Happens-Before: The Formal Definition

What is Happens-Before?

Happens-before (→hb) is a **partial ordering of memory operations** that guarantees:

1. **Temporal ordering:** If A →hb B, then A completes before B starts
2. **Memory visibility:** If A →hb B, then B observes all effects of A

This is the ONLY guarantee of visibility in concurrent programs.

Why "Partial" Ordering?

Partial ordering means not all operations are ordered relative to each other.

```
var a, b int

// Goroutine 1
a = 1  // (1)
b = 2  // (2)

// Goroutine 2
x := b  // (3)
y := a  // (4)

// Within goroutines:
// (1) →hb (2)  ✓ Ordered
// (3) →hb (4)  ✓ Ordered

// Across goroutines:
// (1) ? (3)  ✗ Not ordered
// (2) ? (4)  ✗ Not ordered
```

Unordered operations: Compiler and CPU can reorder them. No visibility guarantees.

The Formal Rules

Rule 1: Intra-Goroutine Ordering (Program Order)

Within a single goroutine, operations execute in program order:

```
func example() {
    x := 1  // (1)
    y := 2  // (2)
    z := 3  // (3)
}

// (1) →hb (2) →hb (3)
```

Important: "Program order" is as observed by the goroutine itself. Compiler can reorder operations as long as observable behavior within the goroutine doesn't change.

```go
func example() {
    a := 1  // (1)
    b := 2  // (2)
    c := a  // (3)
}

// Compiler can reorder (1) and (2) because (2) doesn't affect (3)
// But must execute (1) before (3)
```

Rule 2: Goroutine Creation

Statement before `go` →hb all statements in spawned goroutine.

```go
a := 1     // (1)
b := 2     // (2)
go func() {
    print(a)  // (3)
    print(b)  // (4)
}()

// Guarantees:
// (1) →hb (2) →hb (3)
// (1) →hb (2) →hb (4)
```

Formal: If statement S is before `go f()`, then S →hb first statement in f.

Critical insight: Only statements before `go` are synchronized. Not statements after.

```go
go func() {
    print(a)  // (1)
}()
a = 1  // (2)

// No guarantee: (2) NOT →hb (1)
// Can print 0
```

Rule 3: Goroutine Exit (No Synchronization)

Goroutine exit does NOT synchronize with parent.

```go
go func() {
    a = 1  // (1)
}()  // (2) Returns immediately
print(a)  // (3)

// No guarantee: (1) NOT →hb (3)
```

Why? Parent doesn't wait for child to finish (unless you use WaitGroup/channel).

Rule 4: Channel Send and Receive

Unbuffered Channel

Send →hb receive completes.

```
ch := make(chan int)

// Goroutine 1
a = 1       // (1)
ch <- 0     // (2) Send

// Goroutine 2
<-ch        // (3) Receive completes
print(a)    // (4)

// Chain: (1) →hb (2) →hb (3) →hb (4)
```

Formal: A send on channel C →hb the corresponding receive on C returns.

Intuition: Unbuffered is synchronous rendezvous. Sender blocks until receiver ready.

Buffered Channel

Send →hb receive begins.

Also: Receive →hb send completes (for buffer slot).

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

// Goroutine 1
a = 1       // (1)
ch <- 0     // (2) Send (doesn't block, buffer has space)

// Goroutine 2
<-ch        // (3) Receive begins
print(a)    // (4)

// Chain: (1) →hb (2) →hb (3) →hb (4)
```

Tricky case: Buffer full

```
ch := make(chan int, 1)
ch <- 1   // Fills buffer
ch <- 2   // (1) Blocks until receive

// Other goroutine
<-ch   // (2) Receive
// (2) →hb (1) completes
```

Formal: K-th receive on channel C →hb (K+buffer-size)-th send on C completes.

Example: Buffer size 2, 3rd receive →hb 5th send completes.

Channel Close

Close →hb receive of zero value.

