Thread və Process: Sıfırdan Senior səviyyəyə — Deep Dive Öyrənmə Planı

> **Snapshot:** bu fayl Claude Docs-dakı canlı planın 2026-11-20 tarixli nüsxəsidir. Canlı versiya: https://claude.ai/artifact/3RfryE9eFVgPsAFo18fY3s Giriş testi və özünü yoxla suallarının cavabları: answer-key.md.

Nov 18, 2026 · @Atesh

Planın istifadə qaydası

Plan 12 fazadan ibarətdir: həftədə 8–10 saatla təxminən 26 həftədə (6 ay) sıfırdan senior səviyyəyə çatdırır. Hər faza əvvəlkinin üzərində qurulur, ona görə sıra ilə get. Yol aşağıdan yuxarı gedir: hardware → Linux kernel → runtime (JVM/ART) → Android framework → Kotlin coroutines.

> Diaqram: **Böyük şəkil · 5 qat və onları açan fazalar** — şəkil canlı sənəddədir: https://claude.ai/artifact/3RfryE9eFVgPsAFo18fY3s

Hər faza bir qatı açır; Faza 5 və 11 isə bütün qatlarda işləyən diaqnostika və dizayn bacarıqlarıdır. Yuxarıdakı qatda bir bug görəndə səbəbi çox vaxt bir qat aşağıda axtar.

> Diaqram: **Under-the-hood zənciri · lock()-dan cache coherence-ə 10 addım** — şəkil canlı sənəddədir: https://claude.ai/artifact/3RfryE9eFVgPsAFo18fY3s

Planın mərkəzi zənciri budur: bir `lock()` çağırışı contention olanda beş qatdan keçir, olmayanda isə sağdakı fast path ilə bir CAS-la bitir. Fərq təxminən nanosaniyələrlə mikrosaniyələr arasındadır. Planın sonunda bu diagramın hər addımını mənbə kodu ilə göstərə biləcəksən (Faza 4, 6, 10).

Hər fazanın quruluşu

- **Problem** — faza həll etdiyi konkret sualla açılır.
- **Niyə vacibdir** — bu bilik olmasa nəyin sındığı, sənin real işinlə bağlı.
- **Mövzular (under the hood)** — sadədən mürəkkəbə, dərinlik teqləri ilə; hər bənd həm "nədir", həm "içəridə necə işləyir" sualına cavab verir.
- **Mental model** — diagram və ya addım-addım cədvəl: hər addımda vəziyyət göstərilir.
- **12 sual** — fazanın mərkəzi anlayışı 12 sual modeli ilə açılır.
- **Under-the-hood zənciri** — API çağırışından hardware-ə qədər bir sətir.
- **Digər ekosistemlərdə** — Go, C++, Node.js, Java Loom eyni işi necə görür.
- **Gözünlə gör** — anlayışı canlı müşahidə etmək üçün dəqiq komandalar.
- **Laboratoriya** — hipotez → rəqəmlərlə qurulum → variantlar → ölçülər → gözlənilən nəticə.
- **Öz layihəndə** — fazanı real app-ına tətbiq edən bir tapşırıq.

- **Praktika** — qutucuqlu tapşırıqlar; tamamladıqca işarələ.
- **Özünü yoxla** — "niyə/necə" sualları və (İ) işarəli intervyu sualları; cavab açarı ikinci tab-dadır.
- **Mənbələr** — yalnız həmin fazaya aid fəsillər.
- **Mənimsəmə meyarı** — müşahidə olunan bacarıq və fazanın suallarına 4-cü səviyyədə cavab (aşağıdakı meyara bax). Bu ödənmədən növbəti fazaya keçmə.

Hər mövzu üçün öyrənmə dövrü

1. **Problem** — mövzunun həll etdiyi sualı öz sözlərinlə yaz.
2. **Konsept** — bir mənbədən oxu, ikinci mənbədən təsdiqlə.
3. **Sadə nümunə** — ən kiçik misalı tap və ya qur.
4. **Mental model çək** — yaddaşı, stack-i, state keçidlərini kağızda çək.
5. **Kod yaz** — minimal nümunə (C və ya Kotlin).
6. **Sındır** — bilərəkdən bug yarat və onu müşahidə et: race, deadlock, ANR.
7. **Düzəlt** — bug-ı aradan qaldır və niyə işlədiyini izah et.
8. **Ölç** — benchmark: vaxt, CPU, context switch, cache miss.
9. **İçəri bax** — OS (`strace`, `/proc`), bytecode və assembly, runtime və kernel mənbə kodu.
10. **Yaddaşdan izah et** — 5 dəqiqəlik izahı səsli danış və ya yaz (Feynman texnikası).
11. **Aralıqlı təkrar** — 1, 3, 7 və 21 gün sonra (Həftəlik cədvəl bölməsinə bax).

Elmi əsaslı öyrənmə üsulları

Öyrənmə tədqiqatlarında ən yüksək faydalı sayılan iki üsul özünü yoxlama və aralıqlı təkrardır (Dunlosky və b., 2013). Plan onları belə tətbiq edir:

- **Retrieval practice** — yenidən oxumazdan əvvəl yaddaşdan cavab ver; hər sessiya 3 sualla başlayır.
- **Spaced repetition** — hər faza bitdən 1, 3, 7 və 21 gün sonra təkrar; lüğət sətirləri flashcard-dır.
- **Interleaving** — həftəlik təkrarda köhnə fazalardan sualları qarışdır.
- **Elaboration** — "niyə?" və "necə?" sualları və yaddaşdan izah addımı.
- **Dual coding** — hər mərkəzi anlayışın həm diagramı, həm sözlə izahı var.
- **Worked example → faded example → müstəqil tapşırıq** — laboratoriyalar bu ardıcıllıqla qurulub.
- Yenidən oxumaq və rəngləmək əsas üsul olmamalıdır — tədqiqatlarda faydası aşağı çıxıb.

12 sual modeli

Hər mərkəzi anlayışa bu suallarla yanaş. Tətbiq olunmayan sualı "—" ilə işarələ, cavab uydurma.

1. Nədir?
2. Niyə mövcuddur?
3. Hansı problemi həll edir?
4. Ondan əvvəl bu problem necə həll olunurdu?
5. API necə istifadə olunur?
6. Runtime daxilində nə baş verir?
7. Kernel daxilində nə baş verir?
8. CPU səviyyəsində nə baş verir?
9. Yaddaş səviyyəsində nə baş verir?

10. Qiyməti nədir (latency, yaddaş, contention)?
11. Edge case-lər hansılardır?
12. Alternativlər hansılardır və nə vaxt hansını seçmək lazımdır?

5 səviyyəli cavab meyarı

Hər cavabını bu cədvəllə qiymətləndir. 3-cü səviyyə giriş testindən keçir, 4-cü səviyyə fazadan keçir, 5-ci səviyyə senior-dur.

Səviyyə	Nə deməkdir	Nümunə: mutex
1. Sadə izah	İki cümlə, analogiya ilə	Qapı kilidi: içəridə biri varsa, digəri gözləyir
2. Texniki tərif	Dəqiq terminlər	Eyni anda yalnız bir sahibi olan mutual exclusion primitivi
3. Under the hood	Mexanizm və data strukturları	Contention yoxdursa user-space CAS; varsa `futex_wait` ilə kernel wait queue, unlock-da `futex_wake`
4. Real kod	Minimal nümunəni yazıb işlədə bilirsən	2 thread × 1 000 000 artım mutex ilə düzgün, mutex-siz səhv nəticə verir
5. Production	Real bug, insident və ya trade-off	Lock tutarkən listener çağırmaq deadlock yaratdı; lock-u callback-dən əvvəl buraxdıq

Dərinlik teqləri

- **[Əsas]** — minimum yol: vaxtın azdırsa, yalnız bunları et.
- **[Dərin]** — senior səviyyəsi üçün lazımdır.
- **[Ekspert]** — runtime və kernel mənbə kodu səviyyəsi; ikinci keçiddə də etmək olar.

Hər fazanın "Mövzular" başlığı altında hansı bəndin hansı teqə aid olduğu yazılıb.

İş mühiti

Alət	Nə üçün	Fazalar
Linux (Ubuntu VM, WSL2 və ya bulud server)	C kodu, syscall-lar, /proc fayl sistemi	0–6, 10
gcc/clang, gdb, objdump, readelf	Kompilyasiya, assembly, debug	0–4, 10
strace, perf, htop, top -H	Syscall-ları və scheduling-i izləmək	0–5, 10
JDK 21+, IntelliJ IDEA	JVM tərəfli eksperimentlər	3–7, 9
JMH, jcstress	Benchmark və memory model testləri	4, 6, 7
Android Studio, adb, real ARM64 cihaz	Android eksperimentləri	1–3, 5, 8–11
Perfetto (ui.perfetto.dev), simpleperf	Trace və CPU profili	2, 5, 8, 10, 11
cs.android.com, elixir.bootlin.com	Android və Linux mənbə kodu	3, 4, 8, 10

Android eksperimentlərini mümkün qədər real ARM64 cihazda et. x86 emulator memory ordering və scheduling-i fərqli göstərir.

Qeyd dəftəri: hər faza üçün "anladıqlarım" və "hələ qaranlıq qalanlar" siyahısı saxla. Qaranlıq qalanları növbəti həftənin əvvəlində təkrarla.

Ön şərtlər (Faza 0-dan əvvəl)

Praktikanın yarısı C və Linux terminalında keçir, ona görə aşağıdakı minimumu bilmədən başlama. Bilmirsənsə, Faza 0-dan əvvəl 2–3 həftə əlavə et.

Bilik	Minimum səviyyə	Qısa kurs
C dili	Pointer, struct, `malloc`/`free`, massiv, funksiya pointer-i, `gcc` ilə kompilyasiya	Beej's Guide to C Programming (pulsuz)
Linux terminalı	`ls`, `cd`, `ps`, `top`, `grep`, pipe operatoru, `kill`, arxa plan prosesləri, fayl icazələri	The Linux Command Line (William Shotts, pulsuz)
Kotlin və Java	Class, interface, lambda, generics, exception-lar, collection-lar	Kotlin docs
Data strukturları	Linked list, queue, stack, hash table, heap, binary/red-black tree (ideya səviyyəsində)	istənilən alqoritm kitabı və ya kursu
Say sistemləri	Binary, hex, bit əməliyyatları (AND, OR, XOR, shift) — `ctl` bit packing (Faza 7) və lock word (Faza 4) üçün	CS:APP fəsil 2
Git	`clone`, `commit`, `branch` — praktika kodunu saxlamaq üçün	öz təcrübən

Bütün praktika kodunu bir git repo-da saxla: `phase-00/`, `phase-01/` ... Hər faza üçün README-də "nə qurdum, nə sındırdım, nə ölçdüm" yaz — bu həm də portfolio olur.

Giriş testi və sürətli yol

Başlamazdan əvvəl bu 24 suala 60 dəqiqə ərzində, kitabsız və internetsız cavab ver. Sonra cavablarını "Cavab açarı" tab-ı ilə yoxla və 5 səviyyəli meyara görə qiymətləndir.

Bütün sualların açar fikirləri ayrı tab-dadır: Cavab açarı

Sürətli yol qaydası: fazanın hər iki sualına ən azı 3-cü səviyyədə cavab vermisənsə, həmin fazanı tam keçmə. Yalnız onun laboratoriyasını və mənimsəmə meyarını et, İrəliləyiş cədvəlində statusu "Sürətli yol" qoy. Təcrübəli Android developer üçün adətən Faza 8-in bir hissəsi belə keçir, Faza 0, 6 və 10 isə tam keçilir.

№	Faza	Sual
P1	0	Syscall ilə adi funksiya çağırışı arasında fərq nədir və syscall niyə bahalıdır?
P2	0	Cache line nədir və 2D massivi sütun-sütun gəzmək niyə yavaş ola bilər?
P3	1	1 GB yaddaşı olan process-i `fork()` etmək niyə ucuzdur?
P4	1	Zygote nədir və app process-i necə yaranır?
P5	2	Context switch zamanı nə saxlanılır və harada?
P6	2	Perfetto-da main thread uzun müddət "Runnable" qalırsa, bu nə deməkdir?
P7	3	Eyni process-in thread-ləri nəyi paylaşır, nəyi paylaşmır?
P8	3	Linux-da `pthread_create()` hansı syscall-a çevrilir və onu yeni process yaradılmasından nə fərqləndirir?

