

Concurrency

Aayush Bajaj

July 10, 2026

Contents

1	threads and shared memory	1
1.1	the canonical race	1
1.2	the memory model, honestly	2
2	locks	3
2.1	mutexes and rwlocks	3
2.2	deadlock	3
2.3	dining philosophers, with the asymmetric fix	3
3	atomics and cas	4
4	condition variables	6
5	python's gil, honestly	7
6	async/await: the event loop	8
7	message passing: csp and goroutines	8
8	lock-free wisdom (mostly: don't)	9
9	see also	10

concurrency is structure: many logical tasks in flight, interleaved on however many cpus you have (possibly one). **parallelism** is hardware: tasks literally executing at the same instant. a single-core machine juggling 400 socket connections is concurrent, not parallel; a gpu multiplying matrices is parallel, barely concurrent. you *design* concurrency; you *buy* parallelism (Tanenbaum, Andrew S., 2008).

1 threads and shared memory

a thread is an independent stream of execution inside one address space: own stack and registers, shared everything else. the sharing is the point — and the disease.

1.1 the canonical race

`counter++` is three instructions: load, add, store. two threads running the triple concurrently can interleave so both load the same old value, and one increment evaporates. in c, executed:

```
1 #include <pthread.h>
2 #include <stdio.h>
```

```

3
4 #define THREADS 4
5 #define N 1000000
6
7 long counter = 0; /* shared, unprotected */
8
9 void *worker(void *arg) {
10     for (int i = 0; i < N; i++)
11         counter++; /* load; add; store - three
12             ↪ instructions */
13     return NULL;
14 }
15
16 int main(void) {
17     pthread_t t[THREADS];
18     for (int i = 0; i < THREADS; i++) pthread_create(&t[i], NULL, worker,
19             ↪ NULL);
20     for (int i = 0; i < THREADS; i++) pthread_join(t[i], NULL);
21     printf("expected %d, got %ld, lost %ld updates\n",
22           THREADS * N, counter, (long)THREADS * N - counter);
23     return 0;
24 }

```

```

$ cc -O0 -pthread race.c -o race && ./race && ./race
expected 4000000, got 1086329, lost 2913671 updates
expected 4000000, got 1029777, lost 2970223 updates

```

nearly three quarters of the increments vanished, and the count differs run to run
— the scheduler is part of your program’s semantics now.

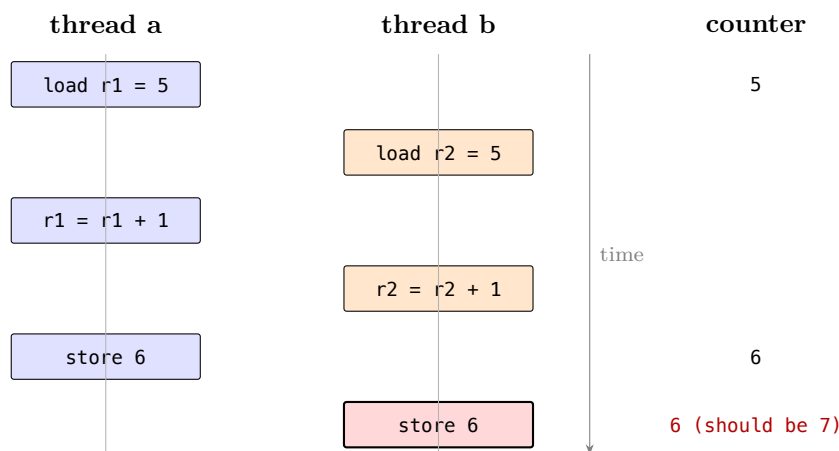


Figure 1: one lost update: both threads load 5 before either stores, so two increments produce 6. every interleaving where the loads both precede the stores loses an update.

1.2 the memory model, honestly

the interleaving picture above is already a *lie of simplification*: compilers reorder and cache your loads and stores, and cpus reorder them again (store buffers, speculative loads), so without synchronisation there is no guarantee another thread ever sees your write, let alone in order. c and c++ resolve this by fiat — a program with a data race has *undefined behaviour*, full stop — and offer a contract instead: if every access to shared data is protected by locks or atomics, the program behaves as if operations interleaved sequentially (“sequential consistency for data-race-free programs”). the practical reading: you are not writing for the hardware’s memory model, you are writing for the contract; **volatile** is not synchronisation; and a race

that “works on my machine” is working by coincidence, at **-00**, until the compiler notices.