```
ch := make(chan int)

// Goroutine 1
a = 1       // (1)
close(ch)   // (2)

// Goroutine 2
<-ch        // (3) Receives zero value
print(a)    // (4)

// Chain: (1) →hb (2) →hb (3) →hb (4)
```

Rule 5: Mutex (sync.Mutex, sync.RWMutex)

Mutex Unlock →hb Subsequent Lock

```
var mu sync.Mutex
var a int

// Goroutine 1
mu.Lock()    // (1)
a = 1        // (2)
mu.Unlock()  // (3)

// Goroutine 2
mu.Lock()    // (4) After (3) releases
print(a)     // (5)
mu.Unlock()

// Chain: (2) →hb (3) →hb (4) →hb (5)
```

Formal: For sync.Mutex m, call n to m.Unlock() →hb call m+1 to m.Lock() returns.

Critical: Unlock happens-before the NEXT Lock (not all future locks, just the first one after).

RWMutex

Multiple readers can hold RLock concurrently (no happens-before between them).

```
var mu sync.RWMutex
var a int

// Goroutine 1 (writer)
mu.Lock()
a = 1
mu.Unlock()  // (1)
```

```
// Goroutine 2 (reader)
mu.RLock()   // (2) After (1)
print(a)     // Sees a=1
mu.RUnlock()

// Goroutine 3 (reader, concurrent with G2)
mu.RLock()   // (3) No ordering with (2)
print(a)     // Sees a=1
mu.RUnlock()

// Guarantees:
// (1) →hb (2)
// (1) →hb (3)
// No relationship between (2) and (3)
```

Formal rules:

1. m.Unlock() →hb m.Lock()
2. m.Unlock() →hb m.RLock()
3. m.RUnlock() (if no concurrent readers remain) →hb m.Lock()

Rule 6: sync.WaitGroup

Done() that causes counter → 0 →hb Wait() returns.

```
var wg sync.WaitGroup
var a int

// Goroutine 1
wg.Add(1)
go func() {
    a = 1        // (1)
    wg.Done()    // (2)
}()

wg.Wait()    // (3) Returns when counter=0
print(a)     // (4)

// Chain: (1) →hb (2) →hb (3) →hb (4)
```

Important: Only the Done() that decrements counter to zero synchronizes with Wait().

```
wg.Add(2)

go func() { wg.Done() }()  // (1) Doesn't sync with Wait
go func() { wg.Done() }()  // (2) This one does (if it's last)

wg.Wait()  // Only (2) →hb Wait()
```

Rule 7: sync/atomic

Atomic write →hb atomic read that observes the write.

```
var a int
var flag int32

// Goroutine 1
a = 42                          // (1)
atomic.StoreInt32(&flag, 1)     // (2)

// Goroutine 2
for atomic.LoadInt32(&flag) == 0 {}  // (3) Spin until sees 1
print(a)                             // (4)

// Chain: (1) →hb (2) →hb (3) →hb (4)
```

Formal: Atomic store S →hb atomic load L that returns value written by S.

Key insight: Atomics synchronize ALL previous operations in the goroutine (not just the atomic variable).

Atomic Read-Modify-Write:

```
// Goroutine 1
a = 1                       // (1)
atomic.AddInt32(&x, 1)      // (2)

// Goroutine 2
v := atomic.AddInt32(&x, 1) // (3) Observes or follows (2)
print(a)                    // (4)

// If (3) observes (2), then (2) →hb (3) →hb (4)
// Chain: (1) →hb (4)
```

Rule 8: sync.Once

once.Do(f) that executes f →hb any once.Do(g) returns.

```
var once sync.Once
var a int

func init() {
    a = 1  // (1)
}

// Multiple goroutines
once.Do(init)  // (2) One goroutine executes
print(a)       // (3) All goroutines see a=1

// (1) →hb (2) completes →hb (3)
```

Rule 9: Package Initialization

init() →hb main.main()

```go
var a int

func init() {
    a = 1  // (1)
}

func main() {
    print(a)  // (2) Always sees a=1
}

// (1) →hb (2)
```

Import order: Package P's init →hb importing package's init.

```go
// Package A
func init() {
    x = 1  // (1)
}

// Package B (imports A)
import "A"
func init() {
    print(A.x)  // (2) Sees x=1
}

// (1) →hb (2)
```

Chaining Happens-Before

Happens-before is **transitive:**

If A →hb B and B →hb C, then A →hb C.