№	Faza	Sual
P9	4	`counter++` niyə atomik deyil? Instruksiya səviyyəsində izah et.
P10	4	Futex niyə "fast" adlanır?
P11	5	Deadlock-un 4 Coffman şərti hansılardır və praktikada hansını sındırırsan?
P12	5	Priority inversion nədir?
P13	6	`volatile` nəyi təmin edir, nəyi etmir?
P14	6	x86-da düzgün işləyən lock-free kod ARM-da niyə sına bilər?
P15	7	`ThreadPoolExecutor`-a sınırsız `LinkedBlockingQueue` verəndə `maximumPoolSize` niyə işə düşmür?
P16	7	`LongAdder` `AtomicLong`-dan nə vaxt və niyə sürətlidir?
P17	8	`Looper.loop()` sonsuz dövrdürsə, main thread niyə CPU yandırmır?
P18	8	AIDL callback hansı thread-də gəlir və bu niyə vacibdir?
P19	9	suspend funksiya bytecode-da nəyə çevrilir?
P20	9	`Dispatchers.IO` və `Dispatchers.Default` fərqi nədir, thread-ləri necə paylaşırlar?
P21	10	`JNIEnv`-i niyə thread-lər arasında paylaşmaq olmaz?
P22	10	Uninterruptible sleep (D state) nədir?
P23	11	Sistemə saniyədə 500 sorğu gəlir, orta latency 200 ms-dir. Sistemdə orta hesabla neçə sorğu olur?
P24	11	Bir jank frame-in səbəbini Perfetto-da necə araşdırarsan?

İrəliləyiş cədvəli

Hər fazaya başlayanda və bitirəndə statusu dəyiş, tarixləri və qeydləri özün yaz. Giriş testindən sonra sürətli keçəcəyin fazaları dərhal "Sürətli yol" et.

Faza	Səviyyə	Status	Başlama	Bitmə	Qeyd
0 · Kompüterin təməli	Beginner	Başlanmayıb			
1 · Process	Beginner	Başlanmayıb			
2 · Context switch və scheduling	Intermediate	Başlanmayıb			
3 · Thread və I/O modelləri	Intermediate	Başlanmayıb			
4 · Sinxronizasiya	Intermediate	Başlanmayıb			
5 · Deadlock və liveness	Intermediate	Başlanmayıb			
6 · Memory model	Advanced	Başlanmayıb			
7 · java.util.concurrent	Advanced	Başlanmayıb			
8 · Android threading	Advanced	Başlanmayıb			
9 · Kotlin Coroutines	Advanced	Başlanmayıb			
10 · Linux kernel və native	Expert	Başlanmayıb			
11 · Senior səviyyə	Senior	Başlanmayıb			

Faza 0 — Kompüterin təməli (1–2 həftə) · Beginner

Öz-özünə öyrənmə sənədi: Faza 0 — Kompüterin təməli: Öz-özünə öyrənmə sənədi — bu fazanı sıfırdan, fəsil-fəsil, təcrübələr və intervyu cavabları ilə öyrədir.

Thread və process əməliyyat sisteminin abstraksiyalarıdır, ona görə əvvəlcə CPU, yaddaş və kernel-in necə işlədiyini bilməlisən. Bu fazanı keçmədən "under the hood" izahlar havada qalır.

Problem: `printf("salam")` yazırsan və mətn ekrana çıxır. Arada CPU, yaddaş və kernel nə edir? Bu sualın tam cavabı Faza 0-dır.

Niyə vacibdir: context switch, cache, memory barrier, syscall kimi sözlər sonrakı fazalarda daim işlədiləcək. Onların fiziki mənasını burada qurursan.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 2, 4, 5, 6 · [Dərin] 3, 7 · [Ekspert] bu fazada yoxdur (pipeline və out-of-order Faza 6-da açılır).

1. **CPU necə işləyir**
   - Register-lər: general purpose, PC (program counter), SP (stack pointer), flags.
   - Instruction cycle: fetch → decode → execute; clock tezliyi.
   - Pipeline və out-of-order execution (Faza 6 üçün toxum).
   - Core, SMT/Hyper-Threading; x86-64 və ARM64 (AArch64) fərqi. Android cihazları ARM64-dür.
2. **Yaddaş iyerarxiyası**
   - Register → L1 (~1 ns) → L2 → L3 → RAM (~100 ns) → flash/SSD.
   - Cache line (adi halda 64 byte), temporal və spatial locality.
   - "Latency numbers every programmer should know" cədvəlini əzbərlə.
3. **Proqram necə yaranır**
   - Source → compiler → assembly → object fayl → linker → executable (ELF).
   - Section-lar: .text, .rodata, .data, .bss; loader proqramı yaddaşa necə yükləyir.
   - Android-da .so faylları da ELF-dir; DEX/OAT isə ART üçündür.
4. **Funksiya çağırışı under the hood**
   - Call stack, stack frame, return address, lokal dəyişənlər.
   - Calling convention: ARM64-də arqumentlər x0–x7-də, return address x30 (LR) register-ində.
   - Stack overflow niyə və necə baş verir.
5. **Əməliyyat sisteminin rolu**
   - Kernel space və user space; privilege səviyyələri (x86 ring 0/3, ARM EL0/EL1).
   - User kodu niyə hardware-ə birbaşa toxuna bilmir.
6. **System call**
   - User → kernel keçidi: ARM64-də `svc #0`, x86-64-də `syscall` instruksiyası.
   - Syscall nömrəsi, libc wrapper (Android-da libc = bionic).
   - Mode switch-in xərci; syscall ilə adi funksiya çağırışının fərqi.
7. **Interrupt və exception**
   - Hardware interrupt (timer, toxunuş, şəbəkə), trap/exception (page fault, sıfıra bölmə).
   - Interrupt handler və interrupt vector table.

- Timer interrupt — preemption-un (Faza 2) əsasıdır.

Mental model: `write()` çağırışının yolu

Addım	Harada	Nə baş verir
1	User space	`printf` mətni libc buffer-inə yazır
2	User space	Terminalda `\n` gələndə libc `write(1, buf, n)` çağırır
3	CPU	`svc #0` (ARM64) və ya `syscall` (x86-64): EL0 → EL1, kernel stack-ə keçid
4	Kernel	Syscall nömrəsinə görə `write` handler-i seçilir, fd 1 yoxlanılır
5	Kernel	Terminal driver-i mətni qəbul edir
6	CPU	User space-ə qayıdış; qaytarılan dəyər yazılan byte sayıdır

12 sual: system call

№	Sual	Qısa cavab
1	Nədir?	User proqramın kernel-dən xidmət istəməsi üçün idarə olunan giriş nöqtəsi
2	Niyə mövcuddur?	User kodu hardware-ə birbaşa toxunmamalıdır: izolyasiya və təhlükəsizlik
3	Hansı problemi həll edir?	Paylaşılan resurslara (disk, şəbəkə, yaddaş) nəzarətli giriş
4	Əvvəl necə idi?	Erkən sistemlərdə (məsələn, MS-DOS) proqramlar hardware-ə birbaşa müraciət edirdi; bir proqram bütün sistemi çökdürə bilirdi
5	API	libc wrapper-ləri: `read`, `write`, `mmap`, `clone`; Android-da libc = bionic
6	Runtime	libc arqumentləri register-lərə qoyur, syscall nömrəsini yazır, xətada `errno`-nu doldurur
7	Kernel	Syscall cədvəlindən handler seçilir, arqumentlər yoxlanılır, iş görülür
8	CPU	`svc`/`syscall` instruksiyası privilege səviyyəsini dəyişir və kernel giriş ünvanına keçir
9	Yaddaş	User buffer-i kernel-ə kopyalanır (`copy_from_user`); kernel ayrıca stack işlədir
10	Qiyməti	Mode switch və cache/TLB təsiri: adi funksiyadan dəfələrlə bahalı; təxminən 100 ns-dən mikrosaniyəyə qədər (CPU və mitiqasiyalardan asılı)
11	Edge case-lər	`EINTR` (signal syscall-ı kəsir), qismən `write`, vDSO (`clock_gettime` kernel-ə girmədən)
12	Alternativlər	vDSO, user-space buffer-ləmə (stdio), batch syscall-lar, `io_uring` (Faza 3)

Under-the-hood zənciri

`printf()` → libc buffer → `write()` wrapper → `svc #0` → kernel syscall cədvəli → `write` handler → terminal driver → ekran

Digər ekosistemlərdə

- Java/Kotlin: `println` → JVM native kodu → libc `write`; eyni syscall.
- Go: Linux-da runtime syscall-ları libc-siz, birbaşa çağırır.
- Node.js: libuv üzərindən eyni syscall-lar.

Gözünlə gör

- `strace -c ./hello` — syscall-ların sayı və vaxtı.
- `ltrace ./hello` — libc çağırışları.
- `perf stat -e cycles,instructions,cache-misses ./prog` — CPU sayğacları.
- `lscpu`, `cat /proc/cpuinfo` — core-lar və cache ölçüləri.

Laboratoriya

Lab 0.1 — Cache təsiri

- **Hipotez:** 2D massivi sütun-sütun gəzmək sətir-sətir gəzməkdən xəyli yavaşdır.
- **Qurulum:** C-də 4096 × 4096 `int` massiv (64 MB); hər element bir dəfə oxunur; 5 təkrarın medianı.
- **Variantlar:** sətir-sətir və sütun-sütun.
- **Ölçülər:** vaxt, `perf stat -e cache-misses`.
- **Gözlənilən nəticə:** sütun variantında cache miss və vaxt dəfələrlə çoxdur, çünki hər addım 16 KB atlayır və yeni cache line-a düşür.

Lab 0.2 — Syscall-ın qiyməti

- **Hipotez:** boş syscall boş funksiyadan dəfələrlə bahalıdır.
- **Qurulum:** 10 000 000 dəfə `noinline` boş funksiya və 10 000 000 dəfə `getppid()` (vDSO-da olmayan syscall).
- **Ölçü:** bir çağırışa düşən orta vaxt (ns).
- **Gözlənilən nəticə:** syscall dəfələrlə bahalıdır; fərqi mode switch ilə izah et.

Öz layihəndə

App-ının cold start-ını syscall tracing aktiv olan Perfetto ilə (ftrace `raw_syscalls`) yaz. Main thread-də ən çox vaxt aparan 3 syscall-ı (məsələn, `openat`, `read`, `futex`) tap.

Praktika

- [ ] C-də hello world yaz; `gcc -S` ilə assembly-yə, `objdump -d` və `readelf -S` ilə section-lara bax.
- [ ] `strace ./hello` işlət; execve, brk, mmap, write, exit_group syscall-larını tap və hər birinin rolunu yaz.
- [ ] Rekursiv funksiya ilə stack overflow yarat; `ulimit -s` dəyişərək fərqi gör.
- [ ] 2D massivi sətir-sətir və sütun-sütun gəz, vaxtı ölç; cache təsirini izah et.
- [ ] gdb-də funksiya çağırışı zamanı SP və PC register-lərinin dəyişməsini addım-addım izlə.

Özünü yoxla

- `printf` niyə ekrana birbaşa yaza bilmir? Arada nə baş verir?
- Syscall niyə adi funksiya çağırışından bahalıdır?
- Timer interrupt olmasaydı, OS sonsuz loop-dakı proqramı necə dayandırardı?
- Massivi sütun-sütun gəzmək niyə dəfələrlə yavaş ola bilər?

- (İ) User space və kernel space-i niyə ayırırıq? Android-da bu sərhəd harada görünür?
- (İ) vDSO nədir və hansı problemi həll edir?

Mənbələr

- *Computer Systems: A Programmer's Perspective* (Bryant, O'Hallaron) — fəsil 1, 3, 6, 8.
- OSTEP — "Introduction" və "Limited Direct Execution" fəsilləri (pulsuz).

Mənimsəmə meyarı: bir C proqramının işə düşdüyü andan `write()` mətni ekrana çatdırana qədər yolu kağızda çəkib izah edə bilirsən.

Faza 1 — Process: nədir, niyə var, necə işləyir (2 həftə) · Beginner

Process işləyən proqramdır: kod + özünəməxsus virtual yaddaş + resurslar (fayllar, UID) + ən azı bir thread. OS hər process-ə "CPU və yaddaş yalnız sənindir" illüziyası yaradır.

Problem: Telefonda 50 app açıqdır və hamısı eyni RAM-dadır. Biri crash olanda digərləri niyə ölmür? Biri digərinin yaddaşını niyə oxuya bilmir?

Niyə var: izolyasiya (bir app crash olanda digəri yaşayır), təhlükəsizlik (bir app digərinin yaddaşını oxuya bilmir) və multiprogramming (bir CPU-da çox proqram). Android-in təhlükəsizlik modeli birbaşa bunun üzərində qurulub.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 2, 5, 6, 8 · [Dərin] 3, 4, 7 · [Ekspert] page table səviyyələri və `mm_struct` daxili (Faza 10-da mənbə kodu ilə).