2 locks

2.1 mutexes and rwlocks

- a **mutex** makes a region of code mutually exclusive: acquire, touch shared state, release. fix for the race above: hold a mutex across the triple (or use an atomic increment, below). granularity is the design decision — one coarse lock is simple and serial; fine-grained locks scale and breed deadlocks.
- a **rwlock** admits many concurrent readers or one writer. worth it only when readers vastly outnumber writers and the critical section is long enough to amortise its (heavier) bookkeeping; under write pressure it degrades to an expensive mutex with a starvation policy problem.

2.2 deadlock

four conditions, all necessary (coffman’s list): **mutual exclusion**, **hold-and-wait** (holding one resource while requesting another), **no preemption** (resources cannot be forcibly taken), and **circular wait** (a cycle in the who-waits-for-whom graph) (Tanenbaum, Andrew S., 2008). break any one and deadlock is impossible; the practical target is the cycle:

- **lock ordering discipline**: impose a global order on locks (by address, by id, by convention) and acquire in that order, always. no order, no cycle, no deadlock.
- alternatives: acquire-all-or-nothing (**trylock** + back off and retry), or timeouts to detect and recover rather than prevent.

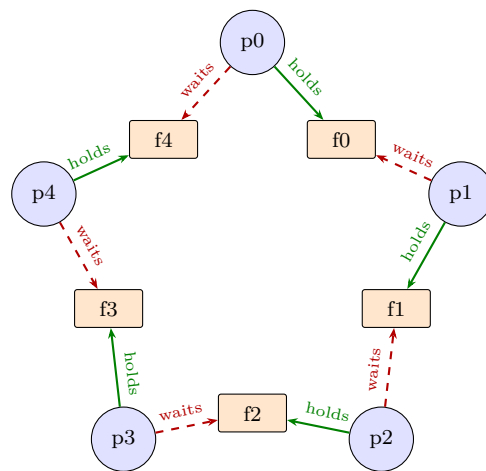


Figure 2: wait-for graph of the symmetric dining philosophers at the fatal instant: everyone holds their left fork and waits for the right. the cycle is the deadlock.

2.3 dining philosophers, with the asymmetric fix

five philosophers, five forks, everyone needs two forks to eat. if everyone grabs their left fork first, there is an interleaving where all five hold one fork and wait forever for the other — the wait-for cycle above. the **asymmetric fix**: one philosopher reaches right-first, so the acquisition order is no longer circular. executed:

```
1 import threading, time
```

```

2 N, MEALS = 5, 50
3
4
5 def philosopher(i, forks, meals, asymmetric):
6     left, right = forks[i], forks[(i + 1) % N]
7     # the fix: the last philosopher reaches for the RIGHT fork first,
8     # breaking the circular wait – lock order is no longer a cycle.
9     first, second = (right, left) if (asymmetric and i == N - 1) else (left,
10     ↪ right)
11     for _ in range(MEALS):
12         with first:
13             time.sleep(0.0002) # widen the trouble window: hold one, want
14             ↪ two
15         with second:
16             meals[i] += 1
17
18 def run(asymmetric):
19     forks = [threading.Lock() for _ in range(N)] # fresh table each run
20     meals = [0] * N
21     ts = [threading.Thread(target=philosopher, args=(i, forks, meals,
22     ↪ asymmetric),
23     ↪ daemon=True) for i in range(N)]
24
25     t0 = time.perf_counter()
26     for t in ts: t.start()
27     deadline = time.monotonic() + 6
28     for t in ts: t.join(timeout=max(0, deadline - time.monotonic()))
29     stuck = sum(t.is_alive() for t in ts)
30     tag = "asymmetric" if asymmetric else "symmetric "
31     print(f"{tag} meals={meals} stuck={stuck}
32     ↪ elapsed={time.perf_counter()-t0:.1f}s")
33
34 run(asymmetric=False) # everyone grabs left first: circular wait
35 run(asymmetric=True) # philosopher 4 grabs right first: no cycle possible

```

symmetric meals=[0, 0, 0, 0, 0] stuck=5 elapsed=6.0s
 asymmetric meals=[50, 50, 50, 50, 50] stuck=0 elapsed=0.0s

the symmetric table deadlocked instantly — zero meals, five threads permanently stuck, given up on after the 6-second timeout — while the asymmetric table finished all 250 meals in a blink. one line of lock-ordering was the entire difference.

