# Worker Pool Pattern

## What is a Worker Pool?

**Worker pool:** A fixed number of goroutines (workers) that process tasks from a shared queue.

**Purpose:**

- **Bound concurrency:** Limit number of concurrent operations
- **Resource control:** Prevent goroutine explosion
- **Throughput:** Process tasks efficiently without overwhelming system

**When to use:**

- Processing many tasks (thousands to millions)
- Each task independent
- Want to limit resource usage (CPU, memory, connections)

## Basic Implementation

```
func workerPool(numWorkers int, tasks <-chan Task) {
    var wg sync.WaitGroup

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

            // Process tasks until channel closed
            for task := range tasks {
                process(task)
            }
        }(i)
    }

    wg.Wait()  // Wait for all workers to finish
}

// Usage:
tasks := make(chan Task, 100)

go workerPool(10, tasks)  // 10 workers

// Send tasks
for _, t := range allTasks {
    tasks <- t
}
close(tasks)  // Signal no more tasks
```

**Key components:**

1. **Task channel:** Queue of work units
2. **Workers:** Fixed number of goroutines
3. **WaitGroup:** Track worker completion

## Advanced Implementation with Cancellation

```
type WorkerPool struct {
    workers  int
    tasks    chan Task
    results  chan Result
    ctx      context.Context
    cancel   context.CancelFunc
    wg       sync.WaitGroup
}

func NewWorkerPool(workers int, bufferSize int) *WorkerPool {
    ctx, cancel := context.WithCancel(context.Background())

    return &WorkerPool{
        workers: workers,
        tasks:   make(chan Task, bufferSize),
        results: make(chan Result, bufferSize),
        ctx:     ctx,
        cancel:  cancel,
    }
}

func (p *WorkerPool) Start() {
    for i := 0; i < p.workers; i++ {
        p.wg.Add(1)
        go p.worker(i)
    }
}

func (p *WorkerPool) worker(id int) {
    defer p.wg.Done()

    for {
        select {
        case task, ok := <-p.tasks:
            if !ok {
                return  // Channel closed
            }

            result := process(task)

            select {
            case p.results <- result:
            case <-p.ctx.Done():
                return
            }
```

```
        case <-p.ctx.Done():
            return
        }
    }
}

func (p *WorkerPool) Submit(task Task) error {
    select {
    case p.tasks <- task:
        return nil
    case <-p.ctx.Done():
        return p.ctx.Err()
    }
}

func (p *WorkerPool) Shutdown() {
    close(p.tasks)      // No more tasks
    p.wg.Wait()         // Wait for workers
    close(p.results)    // Close results channel
}

func (p *WorkerPool) ShutdownNow() {
    p.cancel()          // Cancel all workers
    p.wg.Wait()
    close(p.results)
}

// Usage:
pool := NewWorkerPool(10, 100)
pool.Start()

// Submit tasks
for _, task := range tasks {
    pool.Submit(task)
}

// Collect results
go func() {
    for result := range pool.results {
        handle(result)
    }
}()

pool.Shutdown()
```

## Pattern Variations

### 1. Buffered vs. Unbuffered Task Channel

**Unbuffered (blocking):**

```
tasks := make(chan Task)  // No buffer

// Producer blocks if no worker ready
task := <-tasks  // Worker blocks if no task ready
```

**Pros:** Natural backpressure (producer can't get ahead)
**Cons:** Tight coupling (producer and worker must sync)

**Buffered:**

```
tasks := make(chan Task, 100)  // Buffer size 100

// Producer doesn't block until buffer full
```

**Pros:** Producer and worker decoupled, better throughput
**Cons:** Can hide backpressure issues, uses more memory

**Rule of thumb:** Buffer size = 1-2x number of workers

### 2. Worker Pool with Priority

```
type PriorityPool struct {
    highPriority chan Task
    lowPriority  chan Task
    // ...
}

func (p *PriorityPool) worker() {
    for {
        select {
        case task := <-p.highPriority:
            process(task)
        default:
            select {
            case task := <-p.highPriority:
                process(task)
            case task := <-p.lowPriority:
                process(task)
            }
        }
    }
}

// High-priority tasks processed first
// Fall back to low-priority if no high-priority tasks
```