1. **Program vs process vs thread**
   - Program — diskdə passiv fayl. Process — icra olunan program + resurslar. Thread — process daxilində icra axını.
2. **Address space quruluşu**
   - Aşağıdan yuxarı: text, rodata, data, bss, heap (brk, malloc), mmap bölgəsi (shared library-lər, anonymous mmap), stack (aşağı böyüyür), kernel space.
   - ASLR — ünvanların hər işə düşmədə təsadüfi sürüşdürülməsi.
   - `/proc/<pid>/maps` faylında hər bölgəni gör.
3. **Virtual memory under the hood**
   - Virtual və fiziki ünvan; page (adi halda 4 KB; Android 15-dən 16 KB page size dəstəyi gəlib).
   - Page table (ARM64-də çoxsəviyyəli), MMU, TLB.
   - Page fault (minor və major), demand paging, swap (Android-da zRAM).
   - Copy-on-write (COW) — fork-un ucuz olmasının səbəbi.
4. **PCB — Linux-da `task_struct`**
   - PID, state, saxlanılmış register-lər, scheduling məlumatı.
   - `mm_struct` (yaddaş), `files_struct` (fd cədvəli), credentials (UID/GID), signal handler-lər, parent/children.
5. **Process state-ləri**

- Klassik: New → Ready → Running → Blocked → Terminated.
- Linux: R (running/runnable), S (interruptible sleep), D (uninterruptible sleep), T (stopped), Z (zombie).
- Hər keçidi hansı hadisənin tətiklədiyini yaz (timer interrupt, I/O tamamlandı, exit).

6. **Process həyat dövrü Linux-da**
   - `fork()` — COW ilə kopya yaradır, iki dəfə qayıdır (parent-ə child PID, child-a 0).
   - `exec()` ailəsi — address space-i yeni proqramla əvəz edir.
   - `wait()`/`waitpid()`, `exit()`; zombie (parent wait etmədi), orphan (PID 1 götürür); process ağacı.
7. **File descriptor-lar**
   - fd cədvəli → open file description → inode; stdin/stdout/stderr.
   - fork zamanı miras qalması; `pipe()` ilə iki process arasında əlaqə (IPC-yə giriş).
8. **Android-da process**
   - **Zygote:** init → zygote (app_process) framework class-larını və resursları əvvəlcədən yükləyir → hər app üçün `fork()` edir. COW sayəsində bu səhifələr paylaşılır. `exec()` yoxdur — startup sürəti üçün.
   - Hər app ayrı Linux UID-dir (sandbox) və ayrı SELinux domain-də işləyir.
   - ActivityManagerService process-i başladır (zygote socket ilə), sonra `ActivityThread.main()` çağırılır.
   - `android:process` atributu ilə multi-process app: hər process-də `Application.onCreate` yenidən işləyir, singleton-lar process-lər arasında paylaşılmır.
   - Process importance (foreground, visible, perceptible, service, cached), `oom_score_adj`, lmkd (low memory killer daemon).
   - Process death və state bərpası: ViewModel itir, SavedStateHandle qalır.

> Diaqram: **Process və thread state-ləri · 5 state, 6 keçid** — şəkil canlı sənəddədir: https://claude.ai/artifact/3RfryE9eFVgPsAFo18fY3s

Timer interrupt Running-i Ready-yə qaytarır; I/O və ya lock gözləyən task Blocked-a keçir və hadisə baş verəndə yenidən Ready olur. Perfetto-da Runnable = Ready, Sleeping = S, Uninterruptible Sleep = D deməkdir.

Mental model: `fork()` və copy-on-write

Yuxarıdakı diagram state-ləri göstərir; bu cədvəl isə fork zamanı yaddaşın addım-addım necə dəyişdiyini.

Addım	Parent	Child	Fiziki səhifə
1	`x = 5` səhifə P1-də	—	P1 yalnız parent-in
2	`fork()` çağırır	yaranır	P1 hər ikisinə read-only map olunur, kopya yoxdur
3	—	`x = 7` yazmaq istəyir	Page fault: kernel P1-i yeni P2-yə kopyalayır
4	`x` = 5 görür (P1)	`x` = 7 görür (P2)	İki ayrı səhifə; qalan səhifələr hələ paylaşılır

12 sual: process

№	Sual	Qısa cavab
1	Nədir?	Öz virtual address space-i, resursları və ən azı bir thread-i olan icra olunan proqram
2	Niyə mövcuddur?	İzolyasiya və bir CPU-nun çox proqram arasında bölüşdürülməsi
3	Hansı problemi həll edir?	Proqramların bir-birinin yaddaşını pozması və sistemi çökdürməsi
4	Əvvəl necə idi?	Batch sistemlər: bir dəfədə bir proqram, yaddaş qorunması yox idi
5	API	`fork`, `exec`, `wait`, `exit`, `kill`; Android-da zygote vasitəsilə start, `android:process`
6	Runtime	ActivityManagerService → zygote socket → `fork` → `ActivityThread.main()`
7	Kernel	`task_struct`, `mm_struct`, fd cədvəli, PID ayrılması, scheduler-ə əlavə
8	CPU	MMU page table-ın kökünə baxan register-dən (ARM64 TTBR0) istifadə edir; process dəyişəndə bu dəyişir
9	Yaddaş	Virtual address space, page table-lar, COW səhifələr, zRAM-da swap
10	Qiyməti	Yaradılma (fork + page table), context switch-də TLB təsiri, hər process üçün ayrı ART heap
11	Edge case-lər	Zombie, orphan, lmkd tərəfindən öldürülmə, multi-process app-da ikinci `Application.onCreate`
12	Alternativlər	Thread (eyni address space), isolated process, coroutine (eyni thread-də)

Under-the-hood zənciri

`startActivity()` → ActivityManagerService → zygote socket → `fork()` → COW səhifələr → `ActivityThread.main()` → `Looper.loop()` → `Activity.onCreate()`

Digər ekosistemlərdə

- Chrome: hər sayt üçün ayrı renderer process — izolyasiya üçün.
- Node.js `cluster`, Python `multiprocessing`: bütün core-ları işlətmək üçün process-lər.
- JVM və Go server-ləri: adətən bir process, çox thread və ya goroutine.

Gözünlə gör

- `cat /proc/<pid>/maps`, `cat /proc/<pid>/status` (VmRSS, Threads, PPid).
- `pstree -p`, `ps -o pid,ppid,stat,cmd`.
- Android: `adb shell ps -A -o PID,PPID,NAME`, `adb shell dumpsys meminfo <paket>` (PSS və paylaşılan səhifələr).
- `adb shell cat /proc/<pid>/oom_score_adj` — arxa plana keçəndə dəyişir.

Laboratoriya

Lab 1.1 — Copy-on-write-u ölç

- **Hipotez:** fork-dan dərhal sonra child əlavə fiziki yaddaş tutmur; yaddaş yalnız yazılan səhifələr qədər artır.
- **Qurulum:** 512 MB `malloc` + `memset`, sonra `fork`; child səhifələrin 0%, 25%, 50%, 100%-nə yazır.
- **Ölçülər:** child-ın `/proc/<pid>/smaps_rollup`-dakı Private_Dirty dəyəri, minor page fault sayı (`/usr/bin/time -v`).
- **Gözlənilən nəticə:** Private_Dirty yazılan hissəyə mütənasib artır; 100%-də minor fault sayı təxminən 512 MB / 4 KB = 131 072 olur (THP aktivdirsə, rəqəm fərqli çıxa bilər).

Lab 1.2 — Zombie və orphan

- **Qurulum:** child dərhal çıxır, parent 30 saniyə `wait()` etmir; sonra tərsinə: parent çıxır, child yaşayır.
- **Ölçülər:** `ps` çıxışında Z state, orphan child-ın PPID-sinin dəyişməsi.
- **Gözlənilən nəticə:** Z state parent `wait()` edənə qədər qalır; orphan-ı PID 1 (və ya subreaper) övladlığa götürür.

Öz layihəndə

App-ının neçə process-də işlədiyini yoxla (`adb shell ps -A`). SDK-lar ayrı process açırsa, `Application.onCreate`-də hər process-də lazımsız təkrarlanan init-ləri tap.

Praktika

- [ ] C-də `fork()` yaz, parent və child PID-lərini çap et; fork-dan sonra dəyişəni dəyişərək COW-u müşahidə et.
- [ ] Bilərəkdən zombie yarat və `ps aux` ilə Z state-ini tap; sonra orphan yarat.
- [ ] `fork` + `exec` + `wait` ilə mini shell yaz (`ls`, `echo` işlətsin).
- [ ] `/proc/self/maps` və `/proc/<pid>/status` (VmRSS, Threads) fayllarını oxu və hər sətri izah et.
- [ ] Android: `adb shell ps -A` ilə app-in PPID-sinin zygote64 olduğunu gör.
- [ ] Android: app-i arxa plana atıb `/proc/<pid>/oom_score_adj` dəyişməsini izlə; `adb shell am kill <paket>` ilə process death simulyasiya et.

Özünü yoxla

- `fork()` niyə iki dəfə qayıdır? 1 GB yaddaşı olan process-i fork etmək niyə ucuzdur?
- Zygote niyə fork edir, amma exec etmir?
- İki process eyni virtual ünvana yazanda niyə bir-birinə mane olmur?
- App arxa planda öldürüləndə ViewModel niyə itir, SavedStateHandle niyə qalır?

- (İ) App arxa planda öldürülüb geri qayıdanda nə itir, nə qalır? Bunu hansı mexanizm təmin edir?
- (İ) `android:process` istifadə etməyin üstünlükləri və riskləri nələrdir?

Mənbələr

- OSTEP — Processes, Process API, Address Spaces, Paging, TLBs, Swapping fəsilləri.
- *The Linux Programming Interface* (Michael Kerrisk) — fəsil 6, 24–28.
- Processes and threads overview (Android Developers).

Mənimsəmə meyarı: fork-exec-wait kodunu yaddaşdan yazırsan və app-in işə düşməsini zygote-dan `Activity.onCreate`-ə qədər izah edirsən.

Faza 2 — Context switch və scheduling (2 həftə) · Intermediate

Bir core eyni anda yalnız bir thread icra edir; yüzlərlə thread-in "eyni vaxtda" işləməsi illüziyasını scheduler və context switch yaradır. Jank və ANR-in səbəbi çox vaxt kodda yox, scheduling-dədir.

Problem: 8 core var, 500 runnable thread var — hansı thread nə vaxt, hansı core-da və nə qədər işləyəcək? Main thread niyə bəzən hazır olduğu halda 10 ms gözləyir?

Niyə vacibdir: Perfetto-da main thread-in "Runnable" vəziyyətində gözlədiyini görəndə bunun nə demək olduğunu yalnız bu faza izah edir.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 2, 6 · [Dərin] 3, 5 · [Ekspert] 4 (CFS və EEVDF daxili).

1. **Time sharing və preemption**
    - Cooperative və preemptive multitasking; time slice (quantum).
    - Timer interrupt kernel-ə nəzarəti geri qaytarır — preemption belə işləyir.
2. **Context switch under the hood (addım-addım)**
    1. Interrupt və ya syscall ilə kernel-ə keçid.
    2. Cari thread-in register-ləri kernel stack-ə və `task_struct`-a saxlanılır.
    3. Scheduler növbəti task-ı seçir.
    4. Process fərqlidirsə, page table dəyişir (ARM64 TTBR0, x86 CR3); ASID yoxdursa TLB təmizlənir.
    5. Yeni thread-in register-ləri bərpa olunur, user space-ə qayıdılır.
    - Birbaşa xərc (mikrosaniyələr) və dolayı xərc (soyumuş cache və TLB).
    - Eyni process-in thread-ləri arasında switch process switch-dən ucuzdur. Mode switch context switch demək deyil.
3. **Klassik scheduling alqoritmləri**
    - FIFO, SJF, STCF, Round Robin, Priority, MLFQ, lottery və stride.
    - Metriklər: turnaround time, response time, throughput, fairness, starvation.
4. **Linux scheduler**
    - Scheduling class-lar: stop, deadline, rt, fair, idle.
    - CFS: vruntime, red-black tree, nice dəyərləri (-20..19) və çəki cədvəli.
    - Linux 6.6-dan EEVDF CFS-i əvəz etdi: lag, eligibility, virtual deadline anlayışları.
    - Real-time siyasətlər: SCHED_FIFO, SCHED_RR, SCHED_DEADLINE.
5. **Multicore scheduling**
    - Per-CPU runqueue, load balancing, migration xərci.
    - CPU affinity (`sched_setaffinity`), cache affinity, NUMA (qısa).
6. **Android-da scheduling**
    - Heterogen CPU: big.LITTLE/DynamIQ (little, mid, big/prime core-lar).
    - EAS (Energy Aware Scheduling) — enerji modelinə görə core seçimi.
    - cgroups: cpuset (top-app, foreground, background, system-background), uclamp. Main thread və RenderThread niyə top-app-də böyük core-larda işləyir.

- Thread prioriteti: `Process.setThreadPriority`, `THREAD_PRIORITY_DEFAULT` (0), `BACKGROUND` (10), `DISPLAY` (-4), `URGENT_DISPLAY` (-8) — bunlar Linux nice dəyərləridir. Java `Thread.setPriority(1..10)` xəritələnməsi.
- Perfetto thread state-ləri: Running, Runnable, Runnable (Preempted), Sleeping, Uninterruptible Sleep.

Mental model: Round Robin, 1 core, quantum 4 ms

Üç thread eyni anda hazırdır: A-ya 6 ms, B-yə 3 ms, C-yə 5 ms CPU lazımdır.

