Buffered vs Unbuffered Channels

The Critical Decision

Choosing between buffered and unbuffered channels affects:

- **Synchronization guarantees**
- **Throughput and latency**
- **Goroutine coupling**
- **Deadlock likelihood**
- **Resource consumption**

This is not a performance optimization question. This is a correctness question.

Unbuffered Channels: Synchronous Rendezvous

```go
ch := make(chan int) // No buffer, capacity 0
```

Behavior

Send and receive must meet—neither can proceed without the other.

```go
ch := make(chan int)

// Goroutine 1
ch <- 42  // Blocks until goroutine 2 receives

// Goroutine 2
val := <-ch  // Blocks until goroutine 1 sends
```

Execution Timeline

```
Time:  0ms    10ms    20ms    30ms    40ms
       |       |       |       |       |
G1:    ───────[SEND BLOCKS]───[SEND COMPLETES]───
G2:    ───────────────────────[RECV]──────────────
                               ↑
                         Rendezvous point
```

Both goroutines **synchronize at the channel operation**.

Happens-Before Guarantee

For unbuffered channels:

```
Send happens-before receive completes
```

Example:

```go
var data string
done := make(chan bool)

go func() {
    data = "hello"    // (1) Write
    done <- true      // (2) Send
}()

<-done                // (3) Receive
fmt.Println(data)    // (4) Read

// Guaranteed order: (1) → (2) → (3) → (4)
// Reading `data` is safe
```

When to Use Unbuffered

✅ **Use unbuffered when:**

- You need **exact synchronization** between goroutines
- You want to **signal events** (completion, errors)
- You're **transferring ownership** and want confirmation of receipt
- You want **backpressure** (sender waits for receiver)
- You're implementing **request-response** patterns

Example: Event Signaling

```go
func worker(ready chan struct{}) {
    // Initialize
    doSetup()
    close(ready)  // Signal "I'm ready"
}

func main() {
    ready := make(chan struct{}) // Unbuffered
    go worker(ready)
    <-ready  // Wait until worker is ready
    // Now safe to proceed
}
```

Buffered Channels: Asynchronous Communication

```go
ch := make(chan int, 10) // Buffer capacity 10
```

Behavior

Send blocks only when buffer full; receive blocks only when buffer empty.

```go
ch := make(chan int, 2)
```

```
ch <- 1  // Returns immediately (buffer: [1])
ch <- 2  // Returns immediately (buffer: [1, 2])
ch <- 3  // BLOCKS (buffer full)

val := <-ch  // Receives 1 (buffer: [2])
// Now ch <- 3 can proceed
```

Execution Timeline

```
Time:  0ms    10ms    20ms    30ms    40ms
       |       |       |       |       |
G1:    [SEND][SEND][SEND]──────────────────
        ↓     ↓     ↓
       [1]   [1,2] [1,2,3]
                    ↑
G2:    ───────────────────────[RECV][RECV]──
                               ↓     ↓
                              [2,3]  [3]
```

Goroutines **decouple**—sender doesn't wait for receiver (until buffer full).

Happens-Before Guarantee

For buffered channels:

```
Send happens-before corresponding receive starts
```

Note: Different from unbuffered (send happens-before receive **completes**).

Example:

```go
var data string
done := make(chan bool, 1)  // Buffered

go func() {
    data = "hello"     // (1) Write
    done <- true       // (2) Send (returns immediately)
}()

<-done                 // (3) Receive
fmt.Println(data)     // (4) Read

// Guaranteed order: (1) → (2) happens-before (3) → (4)
// Still safe to read `data`
```

When to Use Buffered

✅ **Use buffered when:**

- Sender and receiver run at **different rates** (producer faster/slower than consumer)
- You want to **decouple goroutines** (avoid blocking sender on slow receiver)
- You're implementing a **work queue** with bounded size

- You want **burst handling** (accept spikes without blocking)
- You're using **buffered returns** (size 1) to prevent leaks

Example: Work Queue

```
func workQueue(capacity int) chan Job {
    jobs := make(chan Job, capacity)  // Buffered

    // Workers can process at their own pace
    for i := 0; i < 10; i++ {
        go func() {
            for job := range jobs {
                process(job)
            }
        }()
    }

    return jobs
}
```

Buffer Size: The Critical Choice

Buffer Size = 0 (Unbuffered)

```
ch := make(chan int)
```

- **Tightest coupling:** Every send waits for receive
- **Strongest sync:** Explicit rendezvous
- **Slowest:** Maximum blocking
- **Simplest reasoning:** Happens-before is obvious

Buffer Size = 1

```
ch := make(chan int, 1)
```

Most common buffered size. Why?

Use case 1: Async result returns (leak prevention)

```
func compute() <-chan Result {
    ch := make(chan Result, 1)  // Buffer size 1

    go func() {
        result := expensiveComputation()
        ch <- result  // Never blocks (buffer has space)
    }()

    return ch
```

```go
    // Even if caller never receives, goroutine completes
}
```

Use case 2: Latest value pattern

```go
type Monitor struct {
    updates chan State  // Buffer size 1
}

func (m *Monitor) NotifyStateChange(state State) {
    select {
    case m.updates <- state:  // Send if space
    default:                  // Drop old value if full
        <-m.updates           // Remove old
        m.updates <- state    // Send new
    }
}
```

Buffer Size = NumWorkers

```go
workers := 10
jobs := make(chan Job, workers)
```

Rationale: Buffer matches concurrency level.

