Bounded Parallelism

What is Bounded Parallelism?

Problem: Processing N items, but can only handle M concurrent operations.

Examples:

- Process 10,000 files, but only 100 concurrent file handles
- Make 1,000 API calls, but rate limit is 10/sec
- Database batch: 50,000 inserts, but pool has 25 connections

Solution: Limit concurrency while still processing all items.

Semaphore Pattern

```go
func processBatchWithSemaphore(items []Item, maxConcurrent int) error {
    sem := make(chan struct{}, maxConcurrent)
    var wg sync.WaitGroup
    errCh := make(chan error, 1)

    for _, item := range items {
        wg.Add(1)

        // Acquire semaphore
        sem <- struct{}{}

        go func(i Item) {
            defer wg.Done()
            defer func() { <-sem }()  // Release semaphore

            if err := process(i); err != nil {
                select {
                case errCh <- err:
                default:
                }
            }
        }(item)
    }

    wg.Wait()
    close(errCh)

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

    return nil
}
```

Worker Pool Pattern

```go
func processBatchWithWorkerPool(items []Item, numWorkers int) error {
    tasks := make(chan Item, numWorkers*2)
    errors := make(chan error, 1)
    var wg sync.WaitGroup

    // Start workers
    for i := 0; i < numWorkers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for item := range tasks {
                if err := process(item); err != nil {
                    select {
                    case errors <- err:
                    default:
                    }
                    return
                }
            }
        }()
    }

    // Send tasks
    for _, item := range items {
        tasks <- item
    }
    close(tasks)

    // Wait for completion
    wg.Wait()
    close(errors)

    // Check errors
    for err := range errors {
        return err
    }

    return nil
}
```

Rate-Limited Batch Processing

```go
import "golang.org/x/time/rate"

func processBatchWithRateLimit(items []Item, rps int) error {
    limiter := rate.NewLimiter(rate.Limit(rps), rps)
```

```go
    for _, item := range items {
        // Wait for rate limiter
        if err := limiter.Wait(context.Background()); err != nil {
            return err
        }

        if err := process(item); err != nil {
            return err
        }
    }

    return nil
}

// Concurrent + rate limited:
func processBatchConcurrentRateLimited(items []Item, workers, rps int) error {
    limiter := rate.NewLimiter(rate.Limit(rps), rps)
    sem := make(chan struct{}, workers)
    var wg sync.WaitGroup

    for _, item := range items {
        // Wait for rate limit
        if err := limiter.Wait(context.Background()); err != nil {
            return err
        }

        // Acquire semaphore
        sem <- struct{}{}
        wg.Add(1)

        go func(i Item) {
            defer wg.Done()
            defer func() { <-sem }()

            process(i)
        }(item)
    }

    wg.Wait()
    return nil
}
```

Batching with Channels

```go
func processBatchWithChannels(items []Item, batchSize int) error {
    batches := make(chan []Item, 10)

    // Send batches
    go func() {
        defer close(batches)
```

```go
        for i := 0; i < len(items); i += batchSize {
            end := i + batchSize
            if end > len(items) {
                end = len(items)
            }

            batch := items[i:end]
            batches <- batch
        }
    }()

    // Process batches concurrently
    var wg sync.WaitGroup
    numWorkers := runtime.NumCPU()

    for i := 0; i < numWorkers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for batch := range batches {
                processBatch(batch)
            }
        }()
    }

    wg.Wait()
    return nil
}
```

Real Example: Image Processing

```go
func processImages(imageFiles []string, maxConcurrent int) error {
    sem := make(chan struct{}, maxConcurrent)
    var wg sync.WaitGroup

    for _, filename := range imageFiles {
        wg.Add(1)
        sem <- struct{}{}

        go func(fn string) {
            defer wg.Done()
            defer func() { <-sem }()

            // Load image
            img, err := loadImage(fn)
            if err != nil {
                log.Printf("Failed to load %s: %v", fn, err)
                return
            }
```

```go
            // Resize
            resized := resize(img, 800, 600)

            // Save
            outFile := strings.Replace(fn, ".jpg", "_thumb.jpg", 1)
            if err := saveImage(outFile, resized); err != nil {
                log.Printf("Failed to save %s: %v", outFile, err)
            }
        }(filename)
    }

    wg.Wait()
    return nil
}
```