Vaxt (ms)	Core-da	Runqueue	Hadisə
0–4	A	B, C	A-nın quantum-u bitdi → preemption
4–7	B	C, A	B bitdi (3 ms)
7–11	C	A	C-nin quantum-u bitdi → preemption
11–13	A	C	A bitdi (cəmi 6 ms)
13–14	C	—	C bitdi (cəmi 5 ms)

Orta turnaround = (13 + 7 + 14) / 3 ≈ 11.3 ms. Response time: A — 0, B — 4, C — 7 ms. Context switch sayı 4-dür. Quantum-u 1 ms etsən, response time yaxşılaşır, amma switch sayı artır — trade-off-u hesabla.

12 sual: context switch

№	Sual	Qısa cavab
1	Nədir?	CPU-nun bir task-ın icrasını saxlayıb başqasına keçməsi
2	Niyə mövcuddur?	Runnable task-lar core-lardan çoxdur; bloklanan task-ın yerinə başqasını işlətmək lazımdır
3	Hansı problemi həll edir?	Bir proqramın CPU-nu inhisara alması; I/O gözləyərkən CPU-nun boş qalması
4	Əvvəl necə idi?	Cooperative multitasking (Windows 3.x, klassik Mac OS): proqram CPU-nu özü buraxmalı idi
5	API	Birbaşa API yoxdur; dolayı: `sched_yield()`, bloklayan çağırışlar, `nice`, `Process.setThreadPriority`
6	Runtime	ART thread state-ini dəyişir (Runnable → Native/Waiting), switch-i isə kernel edir
7	Kernel	`schedule()` → növbəti task-ın seçilməsi → `context_switch()` → `switch_mm` + `switch_to`
8	CPU	Register-lər saxlanılır və bərpa olunur; process fərqlidirsə TTBR0/CR3 dəyişir
9	Yaddaş	Kontekst kernel stack-də və `task_struct`-da saxlanılır; cache və TLB "soyuyur"
10	Qiyməti	Birbaşa təxminən bir neçə mikrosaniyə (aparatdan asılı); dolayı: sonrakı cache miss-lər
11	Edge case-lər	Priority inversion (Faza 5), core-lar arası migration, real-time task-ların digərlərini ac qoyması
12	Alternativlər	User-space scheduling: coroutine-lər, goroutine-lər, virtual thread-lər — kernel-ə girmədən, daha ucuz

Under-the-hood zənciri

Timer interrupt → kernel girişi → tick handler → need_resched bayrağı → `schedule()` → EEVDF növbəti task-ı seçir → `context_switch()` → `switch_mm` + `switch_to` → yeni thread-in register-ləri → user space

Digər ekosistemlərdə

- Go: GMP modeli — goroutine-ləri runtime M (OS thread) və P (processor) üzərində özü planlaşdırır; Go 1.14-dən asinxron preemption var.
- Java 21 virtual thread-ləri: JVM onları ForkJoinPool carrier thread-ləri üzərində planlaşdırır.
- Windows: 0–31 prioritet səviyyəli, preemptive thread scheduler.

Gözünlə gör

- `grep ctxt /proc/<pid>/status` — voluntary və nonvoluntary context switch sayları.
- `perf stat -e context-switches,cpu-migrations ./prog`.
- `perf sched record` və `perf sched latency`.
- `chrt -p <pid>`, `taskset -p <pid>`.
- Android: Perfetto-da thread state track-ı (Running, Runnable, Runnable (Preempted)) və CPU track-ları.

Laboratoriya

Lab 2.1 — Thread sayı və performans

- **Hipotez:** CPU-bound işdə thread sayını core sayından artıq etmək sürəti artırmır, context switch-ləri artırır.
- **Qurulum:** sabit iş (məsələn, 2^30 hash hesablaması) 1, 2, 4, 8, 16, 32, 100, 500, 1000 thread arasında bərabər bölünür.
- **Ölçülər:** ümumi vaxt, `context-switches`, `cpu-migrations`, CPU istifadəsi.
- **Gözlənilən nəticə:** vaxt core sayına qədər azalır, sonra sabitləşir və ya artır; context switch sayı artmağa davam edir.

Lab 2.2 — Pipe ping-pong

- **Qurulum:** iki process pipe ilə 1 byte-ı 1 000 000 dəfə qarşılıqlı ötürür; əvvəlcə bir core-a (`taskset -c 0`), sonra ayrı core-lara bağla.
- **Ölçü:** bir gediş-gəlişin orta vaxtı.
- **Gözlənilən nəticə:** bir core-da hər gediş-gəliş 2 context switch deməkdir; ayrı core-larda isə wakeup gecikməsi görünür.

Öz layihəndə

App-ının scroll trace-ində main thread-in ən uzun "Runnable" intervalını tap. O anda core-u hansı thread tuturdu və niyə?

Praktika

- [ ] Pipe ping-pong ilə iki process arasında context switch xərcini ölç.
- [ ] `perf sched record` və `perf sched latency`, `chrt -f 50`, `taskset -c 0`, `nice -n 19` ilə eksperiment et.
- [ ] Kotlin-də scheduler simulyatoru yaz: FIFO, RR, MLFQ — eyni iş yükü üçün metrikləri müqayisə et.
- [ ] Android: scroll zamanı Perfetto trace yaz; main thread və RenderThread-in hansı core-larda işlədiyini və Runnable vaxtını tap.
- [ ] Background thread-ə `THREAD_PRIORITY_BACKGROUND` ver və trace-də fərqi müqayisə et.

Özünü yoxla

- Context switch zamanı dəqiq nə saxlanılır və harada?
- Eyni process-in thread-ləri arasında switch niyə ucuzdur?
- nice 0 və nice 10 olan iki thread CPU-nu təxminən necə bölüşür?
- Main thread-in Runnable-da 8 ms gözləməsi nəyi göstərir və nə edərdin?

- (İ) Thread-in "Runnable" və "Running" vəziyyətləri arasında fərq nədir? Perfetto-da hər birini necə görürsən?
- (İ) Background thread-ə niyə `THREAD_PRIORITY_BACKGROUND` verməlisən?

Mənbələr

- OSTEP — Limited Direct Execution, Scheduling: Introduction, MLFQ, Lottery, Multiprocessor Scheduling.
- *Linux Kernel Development* (Robert Love) — fəsil 4 (Process Scheduling).
- Linux kernel sənədləri: CFS və EEVDF scheduler təsvirləri (docs.kernel.org).
- Perfetto docs — CPU scheduling və thread state bölmələri.

Mənimsəmə meyarı: Perfetto trace-də bir thread-in niyə vaxtında işləmədiyini (preempted, kiçik core, aşağı prioritet) sübut ilə izah edə bilirsən.

Faza 3 — Thread: nədir, niyə var, necə işləyir (2 həftə) · Intermediate

Thread eyni process daxilində ayrıca icra axınıdır: öz register-ləri və stack-i var, amma heap-i və kodu digər thread-lərlə paylaşır. Linux kernel üçün thread sadəcə resursları paylaşan başqa bir task-dır.

Problem: Server eyni anda 3 sorğu alır, biri database-i 200 ms gözləyir. Bütün proqram bir icra axınıdırsa, qalan iki sorğu da gözləyir. Gözləyən iş digərlərini dayandırmasın deyə nə etməli — həm də ayrı process yaratmadan?

Niyə var: process ağırdır — ayrı address space, bahalı yaradılış və switch, çətin IPC. Thread-lər paylaşılmış yaddaşla ucuz əməkdaşlıq verir. Məqsədlər: responsiveness (UI donmur), parallelism (çox core), I/O gözləməsini gizlətmək.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 2, 5, 6, 9 · [Dərin] 3, 4, 7, 8, 10 · [Ekspert] `clone()`-un kernel tərəfi (Faza 10-da).

1. **Concurrency və parallelism**
   - Concurrency — bir neçə işin zamanca üst-üstə düşməsi (bir core-da da mümkündür). Parallelism — eyni anda fiziki icra (çox core).
   - Amdahl qanunu: proqramın ardıcıl hissəsi (1 − p) sürətlənmənin tavanını müəyyən edir; N — core sayı.

```
S(N) = \frac{1}{(1 - p) + \frac{p}{N}}
```

2. **Thread nəyi paylaşır, nəyi paylaşmır** (cədvəl aşağıda).
3. **Threading modelləri**
   - N:1 (user-level, green threads) — kernel bir thread görür, bloklanma hamısını dayandırır.
   - 1:1 (kernel-level) — Linux və Android belədir.
   - M:N (hibrid) — Go goroutine-ləri; Java virtual thread-ləri və Kotlin coroutine-ləri bu ideyaya yaxındır.
4. **Linux-da thread under the hood**
   - Thread = ayrı `task_struct`, `clone()` syscall-ı ilə yaradılır.
   - Bayraqlar: `CLONE_VM`, `CLONE_FS`, `CLONE_FILES`, `CLONE_SIGHAND`, `CLONE_THREAD`, `CLONE_SETTLS`, `CLONE_CHILD_CLEARTID` — hər birinin nəyi paylaşdırdığını öyrən.
   - TGID (`getpid()`) və TID (`gettid()`) fərqi; NPTL kitabxanası.
5. **pthreads API**
   - `pthread_create`, `pthread_join`, `pthread_detach`, `pthread_exit`, `pthread_self`.
   - Atributlar: stack ölçüsü, detach state; joinable və detached fərqi.
6. **Thread stack**
   - Hər thread-in stack-i `mmap` ilə ayrılır; guard page overflow-da SIGSEGV verir.
   - Ölçülər: Linux-da pthread üçün adi halda 8 MB; Android Java thread-ləri üçün təxminən 1 MB.
   - `Thread(group, runnable, name, stackSize)` konstruktoru; ART `StackOverflowError`-u guard region ilə necə tutur.
7. **Thread-local storage (TLS)**
   - C: `thread_local`, `pthread_key_create`; ARM64-də TPIDR_EL0 register-i TLS ünvanını saxlayır.
   - Java `ThreadLocal` under the hood: hər `Thread` obyektində `ThreadLocalMap`, açar `ThreadLocal`-a WeakReference, dəyər isə strong — thread pool-da memory leak mənbəyi.
   - `Looper.myLooper()` ThreadLocal üzərində qurulub (Faza 8).
8. **Java/ART-da thread under the hood**
   - `Thread.start()` → native `nativeCreate` → `art::Thread` → `pthread_create` → `clone()`. Java obyekti, native ART obyekti və kernel task — üç qatı bir-birinə bağla.
   - `Thread.State`: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED. Diqqət: Java-da RUNNABLE olan thread I/O-da bloklanmış ola bilər.
   - Daemon thread-lər; `interrupt()` mexanizmi (flag + InterruptedException); `Thread.stop()` niyə deprecated-dir.

9. **Android app-in thread-ləri**
    - main, RenderThread, HeapTaskDaemon, FinalizerDaemon, FinalizerWatchdogDaemon, ReferenceQueueDaemon, Signal Catcher, Jit thread pool, `Binder:<pid>_N`, hwuiTask, Profile Saver — hər birinin rolunu yaz.

10. **I/O modelləri və event loop** — thread-lərin yaranma səbəblərindən biri I/O gözləməsidir; Looper (Faza 8), OkHttp və coroutines (Faza 9) bu ideyalar üzərində qurulub.
    - Blocking I/O: `read()` data gələnə qədər thread-i S state-ə salır. "Bir connection = bir thread" modeli və C10K problemi.
    - Non-blocking I/O: `O_NONBLOCK`, `EAGAIN` cavabı, busy polling-in zərəri.
    - I/O multiplexing: `select` → `poll` → `epoll` (`epoll_create`, `epoll_ctl`, `epoll_wait`); level-triggered və edge-triggered fərqi; `eventfd` ilə oyatma.
    - Asinxron I/O: `io_uring` (ideya səviyyəsində); reactor və proactor pattern-ləri.
    - Event loop: bir thread + bir növbə + bir "gözlə" nöqtəsi. Android Looper, Node.js, Netty eyni modeldir.
    - Praktika: C-də `epoll` ilə bir thread-li echo server yaz, sonra "hər client-ə bir thread" versiyası ilə 1000 connection-da yaddaşı müqayisə et.

Paylaşılan (process səviyyəsi)	Hər thread-in özününkü
Kod (.text), global data, heap	TID, register-lər, PC
File descriptor-lar	Stack
Signal handler-lər, cwd	TLS, `errno`
UID/GID, mmap bölgələri	Signal mask, scheduling prioriteti

Mental model: iki thread, bir address space

Hər iki thread eyni `foo()` funksiyasını işlədir: `int counter;` global, `int x = 10;` isə `foo()` daxilində lokal dəyişəndir.