- Producer can submit up to `NumWorkers` jobs without blocking
- Workers never starve (jobs available)
- Bounded memory: Max buffered = worker count

Example:

```go
func processParallel(items []Item) {
    workers := runtime.NumCPU()
    jobs := make(chan Item, workers)
    results := make(chan Result, workers)

    // Start workers
    for i := 0; i < workers; i++ {
        go func() {
            for item := range jobs {
                results <- process(item)
            }
        }()
    }

    // Send jobs
    go func() {
        for _, item := range items {
            jobs <- item
        }
```

```go
        close(jobs)
    }()

    // Collect results...
}
```

Buffer Size = len(items)

```go
items := getItems()
ch := make(chan Item, len(items))
```

When to use:

- You know **exact number of items** upfront
- Want to send all **without blocking**
- Short-lived channel (not long-running queue)

Example:

```go
func scatterGather(items []Item) []Result {
    results := make(chan Result, len(items))  // Exact size

    for _, item := range items {
        go func(it Item) {
            results <- process(it)  // Never blocks
        }(item)
    }

    // Collect exactly len(items) results
    collected := make([]Result, 0, len(items))
    for i := 0; i < len(items); i++ {
        collected = append(collected, <-results)
    }

    return collected
}
```

Buffer Size = Large (100+)

```go
ch := make(chan Message, 1000)
```

When to use:

- High-throughput systems
- Smoothing out **bursty traffic**
- Decoupling fast producer from slow consumer

Warning: Large buffers hide problems.

- Delays detection of slow consumers
- Increases memory usage

- Can hide deadlocks (buffer never fills during testing)

Rule of thumb: If you need buffer > 100, reconsider your design. You might need:

- Dropping policy (discard old messages)
- Multiple consumers (scale out)
- Backpressure (slow down producer)

Comparison Table

Aspect	Unbuffered	Buffered (Size N)
Blocking	Send blocks until recv	Send blocks when full
Synchronization	Explicit rendezvous	Async up to N items
Latency	Higher (blocking)	Lower (non-blocking until full)
Throughput	Lower (tight coupling)	Higher (decoupling)
Memory	None (no buffer)	O(N) per channel
Backpressure	Immediate	Delayed (kicks in when full)
Deadlock risk	Higher (tight coupling)	Lower (loose coupling)
Reasoning	Simpler (explicit sync)	Harder (async effects)

Common Misunderstandings

Myth 1: "Buffered channels are faster"

Truth: They can improve **throughput** (more messages/sec) if sender/receiver speeds differ, but individual operations take the same time (~50-100ns).

Example where buffered is faster:

```go
// Unbuffered: Send blocks until recv
func slowConsumer() {
    ch := make(chan int)

    go func() {
        for i := 0; i < 1000; i++ {
            ch <- i  // Blocks on every send
        }
    }()

    for i := 0; i < 1000; i++ {
        <-ch
        time.Sleep(time.Millisecond)  // Slow consumer
    }
}
// Total time: ~1000ms (send waits for slow recv)

// Buffered: Send doesn't block until full
```

```go
func slowConsumerBuffered() {
    ch := make(chan int, 100)

    go func() {
        for i := 0; i < 1000; i++ {
            ch <- i  // Only blocks when buffer full
        }
    }()

    for i := 0; i < 1000; i++ {
        <-ch
        time.Sleep(time.Millisecond)  // Slow consumer
    }
}
// Total time: Still ~1000ms, but sender finishes early
```

Myth 2: "Buffered channels prevent deadlocks"

Truth: They can **hide** deadlocks by delaying them.

```go
// Unbuffered: Deadlocks immediately
func unbufferedDeadlock() {
    ch := make(chan int)
    ch <- 1  // DEADLOCK: no receiver
}

// Buffered: Deadlocks only when buffer full
func bufferedDeadlock() {
    ch := make(chan int, 5)
    for i := 0; i < 10; i++ {
        ch <- i  // DEADLOCK at i=5: buffer full, no receiver
    }
}
```

Buffered channels **delay** the deadlock, making it harder to detect in small-scale tests.

Myth 3: "Always use buffered size 1 to prevent leaks"

Truth: This works for single-result patterns but isn't a universal solution.

```go
// Good: Single result
func compute() <-chan int {
    ch := make(chan int, 1)
    go func() {
        ch <- expensiveCalc()  // Won't leak even if not received
    }()
    return ch
}

// BAD: Multiple results
func computeMany() <-chan int {
```

```go
    ch := make(chan int, 1)  // Buffer size 1
    go func() {
        for i := 0; i < 100; i++ {
            ch <- i  // Will block at 2nd send if not received
        }
    }()
    return ch
}
// Still leaks if receiver stops early
```

Better for multiple results: Use context cancellation.