the asymmetric fix is just lock ordering in costume: philosopher 4 acquires fork 0 before fork 4, so acquisition follows the global fork order everywhere

3 atomics and cas

- hardware gives indivisible read-modify-write instructions; the workhorse is **cas** (compare-and-swap): atomically, “if the value still equals **expect**, replace it with **new** and report success”. every lock-free structure and every lock implementation bottoms out here.
- a **spinlock** is the smallest thing you can build with it: swing a flag from 0 to 1 with cas; on failure, retry in a loop — burning cpu instead of sleeping. executed sketch (the atomicity is simulated, the algorithm is real):

```

1 import sys, threading, time
2
3 sys.setswitchinterval(1e-5) # frequent preemption: force
4 ↪ contention
5
6 class AtomicFlag:
7     """stand-in for the hardware primitive: one indivisible compare-and-swap.
8     (real cpus give you this as a single instruction – x86 CMPXCHG, arm
9     ↪ LDXR/STXR;

```

```

8  python must fake the indivisibility, but the algorithm above it is the real
   ↪ one.)"""
9  def __init__(self):
10     self._v, self._guard = 0, threading.Lock()
11
12     def compare_exchange(self, expect, new):
13         with self._guard:           # models the instruction's
   ↪ atomicity
14             if self._v == expect:
15                 self._v = new
16                 return True
17             return False
18
19     def store(self, v):
20         with self._guard:
21             self._v = v
22
23 class SpinLock:
24     def __init__(self):
25         self.flag = AtomicFlag()    # 0 = free, 1 = held
26
27     def acquire(self):
28         spins = 0
29         while not self.flag.compare_exchange(0, 1):  # try to swing 0 -> 1
30             spins += 1                                # busy-wait: no queue,
   ↪ no sleep
31         return spins
32
33     def release(self):
34         self.flag.store(0)
35
36 lock, counter = SpinLock(), 0
37 spin_counts = [0] * 4
38 start = threading.Barrier(4)
39
40 def worker(me, n):
41     global counter
42     start.wait()                                # all four hit the lock together
43     for _ in range(n):
44         spin_counts[me] += lock.acquire()
45         counter += 1                            # critical section
46         lock.release()
47
48 threads = [threading.Thread(target=worker, args=(i, 20_000)) for i in range(4)]
49 for t in threads: t.start()
50 for t in threads: t.join()
51 print(f"counter = {counter} (expected {4*20_000})")
52 print(f"wasted spins per thread = {spin_counts}")

```

```

counter = 80000 (expected 80000)
wasted spins per thread = [90685, 101711, 84784, 96901]

```

the counter is exact — mutual exclusion works — but each thread burnt roughly one failed spin per successful acquisition. that is the spinlock trade: superb when critical sections are tens of nanoseconds and the holder is running on another core; pathological when the holder is descheduled and everyone else melts a core waiting. real mutexes spin briefly then park the thread with the kernel.

- **the aba problem:** cas checks the *value*, not the *history*. thread 1 reads a, stalls; thread 2 changes a to b and back to a (freeing and reusing a node, say); thread 1's cas sees “still a” and succeeds — on top of a structurally different world. classic fixes: tag the value with a generation counter (cas on (pointer, count)), or defer reuse (hazard pointers, epochs, gc languages get this free).

4 condition variables

a mutex protects state; a **condition variable** lets threads sleep until the state is interesting. the api contract is subtle in one famous way: `wait()` can return even though nobody signalled (**spurious wakeup**), and by the time you run, another thread may have consumed the state you were promised. hence the iron idiom — **wait in a while loop, never an if**:

```
1 import threading, collections, time
2
3 buf, CAP = collections.deque(), 3
4 cv = threading.Condition()
5
6 def producer():
7     for i in range(8):
8         with cv:
9             while len(buf) >= CAP:           # WHILE, never IF: recheck after
10                 ↪ wake
11                 cv.wait()
12                 buf.append(i)
13                 print(f"produced {i} (buffer={list(buf)})")
14                 cv.notify_all()
15
16 def consumer():
17     for _ in range(8):
18         with cv:
19             while not buf:                   # guards against spurious wakeups
20                 ↪ too
21                 cv.wait()
22                 i = buf.popleft()
23                 cv.notify_all()
24                 time.sleep(0.01)             # slow consumer: producer must
25                 ↪ block
26                 print(f"consumed {i}")
27
28 t1, t2 = threading.Thread(target=producer), threading.Thread(target=consumer)
29 t1.start(); t2.start(); t1.join(); t2.join()
30 print("done, buffer:", list(buf))
```

```
produced 0 (buffer=[0])
produced 1 (buffer=[0, 1])
produced 2 (buffer=[0, 1, 2])
produced 3 (buffer=[1, 2, 3])
consumed 0
produced 4 (buffer=[2, 3, 4])
consumed 1
produced 5 (buffer=[3, 4, 5])
consumed 2
produced 6 (buffer=[4, 5, 6])
consumed 3
produced 7 (buffer=[5, 6, 7])
consumed 4
consumed 5
consumed 6
consumed 7
done, buffer: []
```

