

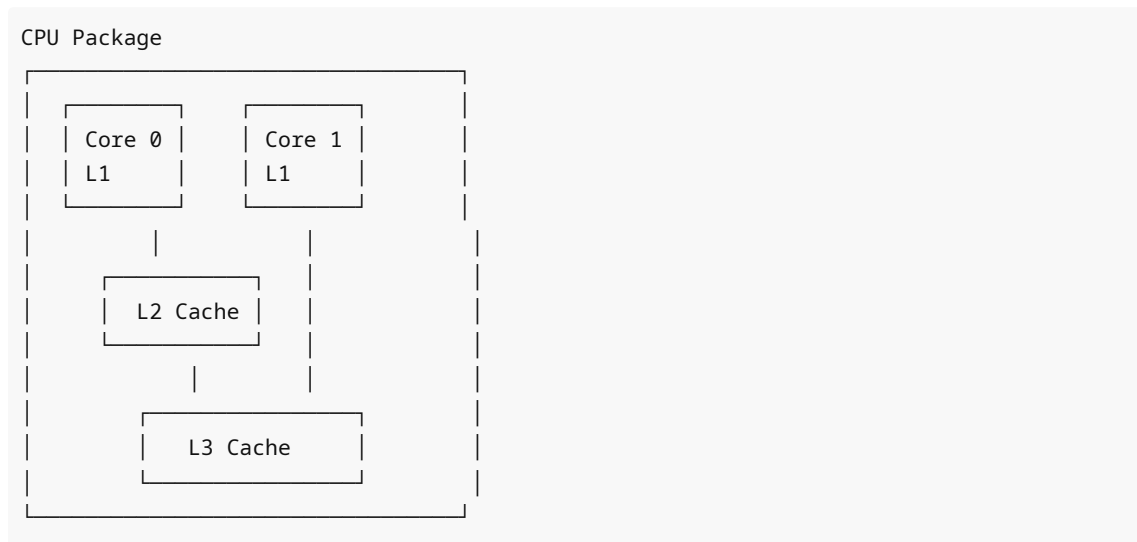
# CPU Cores, Threads, and Context Switching

## Hardware Foundation: Understanding Your Machine

Before writing concurrent code, you must understand the hardware it runs on.

### CPU Cores: The Physical Execution Units

A **CPU core** is a physical processor that can execute instructions. Modern CPUs have multiple cores.

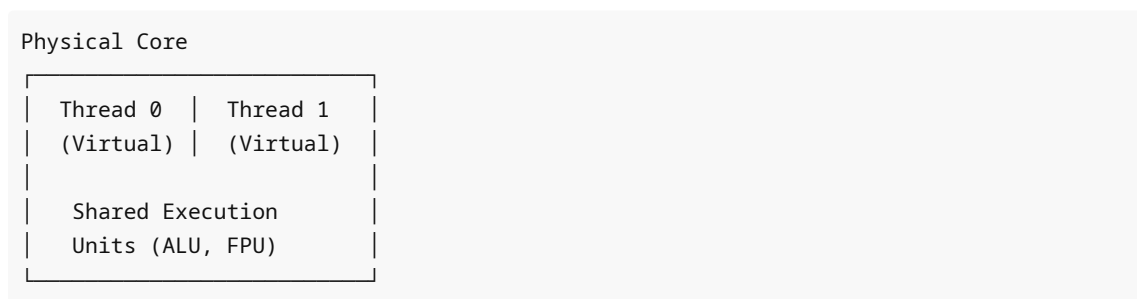


#### Key facts:

- Each core can execute **one instruction stream** at a time
- Cores have their own L1 cache (fastest, smallest)
- Cores may share L2 cache
- All cores typically share L3 cache (slower, larger)

### Hyper-Threading / SMT (Simultaneous Multi-Threading)

Many CPUs support hyper-threading (Intel) or SMT (AMD):



- One physical core presents as **two logical cores**
- They share execution units
- Effective when workloads have different resource needs
- Throughput gain: typically 20-30%, not 2x

#### In Go:

```
runtime.NumCPU() // Returns LOGICAL cores (includes hyper-threading)
// Example: 4 physical cores with hyper-threading → returns 8
```

## OS Threads vs Goroutines

### OS Threads: Heavy and Expensive

#### What it is:

An OS thread is a unit of execution managed by the operating system kernel.

