Concurrent File I/O

File System Basics

Key constraints:

- **Single disk head:** Physical disks can't truly parallelize reads from different locations
- **OS buffering:** Operating system caches frequently accessed files
- **SSDs change game:** Solid-state drives benefit more from concurrency
- **Network filesystems:** NFS,S3 can benefit significantly from parallelism

When concurrency helps:

- Multiple files (each file independent)
- Network I/O (S3, HTTP downloads)
- CPU-bound processing of file data
- SSDs with multiple channels

When concurrency doesn't help:

- Single large file on spinning disk
- Sequential read patterns

Reading Files Concurrently

Multiple Files in Parallel

```go
func readMultipleFiles(filenames []string) (map[string][]byte, error) {
    type result struct {
        filename string
        data     []byte
        err      error
    }

    results := make(chan result, len(filenames))

    // Fan-out: Read files concurrently
    for _, filename := range filenames {
        go func(fn string) {
            data, err := os.ReadFile(fn)
            results <- result{filename: fn, data: data, err: err}
        }(filename)
    }

    // Fan-in: Collect results
    fileData := make(map[string][]byte)
    for range filenames {
        r := <-results
        if r.err != nil {
            return nil, fmt.Errorf("%s: %w", r.filename, r.err)
        }
        fileData[r.filename] = r.data
    }

    return fileData, nil
}
```

Processing Large File in Chunks

```
func processLargeFile(filename string, chunkSize int) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()

    chunks := make(chan []byte, 10)
    results := make(chan error, 1)

    // Start processor goroutines
    var wg sync.WaitGroup
    numWorkers := runtime.NumCPU()

    for i := 0; i < numWorkers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for chunk := range chunks {
                if err := processChunk(chunk); err != nil {
                    select {
                    case results <- err:
                    default:
                    }
                    return
                }
            }
        }()
    }

    // Read file in chunks
    buffer := make([]byte, chunkSize)
    for {
        n, err := file.Read(buffer)
        if n > 0 {
            // Copy buffer (reader reuses it)
            chunk := make([]byte, n)
            copy(chunk, buffer[:n])
            chunks <- chunk
        }

        if err == io.EOF {
            break
        }
        if err != nil {
            close(chunks)
            return err
        }
    }

    close(chunks)
    wg.Wait()

    select {
    case err := <-results:
        return err
    default:
        return nil
    }
}
```

Buffered Reading

```
func readFileBuffered(filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()

    // Buffered reader (default 4KB buffer)
    reader := bufio.NewReader(file)

    for {
        line, err := reader.ReadString('\n')
        if err == io.EOF {
            break
        }
        if err != nil {
            return err
        }

        processLine(line)
    }

    return nil
}
```

Writing Files Concurrently

Multiple Files

```go
func writeMultipleFiles(data map[string][]byte) error {
    errCh := make(chan error, len(data))
    var wg sync.WaitGroup

    for filename, content := range data {
        wg.Add(1)
        go func(fn string, d []byte) {
            defer wg.Done()

            if err := os.WriteFile(fn, d, 0644); err != nil {
                errCh <- fmt.Errorf("%s: %w", fn, err)
            }
        }(filename, content)
    }

    wg.Wait()
    close(errCh)

    // Check for errors
    for err := range errCh {
        return err  // Return first error
    }

    return nil
}
```

Buffered Writing

```go
func writeFileBuffered(filename string, lines []string) error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()

    // Buffered writer (default 4KB buffer)
    writer := bufio.NewWriter(file)
    defer writer.Flush()  // Important!

    for _, line := range lines {
        if _, err := writer.WriteString(line + "\n"); err != nil {
            return err
        }
    }

    return writer.Flush()
}
```

Concurrent Writes to Single File (with Synchronization)

```
type ConcurrentWriter struct {
    file *os.File
    mu   sync.Mutex
}

func NewConcurrentWriter(filename string) (*ConcurrentWriter, error) {
    file, err := os.Create(filename)
    if err != nil {
        return nil, err
    }

    return &ConcurrentWriter{file: file}, nil
}

func (cw *ConcurrentWriter) Write(data []byte) error {
    cw.mu.Lock()
    defer cw.mu.Unlock()

    _, err := cw.file.Write(data)
    return err
}

func (cw *ConcurrentWriter) Close() error {
    cw.mu.Lock()
    defer cw.mu.Unlock()

    return cw.file.Close()
}

// Usage:
writer, _ := NewConcurrentWriter("output.txt")
defer writer.Close()

var wg sync.WaitGroup
for i := 0; i < 10; i++ {
    wg.Add(1)
    go func(n int) {
        defer wg.Done()
        data := fmt.Sprintf("Line %d\n", n)
        writer.Write([]byte(data))
    }(i)
}
wg.Wait()
```