the producer sprints to fill the 3-slot buffer, then lock-steps with the slow consumer — backpressure in twenty lines, the in-process miniature of the broker discussion in

event driven.

5 python's gil, honestly

- the **global interpreter lock** serialises execution of python *bytecode*: one thread interprets at a time, with the interpreter forcing a switch every 5 ms (`sys.setswitchinterval`). it protects the interpreter's own internals (refcounts), not your invariants — compound operations like `counter += 1` or check-then-insert still race across bytecode boundaries and still need locks.
- what escapes it: c extensions release the gil around real work (numpy kernels, hashing, compression), and all blocking i/o releases it — which is why *i/o-bound* threading in python genuinely works.
- what doesn't: cpu-bound pure python. measured on this machine (python 3.13, 4 workers):

```
1 import time, threading, multiprocessing as mp
2
3 def crunch(n):                                # cpu-bound: pure bytecode, no i/o
4     s = 0
5     for i in range(n):
6         s += i * i
7     return s
8
9 N, W = 10_000_000, 4
10
11 if __name__ == "__main__":
12     crunch(N)                                # warm up the specialising interpreter
13
14     t0 = time.perf_counter()
15     for _ in range(W): crunch(N)
16     seq = time.perf_counter() - t0
17
18     t0 = time.perf_counter()
19     ts = [threading.Thread(target=crunch, args=(N,)) for _ in range(W)]
20     for t in ts: t.start()
21     for t in ts: t.join()
22     thr = time.perf_counter() - t0
23
24     t0 = time.perf_counter()
25     with mp.Pool(W) as pool:
26         pool.map(crunch, [N] * W)
27     proc = time.perf_counter() - t0
28
29     print(f"4 x crunch, sequential : {seq:5.2f} s")
30     print(f"4 x crunch, 4 threads   : {thr:5.2f} s  <- gil: one interpreter at
31         ↪ a time")
32     print(f"4 x crunch, 4 processes: {proc:5.2f} s  <- four gils, real
33         ↪ parallelism")
```

```
4 x crunch, sequential : 1.40 s
4 x crunch, 4 threads   : 1.34 s  <- gil: one interpreter at a time
4 x crunch, 4 processes: 0.62 s  <- four gils, real parallelism
```

four threads bought nothing; four processes (one interpreter, one gil each — at the price of pickling everything across) genuinely parallelised.

cpython 3.13 ships an experimental free-threaded build (pep 703) with the gil removed; the ecosystem is still catching up

workload	use	why
cpu-bound python	multiprocessing	sidesteps the gil; per-process memory cost
cpu-bound in numpy/c	threading	extensions release the gil during the real work
i/o-bound, moderate fan-out	threading	blocked threads release the gil; simple code
i/o-bound, huge fan-out	asyncio	thousands of waits on one thread, no stack each

6 asyncio/await: the event loop

one thread, many suspended functions. **await** marks the exact points where a coroutine can be paused; between them it runs uninterrupted (no data races on the gaps — a scheduling guarantee locks can't give you). the loop runs whichever coroutine is ready:

```

1 import asyncio, time
2
3 async def fetch(name, delay):
4     print(f"{time.perf_counter()-T0:4.2f}s {name}: request sent")
5     await asyncio.sleep(delay)           # stand-in for network i/o; yields to
6     ↪ the loop
7     print(f"{time.perf_counter()-T0:4.2f}s {name}: response in")
8     return delay
9
10 async def main():
11     # three "requests" run concurrently on ONE thread: while one awaits,
12     # the event loop runs the others.
13     results = await asyncio.gather(fetch("a", 0.3), fetch("b", 0.2), fetch("c",
14     ↪ 0.1))
15     print(f"{time.perf_counter()-T0:4.2f}s total {sum(results):.1f}s of i/o "
16           f"in {time.perf_counter()-T0:.2f}s wall time")
17
18 T0 = time.perf_counter()
19 asyncio.run(main())

