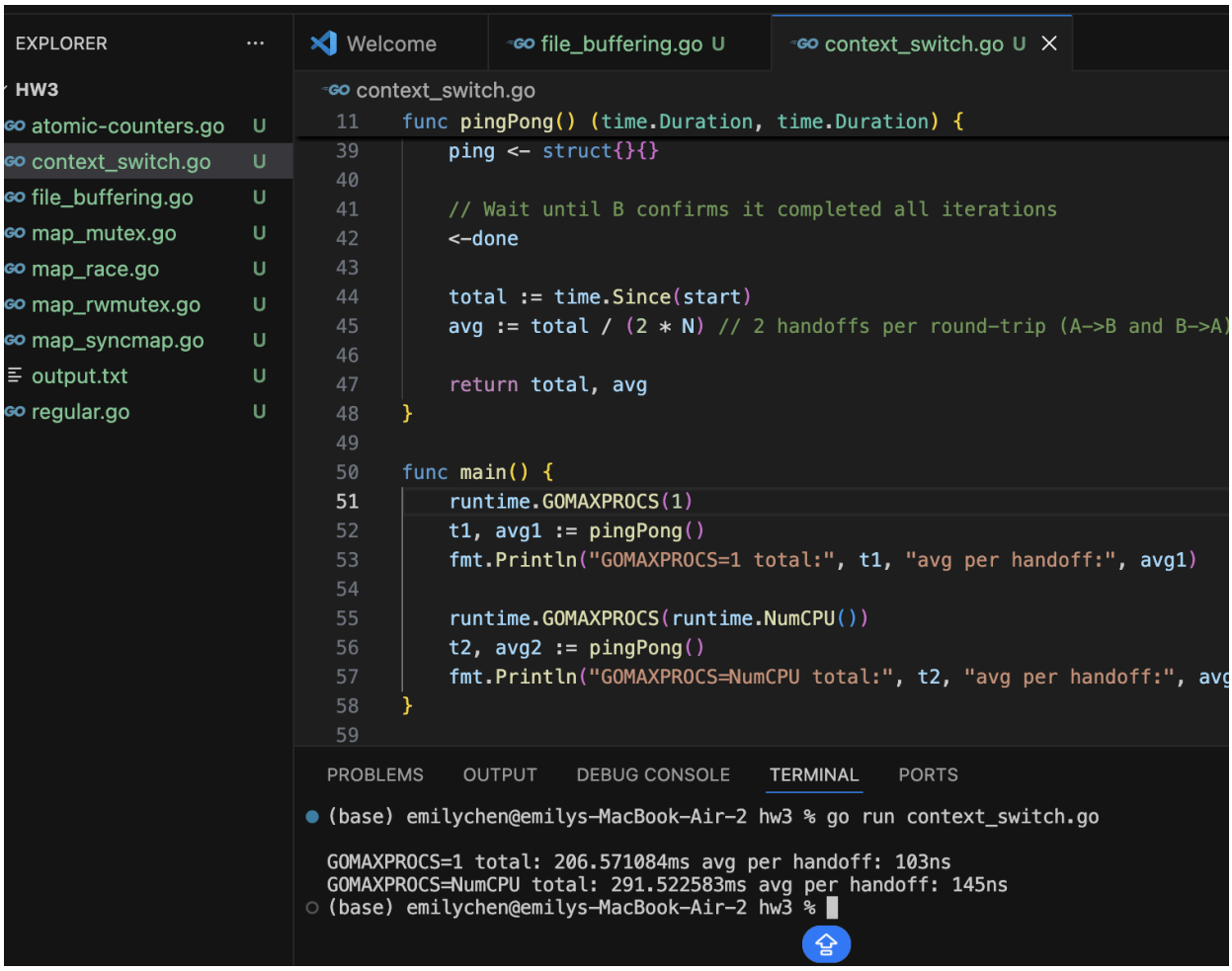


In the unbuffered version, the program called `f.Write()` once per iteration, while in the buffered version the file was wrapped with `bufio.Writer` and data was written to memory first and flushed at the end. The unbuffered run took about **159 ms**, while the buffered run took about **7.5 ms**, making buffered writing more than **20 times faster**.

