# Job Queue Project

## Learning Objectives

Build a production-ready job queue to learn:

- Worker pool pattern with priority queues
- Graceful shutdown with in-flight tracking
- Job persistence and at-least-once delivery
- Retry logic with exponential backoff
- Preventing worker starvation

## Requirements

### Functional Requirements

1. **Worker Pool**
    - Fixed number of workers processing jobs
    - Bounded job queue (backpressure when full)
    - Concurrent job execution
2. **Priority Levels**
    - High, Medium, Low priority jobs
    - High priority processed first
    - No starvation (low priority eventually runs)
3. **Job Lifecycle**
    - Enqueue (blocking if queue full)
    - Processing (worker picks job)
    - Completion (success or failure)
    - Retry (on failure with backoff)
4. **Persistence** (Final only)
    - Jobs persisted to disk
    - Survive process restart
    - At-least-once delivery guarantee

### Non-Functional Requirements

1. **Performance**
    - Process 10,000+ jobs/sec
    - Low latency (<10ms queue → worker)
    - Efficient memory usage
2. **Reliability**
    - No job loss on crash (with persistence)
    - Retry failed jobs (configurable attempts)
    - Graceful shutdown (wait for in-flight)
3. **Observability**

    - Metrics (queued, processing, completed, failed)
    - Job status tracking
    - Worker utilization

## Three Implementations

### 1. Naive Implementation (naive/job_queue.go)

**Approach:**

- Single unbuffered channel for all jobs
- No priorities (FIFO only)
- No persistence (in-memory only)
- No retry logic

**Problems (Intentional):**

- **No backpressure:** Unbounded queue can OOM
- **No priorities:** Critical jobs wait behind low-priority
- **No persistence:** Jobs lost on crash
- **Poor shutdown:** Workers may exit with jobs in-flight
- **No retry:** Failed jobs are lost

**Expected Issues:**

- Memory grows unbounded with fast producers
- Can't prioritize urgent work
- Not production-ready (data loss)

### 2. Improved Implementation (improved/job_queue.go)

**Fixes:**

- **Bounded queues:** Separate channel per priority
- **Priority selection:** High > Medium > Low with select
- **Retry logic:** Exponential backoff (1s, 2s, 4s, 8s)
- **Graceful shutdown:** WaitGroup tracks in-flight jobs
- **Metrics:** Atomic counters for observability

**Improvements:**

- Backpressure prevents OOM
- Priorities work correctly
- Failed jobs retry automatically
- Clean shutdown guaranteed

**Remaining Issues:**

- Still no persistence (in-memory only)
- Priority starvation possible (high priority floods)
- No way to query job status
- Retry state lost on restart

### 3. Final Implementation (final/job_queue.go)

**Production-Ready:**

- **Persistence:** Jobs persisted to disk (at-least-once)
- **Anti-starvation:** Token-based fairness (1 low per 10 high)
- **Job tracking:** Query status by ID
- **Comprehensive metrics:** Per-priority counters
- **Configurable:** Workers, retries, timeouts

**Key Features:**

```
// Job with metadata
type Job struct {
    ID       string
    Priority Priority
    Payload  []byte
    Retries  int
    Created  time.Time
}

// Enqueue with backpressure
err := jq.Enqueue(ctx, job)

// Query status
status := jq.Status(jobID) // Queued, Processing, Completed, Failed

// Metrics
m := jq.Metrics()
fmt.Printf("Completed: %d, Failed: %d\n", m.Completed, m.Failed)
```

**Performance:**

- Throughput: ~50k jobs/sec (4 workers)
- Latency: p50=1ms, p99=10ms
- Persistence overhead: ~10% (batched writes)

## Success Criteria

1. **Understand worker pool:**
    - How many workers for CPU-bound vs I/O-bound jobs?
    - What happens if workers > CPU cores?
    - How to detect worker starvation?
2. **Master priority queues:**
    - Why separate channels per priority?
    - How to prevent starvation?
    - Trade-off: fairness vs throughput
3. **Implement persistence:**
    - At-least-once vs at-most-once vs exactly-once
    - Where to persist (before processing? after?)
    - How to recover on restart?
4. **Handle failures:**

