

Pertemuan 6 — *Concurrency: Thread, Goroutine, Actor, Reactive*

Satu soal, banyak cara mengerjakan banyak hal sekaligus

Hendri Karisma

Semester Ganjil 2026/2027

Program Studi Teknik Informatika, STMIK Tazkia

Alur pertemuan hari ini

1. *Process, thread, dan context switch*
2. *Thread + lock di Java: race condition dan deadlock*
3. *Virtual thread Java 21*
4. *CSP di Go: goroutine, channel, select, worker pool*
5. *Actor model: Erlang dan Akka*
6. *Reactive programming: ReactiveX dan backpressure*
7. *Event loop dan async/await*
8. Pengaruh bahasa: GIL Python, BEAM, JVM, Go
9. Satu soal, tiga model
10. Praktikum dan tugas

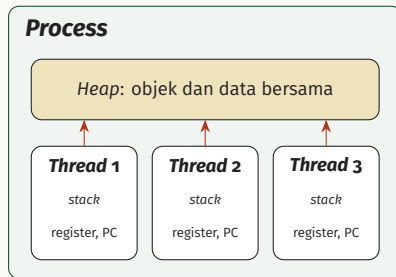
Sasaran pertemuan ini

Anda dapat menulis program *concurrent* dengan beberapa model, mengenali *race condition* dan *deadlock*, dan memilih model yang paling cocok untuk satu masalah berdasarkan risiko dan pengukurannya.

Process, thread, dan context switch

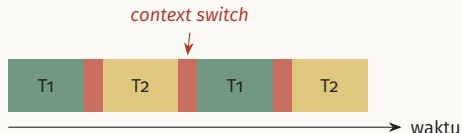
Process dan thread

- ▶ **Process** — program yang sedang berjalan, dengan ruang memori sendiri. Satu *process* tidak bisa membaca memori *process* lain tanpa bantuan sistem operasi.
- ▶ **Thread** — jalur eksekusi di dalam *process*. Semua *thread* dalam satu *process* berbagi *heap* yang sama, tetapi masing-masing punya *stack* dan register sendiri.
- ▶ Berbagi memori itu cepat, dan justru itu sumber bahayanya: dua *thread* bisa mengubah data yang sama pada saat bersamaan.



Context switch: berganti *thread* tidak gratis

- ▶ Satu *core* hanya menjalankan satu *thread* pada satu saat. Penjadwal sistem operasi bergantian memberi giliran.
- ▶ Setiap pergantian menyimpan register dan status *thread* lama, memuat yang baru, dan sering membuat isi *cache* menjadi tidak berguna.
- ▶ Biayanya kira-kira orde mikrodetik. Kecil untuk sekali, terasa ketika terjadi ratusan ribu kali per detik.
- ▶ *Thread* OS juga memakan memori untuk *stack*-nya, sehingga “satu *thread* per koneksi” sulit menampung puluhan ribu koneksi.



Intinya

Model *concurrency* modern (*goroutine*, *virtual thread*, proses BEAM, *event loop*) pada dasarnya berusaha mengurangi dua biaya ini: memori per tugas dan ongkos berganti tugas.

Pengingat: *concurrency* bukan *parallelism*

Concurrency

Struktur program: banyak tugas yang **bisa** berjalan sendiri-sendiri, dalam rentang waktu yang sama. Bisa terjadi di satu *core*.

Parallelism

Eksekusi: banyak tugas **benar-benar** berjalan pada detik yang sama. Butuh lebih dari satu *core*.

Jenis pekerjaan		Yang dibutuhkan	Contoh
<i>I/O-bound</i> (menunggu jaringan, disk)	(menunggu jaringan, disk)	<i>Concurrency</i> : banyak tugas menunggu sekaligus	API gateway, web server, crawler
<i>CPU-bound</i> (menghitung)		<i>Parallelism</i> : banyak <i>core</i> bekerja sekaligus	Kompresi video, perkalian matriks, training

Pertanyaan pertama sebelum memilih model: pekerjaan saya **lebih banyak menunggu** atau **lebih banyak menghitung**?