```

```

0.00s a: request sent
0.00s b: request sent
0.00s c: request sent
0.11s c: response in
0.21s b: response in
0.31s a: response in
0.31s total 0.6s of i/o in 0.31s wall time

```

0.6 seconds of waiting overlapped into 0.31 of wall time, on one thread. the model's one commandment: **never block the loop** — a synchronous `requests.get` or a heavy computation inside a coroutine freezes every other task, because cooperation is the whole scheduler. blocking work gets shipped to a thread (`asyncio.to_thread`) or a process pool.

7 message passing: csp and goroutines

the other church: don't share memory and synchronise — pass messages and let ownership move with the data. go builds the whole style into the language: **goroutines** (runtime-scheduled tasks with kilobyte stacks, cheap enough to spawn per request) and typed **channels** (send blocks until receive on an unbuffered channel — the handoff is the synchronisation). a worker pool, executed:

hoare's communicating sequential processes, 1978; go's slogan is 'do not communicate by sharing memory; instead, share memory by communicating'

```

1 package main
2
3 import (

```

```

4      "fmt"
5      "sync"
6  )
7
8  func worker(id int, jobs <-chan int, results chan<- string, wg *sync.WaitGroup)
9  ↪ {
10     defer wg.Done()
11     for j := range jobs { // receive until channel closed
12         results <- fmt.Sprintf("worker %d squared %d -> %d", id, j,
13             ↪ j*j)
14     }
15 }
16
17 func main() {
18     jobs := make(chan int)           // unbuffered: send blocks until
19     ↪ received
20     results := make(chan string, 8) // buffered: senders don't wait for the
21     ↪ printer
22     var wg sync.WaitGroup
23
24     for w := 1; w <= 3; w++ {
25         wg.Add(1)
26         go worker(w, jobs, results, &wg) // goroutine: ~kilobytes of
27         ↪ stack, cheap
28     }
29     go func() {
30         for j := 1; j <= 6; j++ {
31             jobs <- j
32         }
33         close(jobs) // no more work: ranging workers drain and exit
34     }()
35     go func() { wg.Wait(); close(results) }()
36
37     for r := range results { // main blocks here until results closes
38         fmt.Println(r)
39     }
40 }

```

```

$ go run chan.go
worker 3 squared 3 -> 9
worker 3 squared 4 -> 16
worker 3 squared 5 -> 25
worker 3 squared 6 -> 36
worker 1 squared 1 -> 1
worker 2 squared 2 -> 4

```

note the scrambled output order: three workers raced for jobs and worker 3 was hungriest. no locks appear in the program — the channels *are* the synchronisation, and the shared-state disease has nowhere to live because no state is shared. (deadlock is still possible — two goroutines each blocked sending to the other — the wait-for cycle again, wearing channel clothes.) a from-scratch exercise in exactly this style over real sockets: [lan messenger](#).

8 lock-free wisdom (mostly: don't)

- lock-free algorithms replace “block on contention” with “retry on contention” (cas loops). they shine in exactly two situations: signal handlers/interrupt context where you *cannot* block, and very hot, very small structures (queues, counters) where lock overhead dominates.

- the costs are not linear: aba, memory reclamation (when may a node be freed while lock-free readers roam?), memory-order reasoning per architecture, and bugs that reproduce once a week under production load only.
- the professional default: a coarse mutex, measured. then finer locks, measured. then a *proven* lock-free structure from a library, measured. writing your own is a research project cosplaying as an optimisation — the same escalation discipline as **premature optimisation** everywhere else.
- and one level up, the architectural dodge: the **build-system dag** runs thousands of jobs in parallel with *zero* locks in user code — dependencies express all the ordering, and the scheduler does the rest. structure beats synchronisation.

9 see also

- **build systems** — the dag as a lock-free parallel execution plan
- **lan messenger** — sockets, select loops and message passing in anger
- **event driven** — queues and backpressure between processes instead of threads
- **linux** — the kernel machinery under threads, scheduling and futexes
- **containers** — process isolation when sharing an address space is the wrong idea entirely

references

Tanenbaum, Andrew S. (2008). *Modern Operating Systems*, Pearson.