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*, *async/await*, dan model *runtime*: Node.js, Go, Rust
8. Pengaruh bahasa: GIL Python, BEAM, JVM, Go
9. Satu soal, tiga model
10. Praktikum dan tugas

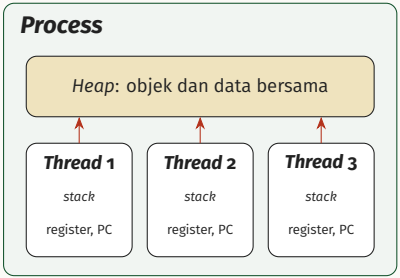
Sasaran pertemuan ini

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*

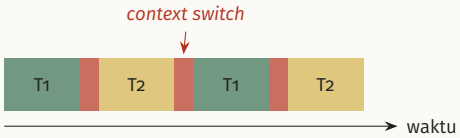
Process dan *thread*

- ***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. *Scheduler* sistem operasi bergantian memberi giliran.
- Setiap pergantian menyimpan register dan status *thread* lama, memuat yang baru, dan sering membuat isi *cache* menjadi tidak berguna.
- *Cost*-nya 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.



Ringkasan

Model *concurrency* modern (*goroutine*, *virtual thread*, proses BEAM, *event loop*) pada dasarnya berusaha mengurangi dua *cost* ini: memori per tugas dan *context-switch cost*.

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/O-bound (menunggu jaringan, disk)	*Concurrency*: banyak tugas menunggu sekaligus	API *gateway*, *web server*, *crawler*
CPU-bound (menghitung)	*Parallelism*: banyak *core* bekerja sekaligus	*Video compression*, 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:

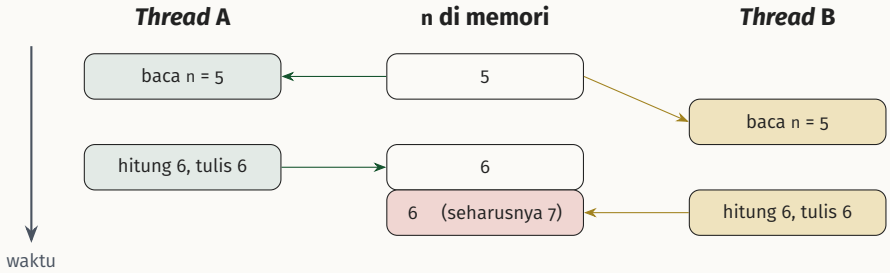
1. baca `n` dari memori
2. tambah satu
3. tulis kembali

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

Ringkasan

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

Bagaimana satu penambahan hilang



Ringkasan

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. *Lock* 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.

Ringkasan

Ketiganya benar. Yang berbeda adalah *cost*-nya: semakin banyak *thread* berebut *lock* 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*: *lock* tidak bisa direbut paksa
4. *Circular wait*: A menunggu B, B menunggu A

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

Menghindari *deadlock*

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

Ringkasan

Di sistem terdistribusi, *lock* melintasi jaringan (*distributed lock*) jauh lebih sulit lagi: pemegang *lock* bisa mati, atau jaringannya putus, sambil tetap "memegang" *lock*. Karena itu desain yang baik meminimalkan kebutuhan akan *lock* 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: berbagi data lewat komunikasi