This large difference happens because each unbuffered write typically results in a system call into the operating system. System calls are expensive due to kernel transitions and filesystem bookkeeping, and performing them 100,000 times dominates the runtime. Buffered writing avoids this cost by batching many small writes in memory and issuing far fewer system calls, each transferring a larger chunk of data to the OS.

The main lesson from this experiment is that buffering trades immediacy for performance. Buffered I/O significantly improves throughput when performing many small writes, but data may remain in memory until it is flushed, which introduces a risk of data loss if the program crashes and requires the programmer to explicitly call `Flush()`. Unbuffered I/O, while simpler and safer in terms of immediacy, performs poorly under high-frequency write workloads. This experiment reinforces a common systems principle: reducing the number of user-to-kernel transitions is critical for achieving good performance.

Context Switching



With `GOMAXPROCS(1)`, the total runtime was about **206 ms**, giving an average handoff cost of approximately **103 ns**. When allowing Go to schedule goroutines across all available CPUs, the total runtime increased to about **291 ms**, with an average handoff cost of approximately **145 ns**. In this experiment, the single–OS-thread configuration was therefore faster.

This happens because the ping-pong workload is strictly sequential and alternating: only one goroutine can make progress at any moment. Allowing multiple OS threads does not introduce any real parallelism, but it does introduce extra coordination overhead. When goroutines are scheduled on different OS threads or cores, the runtime must perform additional synchronization, wakeups, and cache-coherence work when handing off the channel signal.

With `GOMAXPROCS(1)`, both goroutines stay on the same OS thread, so the handoff remains local and avoids cross-thread scheduling and cache traffic, resulting in lower overhead.

This experiment illustrates how context-switching cost increases as isolation boundaries become heavier. Goroutine switches are relatively cheap because they are managed largely in user space and share the same address space. Switching between OS threads or processes is more expensive because it involves kernel scheduling and more CPU state management. Containers do not fundamentally change this picture, since they are still processes from the kernel's perspective, but they may add some accounting overhead. Virtual machines typically have the highest switching cost, as they involve a hypervisor layer and additional virtualized CPU and memory state. Overall, the results reinforce that stronger isolation generally comes with higher context-switching overhead, and that more threads or cores do not automatically lead to better performance for strictly sequential workloads.