Real Example: API Batch Calls

```go
func fetchUsersInBatches(userIDs []int, batchSize, maxConcurrent int) ([]*User,
error) {
    type result struct {
        users []*User
        err   error
    }

    results := make(chan result, (len(userIDs)/batchSize)+1)
    sem := make(chan struct{}, maxConcurrent)
    var wg sync.WaitGroup

    // Process in batches
    for i := 0; i < len(userIDs); i += batchSize {
        end := i + batchSize
        if end > len(userIDs) {
            end = len(userIDs)
        }

        batch := userIDs[i:end]

        wg.Add(1)
        sem <- struct{}{}

        go func(ids []int) {
            defer wg.Done()
            defer func() { <-sem }()

            // API call with batch
            users, err := api.FetchUsers(ids)
            results <- result{users: users, err: err}
        }(batch)
    }
```

```go
    // Wait and close
    go func() {
        wg.Wait()
        close(results)
    }()

    // Collect results
    var allUsers []*User
    for r := range results {
        if r.err != nil {
            return nil, r.err
        }
        allUsers = append(allUsers, r.users...)
    }

    return allUsers, nil
}
```

Bounded Parallelism with Context

```go
func processBatchWithContext(ctx context.Context, items []Item, workers int) error {
    tasks := make(chan Item, workers)
    errors := make(chan error, 1)
    var wg sync.WaitGroup

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

            for {
                select {
                case <-ctx.Done():
                    return

                case item, ok := <-tasks:
                    if !ok {
                        return
                    }

                    if err := process(ctx, item); err != nil {
                        select {
                        case errors <- err:
                        default:
                        }
                        return
                    }
                }
```

```go
            }
        }()
    }

    // Send tasks
    go func() {
        defer close(tasks)

        for _, item := range items {
            select {
            case <-ctx.Done():
                return
            case tasks <- item:
            }
        }
    }()

    // Wait for completion
    wg.Wait()
    close(errors)

    // Check for errors
    select {
    case err := <-errors:
        return err
    default:
        return ctx.Err()
    }
}
```

Dynamic Worker Count

Adjust workers based on system load.

```go
type DynamicPool struct {
    tasks        chan Task
    minWorkers   int
    maxWorkers   int
    currentWorkers atomic.Int32
    wg           sync.WaitGroup
    quit         chan struct{}
}

func (dp *DynamicPool) Start() {
    // Start minimum workers
    for i := 0; i < dp.minWorkers; i++ {
        dp.addWorker()
    }

    // Monitor queue depth
    go dp.scaleWorkers()
```

```go
}

func (dp *DynamicPool) scaleWorkers() {
    ticker := time.NewTicker(5 * time.Second)
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            queueDepth := len(dp.tasks)
            workers := int(dp.currentWorkers.Load())

            // Scale up if queue backing up
            if queueDepth > workers*2 && workers < dp.maxWorkers {
                dp.addWorker()
                log.Printf("Scaled up to %d workers", workers+1)
            }

            // Scale down if idle
            if queueDepth == 0 && workers > dp.minWorkers {
                // Signal one worker to stop (implementation omitted)
            }

        case <-dp.quit:
            return
        }
    }
}

func (dp *DynamicPool) addWorker() {
    dp.currentWorkers.Add(1)
    dp.wg.Add(1)

    go func() {
        defer dp.wg.Done()
        defer dp.currentWorkers.Add(-1)

        for {
            select {
            case task := <-dp.tasks:
                task.Execute()
            case <-dp.quit:
                return
            }
        }
    }()
}
```

Parallel Map/Reduce

```go
func parallelMap(items []Item, workers int, fn func(Item) Result) []Result {
    tasks := make(chan Item, len(items))
    results := make(chan Result, len(items))

    // Start workers
    var wg sync.WaitGroup
    for i := 0; i < workers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for item := range tasks {
                results <- fn(item)
            }
        }()
    }

    // Send tasks
    for _, item := range items {
        tasks <- item
    }
    close(tasks)

    // Wait and close results
    go func() {
        wg.Wait()
        close(results)
    }()

    // Collect results
    var output []Result
    for result := range results {
        output = append(output, result)
    }

    return output
}

func parallelReduce(items []Item, workers int, fn func(Item, Item) Item) Item {
    if len(items) == 0 {
        var zero Item
        return zero
    }

    // Divide into chunks
    chunkSize := (len(items) + workers - 1) / workers
    results := make(chan Item, workers)

    for i := 0; i < len(items); i += chunkSize {
        end := i + chunkSize
        if end > len(items) {
            end = len(items)
```

```go
        }

        chunk := items[i:end]

        go func(c []Item) {
            acc := c[0]
            for _, item := range c[1:] {
                acc = fn(acc, item)
            }
            results <- acc
        }(chunk)
    }

    // Reduce results
    partials := make([]Item, 0, workers)
    for i := 0; i < workers && i < len(items); i++ {
        partials = append(partials, <-results)
    }

    result := partials[0]
    for _, partial := range partials[1:] {
        result = fn(result, partial)
    }

    return result
}
```