#### Characteristics:

- **Size:** 1-2 MB default stack size (Linux)
- **Creation time:** ~1-2 microseconds
- **Scheduling:** Managed by OS kernel (preemptive)
- **Context switch cost:** 1-2 microseconds (involves kernel)

#### Limitations:

- Can't create millions (memory exhaustion)
- Context switching involves kernel mode transitions (expensive)
- Cache pollution on context switches

### Goroutines: Light and Cheap

#### What it is:

A goroutine is a user-space thread managed by the Go runtime.

#### Characteristics:

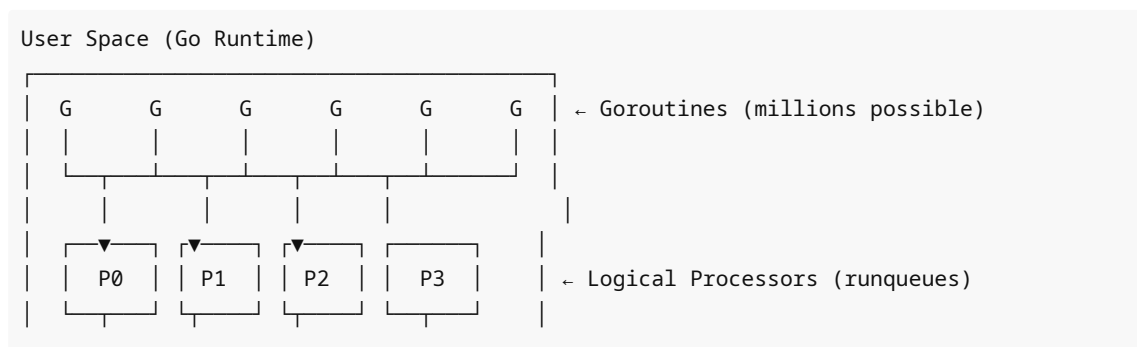
- **Size:** 2 KB initial stack (grows/shrinks dynamically)
- **Creation time:** ~100-200 nanoseconds
- **Scheduling:** Managed by Go runtime (cooperative + preemptive in Go 1.14+)
- **Context switch cost:** ~50-100 nanoseconds (pure user-space)

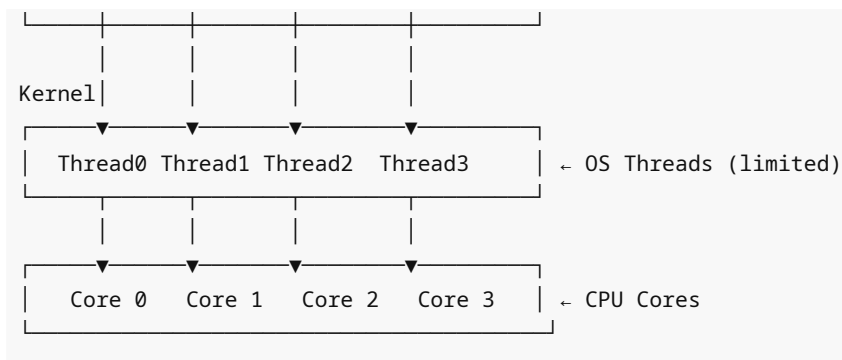
#### Benefits:

- Can create millions (if you have the memory)
- Fast context switching (no kernel involvement)
- Efficient multiplexing onto OS threads

## The Go Scheduler: M:N Threading Model

Go uses an **M:N scheduler**: M goroutines multiplex onto N OS threads.





### Components:

1. **G (Goroutine):** User code wrapped in a goroutine
2. **M (Machine):** OS thread
3. **P (Processor):** Logical processor with a local run queue

### Key principles:

- `P count = GOMAXPROCS` (default: `runtime.NumCPU()`)
- Each `P` has a local queue of runnable goroutines
- Each `M` must have a `P` to execute Go code
- `M` can block; `P` gets reassigned to another `M`

## Context Switching: The Hidden Cost

### What is a Context Switch?

Switching CPU execution from one task to another.