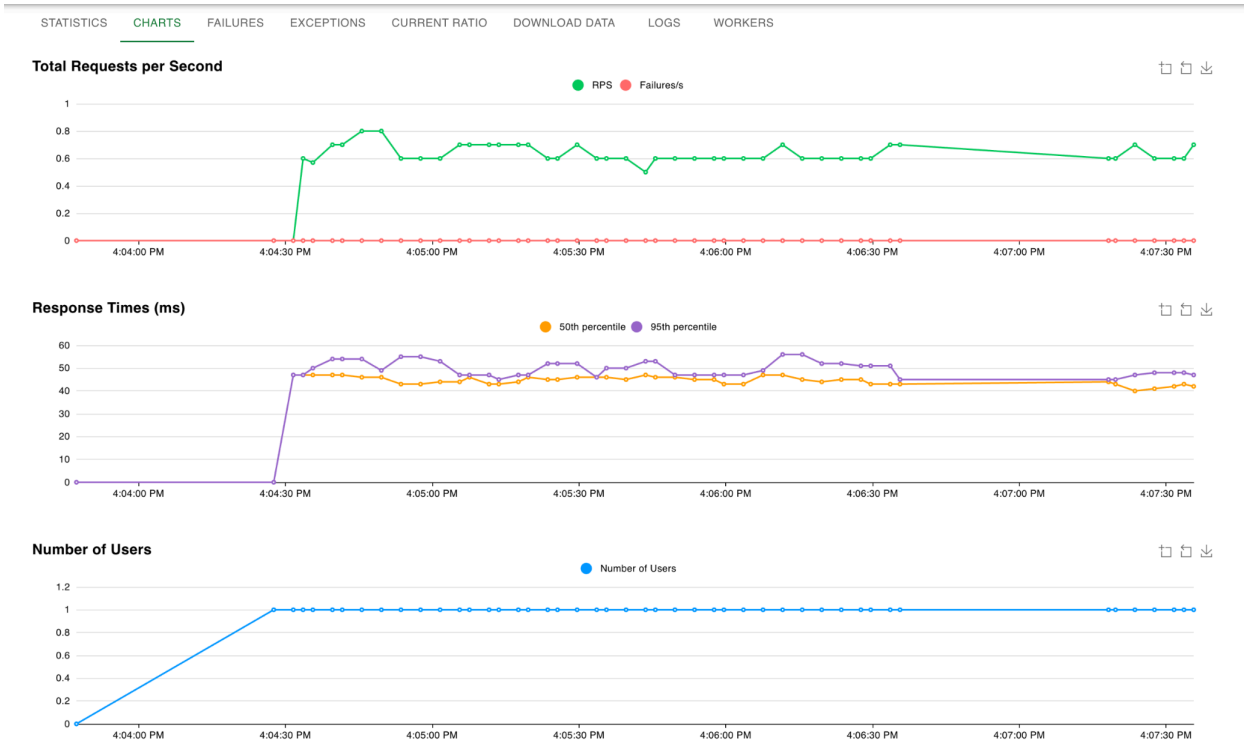
Part III

Locust

```
[+] Running 3/3
 ✔ Network locust-test_default     Create...                    0.0s
 ✔ Container locust-test-worker-1  Cre...                       0.9s
 ✔ Container locust-test-master-1  Cre...                       0.9s
Attaching to master-1, worker-1
master-1  | [2026-02-02 00:02:13,755] 549e885432af/INFO/locust.main: Starting Locust 2.43.2
master-1  | [2026-02-02 00:02:13,756] 549e885432af/INFO/locust.main: Starting web interface at http://0.0.0.0:8089, press enter
to open your default browser.
worker-1  | [2026-02-02 00:02:13,759] ac8f70286fa6/INFO/locust.main: Starting Locust 2.43.2
master-1  | [2026-02-02 00:02:13,769] 549e885432af/INFO/locust.runners: ac8f70286fa6_5fa37617bf164a3d9e1f31e78b838c3d (index 0)
reported as ready. 1 workers connected.
```

STATISTICS | CHARTS | FAILURES | EXCEPTIONS | CURRENT RATIO | DOWNLOAD DATA | LOGS | WORKERS

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	57	0	45	55	81	45.65	38	81	1718.84	0.5	0
POST	POST /albums	18	0	46	54	54	46.25	40	54	100	0.1	0
	Aggregated	75	0	45	54	81	45.79	38	81	1330.32	0.6	0