Thread + lock di Java

Race condition: n++ bukan satu langkah

```
class Penghitung {  
    int n = 0;  
    void tambah() { n++; }  
}  
  
Penghitung p = new Penghitung();  
Runnable kerja = () -> {  
    for (int i = 0; i < 1_000_000; i++) p.tambah();  
};  
Thread a = new Thread(kerja), b = new Thread(kerja);  
a.start(); b.start();  
a.join(); b.join();  
System.out.println(p.n); // sering < 2000000
```

n++ sebenarnya tiga langkah:

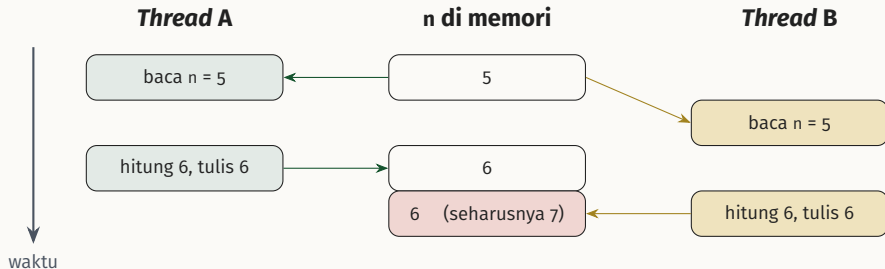
1. baca n dari memori
2. tambah satu
3. tulis kembali

Bila dua *thread* membaca nilai yang sama sebelum salah satunya menulis, satu penambahan **hilang**.

Intinya

Hasilnya berbeda setiap kali dijalankan. Bug *concurrency* sulit karena ia tidak selalu muncul.

Bagaimana satu penambahan hilang



Intinya

Bagian kode yang membaca lalu mengubah data bersama disebut **critical section**. Ia harus dijalankan oleh satu *thread* saja pada satu waktu (*mutual exclusion*).

Tiga cara menambal di Java

synchronized

```
class Penghitung {  
    int n = 0;  
    synchronized void tambah() {  
        n++;  
    }  
}
```

Paling sederhana. Kunci melekat pada objek.

ReentrantLock

```
final Lock kunci = new ReentrantLock();  
void tambah() {  
    kunci.lock();  
    try { n++; }  
    finally { kunci.unlock(); }  
}
```

Lebih fleksibel: tryLock dengan *timeout*, *fairness*. Wajib unlock di *finally*.

AtomicInteger

```
final AtomicInteger n =  
    new AtomicInteger();  
void tambah() {  
    n.incrementAndGet();  
}
```

Tanpa *lock*: memakai instruksi atomik CPU (*compare-and-swap*). Cepat, tetapi hanya untuk operasi sederhana.

Intinya

Ketiganya benar. Yang berbeda adalah ongkosnya: semakin banyak *thread* berebut kunci yang sama, semakin banyak waktu habis untuk menunggu, bukan bekerja.

Deadlock: dua thread saling menunggu selamanya

```
Object rekA = new Object(), rekB = new Object();

// transfer A -> B
new Thread(() -> {
    synchronized (rekA) {
        tidur(50);
        synchronized (rekB) { /* pindah dana */ }
    }
}).start();

// transfer B -> A, urutan kunci terbalik
new Thread(() -> {
    synchronized (rekB) {
        tidur(50);
        synchronized (rekA) { /* pindah dana */ }
    }
}).start();
```

Empat syarat Coffman; *deadlock* terjadi bila keempatnya terpenuhi:

1. *Mutual exclusion*: sumber daya hanya bisa dipegang satu pihak
2. *Hold and wait*: memegang satu, menunggu yang lain
3. *No preemption*: kunci tidak bisa direbut paksa
4. *Circular wait*: A menunggu B, B menunggu A

Diagnosis: `jstack <pid>` menampilkan *thread* yang saling menunggu.

Menghindari *deadlock*

- ▶ **Urutan kunci yang tetap** — selalu kunci rekening dengan nomor lebih kecil dulu. *Circular wait* jadi mustahil.
- ▶ **tryLock dengan timeout** — bila gagal mendapat kunci kedua dalam waktu tertentu, lepaskan kunci pertama dan coba lagi.
- ▶ **Perkecil critical section** — pegang kunci sesingkat mungkin; jangan memanggil jaringan atau disk sambil memegang kunci.
- ▶ **Hindari berbagi** — bila tidak ada data bersama, tidak ada yang perlu dikunci. Inilah gagasan di balik CSP dan *actor model*.