Yaddaş bölgəsi	Thread A	Thread B	Paylaşılır?
`.text` (`foo` kodu)	eyni kod	eyni kod	Bəli
`.data` (`counter`)	eyni `counter`	eyni `counter`	Bəli — race buradan başlayır
Heap (`malloc`, Java obyektləri)	eyni heap	eyni heap	Bəli
Stack A (`x = 10`)	öz `x`-i	—	Xeyr
Stack B (`x = 10`)	—	öz `x`-i	Xeyr
Register-lər, PC, SP	öz dəyərləri	öz dəyərləri	Xeyr

12 sual: thread

№	Sual	Qısa cavab
1	Nədir?	Process daxilində öz register-ləri və stack-i olan icra axını
2	Niyə mövcuddur?	Paralel icra və gözləmə zamanı başqa işin davam etməsi; process-dən ucuzdur

№	Sual	Qısa cavab
3	Hansı problemi həll edir?	Bir axınlı proqramda bloklanmanın hər şeyi dayandırması; çox core-dan istifadə
4	Əvvəl necə idi?	Hər sorğuya ayrı process (fork per request) və ya event loop + callback-lər
5	API	`pthread_create`, `Thread(...).start()`, `Executor`, `HandlerThread`
6	Runtime	ART `art::Thread` obyekti Java `Thread` ilə bağlanır; thread state-ləri, GC üçün suspend point-lər
7	Kernel	`clone()` ilə yeni `task_struct`, eyni `mm_struct` və TGID, scheduler-ə əlavə
8	CPU	Hər thread-in öz register konteksti; TLS ünvanı ARM64-də TPIDR_EL0-da
9	Yaddaş	Ayrı stack (`mmap` + guard page), paylaşılan heap; Android Java thread stack-i təxminən 1 MB
10	Qiyməti	Yaradılma (təxminən onlarla mikrosaniyə), stack yaddaşı, scheduler yükü, sinxronizasiya xərci
11	Edge case-lər	Stack overflow, ThreadLocal leak, daemon thread-in yarımçıq kəsilməsi, udulan `InterruptedException`
12	Alternativlər	Thread pool (Faza 7), event loop (Faza 3.10, Faza 8), coroutine (Faza 9), process

Under-the-hood zənciri

`Thread.start()` → ART `Thread::CreateNativeThread` → bionic `pthread_create` → stack üçün `mmap` + guard page → `clone()` (CLONE_VM, CLONE_THREAD və digər bayraqlarla) → kernel `copy_process()` → yeni `task_struct` runqueue-da → scheduler seçir → `run()`

Digər ekosistemlərdə

- C++: `std::thread` — pthreads üzərində nazik qat.
- Go: goroutine-lər M:N modelindədir; stack kiçik (bir neçə KB) başlayır və lazım olduqca böyüyür.
- Node.js: JavaScript üçün bir thread + fayl I/O və DNS üçün libuv thread pool (default 4 thread).
- Java 21: virtual thread-lər — JVM-in idarə etdiyi yüngül thread-lər.

Gözünlə gör

- `ps -eLf`, `ps -T -p <pid>`, `top -H -p <pid>`.
- `ls /proc/<pid>/task/`, `cat /proc/<pid>/task/<tid>/status`.
- `strace -f -e trace=clone,clone3 ./prog` — bayraqları ilə birlikdə.
- Android: `adb shell ps -T -p <pid>`; `/proc/<pid>/maps`-də thread stack-ləri `[anon:stack_and_tls:<tid>]` adı ilə görünür.

Laboratoriya

Lab 3.1 — Race-in ilk görünüşü

- **Hipotez:** sinxronizasiyasız `counter++` itirilmiş artımlar verir.
- **Qurulum:** 100 thread × 100 000 artım; gözlənilən dəyər 100 × 100 000 = 10 000 000.

- **Variantlar:** plain counter (bu fazada); mutex və atomic variantları Faza 4-də.
- **Ölçülər:** son dəyər, 10 qaçışda minimum və maksimum, vaxt.
- **Gözlənilən nəticə:** nəticə hər dəfə fərqlidir və 10 000 000-dan azdır; səbəbi Faza 4-ün mental modelində görünəcək.

Lab 3.2 — Thread-in qiyməti

- **Qurulum:** 10 000 thread yarat, hər biri 1 saniyə yatsın; sonra eyni işi 100 000 coroutine ilə et.
- **Ölçülər:** yaradılma vaxtı, VmRSS və Threads (`/proc/<pid>/status`); Android cihazda limitə çatanda xəta (`pthread_create failed`).
- **Gözlənilən nəticə:** hər thread ayrıca stack (virtual yaddaş) və kernel task-ı tutur; coroutine-lər heap-də kiçik obyektlərdir.

Lab 3.3 — Event loop — təkamül layihəsinin 1–4-cü addımları (Layihə nərdivanı bölməsinə bax).

Öz layihəndə

App-ının thread siyahısını çıxar (`adb shell ps -T -p <pid>`). Adı ümumi olan thread-ləri (`Thread-12`, `pool-3-thread-1`) tap, hansı kitabxanaya aid olduqlarını müəyyən et və öz thread-lərinə mənalı ad ver.

Praktika

- [ ] C-də iki pthread yarat, global counter-i artırsınlar və join et (race-i Faza 4-də araşdıracaqsan).
- [ ] `ps -T -p <pid>`, `ls /proc/<pid>/task`, `top -H` ilə thread-ləri gör; TID-ləri `gettid()` çıxışı ilə tutuşdur.
- [ ] `strace -f` ilə `pthread_create`-in içində `clone()` və bayraqlarını gör.
- [ ] Kotlin: 10 000 thread yaratmağa cəhd et, yaddaşı və xətanı müşahidə et (Faza 9 üçün motivasiya).
- [ ] ThreadLocal leak-ni thread pool-da reproduce et.
- [ ] Android: `adb shell ps -T -p <pid>` ilə app thread-lərinin siyahısını çıxar, hər birinin rolunu yaz.

Özünü yoxla

- Linux kernel üçün thread ilə process arasında fərq nədir?
- Java-da RUNNABLE vəziyyətindəki thread həqiqətən CPU-da işləyirmi?
- ThreadLocal thread pool-da niyə təhlükəlidir?
- 2000 thread yaratmanın real xərci nədir (yaddaş, scheduler, kernel obyektləri)?

- (İ) Process ilə thread arasında fərq nədir — Linux kernel səviyyəsində?
- (İ) Event loop minlərlə connection-u bir thread-lə necə idarə edir?

Mənbələr

- OSTEP — Concurrency: An Introduction, Thread API.
- *The Linux Programming Interface* — fəsil 29–33 (POSIX threads).
- *Java Concurrency in Practice* (Brian Goetz) — fəsil 1.
- cs.android.com — `art/runtime/thread.cc`, `java_lang_Thread.cc`.

Mənimsəmə meyarı: `Thread.start()` çağırışından kernel-də yeni task-ın scheduler-ə düşməsinə qədər bütün qatları izah edə bilirsən.

Faza 4 — Race condition və sinxronizasiya (3 həftə) · Intermediate

Thread-lər yaddaşı paylaşır, ona görə eyni dataya eyni anda yazanda nəticə qeyri-müəyyən olur. Sinxronizasiya bu xüsusiyyəti nəzarət altına alan alətlər dəstidir: hardware atomik instruksiyalardan `synchronized`-a qədər.

Problem: İki thread eyni `counter`-i 1 000 000 dəfə artırır. Nəticə 2 000 000 olmalıdır, amma hər qaçışda fərqli və daha kiçik ədəd çıxır. Niyə? Və bunu ən ucuz yolla necə düzəltmək olar?

Niyə vacibdir: sinxronizasiyanı bilmədən thread işlətmək gizli bug fabrikidir. Bu bug-lar testdə görünmür, production-da təsadüfi crash kimi çıxır.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 2, 4, 5, 7 · [Dərin] 3, 6, 8 · [Ekspert] 9 (AQS daxili).

1. **Race condition, critical section, data race**
   - Data race: iki thread eyni yaddaşa müraciət edir, ən azı biri yazır, sinxronizasiya yoxdur.
   - Race condition: məntiqi səhv (məsələn, check-then-act). Data race olmadan da ola bilər.
   - `counter++` niyə atomik deyil: load → add → store. Assembly-də özün gör.
2. **Critical section tələbləri**
   - Mutual exclusion, progress, bounded waiting.
   - Peterson alqoritmi və onun müasir CPU-da memory reordering üzündən niyə sındığı (Faza 6-ya körpü).
3. **Hardware primitivləri**
   - Test-and-set, compare-and-swap (CAS), fetch-and-add.
   - ARM64-də LL/SC (`LDXR`/`STXR`) və ARMv8.1 LSE instruksiyaları (`CAS`, `LDADD`); cache səviyyəsində exclusive monitor.
4. **Lock növləri**
   - Spinlock (CAS-loop): qısa critical section və çox core olanda məqsədəuyğundur.
   - Mutex (bloklayan) və adaptive lock (əvvəl spin, sonra yatmaq).
   - **Futex under the hood:** contention yoxdursa syscall-sız user-space CAS; contention olanda `futex(FUTEX_WAIT/FUTEX_WAKE)` ilə kernel wait queue. Demək olar ki, bütün müasir lock-lar bunun üzərində qurulub.
5. **Semaphore, condition variable, monitor**
   - Dijkstra semaphore-u (P/V), counting və binary növləri.
   - Condition variable: `wait` lock-u atomik buraxıb yatır. Spurious wakeup olur, ona görə həmişə `while` loop.
   - Monitor = lock + condition variable-lar.
6. **Digər lock-lar**
   - Reentrant lock (hold count), read-write lock, fair və unfair lock.
   - Lock granularity (coarse və fine-grained), lock striping.
7. **Klassik problemlər**

    - Producer-consumer (bounded buffer), readers-writers, dining philosophers, sleeping barber.
    - Hər birini həm semaphore, həm monitor ilə həll et.
8. **Java `synchronized` under the hood**
    - Hər obyektin monitoru; bytecode-da `monitorenter`/`monitorexit`.
    - ART: obyekt başlığındakı lock word — thin lock (CAS ilə owner thread id + say), contention olanda fat lock-a (Monitor obyekti) inflation.
    - HotSpot-da biased locking (JDK 15-də deprecated) — tarixi kontekst üçün.
    - `wait`/`notify`/`notifyAll` yalnız monitor tutulanda; `IllegalMonitorStateException`.
9. **`java.util.concurrent.locks`**
    - `ReentrantLock` (`tryLock`, `lockInterruptibly`, fairness), `Condition`, `ReadWriteLock`, `StampedLock` (optimistic read).
    - **AQS (AbstractQueuedSynchronizer) daxili:** `volatile int state`, CLH variantı olan FIFO gözləmə növbəsi, `LockSupport.park/unpark` (altında futex), exclusive və shared rejim.
    - `ReentrantLock`, `Semaphore`, `CountDownLatch` — hamısı AQS üzərində qurulub.

Mental model: itirilmiş artım (lost update)

Addım	Thread A	Thread B	`counter` (yaddaş)
1	LOAD r1 ← counter (0)		0
2	ADD r1 = 1		0
3	context switch	LOAD r2 ← counter (0)	0
4		ADD r2 = 1	0
5		STORE counter ← 1	1
6	STORE counter ← 1		1

İki artım oldu, nəticə isə 1-dir. Mutex 1–6 addımlarını bölünməz edir. Atomic əməliyyat (ARM64 LSE `LDADD` və ya `LDXR`/`STXR` dövrü) eyni işi instruksiya səviyyəsində görür.

12 sual: mutex

№	Sual	Qısa cavab
1	Nədir?	Eyni anda yalnız bir sahibi olan mutual exclusion primitivi
2	Niyə mövcuddur?	Paylaşılan mutable state-i critical section-da qorumaq
3	Hansı problemi həll edir?	Race condition və itirilmiş yeniləmələr
4	Əvvəl necə idi?	Tək CPU-da interrupt-ları söndürmək; Peterson və Dekker kimi proqram alqoritmləri
5	API	`pthread_mutex_lock`, `synchronized`, `ReentrantLock`, Kotlin `Mutex` (suspend)
6	Runtime	ART: lock word-da CAS ilə thin lock, contention olanda fat lock (Monitor); AQS: state + CLH növbəsi + park
7	Kernel	Contention-da `futex(FUTEX_WAIT)`: thread wait queue-ya düşür; unlock-da `FUTEX_WAKE`
8	CPU	CAS (`LDXR`/`STXR` və ya LSE `CAS`), acquire/release semantikası (Faza 6)

№	Sual	Qısa cavab
9	Yaddaş	Lock word-un cache line-ı core-lar arasında köçür (MESI); critical section-dakı yazılar unlock-dan sonra görünür
10	Qiyməti	Uncontended: bir CAS (nanosaniyələr); contended: syscall + context switch (mikrosaniyələr) + cache line ping-pong
11	Edge case-lər	Reentrancy, sahibi ölən lock, priority inversion, lock tutarkən callback, unlock-un unudulması (`try/finally`)
12	Alternativlər	Atomics (tək dəyər üçün), lock-free strukturlar, immutability, thread confinement, read-write lock

Under-the-hood zənciri

`synchronized` → `monitorenter` → ART lock word-da CAS → contention: thin lock → fat lock inflation → `futex(FUTEX_WAIT)` → kernel wait queue → scheduler → context switch → `FUTEX_WAKE` → yenidən CAS → cache coherence. Böyük şəkildəki zəncir diagramı bu yolu addım-addım göstərir.