- **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 (*scheduling* 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`

```go
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 *cancellation* ikut menjalar ke *call* berikutnya.

Ringkasan

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

Worker pool: membatasi *concurrency*

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

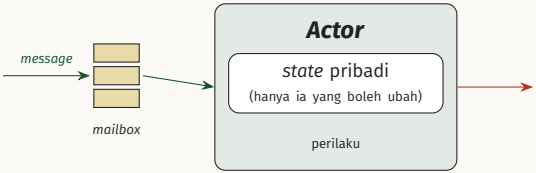
func main() {
    jobs, out := make(chan int, 100), make(chan int, 100)
    var wg sync.WaitGroup
    for w := 0; w < 4; w++ {   // 4 pekerja
        wg.Add(1); go worker(jobs, out, &wg)
    }
    for j := 1; j <= 20; j++ { jobs <- j }
    close(jobs)
    go func() { wg.Wait(); close(out) }()
    total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

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

Actor model: Erlang dan Akka

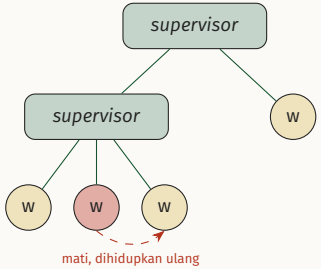
Actor model

- Dirumuskan Carl Hewitt dan rekan (1973). Semua hal adalah ***actor***.
- Tiap *actor* punya *state* pribadi dan sebuah ***mailbox***. Tidak ada yang boleh menyentuh *state*-nya kecuali dirinya sendiri.
- Saat menerima satu *message*, *actor* hanya boleh:
  1. mengirim *message* ke *actor* lain,
  2. membuat *actor* baru,
  3. menentukan perilakunya untuk *message* berikutnya.
- *Message* 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. *Scheduler*-nya *preemptive*: tiap proses diberi jatah langkah, sehingga satu proses rakus tidak membekukan yang lain.
- ***Let it crash*** — jangan menulis penanganan *error* untuk setiap kemungkinan. Biarkan proses yang bermasalah mati, lalu ***supervisor*** menghidupkannya kembali dengan *state* bersih.
- Elixir berjalan di BEAM yang sama dengan sintaks yang lebih modern.



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

Akka (Scala): *actor* bertipe dengan *supervision*

```scala
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
```

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

Dua wujud *actor*: satu mesin dan banyak mesin

Satu mesin (*multi-thread*)

- Ribuan sampai jutaan *actor* dijalankan di atas **segelintir *thread*** (M:N): *dispatcher* Akka/Pekko adalah *thread pool*; BEAM menjalankan satu *scheduler* per *core*.
- Satu *actor* memproses satu *message* pada satu waktu, jadi tidak perlu *lock*. Paralelisme datang dari **banyak *actor*** yang aktif di banyak *core* sekaligus.
- *Message* berpindah lewat memori: cepat (orde mikrodetik) dan praktis tidak hilang.

Banyak mesin (terdistribusi)

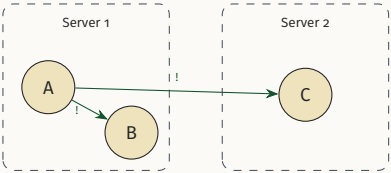
- *Actor* tersebar di beberapa *node*; pengirim tetap hanya memegang alamat (`ActorRef`, PID).
- Contoh: Akka/Pekko Cluster dan Cluster Sharding (misalnya satu *actor* per rekening, disebar ke banyak *node*), Erlang *distributed nodes*, Microsoft Orleans (*virtual actor*), Ray *actor* di Python.
- *Message* melewati jaringan: orde milidetik, bisa hilang, *node* bisa mati atau terpisah oleh *network partition*.

	Satu mesin	**Terdistribusi**
Media *message*	*shared memory* proses	jaringan (TCP) dan *serialization*
Kegagalan	*exception* di satu *actor*	*node* mati, *network partition*
Jaminan *message*	praktis andal	*at-most-once* bawaan; tambah *ack* dan *retry*

Sumber: Akka berganti lisensi ke BSL pada 2022; Apache Pekko adalah *fork* berlisensi Apache 2.0 dengan API yang sama.

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 *message* 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).

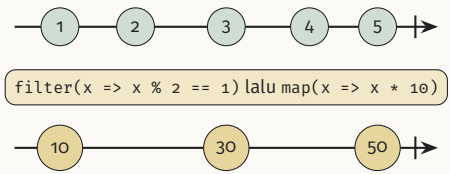
Ringkasan

Actor model disukai di sistem terdistribusi karena model mentalnya sudah memuat "*message* bisa terlambat atau hilang" sejak awal, sehingga tidak ada ilusi panggilan lokal.

Reactive programming

ReactiveX: data sebagai *stream*

- **ReactiveX** (Rx) berawal di Microsoft (Rx.NET), lalu menyebar ke RxJava, RxJS, dan bahasa lain.
- Gagasannya: perlakukan *event* (klik, *message*, hasil HTTP) sebagai ***Observable***, *stream* nilai sepanjang waktu.
- *Stream* 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 *stream* 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 — *stream* besar dengan *backpressure*

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

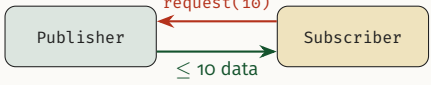
`Flowable` (bukan `Observable`) dipakai bila produsen bisa lebih cepat daripada konsumen.

Ringkasan

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 *message* per detik, konsumen hanya sanggup 1.000. Tanpa kendali, *queue* 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.



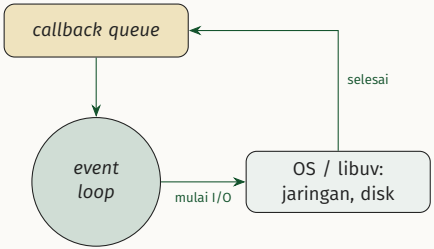
Ringkasan

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 *queue*, 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`