Directory Traversal

Sequential Walk

```
func walkDirectory(root string) ([]string, error) {
    var files []string
    err := filepath.Walk(root, func(path string, info os.FileInfo, err error) error
{
        if err != nil {
            return err
        }

        if !info.IsDir() {
            files = append(files, path)
        }

        return nil
    })

    return files, err
}
```

Concurrent Walk

```
func walkDirectoryConcurrent(root string) ([]string, error) {
    var (
        mu    sync.Mutex
        files []string
        wg    sync.WaitGroup
    )

    err := filepath.Walk(root, func(path string, info os.FileInfo, err error) error
{
        if err != nil {
            return err
        }

        if !info.IsDir() {
            wg.Add(1)
            go func(p string) {
                defer wg.Done()

                // Process file (e.g., read, parse)
                processFile(p)

                // Add to results
                mu.Lock()
                files = append(files, p)
                mu.Unlock()
            }(path)
        }

        return nil
    })

    wg.Wait()
    return files, err
}
```

Parallel Walk with Worker Pool

```
func walkWithWorkerPool(root string, workers int) error {
    type task struct {
        path string
        info os.FileInfo
    }

    tasks := make(chan task, 100)
    var wg sync.WaitGroup

    // Start workers
    for i := 0; i < workers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for t := range tasks {
                processFile(t.path, t.info)
            }
        }()
    }

    // Walk directory
    err := filepath.Walk(root, func(path string, info os.FileInfo, err error) error
{
        if err != nil {
            return err
        }

        if !info.IsDir() {
            tasks <- task{path: path, info: info}
        }

        return nil
    })

    close(tasks)
    wg.Wait()

    return err
}
```

Pipeline: Read → Process → Write

```
func processFilePipeline(inputFiles []string, outputDir string) error {
    // Stage 1: Read files
```

```go
    fileChans := make(chan fileData, len(inputFiles))

    go func() {
        defer close(fileChans)
        for _, filename := range inputFiles {
            data, err := os.ReadFile(filename)
            if err != nil {
                log.Printf("Error reading %s: %v", filename, err)
                continue
            }
            fileChans <- fileData{filename: filename, data: data}
        }
    }()

    // Stage 2: Process (multiple workers)
    processed := make(chan fileData, 10)
    var wg sync.WaitGroup
    numWorkers := runtime.NumCPU()

    for i := 0; i < numWorkers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for fd := range fileChans {
                // Process data
                result := transform(fd.data)
                processed <- fileData{
                    filename: filepath.Join(outputDir, filepath.Base(fd.filename)),
                    data:     result,
                }
            }
        }()
    }

    // Close processed channel when all workers done
    go func() {
        wg.Wait()
        close(processed)
    }()

    // Stage 3: Write results
    for fd := range processed {
        if err := os.WriteFile(fd.filename, fd.data, 0644); err != nil {
            return err
        }
    }

    return nil
}

type fileData struct {
    filename string
    data     []byte
}
```

Real Example: Log File Analyzer

```
type LogAnalyzer struct {
    workers int
}

func (la *LogAnalyzer) Analyze(logFiles []string) (*Stats, error) {
    lines := make(chan string, 1000)
    results := make(chan Stats, la.workers)

    var wg sync.WaitGroup

    // Workers: Process lines
    for i := 0; i < la.workers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()

            stats := Stats{}
            for line := range lines {
                stats.Process(line)
            }

            results <- stats
        }()
    }

    // Read all log files
    for _, filename := range logFiles {
        file, err := os.Open(filename)
        if err != nil {
            close(lines)
            return nil, err
        }

        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
            lines <- scanner.Text()
        }

        file.Close()
    }

    close(lines)
    wg.Wait()
    close(results)
    // Merge results
    finalStats := Stats{}
    for stats := range results {
        finalStats.Merge(stats)
    }

    return &finalStats, nil
}

type Stats struct {
    ErrorCount   int
    WarningCount int
    TotalLines   int
}

func (s *Stats) Process(line string) {
    s.TotalLines++
    if strings.Contains(line, "ERROR") {
        s.ErrorCount++
    } else if strings.Contains(line, "WARN") {
        s.WarningCount++
    }
}

func (s *Stats) Merge(other Stats) {
    s.ErrorCount += other.ErrorCount
    s.WarningCount += other.WarningCount
    s.TotalLines += other.TotalLines
}
```