Digər ekosistemlərdə

- C++: `std::mutex` (Linux-da futex üzərində), RAII üçün `std::lock_guard`.
- Go: `sync.Mutex`-in normal və starvation rejimləri var; kanallar alternativ yanaşmadır.
- C#: `lock` (Monitor), async-dostu `SemaphoreSlim`.
- Kotlin coroutines: `Mutex` thread-i bloklamır, coroutine-i suspend edir (Faza 9).

Gözünlə gör

- `strace -f -e futex ./prog` — contention zamanı futex çağırışları.
- `perf stat -e context-switches` — mutex və spinlock müqayisəsi.
- `perf lock` (dəstəklənən kernel-lərdə) — lock contention statistikası.
- Android: Perfetto-da "monitor contention with owner ..." slice-ları; ANR trace-də "waiting to lock".

Laboratoriya

Lab 4.1 — Race lab (Lab 3.1-in davamı)

- **Hipotez:** mutex və atomic düzgün nəticə verir, amma qiymətləri fərqlidir.
- **Qurulum:** 100 thread × 100 000 artım; gözlənilən dəyər 10 000 000.
- **Variantlar:** plain, mutex, atomic, `LongAdder` (JVM-də).
- **Ölçülər:** düzgünlük, vaxt, CPU, context switch sayı (`perf stat`).
- **Gözlənilən nəticə:** plain səhvdir; atomic adətən mutex-dən sürətlidir; contention artdıqca `LongAdder` önə keçir (Faza 7).

Lab 4.2 — Spinlock və mutex

- **Qurulum:** öz CAS spinlock-un və `pthread_mutex`; critical section uzunluğu 10 ns, 1 µs, 100 µs; thread sayı 2, 8, 32.
- **Ölçülər:** ümumi vaxt, CPU istifadəsi.

- **Gözlənilən nəticə:** qısa critical section və az thread-də spinlock udur; uzun critical section-da və core-dan çox thread-də spinlock CPU yandırır və uduzur.

Lab 4.3 — Bounded queue stress testi

- **Qurulum:** 4 producer, 4 consumer, tutum 16; hər producer 1 000 000 element göndərir; consumer-lər cəmi yoxlayır.
- **Ölçülər:** itən və təkrarlanan element yoxdur (cəm yoxlaması), throughput.
- **Sındır:** `wait()`-i `if` ilə yaz və `notify()` istifadə et — nəticənin sındığını gör, sonra `while` və `notifyAll()` ilə düzəlt.

Öz layihəndə

Kod bazanda `synchronized` və `@Synchronized` istifadələrini tap. Lock tutularkən callback, I/O və ya başqa lock çağırılan yerləri işarələ.

Praktika

- [ ] C və Kotlin-də 2 thread × 1 000 000 `counter++` — itirilmiş artımları gör. Sonra mutex, `AtomicInteger`, `synchronized` ilə düzəlt və JMH ilə sürətləri müqayisə et.
- [ ] CAS ilə öz spinlock-unu yaz, sonra test-and-test-and-set və exponential backoff əlavə et.
- [ ] Bounded blocking queue-nu 3 üsulla yaz: `synchronized` + `wait/notify`, `ReentrantLock` + 2 `Condition`, iki `Semaphore`.
- [ ] Dining philosophers yaz, deadlock-a sal, sonra düzəlt.
- [ ] `strace -f -e futex` ilə contention zamanı futex syscall-larını gör.
- [ ] AQS mənbə kodunu oxu: `acquire()`, `release()`, növbəyə əlavə olunma; `ReentrantLock.Sync`-i izlə.

Özünü yoxla

- Data race olmadan race condition ola bilərmi? Nümunə gətir.
- Condition variable-da niyə `if` yox, `while` işlədilir?
- Futex niyə "fast" adlanır?
- `synchronized` və `ReentrantLock` arasında nə vaxt hansını seçərdin?

- (İ) `synchronized` ilə `ReentrantLock` fərqi nədir və nə vaxt hansını seçərsən?
- (İ) Semaphore ilə mutex arasında fərq nədir?

Mənbələr

- OSTEP — Locks, Lock-based Concurrent Data Structures, Condition Variables, Semaphores.
- The Little Book of Semaphores (Allen Downey, pulsuz).
- *Java Concurrency in Practice* — fəsil 2–5, 13, 14.
- Futexes Are Tricky (Ulrich Drepper).
- JDK mənbəsi: `AbstractQueuedSynchronizer.java`; ART: `art/runtime/monitor.cc`, `lock_word.h`.

Mənimsəmə meyarı: bounded blocking queue-nu kitabsız yazırsan və `synchronized`-ın thin lock-dan futex-ə qədər yolunu izah edirsən.

Faza 5 — Deadlock, livelock, starvation, priority inversion (1 həftə) · Intermediate

Lock düzgünlüyü (safety) təmin edir, amma yeni risk yaradır: thread-lər irəliləməyə bilər (liveness). Android-da bunun ən tanınmış simptomu ANR-dir.

Problem: App testdə normal işləyir, amma production-da ayda bir neçə dəfə ANR verir və trace-də main thread "Blocked" görünür. Kim kimi gözləyir və bu niyə testdə heç vaxt təkrarlanmır?

Niyə vacibdir: senior developer ANR trace-ni açıb "kim kimi gözləyir" zəncirini dəqiqələr içində tapmalıdır.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 3, 5 · [Dərin] 2, 4 · [Ekspert] PI-futex və kernel lockdep (Faza 10-da).

1. **Deadlock**
   - Coffman şərtləri: mutual exclusion, hold and wait, no preemption, circular wait — dördü birlikdə olmalıdır.
   - Resource allocation graph ilə deadlock-u çəkmək.
   - Prevention (bir şərti sındır — ən praktik yol qlobal lock ordering), avoidance (Banker alqoritmi), detection və recovery; `tryLock(timeout)`.
2. **Gizli deadlock-lar**
   - Lock tutarkən yad metod (callback, listener) çağırmaq.
   - Class initialization deadlock (iki class-ın static initializer-ləri bir-birini gözləyir).
   - Main thread-in background thread-i gözləməsi: `runBlocking`, `Future.get()`, `Thread.join()` main-də.
   - Coroutine `Mutex` reentrant deyil — özünü gözləyən coroutine (Faza 9).
3. **Livelock və starvation**
   - Livelock: thread-lər işləyir, bir-birinə yol verir, amma irəliləyiş yoxdur.
   - Starvation: unfair lock, aşağı prioritet, sonsuz yeni iş axını. Fairness-in qiyməti (throughput itkisi).
4. **Priority inversion**
   - Aşağı prioritetli thread lock tutur, yüksək prioritetli gözləyir, orta prioritetli isə CPU-nu tutur.
   - Mars Pathfinder (1997) hadisəsi; həll yolları: priority inheritance, priority ceiling.
   - Android-da: main thread background thread-in lock-unu gözləyir, o thread isə kiçik core-da və background cgroup-da yavaş işləyir → jank və ANR.
5. **Diaqnostika**
   - Thread dump: `jstack`; Android-da `adb shell kill -3 <pid>` (SIGQUIT → Signal Catcher), `/data/anr/` trace-ləri, `adb bugreport`.
   - Trace oxumaq: "waiting to lock <0x...> held by thread N", ART thread state-ləri (Runnable, Blocked, Waiting, Native, Suspended).

- Perfetto-da "monitor contention with owner ..." slice-ları.

Mental model: iki lock-lu deadlock

Addım	Thread 1	Thread 2	Lock A	Lock B
1	`lock(A)` alındı		T1	boş
2		`lock(B)` alındı	T1	T2
3	`lock(B)` istəyir → gözləyir		T1	T2
4		`lock(A)` istəyir → gözləyir	T1	T2
5	yatır	yatır	circular wait	circular wait

Lock ordering qaydası (həmişə əvvəl A, sonra B) 2-ci addımı qeyri-mümkün edir: Thread 2 B-dən əvvəl A-nı istəməlidir.

12 sual: deadlock

№	Sual	Qısa cavab
1	Nədir?	Thread-lərin bir-birinin tutduğu resursu dövri gözləməsi; heç biri irəliləmir
2	Niyə yaranır?	Lock-lar mutual exclusion verir, amma sıra qaydası olmadan dövr yarana bilər
3	Hansı problemi həll edir?	— (deadlock özü problemdir)
4	Tarixçə	Coffman və həmmüəllifləri (1971) dörd şərti formallaşdırdı; OS-lar prevention, avoidance və detection üsulları ilə yanaşdı
5	API	`tryLock(timeout)`, lock ordering qaydası, `ThreadMXBean.findDeadlockedThreads()`
6	Runtime	ART thread dump-da "waiting to lock <0x...> held by thread N" zənciri
7	Kernel	Kernel üçün bunlar sadəcə S state-də yatan thread-lərdir; futex deadlock-u aşkarlamır
8	CPU	CPU boşdur: deadlock-da CPU istifadəsi düşür (livelock-dan fərqli)
9	Yaddaş	Sahiblik məlumatı (owner thread id) lock word-da və ya monitor obyektində saxlanılır
10	Qiyməti	Funksiya və ya bütün sistem dayanır; main thread-də ANR deməkdir
11	Edge case-lər	Tək lock ilə deadlock (reentrant olmayan `Mutex`), class init deadlock, main + `runBlocking`, binder ilə process-lər arası deadlock
12	Alternativlər	Lock ordering, message passing, actor, thread confinement, `tryLock` + backoff

Under-the-hood zənciri

Main thread `synchronized(a)` → lock word-da owner T2 → fat lock → `futex_wait` → kernel wait queue (S) → InputDispatcher 5 s sonra ANR qərarı verir → SIGQUIT → Signal Catcher thread dump yazır → trace bugreport-a düşür

Digər ekosistemlərdə

- Go runtime bütün goroutine-lər bloklananda "all goroutines are asleep - deadlock!" xətası verir.
- Database-lər wait-for graph ilə deadlock-u aşkarlayıb bir transaction-u geri qaytarır.

- Linux kernel öz daxili lock-ları üçün lockdep ilə sıra pozuntularını tapır.

Gözünlə gör

- `jstack <pid>` və ya `jcmd <pid> Thread.print` — "Found one Java-level deadlock" bölməsi.
- Android: `adb shell kill -3 <pid>`, sonra `adb bugreport` içindəki ANR trace-ləri; Perfetto-da lock contention slice-ları.
- `cat /proc/<pid>/task/*/wchan` — thread-lərin kernel-də harada gözlədiyi (məsələn, futex).

Laboratoriya

Lab 5.1 — Deadlock lab

- **Qurulum:** T1: lock A → 100 ms yat → lock B; T2: lock B → 100 ms yat → lock A.
- **Ölçülər:** deadlock-a qədər vaxt, thread dump-dakı zəncir, CPU istifadəsi (təxminən 0%).
- **Düzəlt:** əvvəlcə lock ordering, sonra `tryLock(50 ms)` + random backoff variantı.
- **Gözlənilən nəticə:** ordering ilə deadlock yox olur. Backoff-suz `tryLock` variantında livelock görünə bilər — backoff-un niyə random olmalı olduğunu izah et.

Lab 5.2 — Android ANR lab

- **Qurulum:** background thread lock tutub 10 saniyə yatır; main thread düyməyə klikdə eyni lock-u istəyir.
- **Ölçülər:** ANR-ə qədər vaxt, trace-də "held by" zənciri.
- **Gözlənilən nəticə:** input dispatch timeout-dan (5 s) sonra ANR; trace-də main thread Blocked, sahibi isə Sleeping vəziyyətindədir.

Öz layihəndə

Play Console (Android vitals) və ya Crashlytics-də son ANR-lərdən birini götür. Trace-də zənciri çıxar: main thread nəyi gözləyir, onu kim tutur, niyə?

Praktika

- [ ] İki lock ilə deadlock yarat, `jstack` ilə tap, lock ordering ilə düzəlt.
- [ ] Livelock nümunəsi yaz (iki thread `tryLock` ilə həmişə geri çəkilir), sonra random backoff ilə düzəlt.
- [ ] Android: main thread-də background thread-in tutduğu lock-u gözlədərək bilərəkdən ANR yarat; ANR trace-də zənciri tap və izah et.
- [ ] Perfetto-da lock contention slice-ını tap, owner thread-in həmin anda harada olduğunu (Running, Runnable, Sleeping) müəyyən et.

Özünü yoxla

- Coffman şərtlərindən hansını sındırmaq praktikada ən asandır və niyə?
- Priority inversion-u Android trace-də necə tanıyarsan?
- ANR trace-də "held by" zəncirini necə oxuyursan?

- (İ) Production-da deadlock-u necə diaqnoz edərsən? Hansı alətlərlə?

- (İ) Livelock ilə deadlock CPU istifadəsinə görə necə fərqlənir?

Mənbələr

- OSTEP — Common Concurrency Problems.
- *Java Concurrency in Practice* — fəsil 10 (Avoiding Liveness Hazards).
- ANRs (Android Developers).

Mənimsəmə meyarı: sənə təqdim olunan ixtiyari ANR trace-də kök səbəbi 10 dəqiqə ərzində tapırsan.