- Exponential backoff vs fixed delay
- Max retries before dead letter queue
- Idempotent job handlers

## Usage Examples

### Naive Implementation

```
jq := naive.NewJobQueue(4) // 4 workers
defer jq.Close()

// Enqueue job (blocks if internal buffer full)
jq.Enqueue(naive.Job{
    ID:      "job1",
    Handler: func() error { return processJob() },
})
```

### Final Implementation

```
jq := final.NewJobQueue(final.Config{
    NumWorkers:   4,
    QueueSize:    1000, // per priority
    MaxRetries:   3,
    PersistPath:  "./jobs.db",
    ShutdownTimeout: 30 * time.Second,
})
defer jq.Close()

// Enqueue with priority and context
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()

job := final.Job{
    ID:       uuid.New().String(),
    Priority: final.HighPriority,
    Payload:  []byte(`{"action": "send_email"}`),
}

if err := jq.Enqueue(ctx, job); err != nil {
    log.Fatal(err)
}

// Query status
status := jq.Status(job.ID)
fmt.Printf("Job %s: %s\n", job.ID, status)

// Wait for completion
ticker := time.NewTicker(100 * time.Millisecond)
for range ticker.C {
    if status := jq.Status(job.ID); status == final.Completed {
```

```
            break
        }
    }
```

## Testing Strategy

### Unit Tests

```
cd final
go test -v
go test -race
go test -cover
```

### Test Cases

1. **TestEnqueue** - Verify job queued successfully
2. **TestPriority** - High priority processed first
3. **TestRetry** - Failed jobs retry with backoff
4. **TestGracefulShutdown** - In-flight jobs complete
5. **TestPersistence** - Jobs survive restart
6. **TestBackpressure** - Enqueue blocks when full
7. **TestMetrics** - Counters accurate

### Stress Test

```
func TestStress(t *testing.T) {
    jq := NewJobQueue(Config{NumWorkers: 10, QueueSize: 1000})
    defer jq.Close()

    var wg sync.WaitGroup
    numJobs := 100000

    for i := 0; i < numJobs; i++ {
        wg.Add(1)
        go func(id int) {
            defer wg.Done()
            job := Job{
                ID: fmt.Sprintf("job%d", id),
                Priority: Priority(id % 3),
                Payload: []byte("data"),
            }
            jq.Enqueue(context.Background(), job)
        }(i)
    }

    wg.Wait()

    // Wait for completion
    time.Sleep(5 * time.Second)
```

```
    m := jq.Metrics()
    if m.Completed != uint64(numJobs) {
        t.Errorf("Expected %d completed, got %d", numJobs, m.Completed)
    }
}
```

## Real-World Scenarios

### 1. Background Email Sender

```
jq := final.NewJobQueue(final.Config{
    NumWorkers: 10, // I/O-bound, more workers
    QueueSize:  10000,
    MaxRetries: 3,
})

// Enqueue email job
job := final.Job{
    ID:       emailID,
    Priority: final.MediumPriority,
    Payload:  marshalEmail(to, subject, body),
}

jq.EnqueueWithHandler(context.Background(), job, func(payload []byte) error {
    return sendEmail(payload)
})
```

### 2. Image Processing Pipeline

```
jq := final.NewJobQueue(final.Config{
    NumWorkers: runtime.NumCPU(), // CPU-bound
    QueueSize:  1000,
})

// High priority for user-uploaded images
job := final.Job{
    ID:       imageID,
    Priority: final.HighPriority,
    Payload:  imageData,
}

jq.EnqueueWithHandler(ctx, job, func(payload []byte) error {
    return processImage(payload) // Resize, compress, etc.
})
```