### What must be saved/restored:

```

Execution Context
├─ Program Counter (where in code)
├─ Stack Pointer (where in memory)
├─ Registers (CPU state)
├─ TLS (Thread Local Storage)
└─ FPU state (floating point registers)
  
```

### OS Thread Context Switch (Expensive)

1. Current thread makes syscall or gets preempted
2. CPU traps to kernel mode ← Expensive
3. Kernel saves thread state
4. Kernel picks next thread from scheduler queue
5. Kernel restores new thread state
6. CPU returns to user mode ← Expensive
7. New thread resumes execution

Total: ~1-2 microseconds + cache pollution

### Why expensive:

- Kernel mode transition (user→kernel→user)
- Cache misses (new thread has different memory working set)
- TLB flush (memory address translation cache)

## Goroutine Context Switch (Cheap)

1. Goroutine yields or gets preempted
2. Go scheduler (user space) saves G state ← Cheap (no kernel)
3. Go scheduler picks next G from run queue
4. Go scheduler restores G state
5. Goroutine resumes execution

Total: ~50-100 nanoseconds (pure user-space)

### Why cheap:

- No kernel involvement (no mode switch)
- Smaller context (stack pointer, PC, Go state)
- More predictable (less cache pollution)

## Performance Comparison

Operation	OS Thread	Goroutine	Speedup
Creation	~1-2 $\mu$ s	~100-200 ns	~10-20x
Context switch	~1-2 $\mu$ s	~50-100 ns	~20-40x
Stack size	1-2 MB	2 KB initial	~500-1000x
Max concurrent	~thousands	~millions	~1000x

## When Context Switching Kills Performance

### Example: Thrashing

```
// BAD: Too many goroutines for CPU-bound work
func processItems(items []int) {
    for _, item := range items {
        go func(item int) {
            // CPU-intensive work
            result := heavyComputation(item)
            fmt.Println(result)
        }(item)
    }
}

// If len(items) = 100,000 and NumCPU = 8:
// - 100,000 goroutines created
// - Scheduler thrashing: constant context switches
// - Cache pollution: no data locality
// - Result: Slower than sequential
```

## Real-World Failure: Docker Hub Outage (2018)

### What happened:

Docker Hub became unresponsive during high traffic.

### Root cause:

A service spawned one goroutine per incoming request with no limit. During a traffic spike:

- 50,000+ goroutines created
- Scheduler spent more time context switching than doing work
- CPU usage at 100% but no actual progress (thrashing)
- Memory exhaustion from goroutine stacks

### Fix:

Implemented a worker pool with bounded concurrency (100 workers).

### Lesson:

*Context switching is not free. Unbounded goroutines can cause scheduler thrashing.*

## Cache Effects: The Hidden Performance Killer

### Cache Hierarchy

Access Time & Size (typical)			
Level	Time	Size	Scope
L1	~1 ns	32 KB	Per-core
L2	~3-10 ns	256 KB	Per-core
L3	~20-40 ns	8-32 MB	Shared
RAM	~100 ns	8-64 GB	Shared

### Cache Coherency Cost

When multiple cores access shared memory, CPUs must maintain coherency:

Core 0	Core 1
<hr/>	
Read counter (value: 0)	
→ Copy to L1 cache	
Write counter = 1	
→ L1 cache: 1	
	Read counter
	→ Cache miss!
	→ Must fetch from Core 0
	→ Value: 1

**Cost:** Cache coherency traffic slows down parallel execution.

### In Go:

```
// Shared counter modified by many goroutines
var counter int64

for i := 0; i < 1000; i++ {
```

```
go func() {  
    atomic.AddInt64(&counter, 1) // Cache line bouncing between cores  
}()  
}
```

Each atomic operation causes cache line to bounce between cores. This is called **cache line contention**.