Faza 6 — Memory model: visibility və ordering (3 həftə) · Advanced

Kodun yazdığın ardıcıllıqla icra olunması illüziyadır: compiler və CPU əməliyyatları yerini dəyişir, bir thread-in yazdığını digəri gec və ya başqa sırada görə bilər. Memory model hansı nəticələrin mümkün olduğunu müəyyən edən müqavilədir.

Problem: Thread A `data = 42; ready = true` yazır, Thread B `if (ready) print(data)` edir. B `ready = true` görüb `data = 0` çap edə bilərmi? x86-da hardware bu sıranı qoruyur, ARM isə qorumur; compiler isə hər ikisində yerini dəyişə bilər.

Niyə vacibdir: ARM (Android cihazları) x86-dan zəif ordering-ə malikdir. x86 emulator-da işləyən kod real cihazda sına bilər. `volatile`-ın və `synchronized`-ın həqiqi mənası yalnız burada açılır.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 2, 5, 6, 7 · [Dərin] 3, 4, 8 · [Ekspert] 9 (lock-free və memory reclamation).

1. **Reordering mənbələri**
   - Compiler: dəyəri register-də saxlamaq, hoisting — `while (!flag)` sonsuz loop-a çevrilir.
   - CPU: out-of-order execution, store buffer, invalidate queue, speculative execution.
2. **Cache coherence**
   - MESI protokolu: Modified, Exclusive, Shared, Invalid state-ləri və keçidləri.
   - Coherence (bir ünvan üçün razılıq) və consistency (ünvanlar arası sıra) fərqi.
   - False sharing: iki müstəqil dəyişən eyni 64-byte cache line-da; həlli padding və `@Contended` (JDK daxili).
3. **Memory consistency modelləri**
   - Sequential consistency (Lamport) — intuitiv, amma bahalı.
   - TSO (x86) — yalnız store → load reordering mümkündür.
   - Weak/relaxed (ARM, POWER) — demək olar hər şey yerini dəyişə bilər.
   - Litmus testlər: Store Buffering (SB), Message Passing (MP), IRIW — hər birini kağızda həll et.
4. **Memory barrier-lər (fence)**
   - LoadLoad, LoadStore, StoreStore, StoreLoad.
   - ARM64: `DMB ISH`, `DSB`, `ISB`; acquire/release instruksiyaları `LDAR`/`STLR`. x86: `MFENCE`, `LOCK` prefiksi.
5. **Acquire/release və happens-before**

    - Release yazışından əvvəlki hər şey, acquire oxunuşundan sonra görünür (synchronizes-with).
    - Happens-before "zamanca əvvəl" demək deyil — görünürlük zəmanətidir.
6. **Java Memory Model (JSR-133, JLS §17)**
    - Program order, synchronization order və happens-before qaydaları: monitor unlock → lock, volatile write → read, `Thread.start`, `Thread.join`, final field freeze, static initializer.
    - JMM-də data race tərifi və DRF-SC zəmanəti: data race yoxdursa, proqram sequentially consistent görünür.
    - `volatile` under the hood: ARM64-də `LDAR`/`STLR`; atomiklik deyil, görünürlük və sıra verir.
    - `final` sahələr və safe publication; out-of-thin-air nəticələr.
7. **Double-checked locking**
    - `volatile`-sız niyə sınıqdır (yarı qurulmuş obyektin görünməsi) və düzgün variant.
    - Kotlin `by lazy` modları: SYNCHRONIZED, PUBLICATION, NONE — decompile edib under the hood-a bax.
8. **C++/Rust `memory_order`**
    - relaxed, acquire, release, acq_rel, seq_cst — dildən asılı olmayaraq ən aydın model. NDK-da birbaşa lazım olacaq (Faza 10).
9. **Lock-free proqramlaşdırma**
    - CAS loop; progress zəmanətləri: wait-free, lock-free, obstruction-free.
    - Treiber stack, Michael-Scott queue.
    - ABA problemi və həlli (`AtomicStampedReference`, tagged pointer).
    - Memory reclamation: hazard pointers, epoch-based; Java-da GC bu problemin böyük hissəsini həll edir.
    - `VarHandle`: `getAcquire`, `setRelease`, `getOpaque`, `compareAndExchange`.

Mental model: Message Passing (MP) litmus testi

Başlanğıc: `data = 0`, `ready = false`. Thread A: `data = 42`, sonra `ready = true`. Thread B: `r1 = ready`, sonra `r2 = data`.

Nəticə (r1, r2)	Sequential consistency	x86 (TSO)	ARM, barrier-siz	`volatile` (release/acquire) ilə
(false, 0)	mümkün	mümkün	mümkün	mümkün
(false, 42)	mümkün	mümkün	mümkün	mümkün
(true, 42)	mümkün	mümkün	mümkün	mümkün
(true, 0)	qeyri-mümkün	qeyri-mümkün	**mümkün**	qeyri-mümkün

x86 sütunu compiler-in yerdəyişmə etmədiyini fərz edir. Java-da `volatile`-sız JIT hər iki platformada (true, 0) nəticəsinə icazə verir.

12 sual: `volatile` (Java Memory Model)

№	Sual	Qısa cavab
1	Nədir?	Sahəyə yazının digər thread-lərə görünməsini və ətrafdakı əməliyyatların sırasını zəmanətləyən modifikator
2	Niyə mövcuddur?	Compiler və CPU reordering-i proqramçının gözləntisini pozur

№	Sual	Qısa cavab
3	Hansı problemi həll edir?	Stale dəyər (`while (!flag)` sonsuz loop), yarı qurulmuş obyektin görünməsi
4	Əvvəl necə idi?	JSR-133-dən (Java 5) əvvəl `volatile` semantikası zəif idi; DCL hətta `volatile` ilə sınıq idi
5	API	`@Volatile`, `AtomicReference`, `VarHandle` (`setRelease`, `getAcquire`)
6	Runtime	JIT/ART compiler `volatile` ətrafında yerdəyişmə etmir və lazımi barrier-ləri qoyur
7	Kernel	— (iştirak etmir; tamamilə user space və CPU səviyyəsidir)
8	CPU	ARM64: oxu `LDAR`, yazı `STLR` (acquire/release); x86: yazıdan sonra full fence (`LOCK`-lu instruksiya və ya `MFENCE`)
9	Yaddaş	Store buffer boşalır, dəyər cache coherence ilə digər core-lara çatır
10	Qiyməti	Adi oxu/yazıdan bahalı, lock-dan ucuz; isti dövrlərdə hiss olunur
11	Edge case-lər	`volatile counter++` atomik deyil; `volatile` massivin elementləri `volatile` deyil; qeyri-`volatile` 64-bit `long`/`double` yazısı JLS-ə görə iki hissəyə bölünə bilər
12	Alternativlər	`synchronized`/lock, `Atomic*`, `VarHandle` ilə zəif rejimlər, immutable obyekt + `final` sahələr

Under-the-hood zənciri

`@Volatile var ready` → bytecode-da `ACC_VOLATILE` bayrağı → ART compiler → ARM64 `STLR` / `LDAR` → store buffer boşalır → MESI invalidation → digər core yeni dəyəri görür

Digər ekosistemlərdə

- C++: `std::atomic<T>` + `memory_order_release`/`acquire`; C++-da `volatile` thread sinxronizasiyası üçün deyil.
- Go memory model: happens-before qaydaları kanallar, `sync` və `sync/atomic` üzərində qurulub.
- .NET: `Volatile.Read`/`Volatile.Write`, `Interlocked`.
- Rust: `Ordering::Relaxed`, `Acquire`, `Release`, `AcqRel`, `SeqCst` — C++ modelinin eynisi.

Gözünlə gör

- jcstress çıxışı: hər nəticənin neçə dəfə müşahidə olunduğu (ACCEPTABLE, FORBIDDEN, ACCEPTABLE_INTERESTING).
- `perf c2c` — false sharing-i tapan Linux aləti.
- `perf stat -e cache-misses,L1-dcache-load-misses` — false sharing benchmark-ında.
- Android: `oatdump` ilə ART-ın yaratdığı maşın kodunda `LDAR`/`STLR` instruksiyaları.

Laboratoriya

Lab 6.1 — Görünürlük (visibility)

- **Hipotez:** `volatile`-sız stop flag JIT optimizasiyasından sonra heç vaxt görünməyə bilər.
- **Qurulum:** worker `while (!stop) count++` işlədir; main 1 saniyə sonra `stop = true` yazır.
- **Variantlar:** plain, `@Volatile`, `AtomicBoolean`.

- **Ölçülər:** worker dayanırmı (10 s timeout), dayanma gecikməsi.
- **Gözlənilən nəticə:** plain variantda JIT-dən sonra tez-tez sonsuz dövr; `volatile` variant dərhal dayanır.

Lab 6.2 — False sharing

- **Qurulum:** 2 thread, hər biri öz `long` counter-ini 100 000 000 dəfə artırır.
- **Variantlar:** A — iki counter yan-yana; B — aralarında 64 byte padding.
- **Ölçülər:** vaxt, `perf c2c` və ya cache miss sayı.
- **Gözlənilən nəticə:** A variantı B-dən dəfələrlə yavaşdır, halbuki paylaşılan dəyişən yoxdur.

Lab 6.3 — MP litmus testi jcstress-də

- **Qurulum:** yuxarıdakı MP testi plain və `volatile` variantlarda; mümkünsə ARM64 maşında.
- **Gözlənilən nəticə:** ARM-da plain variantda (true, 0) az sayıda da olsa görünür; `volatile` ilə heç vaxt.

Öz layihəndə

Kod bazanda `@Volatile` və double-checked locking istifadələrini tap. Hər birinin həqiqətən lazım olduğunu və ya `by lazy`, `AtomicReference` ilə əvəz oluna biləcəyini yoxla.

Praktika

- [ ] `while (!stopRequested)` nümunəsini `volatile`-sız yaz və JIT ilə sonsuz loop-u reproduce et.
- [ ] jcstress ilə SB və MP litmus testlərini yaz; mümkünsə ARM maşında (Apple Silicon, ARM server) x86 ilə müqayisə et.
- [ ] JMH ilə false sharing benchmark-ı: padding-li və padding-siz counter-lar.
- [ ] Sınıq double-checked locking yaz, sonra `volatile` ilə düzəlt; `by lazy`-ni decompile et.
- [ ] Treiber stack və Michael-Scott queue yaz, çox thread-li stress test et.
- [ ] ART `oatdump` və ya disassembly ilə `@Volatile` sahənin ARM64-də hansı instruksiyalara çevrildiyinə bax.

Özünü yoxla

- `volatile` atomikliyi təmin edirmi? `volatile` counter++ niyə yenə səhvdir?
- x86-da işləyən kod ARM-da niyə sına bilər? Konkret nümunə ver.
- Happens-before ilə "zamanca əvvəl" eyni şeydirmi?
- ABA problemi nədir və Java-da GC onun bir hissəsini niyə həll edir?

- (İ) Happens-before nədir? JMM-dən 3 qayda say.
- (İ) Double-checked locking niyə `volatile` olmadan sınıqdır?

Mənbələr

- *Rust Atomics and Locks* (Mara Bos) — memory ordering-in ən aydın izahı; Rust bilmək şərt deyil.
- Preshing on Programming — memory ordering və lock-free seriyası.

- Aleksey Shipilëv, "Java Memory Model Pragmatics" (shipilev.net).
- Doug Lea, The JSR-133 Cookbook for Compiler Writers.
- *Is Parallel Programming Hard, And, If So, What Can You Do About It?* (Paul McKenney) — memory barriers əlavəsi.
- *The Art of Multiprocessor Programming* (Herlihy, Shavit) — lock-free strukturlar.

Mənimsəmə meyarı: MP litmus testinin x86 və ARM-da mümkün nəticələrini yazırsan və JMM-in happens-before qaydaları ilə hər birini əsaslandırırsan.

Faza 7 — java.util.concurrent dərindən (2 həftə) · Advanced

`java.util.concurrent` əvvəlki fazalardakı primitivlərin (CAS, AQS, park/unpark) üzərində qurulmuş hazır alətlərdir. Onları mənbə kodu səviyyəsində bilmək — senior ilə middle arasındakı əsas fərqlərdən biridir.

Problem: Hər şəkil yükləməsi üçün yeni `Thread()` yaradırsan. 200 şəkillik siyahını sürətlə scroll edəndə app yavaşlayır və bəzən OOM verir. Neçə thread lazımdır, işlər harada gözləməlidir, artıq iş gələndə nə etməli?

Niyə vacibdir: WorkManager, Room, OkHttp, Coil, kotlinx.coroutines — hamısı bu alətləri və ya onların ideyalarını daxildə işlədir.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 2, 3, 6, 9 · [Dərin] 4, 5, 7, 8 · [Ekspert] 10 və `ConcurrentHashMap` resize daxili.

