HW3 - Report

CS6640 25Fall

Rong Huang

Part II

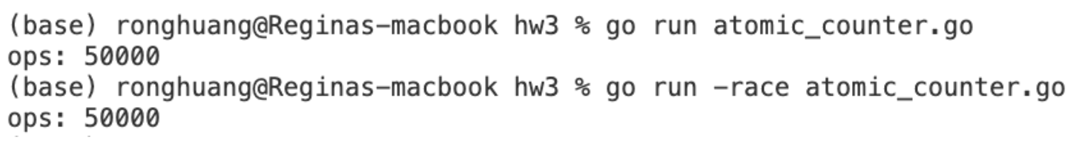
1. Atomic

Code

- atomic_counters.go: uses `atomic.Uint64` with `Add` and `Load`.
- non_atomic.go: uses a plain `uint64` with `ops++`.

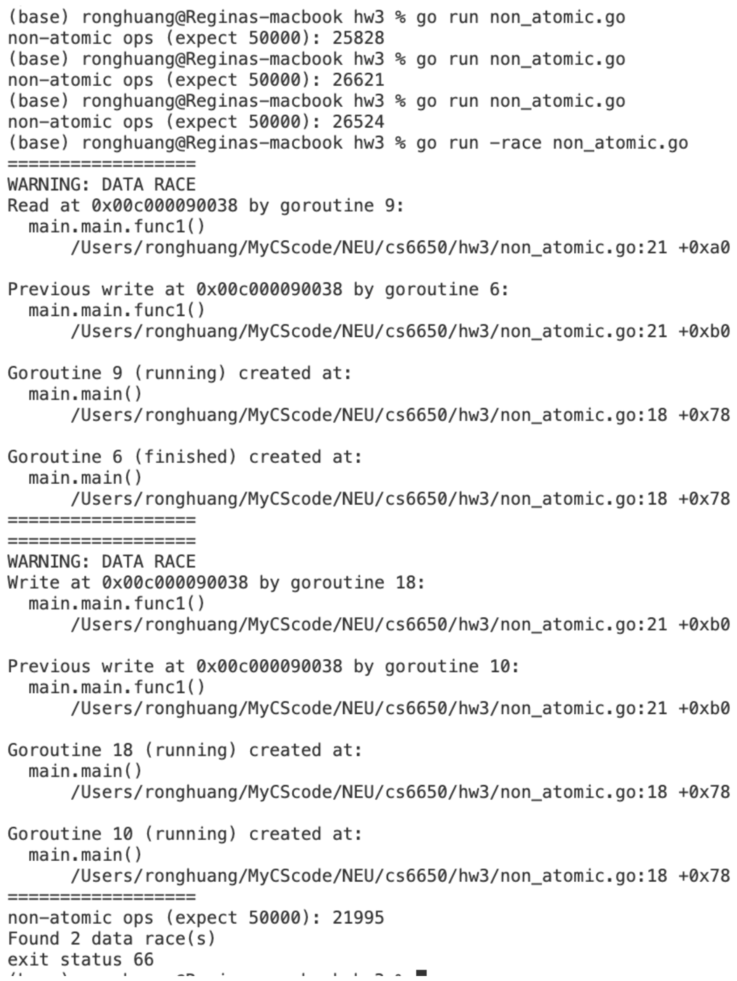
Results

- **Atomic**
  - Output is always `ops: 50000`.
  - Running with `race` shows no warnings.



```
(base) ronghuang@Reginas-macbook hw3 % go run atomic_counter.go
ops: 50000
(base) ronghuang@Reginas-macbook hw3 % go run -race atomic_counter.go
ops: 50000
```

- **Non-atomic**
  - Outputs vary and are always less than 50000 (e.g., 25828, 26621, 26524, 21995).
  - Running with `race` reports `WARNING: DATA RACE` with stack traces.



```
(base) ronghuang@Reginas-macbook hw3 % go run non_atomic.go
non-atomic ops (expect 50000): 25828
(base) ronghuang@Reginas-macbook hw3 % go run non_atomic.go
non-atomic ops (expect 50000): 26621
(base) ronghuang@Reginas-macbook hw3 % go run non_atomic.go
non-atomic ops (expect 50000): 26524
(base) ronghuang@Reginas-macbook hw3 % go run -race non_atomic.go
==================
WARNING: DATA RACE
Read at 0x00c000090038 by goroutine 9:
  main.main.func1()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/non_atomic.go:21 +0xa0

Previous write at 0x00c000090038 by goroutine 6:
  main.main.func1()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/non_atomic.go:21 +0xb0

Goroutine 9 (running) created at:
  main.main()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/non_atomic.go:18 +0x78

Goroutine 6 (finished) created at:
  main.main()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/non_atomic.go:18 +0x78
==================
==================
WARNING: DATA RACE
Write at 0x00c000090038 by goroutine 18:
  main.main.func1()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/non_atomic.go:21 +0xb0

Previous write at 0x00c000090038 by goroutine 10:
  main.main.func1()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/non_atomic.go:21 +0xb0

Goroutine 18 (running) created at:
  main.main()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/non_atomic.go:18 +0x78

Goroutine 10 (running) created at:
  main.main()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/non_atomic.go:18 +0x78
==================
non-atomic ops (expect 50000): 21995
Found 2 data race(s)
exit status 66
```

Observations

- Non-atomic increment (`ops++`) is not safe under concurrency. The read-modify-write sequence can be interleaved across goroutines, causing lost updates.
- The race detector clearly reports concurrent accesses to the same memory location.
- Atomic increment (`ops.Add(1)`) ensures correctness and consistency across runs.

Conclusion

Atomic operations guarantee correctness in concurrent settings with low overhead. Without atomicity, data races occur, results become inconsistent, and the race detector reports errors.

2. Collections

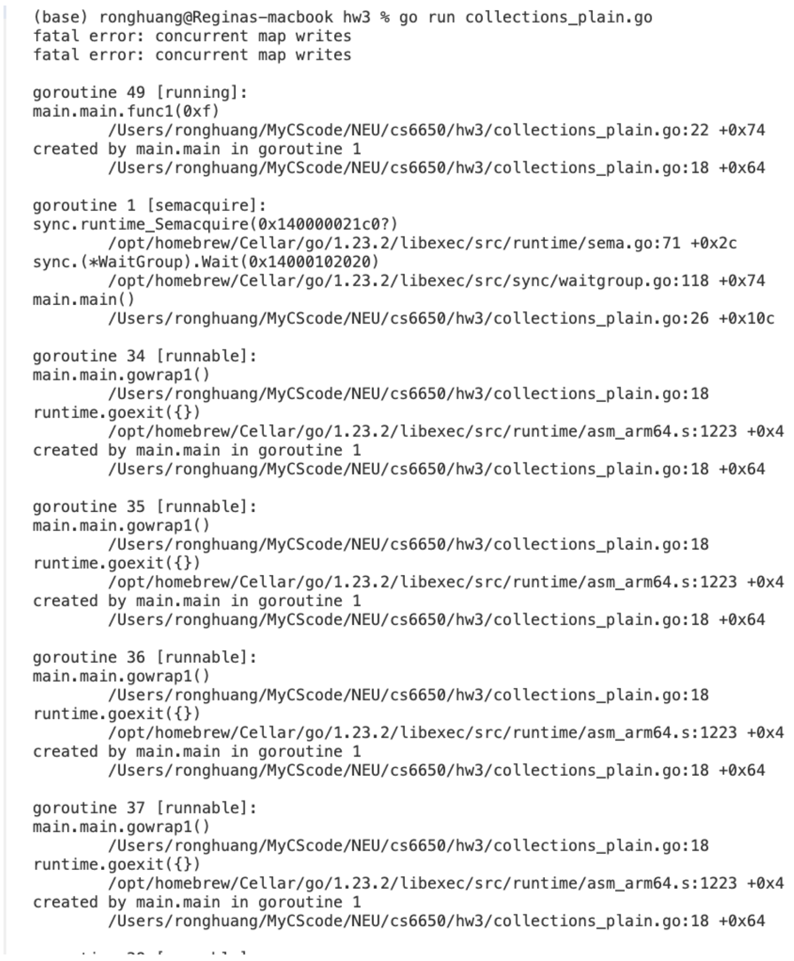
Code

- `collections_plain.go`: Uses a plain `map[int]int` shared across 50 goroutines.