Intinya

Di sistem terdistribusi, *lock* melintasi jaringan (*distributed lock*) jauh lebih sulit lagi: pemegang kunci bisa mati, atau jaringannya putus, sambil tetap “memegang” kunci. Karena itu desain yang baik meminimalkan kebutuhan akan kunci bersama.

Thread pool dan virtual thread (Java 21)

Thread pool klasik — jumlah thread OS dibatasi

```
ExecutorService pool = Executors.newFixedThreadPool(8);  
Future<Integer> f = pool.submit(() -> hitung());
```

Virtual thread — satu tugas, satu thread murah

```
try (var exec =  
    Executors.newVirtualThreadPerTaskExecutor()) {  
    IntStream.range(0, 10_000).forEach(i ->  
        exec.submit(() -> {  
            Thread.sleep(Duration.ofSeconds(1));  
            return i;  
        }));  
} // menunggu semua tugas selesai
```

- ▶ *Virtual thread* dikelola JVM, bukan sistem operasi. Banyak *virtual thread* ditumpangkan pada sedikit *thread OS* (*carrier*).
- ▶ Saat menunggu I/O, *virtual thread* dilepas dari *carrier*-nya, sehingga *carrier* bisa menjalankan tugas lain.
- ▶ Sepuluh ribu tugas yang masing-masing tidur satu detik selesai dalam sekitar satu detik, tanpa sepuluh ribu *thread OS*.
- ▶ Gaya kodenya tetap sinkron dan mudah dibaca. Untuk pekerjaan *CPU-bound*, ia tidak menambah *core*.

Sumber: JEP 444, Java 21 (2023).

CSP di Go: *goroutine* dan *channel*

CSP: berkomunikasi, bukan berbagi

- ▶ **CSP** (*Communicating Sequential Processes*) dirumuskan Tony Hoare (1978): proses-proses berurutan yang hanya bertukar data lewat saluran.
- ▶ Go mengambil gagasan ini: **goroutine** adalah fungsi yang berjalan *concurrent*; **channel** adalah pipa bertipe antar *goroutine*.
- ▶ *Goroutine* sangat murah: *stack* awalnya hanya beberapa KB dan tumbuh bila perlu. Ratusan ribu *goroutine* dalam satu program itu wajar.
- ▶ *Runtime* Go menjadwalkan banyak *goroutine* di atas sedikit *thread* OS (penjadwalan M:N).

Semboyan Go

"Do not communicate by sharing memory; instead, share memory by communicating."

Sumber: Effective Go, dokumentasi resmi Go.



Goroutine, channel, dan sync.WaitGroup

```
func main() {  
    urls := []string{"a.id", "b.id", "c.id"}  
    hasil := make(chan string, len(urls))  
    var wg sync.WaitGroup  
  
    for _, u := range urls {  
        wg.Add(1)  
        go func(u string) {  
            defer wg.Done()  
            hasil <- ambil(u) // kirim ke channel  
        }(u)  
    }  
    wg.Wait() // tunggu semua goroutine  
    close(hasil) // tidak ada lagi yang mengirim  
    for h := range hasil {  
        fmt.Println(h)  
    }  
}
```

- ▶ go f() menjalankan f sebagai *goroutine* baru.
- ▶ ch <- x mengirim, x := <-ch menerima. Pada *channel* tanpa *buffer*, pengirim menunggu sampai ada penerima.
- ▶ WaitGroup menghitung berapa *goroutine* belum selesai.
- ▶ close memberi tahu penerima bahwa data sudah habis, sehingga range berhenti.
- ▶ Deteksi race: go run -race main.go.

select: menunggu beberapa kemungkinan sekaligus

```
hasil := make(chan string, 1)
go func() { hasil <- panggilLayanan() }()

select {
case r := <-hasil:
    fmt.Println("jawaban:", r)
case <-time.After(500 * time.Millisecond):
    fmt.Println("timeout, pakai cadangan")
}
```

- ▶ select menunggu *channel* mana yang siap lebih dulu.
- ▶ Pola ini adalah *timeout* untuk panggilan jaringan, langsung menambal *fallacy* “*latency* itu nol” dari pertemuan 1.
- ▶ Di kode produksi, pola yang sama ditulis dengan `context.WithTimeout` agar pembatalan ikut menjalar ke pemanggilan berikutnya.

Intinya

Buffer 1 pada *channel* `hasil` penting: tanpa itu, *goroutine* pengirim akan tertahan selamanya setelah *timeout* (*goroutine leak*).