### 3. Dynamic Worker Pool (Scaling)

```go
type DynamicPool struct {
    minWorkers int
    maxWorkers int
    current    int
    tasks      chan Task
    mu         sync.Mutex
    // ...
}

func (p *DynamicPool) scaleUp() {
    p.mu.Lock()
    defer p.mu.Unlock()

    if p.current < p.maxWorkers {
        p.current++
        go p.worker()
    }
}

func (p *DynamicPool) scaleDown() {
    p.mu.Lock()
    defer p.mu.Unlock()

    if p.current > p.minWorkers {
        p.current--
        // Send signal to stop one worker
    }
}

// Monitor queue length and scale accordingly
func (p *DynamicPool) monitor() {
    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()

    for range ticker.C {
        queueLen := len(p.tasks)

        if queueLen > p.current*2 {
            p.scaleUp()
        } else if queueLen < p.current/2 {
            p.scaleDown()
        }
    }
}
```

### 4. Worker Pool with Retry

```go
type RetryableTask struct {
    Task
    Retries int
```

```
    MaxRetries int
}

func (p *WorkerPool) worker() {
    for task := range p.tasks {
        err := process(task.Task)

        if err != nil && task.Retries < task.MaxRetries {
            task.Retries++

            // Exponential backoff
            backoff := time.Duration(1<<task.Retries) * time.Second
            time.Sleep(backoff)

            // Requeue
            p.tasks <- task
        } else {
            // Success or max retries reached
            p.results <- Result{Task: task, Err: err}
        }
    }
}
```

## Real-World Example: Image Processing Service

```
package main

import (
    "context"
    "fmt"
    "image"
    "sync"
    "time"
)

type ImageTask struct {
    ID    string
    Image image.Image
}

type ImageResult struct {
    ID        string
    Thumbnail image.Image
    Err       error
}

type ImageProcessor struct {
    workers int
    tasks   chan ImageTask
    results chan ImageResult
```

```
    wg      sync.WaitGroup
    ctx     context.Context
    cancel  context.CancelFunc
}

func NewImageProcessor(workers int) *ImageProcessor {
    ctx, cancel := context.WithCancel(context.Background())

    return &ImageProcessor{
        workers: workers,
        tasks:   make(chan ImageTask, workers*2),
        results: make(chan ImageResult, workers*2),
        ctx:     ctx,
        cancel:  cancel,
    }
}

func (ip *ImageProcessor) Start() {
    for i := 0; i < ip.workers; i++ {
        ip.wg.Add(1)
        go ip.worker(i)
    }
}

func (ip *ImageProcessor) worker(id int) {
    defer ip.wg.Done()

    for {
        select {
        case task, ok := <-ip.tasks:
            if !ok {
                return
            }

            start := time.Now()

            // Simulate processing
            thumbnail := resize(task.Image, 200, 200)

            duration := time.Since(start)
            fmt.Printf("Worker %d processed %s in %v\n",
                id, task.ID, duration)

            result := ImageResult{
                ID:        task.ID,
                Thumbnail: thumbnail,
            }

            select {
            case ip.results <- result:
            case <-ip.ctx.Done():
                return
```

```
            }

        case <-ip.ctx.Done():
            return
        }
    }
}

func (ip *ImageProcessor) Submit(task ImageTask) error {
    select {
    case ip.tasks <- task:
        return nil
    case <-ip.ctx.Done():
        return ip.ctx.Err()
    default:
        return fmt.Errorf("task queue full")
    }
}

func (ip *ImageProcessor) Shutdown(timeout time.Duration) error {
    close(ip.tasks)

    done := make(chan struct{})
    go func() {
        ip.wg.Wait()
        close(done)
    }()

    select {
    case <-done:
        close(ip.results)
        return nil
    case <-time.After(timeout):
        ip.cancel()
        return fmt.Errorf("shutdown timeout")
    }
}

func main() {
    processor := NewImageProcessor(10)
    processor.Start()

    // Submit tasks
    go func() {
        for i := 0; i < 100; i++ {
            task := ImageTask{
                ID:    fmt.Sprintf("image-%d", i),
                Image: loadImage(),
            }
            processor.Submit(task)
        }
    }()
```

```
    // Collect results
    processed := 0
    for result := range processor.results {
        if result.Err != nil {
            fmt.Printf("Error: %v\n", result.Err)
        }
        processed++
        if processed == 100 {
            break
        }
    }

    processor.Shutdown(5 * time.Second)
}
```