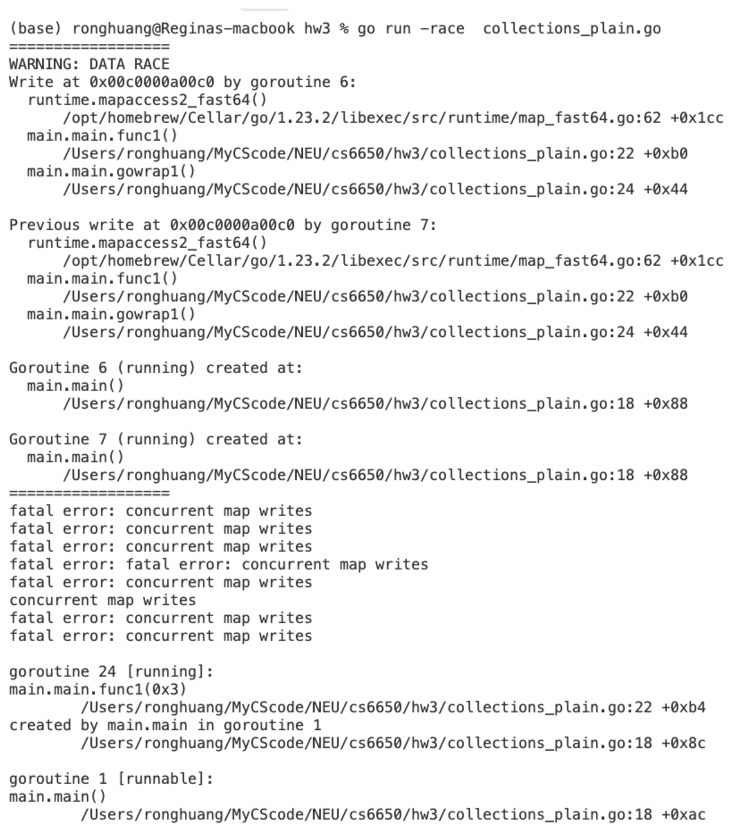
- Each goroutine `g` (0–49) writes 1,000 entries: `m[g*1000 + i] = i` where `i` ranges 0–999.
- Total expected entries: 50,000 distinct key-value pairs.

Results

- Program consistently panics with `fatal error: concurrent map writes`.
- With `race`, the race detector reports concurrent read/write on the same map entry with stack traces.
- Because the runtime stops the program on concurrent writes, no reliable `len(m)` can be observed in my runs.



```
(base) ronghuang@Reginas-macbook hw3 % go run collections_plain.go
fatal error: concurrent map writes
fatal error: concurrent map writes

goroutine 49 [running]:
main.main.func1(0xf)
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:22 +0x74
created by main.main in goroutine 1
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0x64

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0x140000021c0?)
        /opt/homebrew/Cellar/go/1.23.2/libexec/src/runtime/sema.go:71 +0x2c
sync.(*WaitGroup).Wait(0x14000102020)
        /opt/homebrew/Cellar/go/1.23.2/libexec/src/sync/waitgroup.go:118 +0x74
main.main()
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:26 +0x10c

goroutine 34 [runnable]:
main.main.gowrap1()
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18
runtime.goexit({})
        /opt/homebrew/Cellar/go/1.23.2/libexec/src/runtime/asm_arm64.s:1223 +0x4
created by main.main in goroutine 1
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0x64

goroutine 35 [runnable]:
main.main.gowrap1()
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18
runtime.goexit({})
        /opt/homebrew/Cellar/go/1.23.2/libexec/src/runtime/asm_arm64.s:1223 +0x4
created by main.main in goroutine 1
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0x64

goroutine 36 [runnable]:
main.main.gowrap1()
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18
runtime.goexit({})
        /opt/homebrew/Cellar/go/1.23.2/libexec/src/runtime/asm_arm64.s:1223 +0x4
created by main.main in goroutine 1
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0x64

goroutine 37 [runnable]:
main.main.gowrap1()
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18
runtime.goexit({})
        /opt/homebrew/Cellar/go/1.23.2/libexec/src/runtime/asm_arm64.s:1223 +0x4
created by main.main in goroutine 1
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0x64
```



```
(base) ronghuang@Reginas-macbook hw3 % go run -race  collections_plain.go
==================
WARNING: DATA RACE
Write at 0x00c0000a00c0 by goroutine 6:
  runtime.mapaccess2_fast64()
      /opt/homebrew/Cellar/go/1.23.2/libexec/src/runtime/map_fast64.go:62 +0x1cc
  main.main.func1()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:22 +0xb0
  main.main.gowrap1()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:24 +0x44

Previous write at 0x00c0000a00c0 by goroutine 7:
  runtime.mapaccess2_fast64()
      /opt/homebrew/Cellar/go/1.23.2/libexec/src/runtime/map_fast64.go:62 +0x1cc
  main.main.func1()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:22 +0xb0
  main.main.gowrap1()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:24 +0x44

Goroutine 6 (running) created at:
  main.main()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0x88

Goroutine 7 (running) created at:
  main.main()
      /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0x88
==================
fatal error: concurrent map writes
fatal error: concurrent map writes
fatal error: concurrent map writes
fatal error: fatal error: concurrent map writes
fatal error: concurrent map writes
concurrent map writes
fatal error: concurrent map writes
fatal error: concurrent map writes

goroutine 24 [running]:
main.main.func1(0x3)
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:22 +0xb4
created by main.main in goroutine 1
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0x8c

goroutine 1 [runnable]:
main.main()
        /Users/ronghuang/MyCScode/NEU/cs6650/hw3/collections_plain.go:18 +0xac
```

Observations

- Go’s built-in maps are **not safe for concurrent writes**.
- When multiple goroutines write at the same time, the runtime detects this and raises **“fatal error: concurrent map writes”**.
- This happens because map updates involve multiple steps (hashing, bucket placement, resizing), which are **not atomic**.
- As a result, the program may panic before completing, and if it doesn’t, the map length can still be incorrect.

Conclusion

Plain Go maps cannot be used safely with concurrent goroutines. Go chooses to panic rather than risk silent corruption, making the error visible. This experiment shows why synchronization tools (mutex, channels, or `sync.Map`) are required when multiple goroutines share a data structure.

3. Mutex

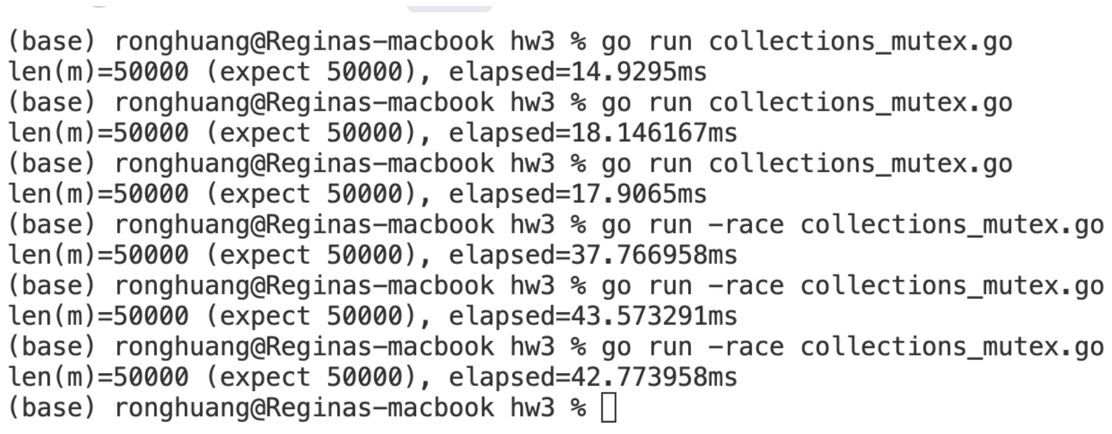
Code

`collections_mutex.go`: wraps a `map[int]int` in a `SafeMap` with a `sync.Mutex`.