Worker pool: membatasi concurrency

```
func worker(jobs <-chan int, out chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    for j := range jobs { out <- j * j }
}

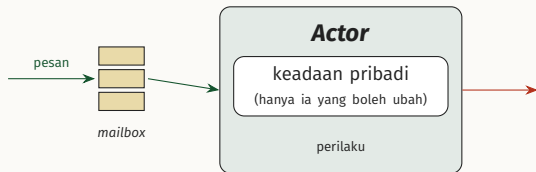
func main() {
    jobs, out := make(chan int, 100), make(chan int, 100)
    var wg sync.WaitGroup
    for w := 0; w < 4; w++ { // 4 pekerja
        wg.Add(1); go worker(jobs, out, &wg)
    }
    for j := 1; j <= 20; j++ { jobs <- j }
    close(jobs)
    go func() { wg.Wait(); close(out) }()
    total := 0
    for r := range out { total += r }
    fmt.Println(total)
}
```

- ▶ Jumlah pekerja tetap (di sini 4), berapa pun jumlah tugasnya.
- ▶ Berguna ketika sumber daya di belakangnya terbatas: koneksi basis data, kuota API, jumlah *core*.
- ▶ <-chan dan chan<- membatasi arah: pekerja hanya boleh menerima dari jobs dan mengirim ke out.
- ▶ Tidak ada *mutex* sama sekali: total hanya disentuh satu *goroutine*.

Actor model: Erlang dan Akka

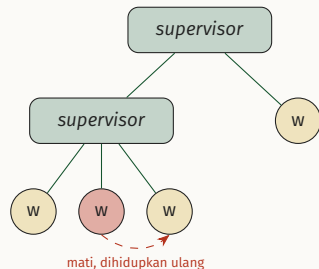
Actor model

- ▶ Dirumuskan Carl Hewitt dan rekan (1973). Semua hal adalah **actor**.
- ▶ Tiap *actor* punya keadaan pribadi dan sebuah **mailbox**. Tidak ada yang boleh menyentuh keadaannya kecuali dirinya sendiri.
- ▶ Saat menerima satu pesan, *actor* hanya boleh:
 1. mengirim pesan ke *actor* lain,
 2. membuat *actor* baru,
 3. menentukan perilakunya untuk pesan berikutnya.
- ▶ Pesan diproses **satu per satu**, sehingga tidak perlu *lock*.



Erlang dan BEAM: “let it crash”

- ▶ Erlang lahir di Ericsson (1980-an) untuk sentral telepon yang tidak boleh mati.
- ▶ Mesin virtualnya, **BEAM**, menjalankan jutaan proses ringan. Penjadwalnya *preemptive*: tiap proses diberi jatah langkah, sehingga satu proses rakus tidak membekukan yang lain.
- ▶ **Let it crash** — jangan menulis penanganan galat untuk setiap kemungkinan. Biarkan proses yang bermasalah mati, lalu **supervisor** menghidupkannya kembali dalam keadaan bersih.
- ▶ Elixir berjalan di BEAM yang sama dengan sintaks yang lebih modern.



Supervision tree: kegagalan diisolasi di cabangnya, tidak menjar ke seluruh sistem.

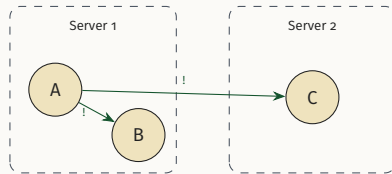
Akka (Scala): actor bertipe dengan supervision

```
object Penghitung {  
  sealed trait Pesan  
  case object Tambah extends Pesan  
  case class Ambil(balas: ActorRef[Int]) extends Pesan  
  def apply(n: Int = 0): Behavior[Pesan] =  
    Behaviors.receiveMessage {  
      case Tambah      => apply(n + 1)  
      case Ambil(balas) => balas ! n; Behaviors.same  
    }  
}  
  
// dibungkus supervisor: mulai ulang bila gagal  
val aman = Behaviors.supervise(Penghitung())  
  .onFailure[Exception](SupervisorStrategy.restart)  
val ref = context.spawn(aman, "penghitung")  
ref ! Penghitung.Tambah
```

- ▶ Keadaan *n* tidak pernah disimpan di variabel bersama; ia ada di *behavior* berikutnya (`apply(n + 1)`).
- ▶ `!` (`tell`) mengirim pesan tanpa menunggu.
- ▶ `ActorRef[Int]` pada `Ambil` adalah alamat untuk membalas: pola *request-response* di atas pesan satu arah.
- ▶ `SupervisorStrategy.restart` membuat *actor* dimulai ulang dengan keadaan awal ketika terjadi *exception*.