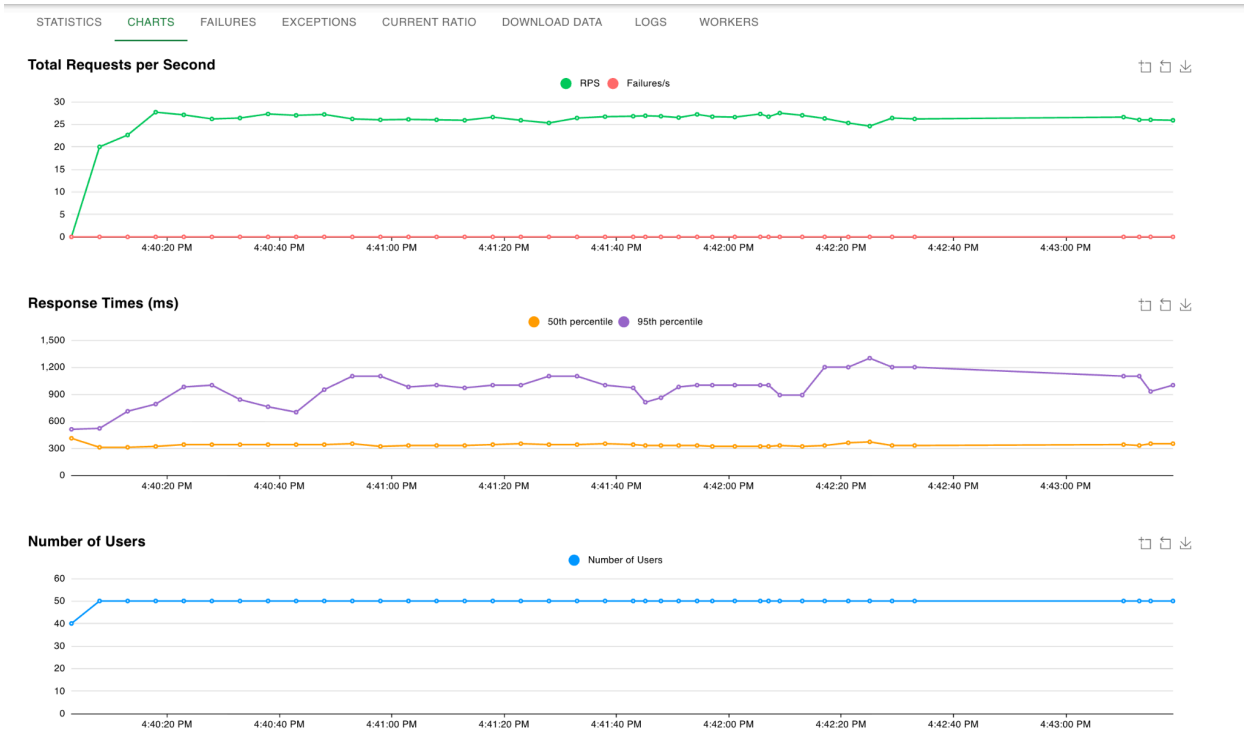


In this experiment, both GET and POST requests completed with zero failures, confirming the correctness of the system. Since real-world workloads are typically read-heavy, optimizing GET performance is critical. Using a hashmap provides fast O(1) lookups for GET requests, while POST requests incur additional overhead due to JSON parsing and state mutation. Under low load, the latency difference is minimal, but under higher concurrency, write operations would likely experience more contention and higher tail latency. This highlights the tradeoff between simplicity, correctness, and scalability.

Load test

STATISTICS CHARTS FAILURES EXCEPTIONS CURRENT RATIO DOWNLOAD DATA LOGS WORKERS

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	2843	0	370	1100	1900	493.23	110	2413	1055833.16	20.6	0
POST	POST /albums	915	0	52	84	140	56.67	31	403	100	4	0
	Aggregated	3758	0	330	1000	1900	386.93	31	2413	798782.65	24.6	0



I used Locust to perform load testing on the HTTP server from the previous homework. Locust was run locally using Docker Compose with one master and one worker. The test was configured with 50 concurrent users, a ramp-up rate of 10 users per second, and a 3:1 ratio of GET to POST requests. GET requests retrieved the `/albums` endpoint, while POST requests inserted new album records.

Because load tests can also stress the machine generating traffic, I monitored resource usage using `docker stats` during the experiment. **The Locust worker stayed around 28% CPU usage** and the master stayed below 1%, with low memory usage overall. This indicates that the load generator was not a bottleneck and that the observed performance reflects the server's behavior.

During the test, there were no failures for either GET or POST requests, and throughput remained stable at around 25–30 requests per second. However, GET requests showed

significantly higher latency than POST requests, especially in the p95 and p99 percentiles. This is because GET requests returned the full albums list, which grew over time as POST requests added more data. The larger response size increased serialization, memory, and network overhead, leading to noticeable tail latency under concurrent load.

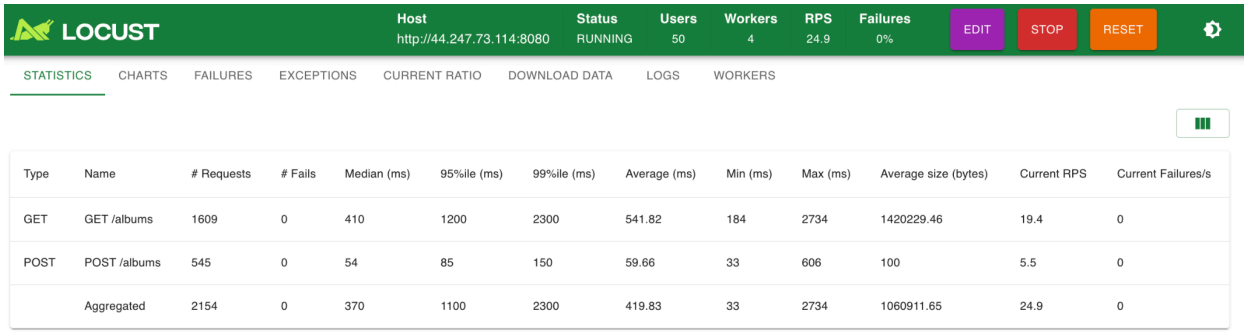
Overall, the system handled concurrent traffic correctly without errors, but the results highlight how read-heavy workloads with large response payloads can impact tail latency even when using efficient data structures.