- `Set(k,v)`: `Lock` → write → `Unlock`
- `Len()`: `Lock` → `len(m)` → `Unlock`
- 50 goroutines × 1,000 writes each; print `len(m)` and elapsed time.

Results

- Correctness: always `len(m)=50000 (expect 50000)`.
- Timings:
  - Mean elapsed time (no race): ~16.99ms.
  - Mean elapsed time (with `race`): ~41.37 ms.



```
(base) ronghuang@Reginas-macbook hw3 % go run collections_mutex.go
len(m)=50000 (expect 50000), elapsed=14.9295ms
(base) ronghuang@Reginas-macbook hw3 % go run collections_mutex.go
len(m)=50000 (expect 50000), elapsed=18.146167ms
(base) ronghuang@Reginas-macbook hw3 % go run collections_mutex.go
len(m)=50000 (expect 50000), elapsed=17.9065ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_mutex.go
len(m)=50000 (expect 50000), elapsed=37.766958ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_mutex.go
len(m)=50000 (expect 50000), elapsed=43.573291ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_mutex.go
len(m)=50000 (expect 50000), elapsed=42.773958ms
(base) ronghuang@Reginas-macbook hw3 % 
```

Observations

- Mutex enforces exclusive access, preventing crashes and ensuring correctness.
- Lock/unlock on every operation introduces contention and slows execution.
- Race detector amplifies overhead but confirms absence of data races.

Conclusion

Using a `sync.Mutex` around a shared map makes concurrent writes safe and deterministic (`len = 50,000`). The tradeoff is lower throughput from lock contention, but it is the simplest correct solution for safe concurrent access.

4. RWMutex

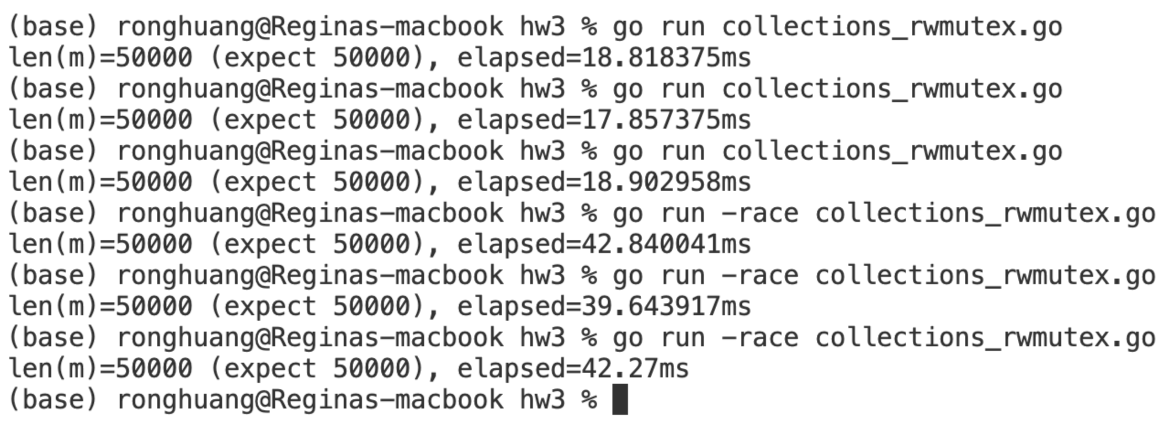
Code

`collections_rwmutex.go`

- Replaces the `sync.Mutex` with a `sync.RWMutex` to allow concurrent reads and exclusive writes.
- Write operations still use `Lock()` for exclusive access when writing.
- Read operation (`Len()`) uses `RLock()` for shared access when reading the map length.

Results

- **Correctness**: Always `len(m) = 50000 (expect 50000)`.
- **Timings**:
  - Mean elapsed time (no race): ~18.53 ms
  - Mean elapsed time (with `race`): ~41.57 ms



```
(base) ronghuang@Reginas-macbook hw3 % go run collections_rwmutex.go
len(m)=50000 (expect 50000), elapsed=18.818375ms
(base) ronghuang@Reginas-macbook hw3 % go run collections_rwmutex.go
len(m)=50000 (expect 50000), elapsed=17.857375ms
(base) ronghuang@Reginas-macbook hw3 % go run collections_rwmutex.go
len(m)=50000 (expect 50000), elapsed=18.902958ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_rwmutex.go
len(m)=50000 (expect 50000), elapsed=42.840041ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_rwmutex.go
len(m)=50000 (expect 50000), elapsed=39.643917ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_rwmutex.go
len(m)=50000 (expect 50000), elapsed=42.27ms
(base) ronghuang@Reginas-macbook hw3 % 
```

Observations

- **Did this change anything?** No. Performance got 9% worse.
- **Why not?** All 50,000 operations are writes. RWMutex allows multiple concurrent readers, but writers still need exclusive access. Since we only write, every operation needs an exclusive lock just like regular Mutex. The RWMutex adds extra overhead to manage read vs write states but provides no benefit.

Conclusion

Lesson learned: RWMutex only helps when you have many reads that can run concurrently. For write only workloads, RWMutex is slower than Mutex due to extra overhead with no parallelism gain. Choose synchronization based on your workload: use RWMutex for read heavy scenarios, use simple Mutex for write heavy or balanced scenarios.

5. Sync.Map

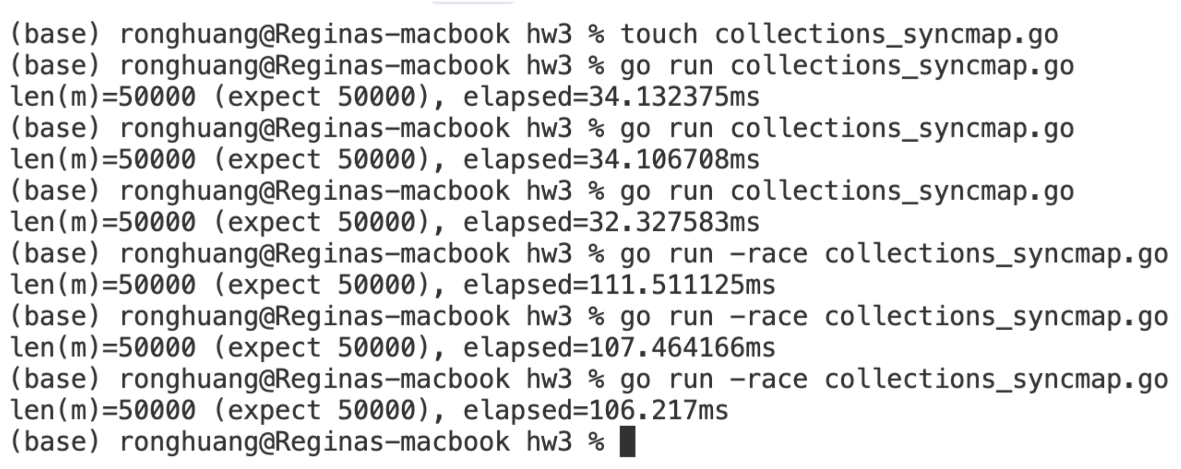
Code

collections_syncmap.go

- Used `sync.Map` directly without external locking
- `Store(key, value)` for writes
- `Range()` to count entries

Results

- **Correctness**: Always `len(m) = 50000 (expect 50000)`.
- **Timings**:
  - Mean elapsed time (no race): ~33.52 ms
  - Mean elapsed time (with `race`): ~108.40 ms



```
(base) ronghuang@Reginas-macbook hw3 % touch collections_syncmap.go
(base) ronghuang@Reginas-macbook hw3 % go run collections_syncmap.go
len(m)=50000 (expect 50000), elapsed=34.132375ms
(base) ronghuang@Reginas-macbook hw3 % go run collections_syncmap.go
len(m)=50000 (expect 50000), elapsed=34.106708ms
(base) ronghuang@Reginas-macbook hw3 % go run collections_syncmap.go
len(m)=50000 (expect 50000), elapsed=32.327583ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_syncmap.go
len(m)=50000 (expect 50000), elapsed=111.511125ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_syncmap.go
len(m)=50000 (expect 50000), elapsed=107.464166ms
(base) ronghuang@Reginas-macbook hw3 % go run -race collections_syncmap.go
len(m)=50000 (expect 50000), elapsed=106.217ms
(base) ronghuang@Reginas-macbook hw3 % █
```