```go
a = 1      // (1)
ch <- 0    // (2) Send
// Receiver
<-ch       // (3) Receive
print(a)   // (4)

// (1) →hb (2) (same goroutine)
// (2) →hb (3) (channel rule)
// (3) →hb (4) (same goroutine)
// Chain: (1) →hb (4)
```

This is how you reason about concurrency: Build chains of happens-before.

The Absence of Happens-Before

If A NOT →hb B and B NOT →hb A, operations are concurrent (unordered).

Consequences:

1. Compiler can reorder them
2. CPU can reorder them
3. B might not see A's effects ever
4. Partial effects can be observed

```
var x, y int

// Goroutine 1
x = 1  // (1)
y = 1  // (2)

// Goroutine 2
a := y  // (3)
b := x  // (4)

// No happens-before between goroutines
// Possible outcomes:
// a=0, b=0  (neither write visible)
// a=1, b=0  (only y write visible)
// a=0, b=1  (only x write visible)
// a=1, b=1  (both visible)
```

Real-World Example: The Publication Race

A common bug pattern where publishing a pointer races with using it.

Unsafe Publication

```
type Config struct {
    Host string
    Port int
}

var config *Config

func Load() {
    cfg := &Config{
        Host: "localhost",  // (1)
        Port: 8080,         // (2)
    }
    config = cfg  // (3) Publication
}

func Use() {
    if config != nil {  // (4) Check pointer
        connect(config.Host, config.Port)  // (5) Use fields
    }
```

```
}

// Thread 1: Load()
// Thread 2: Use()

// Possible: (3) →hb (4) but (1,2) NOT →hb (5)
// Result: See config != nil, but Host and Port not initialized!
```

Why? No happens-before between (1,2) and (5). Compiler/CPU can reorder.

Safe Publication

```
var (
    config *Config
    mu     sync.Mutex
)

func Load() {
    cfg := &Config{
        Host: "localhost",  // (1)
        Port: 8080,         // (2)
    }
    mu.Lock()
    config = cfg  // (3)
    mu.Unlock()   // (4)
}

func Use() {
    mu.Lock()        // (5)
    cfg := config    // (6)
    mu.Unlock()

    if cfg != nil {
        connect(cfg.Host, cfg.Port)  // (7)
    }
}

// Chain: (1) →hb (2) →hb (3) →hb (4) →hb (5) →hb (6) →hb (7)
// Always see fully initialized Config
```

Real-World Failure: Cloudflare BGP Route Leak (2020)

Date: June 2020
Incident: Cloudflare route leak affecting global traffic

Root cause (simplified): Concurrent update to routing table without proper synchronization.

```
// Simplified version of bug pattern
type RouteTable struct {
    routes map[string]*Route
```

```
}

var table *RouteTable

func updateRoutes(newRoutes []*Route) {
    // Create new table
    newTable := &RouteTable{
        routes: make(map[string]*Route),
    }

    for _, r := range newRoutes {
        newTable.routes[r.Prefix] = r  // (1) Multiple writes
    }

    table = newTable  // (2) Publish new table
    // No synchronization!
}

func lookupRoute(prefix string) *Route {
    return table.routes[prefix]  // (3) Read without sync
}

// Problem: (1) NOT →hb (3)
// Reader can see partially constructed map
// Can read stale 'table' pointer or partially filled routes
```

Impact: Traffic routed to wrong destinations, global outage.

Fix: Add mutex around table updates and reads.