File Watching

```
import "github.com/fsnotify/fsnotify"

func watchDirectory(dir string) error {
    watcher, err := fsnotify.NewWatcher()
    if err != nil {
        return err
    }
    defer watcher.Close()

    done := make(chan bool)

    go func() {
        for {
            select {
            case event := <-watcher.Events:
                switch {
                case event.Op&fsnotify.Create == fsnotify.Create:
                    log.Println("Created:", event.Name)
                    go processFile(event.Name)

                case event.Op&fsnotify.Write == fsnotify.Write:
                    log.Println("Modified:", event.Name)

                case event.Op&fsnotify.Remove == fsnotify.Remove:
                    log.Println("Removed:", event.Name)
                }

            case err := <-watcher.Errors:
                log.Println("Error:", err)
            }
        }
    }()

    err = watcher.Add(dir)
    if err != nil {
        return err
    }

    <-done
    return nil
}
```

Common Mistakes

Mistake 1: Forgetting to Flush Buffered Writer

```
// WRONG: Data lost
writer := bufio.NewWriter(file)
writer.WriteString("data")
file.Close()  // Doesn't flush buffer!

// Fix:
writer := bufio.NewWriter(file)
writer.WriteString("data")
writer.Flush()  // Flush before closing
file.Close()
```

Mistake 2: Not Checking Scanner Errors

```
// WRONG: Ignores errors
scanner := bufio.NewScanner(file)
for scanner.Scan() {
    process(scanner.Text())
}
// Missing scanner.Err() check!

// Fix:
```

```
scanner := bufio.NewScanner(file)
for scanner.Scan() {
    process(scanner.Text())
}
if err := scanner.Err(); err != nil {
    return err
}
```

Mistake 3: Race on Shared Slice

```
// WRONG: Race condition
var files []string

filepath.Walk(root, func(path string, info os.FileInfo, err error) error {
    go func() {
        files = append(files, path) // RACE!
    }()
    return nil
})

// Fix: Use mutex
var (
    files []string
    mu    sync.Mutex
)

filepath.Walk(root, func(path string, info os.FileInfo, err error) error {
    go func() {
        mu.Lock()
        files = append(files, path)
        mu.Unlock()
    }()
    return nil
})
```

Performance Tips

1. **Use buffered I/O:** `bufio.Reader/Writer` for small reads/writes
2. **Parallelize multiple files** (easy wins)
3. **For single file:** Process after reading, don't try to parallelize reads
4. **Worker pools for processing:** Match to NumCPU
5. **SSDs benefit from concurrency** more than spinning disks

Interview Questions

Q: "When does concurrent file I/O help?"

"Helps when: multiple independent files, network filesystems (S3, NFS), CPU-bound processing after read, SSDs. Doesn't help: single file sequential read on spinning disk (single head can't parallelize). Pattern: read file sequentially, parallelize processing in worker pool."

Q: "How do you handle concurrent writes to same file?"

"File writes aren't thread-safe. Solutions: 1) Mutex around writes (serializes, simple), 2) Channel to single writer goroutine (serializes, clean), 3) Append atomic writes (limited use), 4) Write to separate files then merge. Choose: mutex if simple, channel if better separation."

Q: "What's the difference between os.ReadFile and bufio?"

"os.ReadFile: reads entire file to memory, single call. bufio.Reader: reads in chunks, preserves memory. Use ReadFile for small files (< MB), bufio for large files or line-by-line processing. bufio uses buffer (default 4KB), reduces syscalls."

Key Takeaways

1. **Concurrent file reads help for multiple files**
2. **Single file: read sequential, parallelize processing**
3. **Always flush buffered writers before closing**
4. **Check scanner.Err() after loop**
5. **Concurrent writes need synchronization (mutex/channel)**
6. **Use bufio for large files**
7. **Worker pool for processing file contents**
8. **Directory traversal benefits from parallelism**
9. **SSDs benefit more from concurrency than spinning disks**
10. **File watching enables real-time processing**

Exercises

1. Build concurrent directory walker with worker pool (N workers).

2. Implement log analyzer that processes multiple GB files in parallel.

3. Create pipeline: read CSV → parse → transform → write JSON.

4. Benchmark: Compare sequential vs. concurrent file processing (10 files, 100 files).

5. Build file watcher that processes new files as they arrive.

Next: graceful-shutdown.md - Cleanly shutting down concurrent systems.