Observations

- sync.Map is slowest for this write only workload (2x slower than Mutex)
- No explicit locking code needed, cleaner API
- Uses internal sharding and atomic operations which add overhead

Conclusion

Here’s a quick comparison of the three approaches used in this experiment:

Approach	Use Case	**Performance**	Correctness	Best Use Case
Plain Map	No synchronization, directly shared map	Crashes on concurrent writes	Fatal error	Not suitable for concurrent write-heavy operations.
Mutex	Single writer, single reader	16.99ms(baseline)	Correct, deterministic	Suitable for **write-heavy** workloads, but slower due to locking overhead.
RWMutex	Many readers, single writer	18.53ms(1.09x slower)	Correct, deterministic	Suitable for **read-heavy** workloads, but adds overhead for write-heavy tasks.
Sync.Map	Multiple concurrent readers and writers	33.52ms(1.97x slower)	Correct, deterministic	Optimized for concurrent access where writes are distributed across goroutines.

- **Mutex**: Fastest for write heavy workloads. Simple and efficient when contention is manageable.
- **RWMutex**: No benefit for writes. Would excel with many concurrent reads.
- **sync.Map**: Slowest for pure writes but shines in two cases: (1) when different goroutines access different keys, (2) write once, read many patterns.

If reads dominated: RWMutex would become fastest (parallel reads). sync.Map would also improve relative to Mutex.

Lesson learned: sync.Map's overhead is only justified for specific patterns. For general concurrent map access with mixed operations, Mutex is often the best choice. Use sync.Map for cache scenarios or when goroutines work on disjoint key sets.

6. File Access

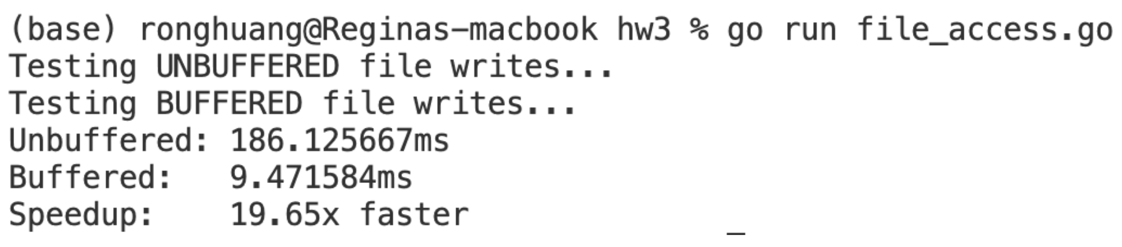
Code

- Unbuffered: Direct `f.Write([]byte(...))` for each iteration

- Buffered: `bufio.Writer` with `w.WriteString(...)` then `w.Flush()`
- Both write 100,000 identical lines

Results

- Unbuffered: 186.13ms
- Buffered: 9.47ms
- Speedup: 19.65x faster



```
(base) ronghuang@Reginas-macbook hw3 % go run file_access.go
Testing UNBUFFERED file writes...
Testing BUFFERED file writes...
Unbuffered: 186.125667ms
Buffered:   9.471584ms
Speedup:    19.65x faster
```

Observations

- Buffered I/O is dramatically faster (nearly 20x improvement)
- Unbuffered makes 100,000 system calls (one per write)
- Buffered I/O reduces system calls, since it accumulates data in a 4KB memory buffer and only makes system calls when the buffer is full or flushed.

Conclusion

What's happening? Every f.Write() triggers a system call, causing a switch from user space to kernel space and back, which adds significant overhead. With 100,000 writes, unbuffered I/O incurs this overhead 100,000 times. Buffered I/O, on the other hand, accumulates data in memory and only triggers a system call when necessary, dramatically reducing the frequency of system calls.

Lesson learned: I/O operations can be costly due to system call overhead. Buffering trades memory (using a 4KB buffer by default) for speed, providing a significant performance boost —up to 20x faster in this case. This principle extends to distributed systems as well, where batching network requests reduces round-trip time, similar to how buffering reduces syscall overhead. Unless immediate persistence is critical (such as for error logs or financial transactions), always opt for buffered I/O for better performance.

7. Context Switching

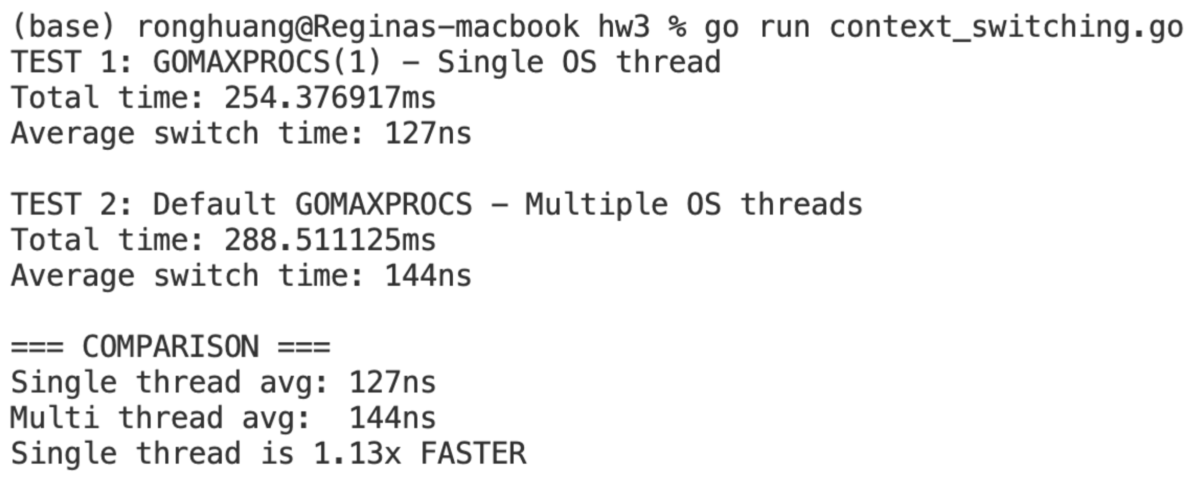
Code

context_switching.go

- Ping-pong between 2 goroutines using unbuffered channels
- 1 million round trips (2 million context switches)
- Test with GOMAXPROCS(1) vs GOMAXPROCS(NumCPU)

Results

- **Single thread (GOMAXPROCS=1)**
  - Total time: 254.38ms
  - Avg switch: 127ns
- **Multiple threads (GOMAXPROCS=8)**
  - Total time: 288.51ms
  - Avg switch: 144ns
- **Winner**: Single thread is 1.13x faster



```
(base) ronghuang@Reginas-macbook hw3 % go run context_switching.go
TEST 1: GOMAXPROCS(1) - Single OS thread
Total time: 254.376917ms
Average switch time: 127ns

TEST 2: Default GOMAXPROCS - Multiple OS threads
Total time: 288.511125ms
Average switch time: 144ns

=== COMPARISON ===
Single thread avg: 127ns
Multi thread avg:  144ns
Single thread is 1.13x FASTER
```

Observations

- Single OS thread performs better for pure channel communication (127ns vs 144ns)
- 17ns difference per switch adds up over 2 million switches
- Goroutines on same OS thread switch via user space scheduler
- Goroutines on different OS threads require kernel involvement and CPU cache synchronization

Conclusion

Why is single thread faster? When goroutines run on the same OS thread, switching is just a function call in user space. Go's scheduler moves the stack pointer and program counter without kernel involvement. With multiple OS threads, channel operations must synchronize across CPU cores, involving memory barriers and potential cache invalidation.