Location transparency: jembatan ke sistem terdistribusi

- Pengirim hanya memegang ActorRef. Ia tidak tahu, dan tidak perlu tahu, apakah *actor* tujuannya di *thread* yang sama, di *core* lain, atau di server lain.
- Akka Cluster dan Erlang/OTP memakai gagasan ini untuk menyebar *actor* ke banyak mesin.
- Tetapi jaringan tetap jaringan: pengiriman pesan antar mesin bisa hilang. Akka secara bawaan menjanjikan *at-most-once delivery*; bila perlu jaminan lebih, aplikasi menambah *acknowledgement* dan *retry* sendiri.



Kode A sama untuk mengirim ke B (lokal) maupun ke C (jarak jauh).

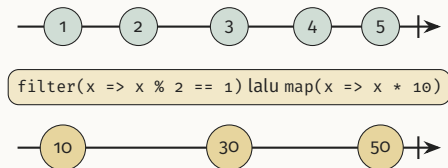
Intinya

Ini alasan *actor model* disukai di sistem terdistribusi: model mentalnya sudah “pesan bisa terlambat atau hilang” sejak awal, sehingga tidak ada ilusi panggilan lokal.

Reactive programming

ReactiveX: data sebagai aliran

- ▶ **ReactiveX** (Rx) berawal di Microsoft (Rx.NET), lalu menyebar ke RxJava, RxJS, dan bahasa lain.
- ▶ Gagasannya: perlakukan peristiwa (klik, pesan, hasil HTTP) sebagai **Observable**, aliran nilai sepanjang waktu.
- ▶ Aliran diolah dengan **operator** seperti `map`, `filter`, `debounce`, `merge`, `switchMap` — mirip mengolah *list*, tetapi nilainya datang belakangan.



Marble diagram: tiap bola satu nilai; garis tegak di kanan berarti aliran selesai.

RxJS dan RxJava dalam praktik

RxJS — kotak pencarian yang tidak membanjiri server

```
fromEvent(kotak, 'input').pipe(  
  map(e => e.target.value),  
  debounceTime(300),           // tunggu jeda ketik  
  distinctUntilChanged(),  
  switchMap(q =>               // batalkan yang lama  
    ajax.getJSON('/cari?q=' + q))  
).subscribe(tampilkan);
```

RxJava — aliran besar dengan *backpressure*

```
Flowable.range(1, 1_000_000)  
  .onBackpressureBuffer(10_000)  
  .observeOn(Schedulers.computation())  
  .map(x -> proses(x))  
  .subscribe(System.out::println);
```

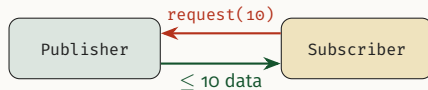
Flowable (bukan Observable) dipakai bila produsen bisa lebih cepat daripada konsumen.

Intinya

Contoh RxJS menunjukkan kekuatan Rx: logika “tunggu pengguna berhenti mengetik, abaikan yang sama, batalkan permintaan lama” cukup ditulis dalam empat operator.

Backpressure dan Reactive Streams

- ▶ Masalah: produsen mengirim 10.000 pesan per detik, konsumen hanya sanggup 1.000. Tanpa kendali, antrian membengkak sampai memori habis.
- ▶ **Backpressure**: konsumen yang menentukan laju. Ia meminta `request(n)`, produsen hanya boleh mengirim sebanyak itu.
- ▶ Spesifikasi **Reactive Streams** membakukan empat antarmuka: `Publisher`, `Subscriber`, `Subscription`, `Processor`. Java 9 memasukkannya sebagai `java.util.concurrent.Flow`.
- ▶ Dipakai oleh RxJava, Project Reactor (Spring WebFlux), dan Akka Streams.



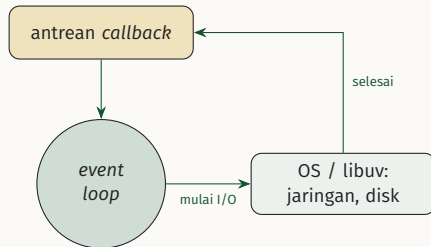
Intinya

Backpressure adalah gagasan yang sama dengan *flow control* TCP pada pertemuan 2: penerima memberi tahu pengirim berapa yang sanggup ia terima.