```python
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())
```

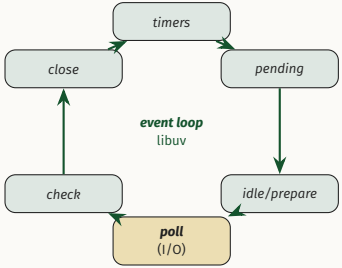
JavaScript (Node.js)

```javascript
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 database* sinkron) di dalam fungsi `async` membekukan seluruh *event loop*.

Di balik layar: model *runtime*

Node.js lebih dalam: satu *thread* JavaScript, banyak *thread* di belakangnya



- ***Network I/O*** (socket, HTTP) tidak memakai *thread* tambahan: libuv mendaftarkannya ke *kernel* lewat **epoll** (Linux), **kqueue** (macOS/BSD), atau **IOCP** (Windows), lalu dibangunkan saat data siap.
- Pekerjaan yang tidak punya API *nonblocking* di OS dikirim ke ***thread pool* libuv** (*default* 4 *thread*, `UV_THREADPOOL_SIZE`): *file* I/O, `dns.lookup`, kripto (`pbkdf2`, `scrypt`), dan *compression* `zlib`.
- Kerja CPU berat milik Anda sendiri dipindah ke **`worker_threads`**: *thread* terpisah dengan *event loop* dan memori JS sendiri, bicara lewat *message*.

`setTimeout` jalan di fase *timers*, *callback* I/O di fase *poll*, `setImmediate` di fase *check*.

Sumber: Dokumentasi Node.js, “The Node.js Event Loop” dan “Don’t Block the Event Loop”; dokumentasi libuv.

Mengapa Node.js unggul untuk I/O, tetapi rapuh untuk CPU

```
// I/O-bound: 10.000 koneksi menunggu,
// tetap satu thread JS -- murah.
app.get('/saldo', async (req, res) => {
  const s = await db.query('SELECT ...');
  res.json(s);
});

// CPU-bound: memblokir SEMUA request.
app.get('/laporan', (req, res) => {
  let x = 0;
  for (let i = 0; i < 5e9; i++) x += i; // beku
  res.json({ x });
});

// Perbaikan: pindahkan ke worker thread.
const { Worker } = require('node:worker_threads');
new Worker('./hitung-laporan.js');
```

Polanya

- Selama kode Anda cepat dan sering "melepas giliran" (`await`), satu *thread* sanggup melayani ribuan koneksi.
- Satu *loop* panjang, `JSON.parse` raksasa, atau *regex* yang *backtracking* parah membuat *latency* semua pengguna melonjak.
- Gejalanya di produksi: *event loop lag* naik, P99 naik, padahal CPU hanya satu *core* yang sibuk.

Ringkasan

Satu *event loop* = *concurrency* tinggi tanpa *parallelism*. Butuh banyak *core*? Jalankan beberapa proses (*cluster*) atau `worker_threads`.

