**Samonorom Chum**

**Wednesday May 8, 2025**

**EECE 4811/5811 Operating System Spring 2025**

**Professor Tseng**

**HW8**

1. **Safe Logger: Exploring Synchronization and fsync in Go (6 pt)**

* **Overview**: In this assignment, you will build a simplified logging service in Go that writes log entries to a file. The logger will support concurrent access from multiple goroutines, simulating client requests. You will implement three different versions of the logger to explore synchronization and the importance of flushing data to disk with fsync.
* **Objectives/Concepts**: By the end of this assignment, you should be able to:
  + Understand the use and impact of fsync for ensuring data durability.
  + Explore synchronization in Go using mutexes vs. channels.
  + Observe the effect of race conditions and data corruption when synchronization is missing or incorrect.
  + Build basic I/O programs in Go that reflect real-world systems behavior.
* **Log entry format**
  + Each log entry should include:

1. Log level (INFO, WARN, ERROR)
2. Context (e.g., request ID or user ID)
3. Message

That is the format is [timestamp] [LEVEL] [context] message

* Example:

[2025-05-01 14:02:11] [INFO] [req-723] Starting processing

* Note: You can generate random or fixed strings for the log level, context, and message.

**Specification of your THREE loggers:**

■ **Naive Logger**:

1. No synchronization.
2. Observe what happens when multiple goroutines write to the file concurrently.
3. Use fsync after every write.

■ **Mutex Logger**:

1. Use a global sync.Mutex to protect file writes.
2. Use fsync after every write.
3. Add batching: only call fsync after every 10 log entries.

**■ Channel Logger:**

1. Use a chan string or chan LogEntry to send log messages to a dedicated logger goroutine.

(LogEntry is your self-defined data structure.)

1. The logger goroutine handles writing to the file.
2. Use fsync after every write.
3. Add batching: only call fsync after every 10 log entries.

■ Note: Batching means the logger can buffer 10 writes in memory and then flush them to disk with one fsync. You can still write to the file immediately

— just delay the call to fsync.

# File List

* log\_entry.go
* naive\_logger.go
* mutex\_logger.go
* channel\_logger.go
* benchmark.go
* Makefile

# How to Run

1. Within the hw8\_logger/ folder, enter the following:

**“make run” in Ubuntu**

1. Alternatively, go run \*.go is the “wildcard that will run all go files in the same folder.

**go run \*.go in Ubuntu**

# Benchmarking:

■ Write a test program to simulate concurrent logging:

1. Create 5 to 10 goroutines, each writing 50 log entries to the logger.
2. Time how long it takes to complete all writes for each logger.
3. Collect and compare the results.

## Benchmark Results

|  |  |  |  |
| --- | --- | --- | --- |
| **Trial** | **NaiveLogger (ms)** | **MutexLogger (ms)** | **ChannelLogger (ms)** |
| 0 | 161.923402 | 54.26341 | 51.005124 |
| 1 | 164.697075 | 44.597054 | 46.476036 |
| 2 | 162.55825 | 48.027388 | 46.169921 |
| 3 | 161.825682 | 48.528398 | 44.926044 |
| 4 | 166.104414 | 46.11543 | 47.148445 |
| 5 | 161.97513 | 45.435929 | 47.544871 |

*Table 1: Benchmark Results.*

# Evaluation

1. **What problems did you observe in the naive logger?**
2. **Did you see corrupted or interleaved output?**
3. **Did any data go missing?**
4. **How can you detect a data race (e.g., using go run -race)?**

* The Naive Logger uses no synchronization, so multiple goroutines access and write to the file concurrently.
* This causes **interleaved and occasionally corrupted log entries**, such as mixed timestamps or incomplete lines.
* It also performs fsync after every write, leading to **significantly higher latency**.
* Running with go run -race \*.go **detects race conditions** in file access, confirming unsafe concurrent behavior.

1. **How did the mutex and channel loggers solve these problems?**

* **Mutex Logger**: Used a sync.Mutex to **serialize access** to the shared file resource. Only one goroutine writes at a time, eliminating data races.
* Additionally, it **batches fsync** every 10 writes, which significantly improves performance.
* Used a chan LogEntry to queue log writes. A dedicated goroutine performs all file I/O and batching. This architecture:
  + Avoid locking in the calling goroutines
  + Ensures all writes are serialized safely
  + Maintains log integrity and high throughput

1. **Compare performance across all three implementations.**
2. **How long did each take to complete?**
3. **How did batching affect performance?**

* ChannelLogger was consistently the fastest due to non-blocking behavior and dedicated file access.
* MutexLogger was a close second, slower only because locking adds slight overhead.
* NaiveLogger was the slowest due to fsync after every write and no batching.

1. **When is fsync necessary? Why is it expensive?**

* fsync ensures that data is flushed from OS buffers to physical disk, making it durable in case of system crash or power failure.
* It is expensive because it blocks the caller until the write is physically completed, which includes waiting for the hardware.
* Calling fsync after every log is inefficient, which is why batching it (every 10 logs) is a performance optimization.

# References:

1. ChatGPT, “Assistance with HW8 Exploring Synchronization and fsync in Go,” OpenAI, personal communication, May. 8, 2025.

# ChatGPT Prompts:

1. HW8 Exploring Synchronization and fsync in Go.
2. Help me get started on HW8.
3. Yes (starter code)
4. generate benchmark.go
5. yes (help with implementation)
6. PDF
7. I'm getting this error msg "./benchmark.go:14:31: undefined: LogEntry ...
8. would having a go.mod affect it?
9. yes, and they all have package main on top, no error in VS Code but still undefined issues when running benchmark.go.
10. yes
11. how do I run Makefile again?
12. Help me generate the report, answer the questions if you can, and make a list of the prompt used for HW8 in a simple plain text list