Lesson learned: Goroutine switching is faster than OS thread context switches. This experiment shows even goroutine switches get slower when crossing OS thread boundaries. This explains why Go can efficiently handle millions of goroutines but only thousands of OS threads. For communication heavy workloads, sometimes less parallelism (fewer OS threads) gives better performance due to reduced synchronization overhead.

Part III: Making Threads work hard with Load-Testing!

1. Locust

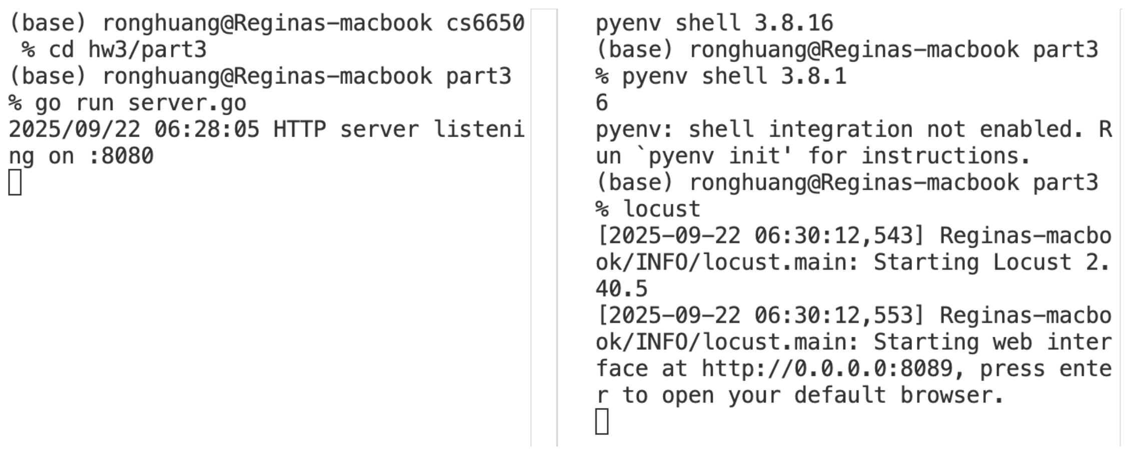
Code

server.go, locustfile.py

- **Server**: HTTP server with RWMutex protecting a map-based key-value store
- **Locustfile**: Equal-weight tasks testing GET and POST endpoints with 1-2 second wait times
- **Configuration**: Single user, single worker to establish baseline performance

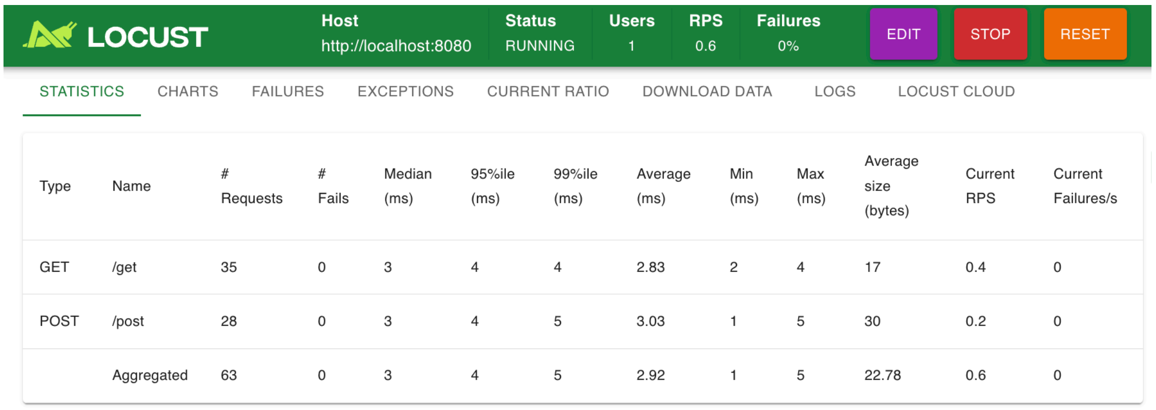
Results

- **GET**: 35 requests, 2.83ms average, 4ms at 99%ile
- **POST**: 28 requests, 3.03ms average, 5ms at 99%ile
- **Failures**: 0% - no errors encountered
- **RPS**: 0.6 requests per second (limited by wait_time)



```
(base) ronghuang@Reginas-macbook cs6650
 % cd hw3/part3
(base) ronghuang@Reginas-macbook part3
% go run server.go
2025/09/22 06:28:05 HTTP server listeni
ng on :8080
```

```
pyenv shell 3.8.16
(base) ronghuang@Reginas-macbook part3
% pyenv shell 3.8.1
6
pyenv: shell integration not enabled. R
un `pyenv init' for instructions.
(base) ronghuang@Reginas-macbook part3
% locust
[2025-09-22 06:30:12,543] Reginas-macbo
ok/INFO/locust.main: Starting Locust 2.
40.5
[2025-09-22 06:30:12,553] Reginas-macbo
ok/INFO/locust.main: Starting web inter
face at http://0.0.0.0:8089, press ente
r to open your default browser.
```



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	35	0	3	4	4	2.83	2	4	17	0.4	0
POST	/post	28	0	3	4	5	3.03	1	5	30	0.2	0
	Aggregated	63	0	3	4	5	2.92	1	5	22.78	0.6	0

Observations

Do you see any failures? No failures observed. RWMutex ensures thread safety even under concurrent requests.

GET vs POST Performance: GET is slightly faster (2.83ms vs 3.03ms). The difference is minimal with one user because there's no lock contention. GET uses `RLock()` which allows concurrent reads, while POST uses `Lock()` for exclusive access. This advantage becomes more pronounced with multiple users.

Percentiles Matter: While averages are similar, the 99%ile shows POST has higher variance (5ms vs 4ms). This indicates occasional slower requests, likely when garbage collection or other system activities coincide with the exclusive lock.

Conclusion

Real-world Implications: Most applications are read-heavy (80/20 rule), making RWMutex ideal. Read operations dominate in scenarios like web content, user profiles, and product catalogs. This impacts our data structure choice - RWMutex over regular Mutex provides better scalability for typical workloads.

Tradeoffs:

- RWMutex adds complexity but enables read parallelism
- Single user tests don't reveal concurrency benefits
- Green threads (Locust) generate load efficiently, similar to goroutines vs OS threads from Part II

The test confirms our server handles basic load correctly. Next steps involve testing with 50 users to observe real concurrency effects and the 3:1 GET/POST ratio to simulate production traffic patterns.

2. Local Test

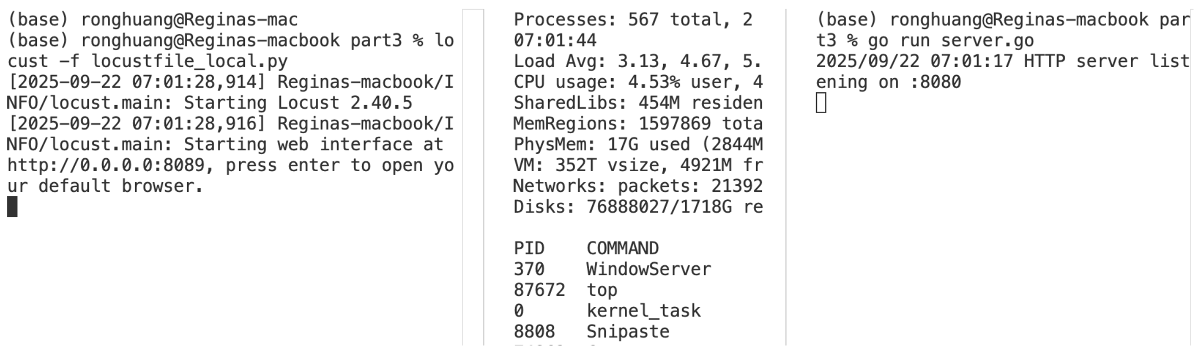
Code

locustfile_local.py

- Configuration: Modified locustfile with `@task(3)` for GET and `@task(1)` for POST to achieve 3:1 ratio
- Wait time: Reduced to `between(0.5, 1.5)` for higher load generation
- Setup: 50 users, 10 users/second ramp-up, single worker