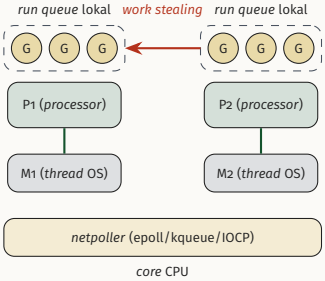
Runtime lain yang memakai *event loop*

Runtime	**Cara kerjanya**	**Catatan**
JavaScript di peramban	Satu *main thread* dengan *task queue* dan *microtask queue* (`Promise`)	Kode berat membekukan UI; pakai Web Worker
Python `asyncio`	*Event loop* di satu *thread*, *coroutine* dengan `async`/`await`	*Library* sinkron (misalnya `requests`) memblokir *loop*; perlu *library async*
Netty, Vert.x (JVM)	Beberapa *event loop*, masing-masing terikat satu *thread*; Vert.x menyebutnya *multi-reactor*	Aturan emasnya sama: *don't block the event loop*
Nginx	Satu proses *master* + beberapa *worker*, tiap *worker* satu *event loop* (epoll)	Itu sebabnya Nginx kuat melayani banyak koneksi dengan memori kecil
Redis	Eksekusi perintah di satu *thread*; *I/O threads* opsional sejak Redis 6	Satu perintah lambat (`KEYS *`) menahan semua klien
Dart / Flutter	*Event loop* per *isolate*; *isolate* tidak berbagi memori	Kerja berat dipindah ke *isolate* lain

Ringkasan

Polanya sama di mana-mana: satu *loop* per *core* atau per proses, I/O tidak pernah ditunggu sambil diam, dan kerja CPU berat harus dipindah ke tempat lain.

Go: *goroutine* dan *scheduler* M:N (model G-M-P)



- **G** = *goroutine* (*stack* awal kecil, beberapa KB, bisa tumbuh); **M** = *thread* OS; **P** = "izin" menjalankan kode Go, jumlahnya `GOMAXPROCS` (*default*: jumlah *core*).
- Tiap P punya *run queue* lokal. P yang menganggur **mencuri** G dari P lain (*work stealing*).
- Sejak Go 1.14 ada ***asynchronous preemption***: *goroutine* yang *loop* lama tetap bisa disela.
- Saat *goroutine* menunggu jaringan, ia "diparkir" dan *netpoller* membangunkannya ketika data siap. M-nya bebas menjalankan G lain.
- *System call* yang memblokir: M dilepas dari P, P diberikan ke M lain.

Sumber: Dokumentasi runtime Go dan *design doc* "Scalable Go Scheduler".

Go terasa sinkron, padahal di bawahnya *nonblocking*

```
func ambil(url string, hasil chan<- int) {
    resp, err := http.Get(url) // terlihat memblokir
    if err != nil { hasil <- -1; return }
    defer resp.Body.Close()
    hasil <- resp.StatusCode
}

func main() {
    hasil := make(chan int)
    for _, u := range urls {   // 10.000 URL
        go ambil(u, hasil)     // 10.000 goroutine
    }
    for range urls { fmt.Println(<-hasil) }
}
```

- Kodenya lurus seperti program sinkron biasa, tanpa *callback* dan tanpa `await`.
- `http.Get` hanya memblokir *goroutine*-nya. *Runtime* memarkirnya, lalu menjalankan *goroutine* lain di *thread* yang sama.
- Hasilnya: kenyamanan gaya sinkron dengan efisiensi *event loop*, sekaligus paralel di semua *core*.
- Harganya: *runtime* dan *garbage collector* ikut di setiap program (dibahas di pertemuan 10).

Rust: *async* tanpa *runtime* bawaan

```rust
// Cargo.toml: tokio = { features = ["full"] }
#[tokio::main]                // multi-thread runtime
async fn main() {
    let mut tugas = Vec::new();
    for u in urls() {
        // spawn: future dijadwalkan di thread pool
        tugas.push(tokio::spawn(async move {
            reqwest::get(u).await.map(|r| r.status())
        }));
    }
    for t in tugas { println!("{:?}", t.await); }
}
```

- *Compiler* mengubah `async fn` menjadi ***future***: sebuah *state machine*. Tidak ada alokasi *stack* per tugas seperti *goroutine*; ini yang disebut *zero-cost*.
- *Future* bersifat *lazy*: tidak berjalan sampai di-`await` atau diserahkan ke ***executor***.
- Bahasa tidak membawa *executor*. Anda memilih: **Tokio** (*multi-thread*, *work stealing*, paling umum), async-std, atau smol.
- *Trait* `Send` dan `Sync` membuat *compiler* menolak data yang tidak aman dibagi antar-*thread*, sebelum program berjalan.