Amdahl's Law

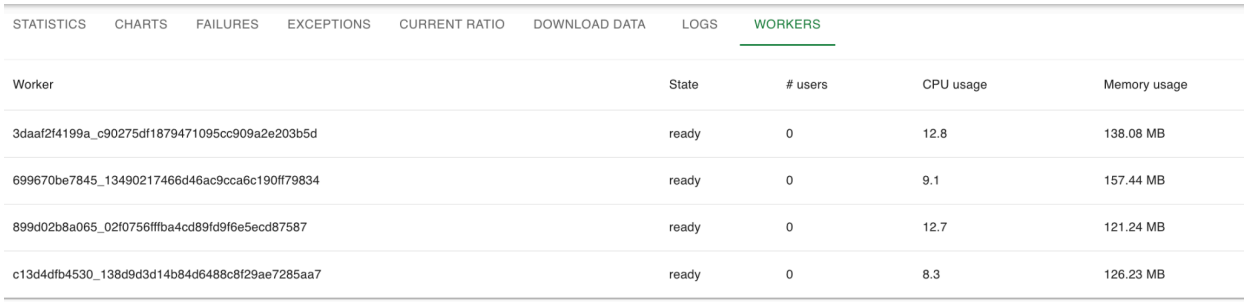
4 worker node:

```
(base) emilychen@emilys-MacBook-Air-2 locust-test % ls
__pycache__             docker-compose.yml      locustfile.py
(base) emilychen@emilys-MacBook-Air-2 locust-test % docker compose up --scale worker=4

[+] Running 3/3
 ✔ Container locust-test-worker-4  Created
 ✔ Container locust-test-worker-2  Created
 ✔ Container locust-test-worker-3  Created
Attaching to master-1, worker-1, worker-2, worker-3, worker-4
worker-4  | [2026-02-02 00:55:22,551] c13d4dfb4530/INFO/locust.main: Starting Locust 2.43.2
master-1  | [2026-02-02 00:55:22,553] 3f04b1d55cbd/INFO/locust.main: Starting Locust 2.43.2
master-1  | [2026-02-02 00:55:22,556] 3f04b1d55cbd/INFO/locust.main: Starting web interface at http://0.0.
to open your default browser.
master-1  | [2026-02-02 00:55:22,576] 3f04b1d55cbd/INFO/locust.runners: c13d4dfb4530_138d9d3d14b84d6488c8f
reported as ready. 1 workers connected.
worker-3  | [2026-02-02 00:55:22,690] 699670be7845/INFO/locust.main: Starting Locust 2.43.2
master-1  | [2026-02-02 00:55:22,716] 3f04b1d55cbd/INFO/locust.runners: 699670be7845_13490217466d46ac9cca6
reported as ready. 2 workers connected.
worker-2  | [2026-02-02 00:55:22,861] 3daaf2f4199a/INFO/locust.main: Starting Locust 2.43.2
master-1  | [2026-02-02 00:55:22,882] 3f04b1d55cbd/INFO/locust.runners: 3daaf2f4199a_c90275df1879471095cc9
reported as ready. 3 workers connected.
worker-1  | [2026-02-02 00:55:23,048] 899d02b8a065/INFO/locust.main: Starting Locust 2.43.2
master-1  | [2026-02-02 00:55:23,057] 3f04b1d55cbd/INFO/locust.runners: 899d02b8a065_02f0756fffba4cd89fd9f
reported as ready. 4 workers connected.
```



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	1609	0	410	1200	2300	541.82	184	2734	1420229.46	19.4	0
POST	POST /albums	545	0	54	85	150	59.66	33	606	100	5.5	0
	Aggregated	2154	0	370	1100	2300	419.83	33	2734	1060911.65	24.9	0