Event loop dan async/await

Event loop: satu thread, tidak pernah menunggu diam

- ▶ Node.js (2009) menjalankan kode JavaScript di **satu thread**. Operasi I/O diserahkan ke sistem operasi atau *thread pool* milik libuv.
- ▶ Saat I/O selesai, *callback*-nya masuk antrean, lalu *event loop* menjalankannya satu per satu.
- ▶ Karena tidak ada dua kode JavaScript yang berjalan bersamaan, tidak ada *race* pada variabel biasa.
- ▶ Harganya: satu perhitungan berat (misalnya *loop* besar) membekukan **semua** permintaan lain.



async/await: kode asinkron yang terbaca seperti sinkron

Python asyncio

```
import asyncio, time

async def ambil(i):
    await asyncio.sleep(1)    # pura-pura menunggu jaringan
    return i

async def main():
    t = time.perf_counter()
    hasil = await asyncio.gather(
        *(ambil(i) for i in range(100)))
    print(len(hasil), time.perf_counter() - t)
    # sekitar 1 detik, bukan 100 detik

asyncio.run(main())
```

JavaScript (Node.js)

```
const respons = await Promise.all(
    urls.map(u => fetch(u))
);
```

- ▶ await berarti “lepaskan giliran sampai hasil ini siap”.
- ▶ Seratus permintaan menunggu **bersamaan**, tetap di satu *thread*.
- ▶ Jebakan: memanggil fungsi yang memblokir (misalnya `time.sleep` atau *driver* basis data sinkron) di dalam fungsi `async` membekukan seluruh *event loop*.

Pengaruh bahasa pada *concurrency*

Bahasa menentukan model yang murah dipakai

Bahasa	Unit <i>concurrency</i> bawaan	Akibat praktis
Python (CPython)	<i>Thread</i> OS, tetapi dibatasi GIL : hanya satu <i>thread</i> menjalankan <i>bytecode</i> Python pada satu saat	<i>Thread</i> berguna untuk I/O, tidak untuk CPU. Untuk CPU: multiprocessing atau pustaka <i>native</i> (NumPy). Versi <i>free-threaded</i> tanpa GIL mulai diuji sejak Python 3.13
Java	<i>Thread</i> OS; <i>virtual thread</i> sejak Java 21	<i>Thread</i> benar-benar paralel. <i>Virtual thread</i> membuat gaya “satu tugas satu <i>thread</i> ” murah lagi
Go	<i>Goroutine</i> dengan penjadwal M:N	Paralel di semua <i>core</i> secara bawaan; ratusan ribu <i>goroutine</i> wajar
Erlang/Elixir	Proses BEAM, penjadwal <i>preemptive</i>	Jutaan proses, isolasi kegagalan per proses; lemah untuk hitungan angka berat
JavaScript	<i>Event loop</i> satu <i>thread</i> ; <i>worker thread</i> terpisah	Unggul untuk I/O; kerja CPU harus dipindah ke <i>worker</i>
Rust	<i>Thread</i> OS dan <i>async</i> ; kompilator menolak <i>data race</i>	Kesalahan berbagi data ditangkap sebelum program berjalan

Satu soal, tiga model

Soal: saldo bersama yang diperbarui ribuan kali

- ▶ Sebuah dompet digital punya satu saldo. Seribu tugas masing-masing menambah saldo sebanyak 1.000 kali. Hasil akhir **wajib** tepat 1.000.000.
- ▶ Kita selesaikan dengan tiga model, lalu bandingkan kebenaran, kecepatan, dan kemudahan membacanya.

Model	Siapa pemilik saldo?	Bagaimana perubahan masuk?
Java <i>thread</i> + <i>lock</i>	Variabel bersama, dijaga kunci	Tiap <i>thread</i> mengunci, menambah, melepas
Go <i>goroutine</i> + <i>channel</i>	Satu <i>goroutine</i> pemilik	<i>Goroutine</i> lain mengirim “tambah 1” lewat <i>channel</i>
Akka <i>actor</i>	Satu <i>actor</i> Penghitung	Pesan Tambah masuk <i>mailbox</i> , diproses satu per satu

Intinya

Model pertama **membagi data lalu menjaganya**. Dua model terakhir **tidak membagi data sama sekali**: hanya satu pemilik, yang lain mengirim pesan.