```
var (
    table *RouteTable
    mu    sync.RWMutex
)

func updateRoutes(newRoutes []*Route) {
    newTable := &RouteTable{ /* ... */ }

    mu.Lock()
    table = newTable
    mu.Unlock()
}

func lookupRoute(prefix string) *Route {
    mu.RLock()
    defer mu.RUnlock()
    return table.routes[prefix]
}
```

Proving Correctness with Happens-Before

To prove code is race-free:

1. Identify all shared memory accesses
2. For each pair (write, read) or (write, write):
    - Find synchronization operation that creates happens-before
    - Build chain from write to read
3. If no chain exists → race

Example: Analyzing a cache

```
type Cache struct {
    mu    sync.Mutex
    items map[string]int
}

func (c *Cache) Get(key string) (int, bool) {
    c.mu.Lock()          // (1)
    v, ok := c.items[key]  // (2) Read
    c.mu.Unlock()
    return v, ok
}

func (c *Cache) Set(key string, val int) {
    c.mu.Lock()          // (3)
    c.items[key] = val   // (4) Write
    c.mu.Unlock()        // (5)
}

// Scenario: G1 calls Set, G2 calls Get
// Write: (4)
// Read: (2)
// Need: (4) →hb (2)

// If Set happens first:
// Chain: (4) →hb (5) (same goroutine)
//        (5) →hb (1) (unlock →hb lock)
//        (1) →hb (2) (same goroutine)
// Chain: (4) →hb (2) ✓

// If Get happens first:
// Get acquires lock before Set
// No conflict (read before write is always safe)

// Conclusion: Race-free ✓
```

Interview Traps

Trap 1: "I read after I write, so it's ordered"

```go
a = 1       // Write
print(a)    // Read
```

Trick question: "Is this safe in another goroutine?"

Wrong: "Yes, write happens before read."
Correct: "Only in the SAME goroutine. In another goroutine, no happens-before without synchronization."

Trap 2: "Happens-before means happens earlier in time"

Wrong.
Correct: "Happens-before is about visibility guarantees, not wall-clock time. Two operations can happen at the same nanosecond but still have happens-before relationship (or not)."

Trap 3: "Once I see a write, all future reads see it"

```go
// Goroutine 1
a = 1

// Goroutine 2
if a == 1 {  // Sees it once
    // Will future reads see a=1?
}
```

Wrong: "Yes, once visible, always visible."
Correct: "Without synchronization, CPU caching can cause reads to oscillate between old and new values. Need happens-before for guaranteed visibility."

Trap 4: "Transitive closure gives me ordering"

```go
var a, b int
ch1, ch2 := make(chan int), make(chan int)

// G1
a = 1      // (1)
ch1 <- 0   // (2)

// G2
<-ch1      // (3)
b = 1      // (4)
ch2 <- 0   // (5)

// G3
<-ch2      // (6)
print(b)   // (7) Sees b=1? YES
print(a)   // (8) Sees a=1? YES
```

Trick: "Does G3 see both writes?"

Correct: "Yes, because of transitivity:
(1) →hb (2) →hb (3) →hb (4) →hb (5) →hb (6) →hb (7) AND (8)
Chain exists, so G3 sees both."

Key Takeaways

1. **Happens-before is partial ordering** (not total)
2. **Only synchronization creates happens-before across goroutines**
3. **Must build explicit happens-before chains** to prove visibility
4. **No happens-before = no guarantee of visibility**
5. **Transitivity allows chaining** across multiple sync points
6. **All synchronization primitives have defined happens-before semantics**
7. **Always reason with happens-before, not time/intuition**

What You Should Be Thinking Now

- "How do I find the happens-before chain in my code?"
- "What's the difference between memory visibility and execution order?"
- "Why doesn't time create happens-before?"

Next: visibility-vs-ordering.md - Understanding the distinction between memory visibility and execution order.

Exercises

1. Draw happens-before diagrams for:
   - Unbuffered channel communication between 3 goroutines
   - Mutex protecting 2 writers and 3 readers
   - WaitGroup with 4 workers

2. Find the race in this code by identifying missing happens-before:

```
var config *Config
func Load() { config = &Config{} }
func Use() { if config != nil { use(config) } }
```

3. Explain why this is a race even though writes are "atomic":

```
var x int64  // 64-bit aligned
go func() { x = 1 }()
go func() { print(x) }()
```

4. Build happens-before chain for:

```
var a int
ch := make(chan int, 1)
go func() { a = 1; ch <- 0 }()
<-ch
print(a)
```

Don't continue until you can: "For any shared memory access, derive the happens-before chain that makes it safe."