## GOMAXPROCS: Tuning Parallel Execution

```
runtime.GOMAXPROCS(n) // Set number of OS threads executing Go code
```

**Default:** `runtime.NumCPU()` (all logical cores)

### When to Change

#### Decrease GOMAXPROCS:

- When you want to limit CPU usage (e.g., on shared machine)
- When testing concurrency bugs (reproduce with GOMAXPROCS=1)
- When profiling to reduce noise

#### Increase GOMAXPROCS:

- (Rarely needed) Never set higher than `NumCPU()` in production
- Reasons: More context switching overhead, no additional parallelism

#### In containers/cloud:

```
// Problem: NumCPU() might return host core count, not container limit  
runtime.GOMAXPROCS(runtime.NumCPU()) // Might use too many threads  
  
// Solution: Use CPU quota  
// Docker: --cpus=2.0 sets limit to 2 cores  
// Go 1.5+ respects this by default
```

## Work Stealing: How Go Balances Load

When a `P` runs out of goroutines in its local queue:

1. Check global run queue
2. Steal from other `P`'s local queues (take half)
3. Poll network (for I/O-ready goroutines)
4. Try again

#### Benefits:

- Automatic load balancing
- Efficient use of all cores
- No manual worker assignment needed

#### Cost:

- Contention on stealing (rare)
- Randomized stealing can cause cache misses

## Interview Traps

### Trap 1: "Goroutines run on threads"

**Incomplete.** Goroutines multiplex onto a pool of OS threads managed by the Go runtime. Multiple goroutines share each thread.

**Correct answer:**

"Goroutines are scheduled by the Go runtime onto a pool of OS threads. The runtime maintains an M:N mapping where M goroutines multiplex onto N OS threads ( $N = GOMAXPROCS$ ). This allows millions of goroutines with only a handful of threads."

### Trap 2: "Setting GOMAXPROCS(1000) gives me 1000 cores of parallelism"

**Wrong.** You can't create cores in software.

**Correct answer:**

"`GOMAXPROCS` limits the number of OS threads executing Go code simultaneously, which is bounded by physical CPU cores. Setting it higher than `runtime.NumCPU()` provides no benefit and adds context switching overhead."

### Trap 3: "More goroutines = better performance"

**Wrong.** Beyond a certain point, context switching overhead dominates.

**Correct answer:**

"Performance depends on workload type. For I/O-bound work, many goroutines help by overlapping wait times. For CPU-bound work, optimal goroutine count is typically close to the number of CPU cores. Excess goroutines cause scheduler thrashing."

### Trap 4: "Context switching is cheap in Go, so it doesn't matter"

**Misleading.** Goroutine context switches are ~20-40x cheaper than thread switches, but still have cost.

**Correct answer:**

"Goroutine context switches are cheaper than OS thread switches (~100ns vs ~1-2µs), but they're not free. With millions of switches per second, the cost accumulates. Bounded concurrency prevents scheduler thrashing."

## Key Takeaways

1. **CPU cores** are the physical limit on parallelism
2. **Goroutines** are 1000x lighter than OS threads
3. **Context switching** has cost—cheap but not free
4. **GOMAXPROCS** should usually equal `NumCPU()`
5. **Unbounded goroutines** cause scheduler thrashing
6. **Cache coherency** slows down shared memory access
7. **Work stealing** provides automatic load balancing

## What You Should Be Thinking Now

- "How do I limit goroutine creation in my code?"

- "What happens when goroutines block on I/O?"
- "How do I reason about performance with goroutines?"
- "Why is concurrent code so hard to debug?"

**Next:** [why-concurrency-is-hard.md](#) - We'll explore why concurrent programming is fundamentally difficult.

---

## Exercises (Do These Before Moving On)

1. Run this to see your hardware:

```
fmt.Println("Logical CPUs:", runtime.NumCPU())
fmt.Println("GOMAXPROCS:", runtime.GOMAXPROCS(0))
```

2. Benchmark goroutine creation:

```
func BenchmarkGoroutineCreation(b *testing.B) {
    for i := 0; i < b.N; i++ {
        go func() {}()
    }
}
```

3. Create a program that spawns 1 million goroutines (each just sleeps). Monitor memory usage. Compare to creating 1 million OS threads (hint: you can't—your system will crash).
4. Explain to someone: "Why can Go have millions of goroutines but systems can't have millions of threads?"

Don't continue until you understand why `GOMAXPROCS > NumCPU()` is usually harmful.