Tiga jawaban

Java

```
AtomicLong saldo = new AtomicLong();
try (var ex = Executors
    .newVirtualThreadPerTaskExecutor()) {
    for (int t = 0; t < 1000; t++)
        ex.submit(() -> {
            for (int i = 0; i < 1000; i++)
                saldo.incrementAndGet();
        });
}
System.out.println(saldo.get());
```

Akka: lihat actor Penghitung di slide sebelumnya; kirim Tambah 1.000.000 kali lalu Ambil.

Go

```
tambah := make(chan int, 1024)
selesai := make(chan int)
go func() {                                // pemilik tunggal
    saldo := 0
    for d := range tambah { saldo += d }
    selesai <- saldo
}()
var wg sync.WaitGroup
for t := 0; t < 1000; t++ {
    wg.Add(1)
    go func() { defer wg.Done()
        for i := 0; i < 1000; i++ { tambah <- 1 } }()
    }
wg.Wait(); close(tambah)
fmt.Println(<-selesai)
```

Membandingkan model *concurrency*

Model	Paling cocok untuk	Risiko khas	Alat bantu deteksi
<i>Thread + lock</i>	Data bersama kecil, operasi sederhana	<i>Race</i> , <i>deadlock</i> , kunci jadi rebutan	<i>jstack</i> , <i>thread sanitizer</i>
<i>Goroutine + channel</i>	Layanan jaringan, <i>pipeline</i> , <i>worker pool</i>	<i>Goroutine leak</i> , <i>channel macet</i>	<code>go run -race</code> , <i>pprof</i>
<i>Actor</i>	Banyak entitas berkeadaan (akun, sesi, perangkat), sistem terdistribusi	<i>Mailbox</i> menumpuk, pesan hilang antar mesin	<i>Metric</i> ukuran <i>mailbox</i> , <i>tracing</i>
<i>Reactive streams</i>	Aliran peristiwa, UI, <i>streaming data</i>	Rantai operator sulit di- <i>debug</i>	<i>Marble test</i> , log per operator
<i>Event loop / async</i>	Banyak koneksi I/O, kode sederhana	Satu tugas berat mem-bekukan semua	<i>Event loop lag metric</i>

Intinya

Tidak ada model terbaik. Pilih berdasarkan bentuk masalah, lalu **ukur**: kebenaran dulu (hasil selalu tepat), baru kecepatan.

Praktikum dan tugas

Praktikum: satu soal, tiga model

1. Jalankan versi Java **tanpa** sinkronisasi sepuluh kali; catat sebaran hasilnya.
2. Perbaiki dengan `synchronized`, `ReentrantLock`, dan `AtomicLong`. Ukur waktunya untuk 2, 8, dan 1000 *thread*.
3. Tulis versi Go dengan *channel*; jalankan dengan detektor *race*:

```
go run -race saldo.go  
time go run saldo.go
```

4. Tulis versi Akka (proyek templat `sbt new akka/akka-scala-seed.g8`, atau gunakan Apache Pekko yang API-nya setara).
5. Isi tabel: model, hasil (benar/salah), waktu median dari 5 kali jalan, jumlah baris kode.

Intinya

Jangan hanya melaporkan waktu. Jelaskan **mengapa** model tertentu lebih lambat: apakah karena rebutan kunci, antrean *channel*, atau ongkos pesan per *actor*?

Tugas untuk pertemuan berikutnya

1. Selesaikan laporan praktikum “satu soal, tiga model” (tabel pengukuran + analisis satu halaman).
2. Untuk tugas besar UTS: tentukan model *concurrency* di tiap layanan Anda dan alasannya. Di mana ada data bersama? Siapa pemiliknya?
3. Baca van Steen & Tanenbaum Bab 3 (*processes*) dan Bab 5 bagian sinkronisasi jam.
4. Pasang kind atau k3d, dan kubectl; pastikan kind `create cluster` berhasil di laptop Anda.

Pertemuan depan

Dari banyak *thread* di satu mesin ke banyak salinan data di banyak mesin: waktu dan urutan, replikasi, konsistensi, Raft, dan Kubernetes yang memulihkan diri.

Terima kasih

Pertanyaan?