Worker	State	# users	CPU usage	Memory usage
3daaf2f4199a_c90275df1879471095cc909a2e203b5d	ready	0	12.8	138.08 MB
699670be7845_13490217466d46ac9cca6c190ff79834	ready	0	9.1	157.44 MB
899d02b8a065_02f0756fffba4cd89fd9f6e5ecd87587	ready	0	12.7	121.24 MB
c13d4dfb4530_138d9d3d14b84d6488c8f29ae7285aa7	ready	0	8.3	126.23 MB

I first ran the load test with **1 worker**, then repeated the same test with **4 workers**, keeping everything else the same: 50 users, 10 users per second ramp-up, and a 3:1 GET-to-POST ratio. When I increased the number of workers from 1 to 4, I expected throughput to increase, but the results showed that total RPS stayed roughly the same, around 25 requests per second.

This means throughput did **not** scale linearly with the number of workers. According to **Amdahl's Law**, this is expected. Even if part of the workload can be parallelized, any serial portion of the system limits the overall speedup. Adding more workers only helps until those serial components dominate.

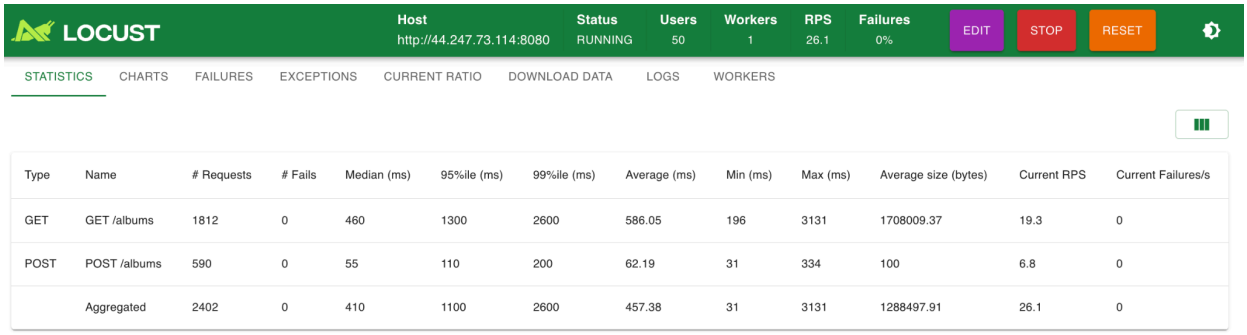
In this case, the bottleneck is not the load generator. CPU usage on the Locust containers stayed well below saturation, so the limitation comes from the server. Both GET and POST requests interact with a shared in-memory hashmap. POST requests modify the map, and GET requests read from it. If the map is protected by a mutex or RWMutex, writes must be serialized, and readers may also block when a writer is active. As concurrency increases, this shared access becomes a serial section that prevents linear scaling.

This also explains why GET requests show high tail latency. GET responses return the entire albums collection, which grows over time as POST requests add data. Under concurrent access, large reads combined with synchronization around the shared map increase contention and amplify p95 and p99 latency.

Overall, increasing the number of workers did not significantly improve throughput because the server contains unavoidable serial sections, consistent with Amdahl's Law. The shared

hashmap access pattern, especially mixed reads and writes, directly contributes to the observed scaling limits and tail latency.

Context Switching



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	GET /albums	1812	0	460	1300	2600	586.05	196	3131	1708009.37	19.3	0
POST	POST /albums	590	0	55	110	200	62.19	31	334	100	6.8	0
	Aggregated	2402	0	410	1100	2600	457.38	31	3131	1288497.91	26.1	0

After switching from `HttpUser` to `FastHttpUser`, I did not observe a significant improvement in overall throughput. The total RPS stayed around the same and there were still no failures. This suggests that Locust was not the bottleneck in my experiment. Even though `FastHttpUser` is more efficient and reduces client-side overhead, the server itself became the limiting factor. According to Amdahl's Law, improving one part of the system does not increase overall performance if another part remains serial or constrained. In my case, factors like server CPU usage or shared data structure access likely limited the throughput, so making the load generator faster did not lead to higher performance.