## Common Mistakes

### Mistake 1: Forgetting to Close Task Channel

```
// WRONG: Workers wait forever
pool.Submit(task)
// Never closes tasks channel

// Workers:
for task := range tasks {}  // Blocks forever
```

**Fix:**

```
// Close when done
pool.Shutdown()  // Closes tasks channel
```

### Mistake 2: Closing Channel Too Early

```
// WRONG:
close(tasks)
tasks <- newTask  // Panic: send on closed channel
```

**Fix:**

```
// Ensure all sends before close
close(tasks)
```

### Mistake 3: Not Draining Results Channel

```
pool.Shutdown()
// WRONG: Results channel still has data
```

```
// Workers block on send to results
```

**Fix:**

```
// Drain results before shutdown
go func() {
    for range pool.results {}
}()
pool.Shutdown()
```

### Mistake 4: Unbounded Task Queue

```
// WRONG:
tasks := make(chan Task)  // Unbuffered

for _, t := range millionTasks {
    tasks <- t  // Blocks, producers stuck
}
```

**Fix:**

```
tasks := make(chan Task, 1000)  // Buffered
// Or use semaphore pattern
```

## Performance Considerations

### Optimal Worker Count

**CPU-bound tasks:**

```
workers := runtime.NumCPU()
```

**I/O-bound tasks:**

```
workers := runtime.NumCPU() * 10  // Or more
```

**Benchmark to find optimal:**

```
func BenchmarkWorkerPool(b *testing.B) {
    for workers := 1; workers <= 32; workers *= 2 {
        b.Run(fmt.Sprintf("Workers-%d", workers), func(b *testing.B) {
            pool := NewWorkerPool(workers, 100)
            pool.Start()
            // ... benchmark logic
        })
    }
}
```

### Buffer Size

**Too small:** Contention, workers idle
**Too large:** Memory waste, hides backpressure

**Rule:** Buffer = 1-2x workers

### Graceful Shutdown

**Immediate (ShutdownNow):**

- Cancel context
- Workers exit immediately
- Tasks in queue dropped

**Graceful (Shutdown):**

- Close task channel
- Workers finish current tasks
- All queued tasks processed

## Interview Coding Challenge

**Problem:** Implement rate-limited worker pool (max N req/sec).

```
type RateLimitedPool struct {
    workers   int
    tasks     chan Task
    results   chan Result
    rateLimit int  // Requests per second
    limiter   *time.Ticker
    wg        sync.WaitGroup
}

func NewRateLimitedPool(workers, rateLimit int) *RateLimitedPool {
    return &RateLimitedPool{
        workers:   workers,
        tasks:     make(chan Task, workers*2),
        results:   make(chan Result, workers*2),
        rateLimit: rateLimit,
        limiter:   time.NewTicker(time.Second / time.Duration(rateLimit)),
    }
}

func (p *RateLimitedPool) worker() {
    defer p.wg.Done()

    for task := range p.tasks {
        <-p.limiter.C  // Wait for rate limit token
        result := process(task)
        p.results <- result
    }
}
```

```go
func (p *RateLimitedPool) Start() {
    for i := 0; i < p.workers; i++ {
        p.wg.Add(1)
        go p.worker()
    }
}

func (p *RateLimitedPool) Shutdown() {
    close(p.tasks)
    p.wg.Wait()
    p.limiter.Stop()
    close(p.results)
}
```

## Key Takeaways

1. **Worker pool bounds concurrency** (prevent goroutine explosion)
2. **Fixed workers process tasks from shared queue**
3. **Buffer size: 1-2x number of workers**
4. **CPU-bound: workers = NumCPU()**
5. **I/O-bound: workers = NumCPU() * 10+**
6. **Always close task channel when done**
7. **Drain results channel before shutdown**
8. **Use context for cancellation**
9. **Graceful shutdown: close tasks, wait WaitGroup**
10. **Immediate shutdown: cancel context**

## Exercises

1. Implement basic worker pool with 10 workers processing 1000 tasks.
2. Add priority queue: high-priority tasks processed first.
3. Implement dynamic scaling: add workers if queue > threshold.
4. Add retry logic: failed tasks retry with exponential backoff.
5. Benchmark optimal worker count for CPU-bound vs. I/O-bound tasks.

**Next:** fan-in-fan-out.md - Splitting and merging data streams.