Results

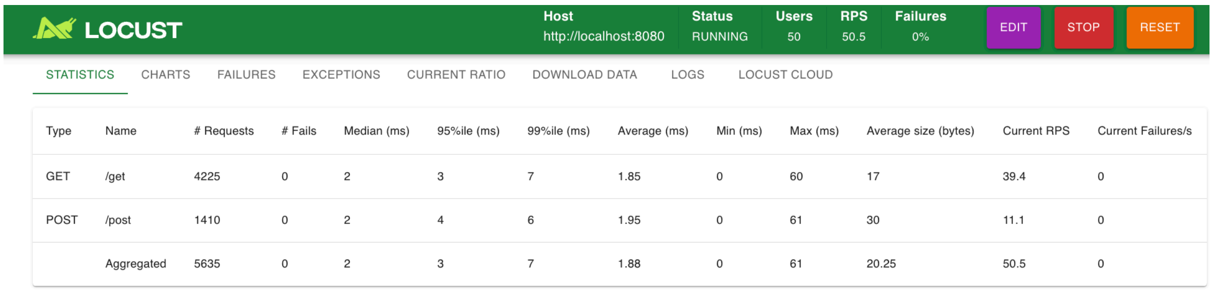
- **GET**: 4225 requests, 1.85ms average, 7ms 99%ile, 39.4 RPS
- **POST**: 1410 requests, 1.95ms average, 6ms 99%ile, 11.1 RPS
- **Ratio achieved**: 3:1 (exactly as configured)
- **Total RPS**: 50.5 with 0% failures
- **CPU usage**: Remained under 10% throughout test



```
(base) ronghuang@Reginas-mac
(base) ronghuang@Reginas-macbook part3 % lo
cust -f locustfile_local.py
[2025-09-22 07:01:28,914] Reginas-macbook/I
NFO/locust.main: Starting Locust 2.40.5
[2025-09-22 07:01:28,916] Reginas-macbook/I
NFO/locust.main: Starting web interface at
http://0.0.0.0:8089, press enter to open yo
ur default browser.
```

```
Processes: 567 total, 2
07:01:44
Load Avg: 3.13, 4.67, 5.
CPU usage: 4.53% user, 4
SharedLibs: 454M residen
MemRegions: 1597869 tota
PhysMem: 17G used (2844M
VM: 352T vsize, 4921M fr
Networks: packets: 21392
Disks: 76888027/1718G re

PID    COMMAND
370    WindowServer
87672  top
0      kernel_task
8808   Snipaste
```

```
(base) ronghuang@Reginas-macbook par
t3 % go run server.go
2025/09/22 07:01:17 HTTP server list
ening on :8080
```



Host	Status	Users	RPS	Failures
http://localhost:8080	RUNNING	50	50.5	0%

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	4225	0	2	3	7	1.85	0	60	17	39.4	0
POST	/post	1410	0	2	4	6	1.95	0	61	30	11.1	0
	Aggregated	5635	0	2	3	7	1.88	0	61	20.25	50.5	0

Observations

With 50 users hitting the server, GET performs slightly better than POST (1.85ms vs 1.95ms). The difference is smaller than expected - probably because the operations are so simple that locks barely matter. CPU stayed under 10%, so the system wasn't even stressed. The 3:1 ratio worked perfectly, mimicking how most real apps have way more reads than writes.

Conclusion

Locust made it easy to simulate 50 concurrent users without writing complex threading code. The server handled 50+ RPS no problem, with zero failures showing the RWMutex prevents races. The small performance gap between GET and POST shows that for simple operations, lock overhead is minimal. This baseline test proves the server can handle some concurrent load.

3. Amdahl's Law

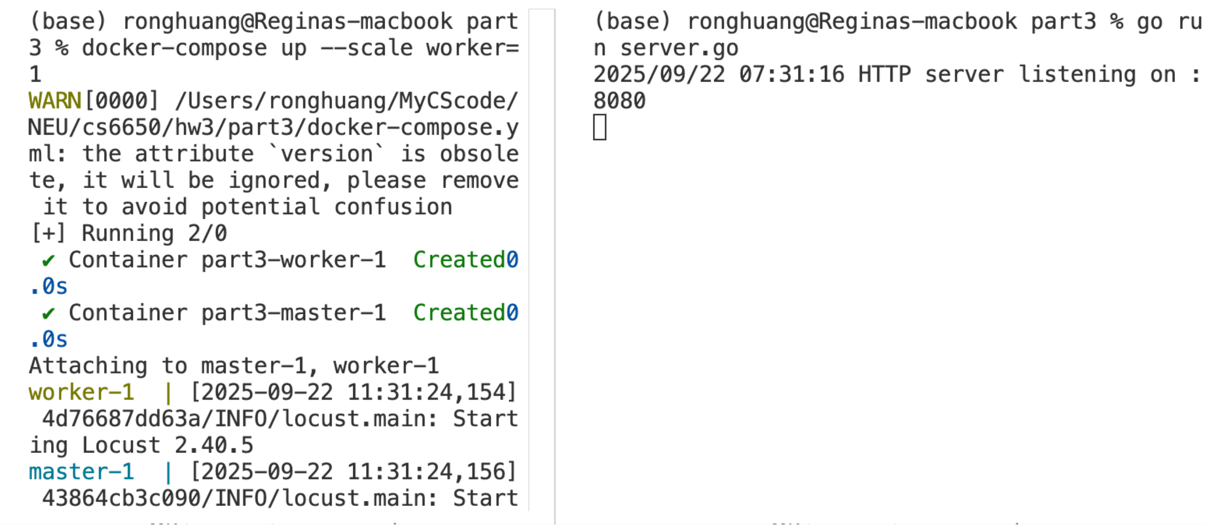
Code

- Docker Compose with 1 vs 4 workers
- Same test parameters (50 users, 3:1 ratio)

Results

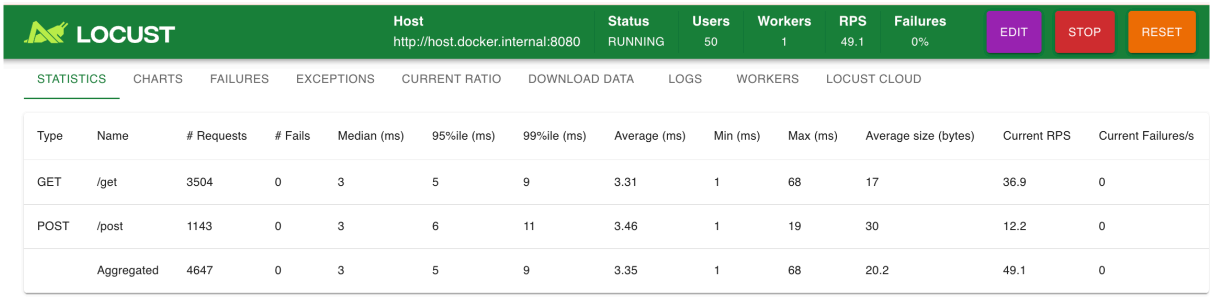
- **1 Worker**: 49.1 RPS, 3.35ms avg
- **4 Workers**: 49.2 RPS, 3.61ms avg
- **Speedup**: 1.002x (essentially zero)