Ringkasan model *runtime*: cocok untuk apa, jebakannya apa

Model	Contoh	Cocok untuk	Jebakan
Event loop satu *thread*	Node.js, `asyncio`, Redis, Nginx *worker*	Banyak koneksi, I/O-*bound*	Satu kerja CPU memblokir semua
Platform thread + *lock*	Java, C++, C#	Kerja CPU paralel, kode lama	*Race*, *deadlock*; *thread* mahal
Virtual thread	Java 21	Gaya "satu *request* satu *thread*" dalam skala besar	Blok `synchronized` lama bisa "menyematkan" *thread*
Goroutine M:N	Go	Layanan jaringan, I/O + CPU sekaligus	*Goroutine leak*, jeda GC
Future + *executor*	Rust (Tokio)	*Latency* rendah dan stabil, hemat memori	Kurva belajar; memanggil fungsi *blocking* di dalam *async*
Actor	Erlang/Elixir, Akka/Pekko	Banyak entitas *stateful*, sistem tahan gagal	*Mailbox* menumpuk

Ringkasan

Pertanyaan pertama selalu sama: pekerjaan Anda **menunggu** (I/O-*bound*) atau **menghitung** (CPU-*bound*)? Jenis pekerjaan itu menentukan model mana yang murah.

Pengaruh bahasa pada *concurrency*

Bahasa menentukan model yang murah dipakai

Bahasa	**Unit *concurrency* bawaan**	**Akibat praktis**
Python (CPython)	*Thread* OS, tetapi dibatasi **GIL**: hanya satu *thread* menjalankan *bytecode* Python pada satu saat	*Thread* berguna untuk I/O, tidak untuk CPU. Untuk CPU: `multiprocessing` atau pustaka *native* (NumPy). Versi *free-threaded* tanpa GIL mulai diuji sejak Python 3.13
Java	*Thread* OS; *virtual thread* sejak Java 21	*Thread* benar-benar paralel. *Virtual thread* membuat gaya "satu tugas satu *thread*" murah lagi
Go	*Goroutine* dengan *scheduler* M:N	Paralel di semua *core* secara bawaan; ratusan ribu *goroutine* wajar
Erlang/Elixir	Proses BEAM, *scheduler preemptive*	Jutaan proses, isolasi kegagalan per proses; lemah untuk hitungan angka berat
JavaScript	*Event loop* satu *thread*; *worker thread* terpisah	Unggul untuk I/O; kerja CPU harus dipindah ke *worker*
Rust	*Thread* OS dan *async*; *compiler* menolak *data race*	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 *thread* + *lock*	Variabel bersama, dijaga *lock*	Tiap *thread* mengunci, menambah, melepas
Go *goroutine* + *channel*	Satu *goroutine* pemilik	*Goroutine* lain mengirim “tambah 1” lewat *channel*
Akka *actor*	Satu *actor* `Penghitung`	*Message* `Tambah` masuk *mailbox*, diproses satu per satu

Ringkasan

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

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
Thread + lock	Data bersama kecil, operasi sederhana	*Race*, *deadlock*, *lock* jadi rebutan	`jstack`, *thread sanitizer*
Goroutine + channel	Layanan jaringan, *pipeline*, *worker pool*	*Goroutine leak*, *channel* macet	`go run -race`, `pprof`
Actor	Banyak entitas *stateful* (akun, sesi, perangkat), sistem terdistribusi	*Mailbox* menumpuk, *message* hilang antar mesin	*Metric* ukuran *mailbox*, *tracing*
Reactive streams	*Event stream*, UI, *streaming* data	Rantai operator sulit di-*debug*	*Marble test*, log per operator
Event loop / *async*	Banyak koneksi I/O, kode sederhana	Satu tugas berat membekukan semua	*Event loop lag metric*

Ringkasan

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** *synchronization* 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.

Ringkasan

Jangan hanya melaporkan waktu. Jelaskan **mengapa** model tertentu lebih lambat: apakah karena rebutan *lock*, *channel* yang penuh, atau *cost message* 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 *synchronization* 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, *replication*, *consistency*, Raft, dan Kubernetes yang memulihkan diri.

Terima kasih

Pertanyaan?