Real-World Failure: Buffer Size Matters

Company: Social media platform (2020)

What happened:
During a traffic spike, HTTP handlers started timing out. CPU usage was normal. Memory usage exploded.

Root cause:

```go
type Handler struct {
    notificationQueue chan Notification  // Unbuffered
}

func (h *Handler) handleRequest(w http.ResponseWriter, r *http.Request) {
    // Process request
    notification := createNotification(r)

    h.notificationQueue <- notification  // Blocks if consumer slow
    // Handler blocked → can't handle new requests

    w.WriteHeader(http.StatusOK)
}
```

Problem:

- Notification consumer was slow (DB writes)
- Handlers blocked sending notifications
- No handlers available for new requests
- HTTP timeout errors

Fix:

```go
type Handler struct {
    notificationQueue chan Notification  // Buffered (size 1000)
}

func (h *Handler) handleRequest(w http.ResponseWriter, r *http.Request) {
    notification := createNotification(r)

    select {
```

```
    case h.notificationQueue <- notification:
        // Sent successfully
    default:
        // Queue full, drop notification (log for monitoring)
        metrics.Inc("notifications_dropped")
    }

    w.WriteHeader(http.StatusOK)
}
```

Lessons:

1. Unbuffered channels couple goroutines tightly
2. Buffering decouples sender from slow receiver
3. Always have a drop policy when buffer fills
4. Monitor queue depth in production

Decision Tree: Buffered or Unbuffered?

```
Do you need exact synchronization (rendezvous)?
├─ YES → Unbuffered
│   └─ Examples: event signaling, handshake, barrier
│
└─ NO → Buffered
    │
    ├─ Single result return?
    │   └─ YES → Buffer size 1
    │
    ├─ Known number of items?
    │   └─ YES → Buffer size = len(items)
    │
    ├─ Work queue with N workers?
    │   └─ YES → Buffer size = N
    │
    ├─ High throughput, bursty traffic?
    │   └─ YES → Larger buffer (10-1000)
    │       └─ + Implement drop policy
    │
    └─ Unsure?
        └─ Start with unbuffered (safer)
           Monitor in production
           Add buffering if needed
```

Interview Traps

Trap 1: "Buffered channels are always better"

Wrong. Buffering introduces complexity and can hide bugs.

Correct answer:
"Unbuffered channels provide stronger synchronization guarantees and simpler reasoning. Buffered channels

decouple sender/receiver, improving throughput when they run at different speeds, but can hide deadlocks and coordination issues. Use unbuffered by default; add buffering when profiling shows it helps."

Trap 2: "Buffer size should be large to avoid blocking"

Wrong. Large buffers consume memory and hide slow consumers.

Correct answer:
"Buffer size should match your concurrency model: size 1 for async returns, size N for N workers, or bounded based on memory constraints. Large buffers (100+) can hide slow consumer problems and delay backpressure. Better to fix the slow consumer or implement a drop policy."

Trap 3: "Unbuffered channels are slower"

Misleading. Individual operations take the same time; throughput differs.

Correct answer:
"Both unbuffered and buffered channel operations take ~50-100ns. Unbuffered channels block sender until receiver is ready, which can reduce throughput if sender/receiver speeds differ. Buffered channels improve throughput by decoupling, but don't make individual operations faster."

Trap 4: "This unbuffered deadlock would be fixed with a buffer"

```
ch := make(chan int)
ch <- 1  // Deadlock
```

Lazy answer: "Add buffer: `make(chan int, 1)` "

Correct answer:
"This is a design issue, not a buffering issue. The problem is sending without a receiver. Buffering would delay the deadlock, not fix it. Correct solutions: 1) Use a goroutine for the send, 2) Use a receiver on the other end, 3) Redesign to avoid this pattern."

Key Takeaways

1. **Unbuffered = synchronous rendezvous** (tight coupling)
2. **Buffered = async up to capacity** (loose coupling)
3. **Buffer size 1** is most common for async returns
4. **Buffer size N** matches N workers pattern
5. **Large buffers hide problems** (slow consumers, deadlocks)
6. **Start unbuffered** (simpler), add buffering when needed
7. **Always implement drop policy** for bounded buffers
8. **Happens-before guarantees differ** (send-before-recv-completes vs send-before-recv-starts)

What You Should Be Thinking Now

- "How do I wait on multiple channels?"
- "How do I implement timeouts?"
- "What's the select statement?"
- "How do I handle non-blocking channel operations?"

Next: select.md - Multiplexing channels with select.

Exercises (Do These Before Moving On)

1. Write two versions of a program: one with unbuffered, one with buffered channels. Measure throughput difference.
2. Create a deadlock with an unbuffered channel. "Fix" it with a buffer. Explain why this is a band-aid, not a fix.
3. Implement a work queue with buffer size matching worker count.
4. Write code that uses buffer size 1 for an async result return. Verify it doesn't leak.
5. Implement a notification queue with a drop policy when buffer is full.

Don't continue until you can explain: "When is unbuffered better than buffered, despite potential blocking?"