1 worker



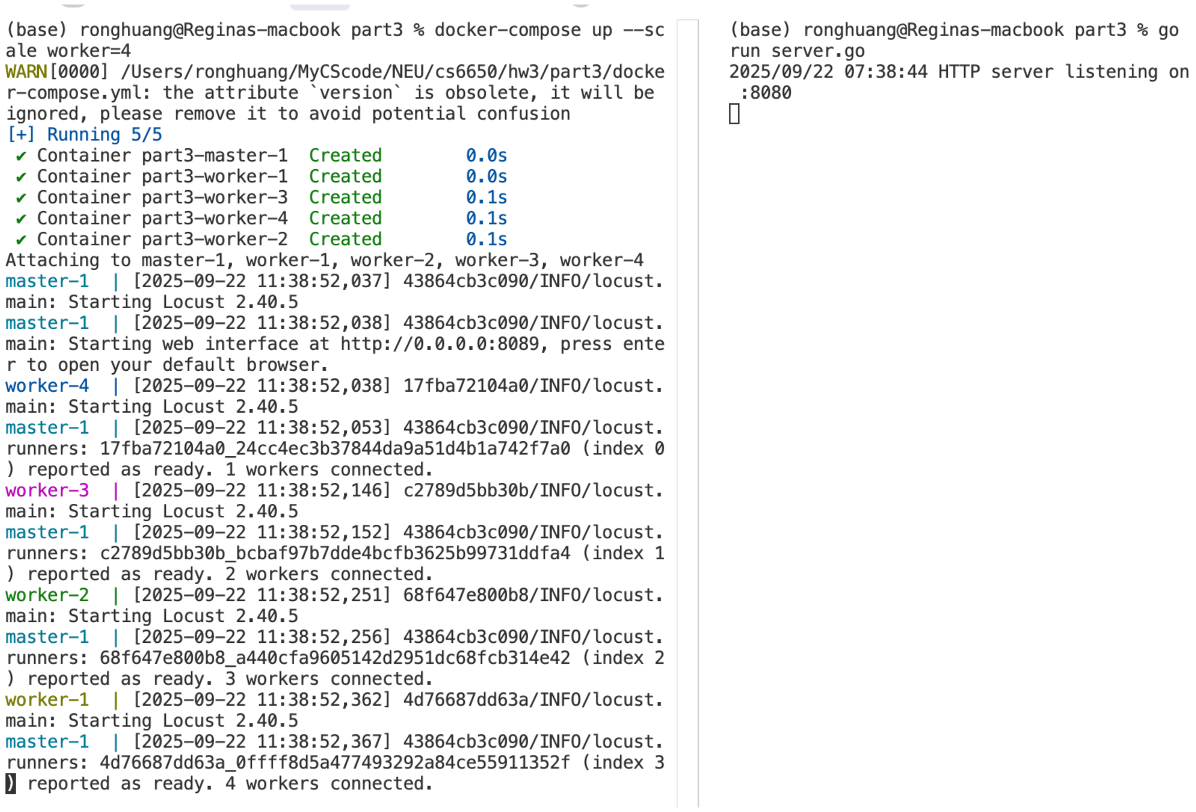
```
(base) ronghuang@Reginas-macbook part
3 % docker-compose up --scale worker=
1
WARN[0000] /Users/ronghuang/MyCScode/
NEU/cs6650/hw3/part3/docker-compose.y
ml: the attribute `version` is obsole
te, it will be ignored, please remove
 it to avoid potential confusion
[+] Running 2/0
 ✔ Container part3-worker-1  Created0
.0s
 ✔ Container part3-master-1  Created0
.0s
Attaching to master-1, worker-1
worker-1  | [2025-09-22 11:31:24,154]
 4d76687dd63a/INFO/locust.main: Start
ing Locust 2.40.5
master-1  | [2025-09-22 11:31:24,156]
 43864cb3c090/INFO/locust.main: Start
```

```
(base) ronghuang@Reginas-macbook part3 % go ru
n server.go
2025/09/22 07:31:16 HTTP server listening on :
8080
```



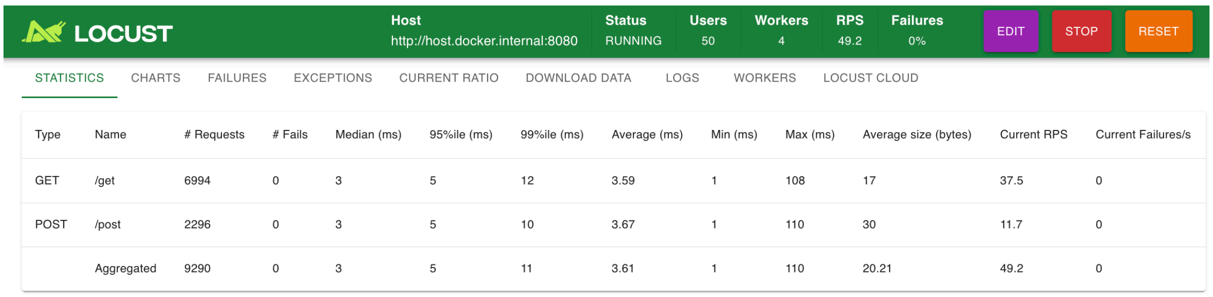
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/get	3504	0	3	5	9	3.31	1	68	17	36.9	0
POST	/post	1143	0	3	6	11	3.46	1	19	30	12.2	0
	Aggregated	4647	0	3	5	9	3.35	1	68	20.2	49.1	0

4 worker



```
(base) ronghuang@Reginas-macbook part3 % docker-compose up --sc
ale worker=4
WARN[0000] /Users/ronghuang/MyCScode/NEU/cs6650/hw3/part3/docke
r-compose.yml: the attribute `version` is obsolete, it will be
ignored, please remove it to avoid potential confusion
[+] Running 5/5
 ✔ Container part3-master-1  Created         0.0s
 ✔ Container part3-worker-1  Created         0.0s
 ✔ Container part3-worker-3  Created         0.1s
 ✔ Container part3-worker-4  Created         0.1s
 ✔ Container part3-worker-2  Created         0.1s
Attaching to master-1, worker-1, worker-2, worker-3, worker-4
master-1  | [2025-09-22 11:38:52,037] 43864cb3c090/INFO/locust.
main: Starting Locust 2.40.5
master-1  | [2025-09-22 11:38:52,038] 43864cb3c090/INFO/locust.
main: Starting web interface at http://0.0.0.0:8089, press ente
r to open your default browser.
worker-4  | [2025-09-22 11:38:52,038] 17fba72104a0/INFO/locust.
main: Starting Locust 2.40.5
master-1  | [2025-09-22 11:38:52,053] 43864cb3c090/INFO/locust.
runners: 17fba72104a0_24cc4ec3b37844da9a51d4b1a742f7a0 (index 0
) reported as ready. 1 workers connected.
worker-3  | [2025-09-22 11:38:52,146] c2789d5bb30b/INFO/locust.
main: Starting Locust 2.40.5
master-1  | [2025-09-22 11:38:52,152] 43864cb3c090/INFO/locust.
runners: c2789d5bb30b_bcbaf97b7dde4bcfb3625b99731ddfa4 (index 1
) reported as ready. 2 workers connected.
worker-2  | [2025-09-22 11:38:52,251] 68f647e800b8/INFO/locust.
main: Starting Locust 2.40.5
master-1  | [2025-09-22 11:38:52,256] 43864cb3c090/INFO/locust.
runners: 68f647e800b8_a440cfa9605142d2951dc68fcb314e42 (index 2
) reported as ready. 3 workers connected.
worker-1  | [2025-09-22 11:38:52,362] 4d76687dd63a/INFO/locust.
main: Starting Locust 2.40.5
master-1  | [2025-09-22 11:38:52,367] 43864cb3c090/INFO/locust.
runners: 4d76687dd63a_0ffff8d5a477493292a84ce55911352f (index 3
) reported as ready. 4 workers connected.
```

```
(base) ronghuang@Reginas-macbook part3 % go
run server.go
2025/09/22 07:38:44 HTTP server listening on
 :8080
```



Host	Status	Users	Workers	RPS	Failures
http://host.docker.internal:8080	RUNNING	50	4	49.2	0%

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	6994	0	3	5	12	3.59	1	108	17	37.5	0
POST	/post	2296	0	3	5	10	3.67	1	110	30	11.7	0
	Aggregated	9290	0	3	5	11	3.61	1	110	20.21	49.2	0

Observations

Expected 4x throughput with 4x workers. Got 0.2% improvement. The server, not load generation, is the bottleneck. All workers hit the same RWMutex-protected hashmap. More workers just mean more contention for the same lock, slightly increasing response times without improving throughput.

Conclusion

Amdahl's Law states speedup is limited by sequential portions. Here, the server's synchronized map access is that sequential bottleneck. No amount of parallel load generation (more workers) can overcome the server's inherent serialization. This mirrors real distributed systems where databases become bottlenecks regardless of application server count.