### 3. Webhook Delivery System

```
jq := final.NewJobQueue(final.Config{
    NumWorkers:  20, // I/O-bound (network requests)
    QueueSize:   50000,
    MaxRetries:  5, // Webhooks can fail temporarily
    PersistPath: "/var/lib/webhooks.db",
})

// Critical webhooks get high priority
priority := final.MediumPriority
if webhook.IsCritical {
    priority = final.HighPriority
}

job := final.Job{
    ID:       webhookID,
    Priority: priority,
    Payload:  marshalWebhook(webhook),
}

jq.EnqueueWithHandler(ctx, job, func(payload []byte) error {
    return deliverWebhook(payload)
})
```

## Common Pitfalls

### Pitfall 1: Too Many Workers (CPU-Bound)

```
// ❌ WRONG: 100 workers for CPU-bound work
jq := NewJobQueue(Config{NumWorkers: 100})

// ✅ CORRECT: Match CPU cores
jq := NewJobQueue(Config{NumWorkers: runtime.NumCPU()})
```

### Pitfall 2: Priority Starvation

```
// ❌ WRONG: Always pick high priority
select {
case job := <-highQueue:
    process(job)
case job := <-lowQueue: // Never selected if high queue busy
    process(job)
}

// ✅ CORRECT: Token-based fairness
tokens := 10
for {
    if tokens > 0 {
        select {
        case job := <-highQueue:
```

```
            process(job)
            tokens--
        default:
            // High queue empty, reset tokens
            tokens = 10
        }
    } else {
        // Process 1 low priority, then reset
        select {
        case job := <-lowQueue:
            process(job)
            tokens = 10
        default:
            tokens = 10
        }
    }
}
```

### Pitfall 3: Forgetting WaitGroup.Add

```
// ❌ WRONG: Add inside goroutine (race)
go func() {
    wg.Add(1) // Main might call Wait() before this
    defer wg.Done()
    work()
}()

// ✅ CORRECT: Add before spawning
wg.Add(1)
go func() {
    defer wg.Done()
    work()
}()
```

### Pitfall 4: Not Handling Context Cancellation

```
// ❌ WRONG: Enqueue ignores context
func (jq *JobQueue) Enqueue(ctx context.Context, job Job) {
    jq.queue <- job // Blocks forever if queue full
}

// ✅ CORRECT: Respect context
func (jq *JobQueue) Enqueue(ctx context.Context, job Job) error {
    select {
    case jq.queue <- job:
        return nil
    case <-ctx.Done():
        return ctx.Err()
```

```
    }
}
```

### Pitfall 5: Losing Jobs on Shutdown

```
// ❌ WRONG: Close queue immediately
close(jq.queue)

// ✅ CORRECT: Wait for in-flight jobs
jq.shutdown() // Signal workers to stop
jq.wg.Wait()  // Wait for in-flight jobs
close(jq.queue)
```

## Interview Discussion Points

1. **How many workers for I/O-bound vs CPU-bound?**
    - CPU-bound: NumCPU (more causes contention)
    - I/O-bound: NumCPU × 10-100 (waiting for I/O)
    - Benchmark to find optimal
2. **How to prevent priority starvation?**
    - Token-based: Process N high, then 1 low
    - Weighted: Pick high 90%, low 10%
    - Age-based: Increase priority over time
3. **At-least-once vs at-most-once vs exactly-once?**
    - At-least-once: Persist before processing (may process twice on crash)
    - At-most-once: Process then persist (may lose on crash)
    - Exactly-once: Distributed transaction (complex, slow)
    - Most systems use at-least-once + idempotent handlers
4. **How to recover jobs after crash?**
    - Persist job state to disk/database
    - On startup, read all Queued/Processing jobs
    - Re-enqueue for processing
    - Use WriteAheadLog for durability
5. **Why exponential backoff?**
    - Avoids thundering herd (all retries at once)
    - Gives failing service time to recover
    - Formula: delay = min(maxDelay, baseDelay × 2^attempt)

## Next Steps

After mastering job-queue:

1. **cache/** - Concurrent LRU cache with sharding
2. **web-crawler/** - Bounded concurrency and politeness
3. **connection-pool/** - Database pool with circuit breaker

4. **pub-sub/** - Topic-based messaging with fan-out

**Build production-ready concurrent systems!** 🚀