Choosing Worker Count

```go
// CPU-bound work:
workers := runtime.NumCPU()

// I/O-bound work (network, disk):
workers := runtime.NumCPU() * 10

// Memory-intensive:
workers := runtime.NumCPU() / 2

// Rate-limited API:
workers := rateLimit / avgRequestsPerSecond

// Database operations:
workers := db.Stats().MaxOpenConnections

// Benchmark to find optimal:
func BenchmarkWorkers(b *testing.B) {
    workerCounts := []int{1, 2, 4, 8, 16, 32}

    for _, workers := range workerCounts {
        b.Run(fmt.Sprintf("workers=%d", workers), func(b *testing.B) {
```

```go
            for i := 0; i < b.N; i++ {
                processBatch(items, workers)
            }
        })
    }
}
```

Common Mistakes

Mistake 1: Unbounded Concurrency

```go
// WRONG: Creates 1 million goroutines
for _, item := range items {  // 1 million items
    go process(item)  // 1 million goroutines!
}

// Fix: Bounded concurrency
sem := make(chan struct{}, 100)
for _, item := range items {
    sem <- struct{}{}
    go func(i Item) {
        defer func() { <-sem }()
        process(i)
    }(item)
}
```

Mistake 2: Wrong Worker Count

```go
// WRONG: Way too many workers for CPU-bound work
workers := 1000  // But only 8 CPUs!

// Fix: Match to resource
workers := runtime.NumCPU()
```

Mistake 3: Not Waiting for Completion

```go
// WRONG: Returns before processing done
func processBatch(items []Item) {
    for _, item := range items {
        go process(item)
    }
    return  // WRONG: Doesn't wait!
}

// Fix: Wait for goroutines
func processBatch(items []Item) {
    var wg sync.WaitGroup
    for _, item := range items {
```

```go
        wg.Add(1)
        go func(i Item) {
            defer wg.Done()
            process(i)
        }(item)
    }
    wg.Wait()
}
```

Performance Benchmarks

```go
func BenchmarkBoundedVsUnbounded(b *testing.B) {
    items := makeTestItems(1000)

    b.Run("unbounded", func(b *testing.B) {
        for i := 0; i < b.N; i++ {
            var wg sync.WaitGroup
            for _, item := range items {
                wg.Add(1)
                go func(i Item) {
                    defer wg.Done()
                    process(i)
                }(item)
            }
            wg.Wait()
        }
    })

    b.Run("bounded-100", func(b *testing.B) {
        for i := 0; i < b.N; i++ {
            processBatchWithSemaphore(items, 100)
        }
    })
}

// Results (typical):
// unbounded:     High memory, context switch overhead
// bounded-100:   Lower memory, better throughput
```

Interview Questions

Q: "Semaphore vs. worker pool for bounded parallelism?"

"Semaphore: goroutine per task, acquire/release pattern, simple for short tasks, higher memory (goroutine per item). Worker pool: fixed goroutines, task queue, better for long-running, lower memory (fixed goroutines). Choose semaphore for simple short tasks, worker pool for sustained load or long tasks."

Q: "How do you choose optimal worker count?"

"Depends on bottleneck. CPU-bound: NumCPU (avoid oversubscription). I/O-bound: NumCPU * 10-100 (goroutines block on I/O). Memory-intensive: Lower (avoid OOM). Rate-limited: Match rate limit. Database: Match connection pool size. Best practice: Benchmark different worker counts, measure throughput and latency."

Q: "What happens if you don't bound concurrency?"

"Memory exhaustion (each goroutine ~2KB), GC pressure (tracking millions of goroutines), scheduler overhead (context switching), resource exhaustion (file handles, connections), system instability. Example: Processing 1M items unbounded = 2GB + overhead. With 100 workers = 200KB."

Key Takeaways

1. **Always bound concurrency for large batches**
2. **Semaphore: simple, goroutine per task**
3. **Worker pool: efficient, fixed goroutines**
4. **CPU-bound: workers = NumCPU**
5. **I/O-bound: workers = NumCPU * 10-100**
6. **Rate limiting complements bounded parallelism**
7. **Use WaitGroup to wait for completion**
8. **Context for cancellation**
9. **Benchmark to find optimal worker count**
10. **Monitor queue depth and latency**

Exercises

1. Implement bounded batch processor with semaphore, benchmark different limits.

2. Build worker pool with dynamic worker count (scales 1-100 based on queue).

3. Create rate-limited batch API client (100 req/sec, 10 concurrent).

4. Benchmark: Compare semaphore vs. worker pool for 10,000 items.

5. Implement parallel map/reduce with bounded workers.

Next: avoiding-goroutine-leaks.md - Detecting and preventing goroutine leaks.