4. Context Switching

Code

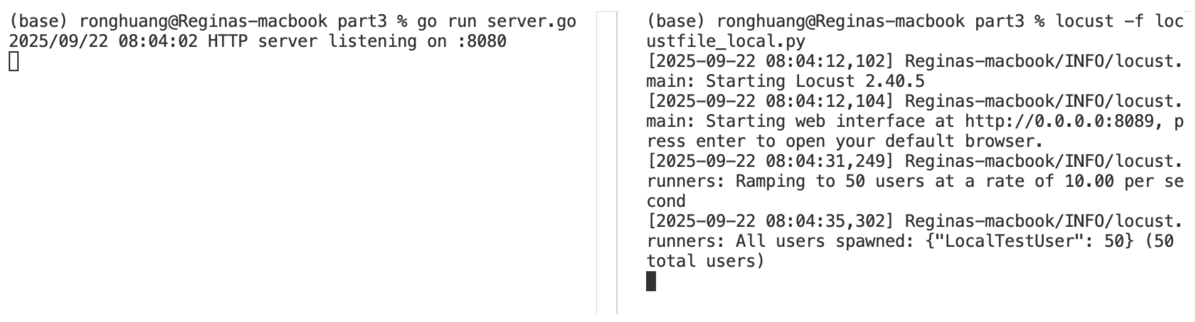
locustfile_fast.py

- Changed `HttpUser` to `FastHttpUser` in locustfile
- Same test parameters (50 users, 3:1 ratio)

Results

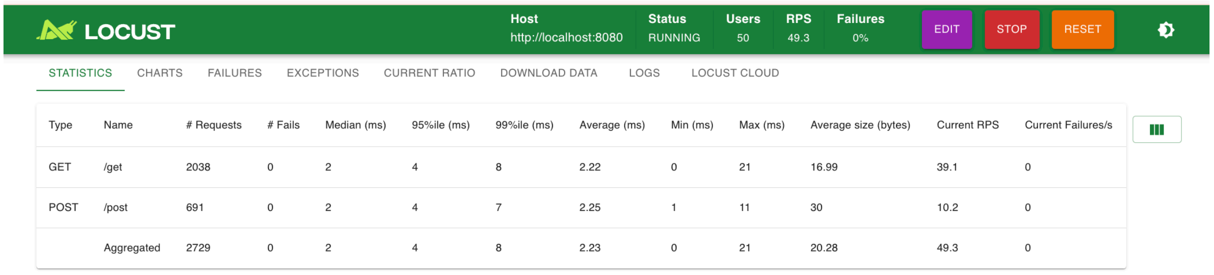
- **HttpUser**: 49.3 RPS, 2.23ms avg response
- **FastHttpUser**: 48.6 RPS, 0.95ms avg response
- **Improvement**: 57% faster response time, similar RPS

httpuser



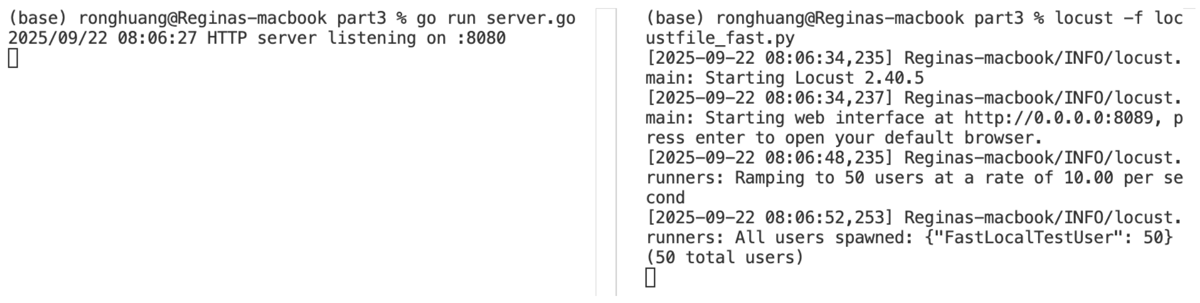
```
(base) ronghuang@Reginas-macbook part3 % go run server.go
2025/09/22 08:04:02 HTTP server listening on :8080
```

```
(base) ronghuang@Reginas-macbook part3 % locust -f loc
ustfile_local.py
[2025-09-22 08:04:12,102] Reginas-macbook/INFO/locust.
main: Starting Locust 2.40.5
[2025-09-22 08:04:12,104] Reginas-macbook/INFO/locust.
main: Starting web interface at http://0.0.0.0:8089, p
ress enter to open your default browser.
[2025-09-22 08:04:31,249] Reginas-macbook/INFO/locust.
runners: Ramping to 50 users at a rate of 10.00 per se
cond
[2025-09-22 08:04:35,302] Reginas-macbook/INFO/locust.
runners: All users spawned: {"LocalTestUser": 50} (50
total users)
```



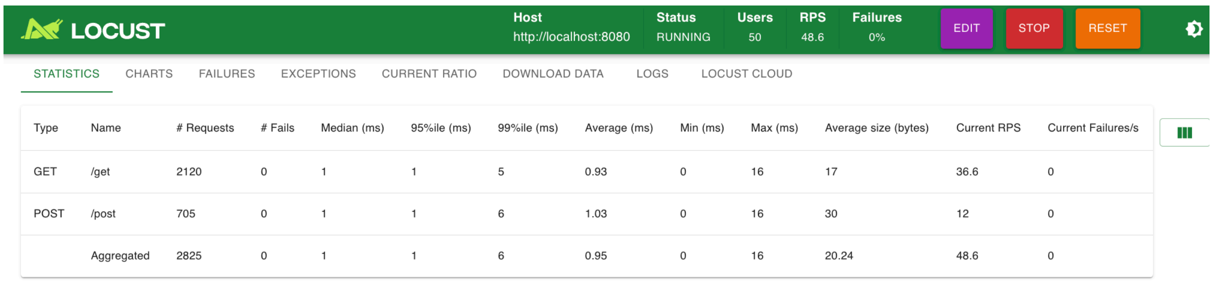
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/get	2038	0	2	4	8	2.22	0	21	16.99	39.1	0
POST	/post	691	0	2	4	7	2.25	1	11	30	10.2	0
	Aggregated	2729	0	2	4	8	2.23	0	21	20.28	49.3	0

fasthttpuser



```
(base) ronghuang@Reginas-macbook part3 % go run server.go
2025/09/22 08:06:27 HTTP server listening on :8080
```

```
(base) ronghuang@Reginas-macbook part3 % locust -f loc
ustfile_fast.py
[2025-09-22 08:06:34,235] Reginas-macbook/INFO/locust.
main: Starting Locust 2.40.5
[2025-09-22 08:06:34,237] Reginas-macbook/INFO/locust.
main: Starting web interface at http://0.0.0.0:8089, p
ress enter to open your default browser.
[2025-09-22 08:06:48,235] Reginas-macbook/INFO/locust.
runners: Ramping to 50 users at a rate of 10.00 per se
cond
[2025-09-22 08:06:52,253] Reginas-macbook/INFO/locust.
runners: All users spawned: {"FastLocalTestUser": 50}
(50 total users)
```



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	2120	0	1	1	5	0.93	0	16	17	36.6	0
POST	/post	705	0	1	1	6	1.03	0	16	30	12	0
	Aggregated	2825	0	1	1	6	0.95	0	16	20.24	48.6	0

Observations

FastHttpUser dramatically reduced response times (2.23ms to 0.95ms) but RPS stayed constant around 49. This shows the bottleneck is the server, not the client. FastHttpUser processes responses more efficiently (C-based vs pure Python), reducing client-side latency, but can't overcome the server's ~49 RPS limit.

Conclusion

FastHttpUser's efficiency mirrors Part II's goroutines vs OS threads, lighter implementation reduces overhead. The 57% response time improvement means less CPU usage on the load generator, allowing you to simulate more users from the same machine. However, actual throughput depends on the server's capacity, not just client efficiency. For load testing at scale, FastHttpUser lets you generate more load with fewer resources.