1. **Executor framework**
   - `Executor` → `ExecutorService` → `ThreadPoolExecutor`; hər tapşırıq üçün yeni thread yaratmağın niyə pis olduğu.
2. **`ThreadPoolExecutor` under the hood**
   - Parametrlər: `corePoolSize`, `maximumPoolSize`, `keepAliveTime`, `workQueue`, `ThreadFactory`, `RejectedExecutionHandler` (Abort, CallerRuns, Discard, DiscardOldest).
   - `ctl` sahəsi: bir `AtomicInteger`-də runState (yuxarı 3 bit) + workerCount (aşağı 29 bit).
   - `execute()` alqoritmi: core → queue → max → reject. Worker class-ı AQS-dən törəyir.
   - Tələ: `LinkedBlockingQueue` (sınırsız) ilə pool heç vaxt `maximumPoolSize`-a çatmır.
3. **Pool ölçüsü**
   - CPU-bound iş: təxminən core sayı qədər thread.
   - I/O-bound iş (JCIP düsturu): N_cpu — core sayı, U — hədəf CPU istifadəsi (0..1), W/C — gözləmə vaxtının hesablama vaxtına nisbəti.

```
N_{threads} = N_{cpu} \times U_{cpu} \times \left(1 + \frac{W}{C}\right)
```

4. **`ScheduledThreadPoolExecutor` və `ForkJoinPool`**
   - `DelayedWorkQueue` (binary heap).
   - Work-stealing: hər worker-in öz deque-i; öz başından LIFO götürür, başqasının quyruğundan FIFO oğurlayır.

    - Parallel stream-lər və `CompletableFuture`-ın default pool-u.
5. **`Future` və `CompletableFuture`**
    - Kompozisiya: `thenApply`, `thenCompose`, `allOf`; async və qeyri-async metodların hansı thread-də işlədiyi.
6. **Concurrent kolleksiyalar**
    - `ConcurrentHashMap` (Java 8+): boş bin-ə CAS ilə yazış, dolu bin-də `synchronized`, 8-dən çox elementdə TreeBin, resize zamanı ForwardingNode və köməkçi thread-lər, size üçün CounterCell.
    - `CopyOnWriteArrayList` — nə vaxt yaxşıdır (çox oxuma, az yazış, listener siyahıları).
    - BlockingQueue növləri: `ArrayBlockingQueue` (bir lock), `LinkedBlockingQueue` (iki lock: putLock/takeLock), `SynchronousQueue` (tutumsuz), `PriorityBlockingQueue`, `DelayQueue`, `LinkedTransferQueue`.
    - `ConcurrentLinkedQueue` — Michael-Scott alqoritmi (Faza 6).
7. **Synchronizer-lər**
    - `CountDownLatch`, `CyclicBarrier`, `Phaser`, `Semaphore`, `Exchanger` — hər birinin real istifadə ssenarisi.
8. **Atomics**
    - `AtomicInteger`, `AtomicLong`, `AtomicReference`.
    - `LongAdder` və `LongAccumulator`: `Striped64`, Cell massivi, contention zamanı hash ilə yayma; `sum()` dəqiq snapshot deyil.
9. **Thread-safety strategiyaları**
    - Immutability, thread confinement (UI thread, ThreadLocal), stack confinement.
    - Safe publication üsulları: static initializer, `volatile`, `final`, lock ilə qorunan sahə.
10. **Virtual thread-lər (Java 21, Project Loom)**
    - Continuation, carrier thread-lər (ForkJoinPool üzərində), mount/unmount.
    - Pinning: `synchronized` daxilində blocking — JDK 24-də JEP 491 ilə əsasən aradan qaldırıldı.
    - Kotlin coroutine-ləri ilə konseptual müqayisə (stackful və stackless). Android-da əsas alət coroutine-lərdir; virtual thread-ləri JVM tərəfi və intervyu üçün öyrən.

Mental model: `ThreadPoolExecutor.execute()` qərar cədvəli

Pool: core 2, max 4, `ArrayBlockingQueue(2)`, `AbortPolicy`. Bütün tapşırıqlar uzun işləyir.

Tapşırıq	Əvvəl thread	Əvvəl queue	Qərar	Nəticə
T1	0	0	thread < core → yeni worker	1 thread
T2	1	0	thread < core → yeni worker	2 thread
T3	2	0	core doludur → queue-ya	queue 1
T4	2	1	queue-ya	queue 2
T5	2	2	queue doludur, thread < max → yeni worker	3 thread
T6	3	2	yeni worker	4 thread
T7	4	2	queue və max doludur → reject	`RejectedExecutionException`

Sınırsız queue-da T5 addımı heç vaxt gəlmir — `maximumPoolSize` bu səbəbdən işə düşmür.

12 sual: thread pool

№	Sual	Qısa cavab
1	Nədir?	Tapşırıqları təkrar istifadə olunan worker thread-lərdə icra edən mexanizm
2	Niyə mövcuddur?	Thread yaratmaq bahalıdır; sınırsız thread sistemi çökdürür
3	Hansı problemi həll edir?	Yaradılma xərci, resursların sınırsız istifadəsi, yük altında idarəsiz davranış
4	Əvvəl necə idi?	Hər tapşırıq üçün yeni thread (thread-per-task)
5	API	`Executors.newFixedThreadPool`, `ThreadPoolExecutor(...)`, `ForkJoinPool`, `asCoroutineDispatcher()`
6	Runtime	`ctl` (runState + workerCount), AQS üzərində Worker, `getTask()` ilə queue-dan `take()` və ya `poll(keepAlive)`
7	Kernel	Boş worker `LockSupport.park` ilə futex-də yatır; yeni tapşırıq `unpark` → futex wake
8	CPU	İşlər core-lar arasında paylanır; worker eyni core-da qalanda cache-dən faydalanır
9	Yaddaş	Queue-dakı tapşırıqlar heap-dədir; sınırsız queue OOM riskidir; hər worker-in öz stack-i
10	Qiyməti	Queue lock-u üstündə contention, handoff gecikməsi; səhv ölçü — ya ac thread-lər, ya da həddən artıq thread
11	Edge case-lər	Sınırsız queue ilə max-ın işləməməsi, `submit`-də udulan exception, pool daxilindən eyni pool-u gözləmək (thread starvation deadlock), unudulan `shutdown`
12	Alternativlər	`ForkJoinPool` (work-stealing), coroutine dispatcher-ləri, virtual thread-lər

Under-the-hood zənciri

`executor.execute(task)` → `ctl` üzərində CAS ilə workerCount → `workQueue.offer()` (lock + Condition) → `notEmpty.signal()` → `LockSupport.unpark(worker)` → `futex(FUTEX_WAKE)` → scheduler worker-i işə salır → `task.run()`

Digər ekosistemlərdə

- Go: pool yoxdur — goroutine ucuzdur, runtime özü work-stealing scheduler-dir; məhdudiyyət üçün semaphore və ya kanal.
- C#: `ThreadPool` + `Task`; thread sayını hill-climbing alqoritmi ilə avtomatik tənzimləyir.
- Node.js: fayl I/O üçün libuv thread pool (default 4).
- Kotlin: `Dispatchers.Default` və `IO` öz CoroutineScheduler-i üzərində (Faza 9).

Gözünlə gör

- `jstack` və ya `jcmd <pid> Thread.print` — worker-lər `LinkedBlockingQueue.take`-də WAITING vəziyyətində.
- JFR (Java Flight Recorder) — thread park və lock contention event-ləri.
- Android: Perfetto-da pool thread-lərinin Running/Sleeping nisbəti; Android Studio CPU Profiler.

Laboratoriya

Lab 7.1 — Öz thread pool-un

- **Qurulum:** `BlockingQueue` + N worker. Feature-ləri bir-bir əlavə et: `submit()`, `shutdown()`, `awaitTermination()`, bounded queue, rejection, exception idarəsi, `Future` nəticəsi.
- **Ölçü:** 1 000 000 qısa tapşırıqda throughput; `ThreadPoolExecutor` ilə müqayisə.
- **Gözlənilən nəticə:** queue lock-u darboğaz olur. Onu necə azaltmaq olar: batching, hər worker-ə ayrı queue, sonra work-stealing.

Lab 7.2 — Pool ölçüsü

- **Qurulum:** 1000 I/O-bound tapşırıq (50 ms gözləmə + 5 ms CPU); pool ölçüləri 1, 4, 16, 64, 88, 128.
- **Düstur:** 8 core, U = 1, W/C = 50 / 5 = 10 → N = 8 × 1 × (1 + 10) = 88.
- **Ölçülər:** ümumi vaxt, CPU istifadəsi.
- **Gözlənilən nəticə:** vaxt 88-ə qədər azalır, sonra CPU doyduğu üçün sabitləşir.

Lab 7.3 — `AtomicLong` və `LongAdder`: JMH ilə 1, 4, 16 thread-də artım sürəti.

Öz layihəndə

App-ında yaradılan bütün Executor-ları tap (`Executors.`, `ThreadPoolExecutor(`, `newSingleThreadExecutor`). Hər birinin queue növünü və ölçüsünü yaz, sınırsız olanları işarələ.

Praktika

- [ ] Öz sadə thread pool-unu yaz: BlockingQueue + worker thread-lər + `shutdown()` və `awaitTermination()`.
- [ ] `ThreadPoolExecutor.execute()` mənbəsini oxu və axını kağıza çək.
- [ ] `LinkedBlockingQueue`-lu pool-un `maximumPoolSize`-a niyə çatmadığını testlə göstər.
- [ ] JMH: `AtomicLong`, `LongAdder` və `synchronized`-ı 1, 4, 16 thread-də müqayisə et.
- [ ] `ConcurrentHashMap.putVal()` mənbəsini oxu və hər budağını izah et.
- [ ] `ForkJoinPool` ilə paralel merge sort yaz və work-stealing-i log-larla izlə.

Özünü yoxla

- `Executors.newFixedThreadPool`-un sınırsız queue-su production-da hansı problemi yaradır?
- Work-stealing niyə səmərəlidir və deque-in iki ucu niyə fərqli istifadə olunur?
- `LongAdder` nə vaxt `AtomicLong`-dan yaxşıdır, nə vaxt yox?

- (İ) Thread pool-un ölçüsünü necə seçərsən? CPU-bound və I/O-bound fərqi nədir?
- (İ) Work-stealing nədir və niyə deque istifadə edir?

Mənbələr

- *Java Concurrency in Practice* — fəsil 6–8, 11, 15, 16.
- JDK mənbə kodu: `java.util.concurrent` paketi (Doug Lea-nın şərhləri özü dərslikdir).

- JEP 444 (Virtual Threads), JEP 491 (Synchronize Virtual Threads without Pinning).

Mənimsəmə meyarı: istənilən iş yükü üçün pool parametrlərini və queue növünü əsaslanmış şəkildə seçib izah edə bilirsən.

Faza 8 — Android threading modeli under the hood (3 həftə) · Advanced

Android-in bütün UI-ı bir thread-də — main thread-də — Looper/MessageQueue dövrü ilə işləyir; frame-ləri VSYNC idarə edir, IPC-ni Binder thread pool daşıyır. Bu fazada əvvəlki 7 fazanın hamısı Android framework-ün içində bir araya gəlir.

Problem: 120 Hz ekranda hər frame üçün 8.3 ms vaxtın var. Bir `onBindViewHolder` 12 ms çəkir — nə baş verir, istifadəçi nə görür və sistem bunu necə hiss edir?

Niyə vacibdir: jank, ANR, `CalledFromWrongThreadException`, binder-dən gələn gözlənilməz callback thread-i — hər biri bu mexanizmlərin birbaşa nəticəsidir.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 2, 4, 5, 6, 9 · [Dərin] 3, 7, 10, 11, 12 · [Ekspert] 8 (ART thread suspension və GC).

1. **Niyə tək UI thread**
   - UI toolkit thread-safe deyil; hər View-a lock qoymaq həm yavaş, həm deadlock riskidir (input aşağıdan yuxarı, layout yuxarıdan aşağı gedir — əks lock sırası).
   - `ViewRootImpl.checkThread()` → `CalledFromWrongThreadException`.
   - Compose: composition main thread-də işləyir; snapshot state sistemi isə başqa thread-dən yazışa icazə verir (MVCC ideyası).
2. **Looper, MessageQueue, Handler, Message under the hood**
   - `Looper.prepare()` → `ThreadLocal<Looper>`; bir thread-ə bir Looper.
   - `Looper.loop()`: sonsuz dövr — `queue.next()` → `msg.target.dispatchMessage(msg)` → `msg.recycleUnchecked()`.
   - MessageQueue: `when` sahəsinə görə sıralanmış linked list; `enqueueMessage()`.
   - `next()` → `nativePollOnce(ptr, timeout)` → native Looper → `epoll_wait` (oyanma üçün eventfd). Main thread boş olanda CPU yandırmır, kernel-də yatır.
   - Message pool (`obtain`/`recycle`, maksimum 50), sync barrier + async message-lər (Choreographer VSYNC-i prioritetləşdirmək üçün işlədir), `IdleHandler`.
   - Handler: `post`, `postDelayed` (`SystemClock.uptimeMillis` — deep sleep-də dayanır), `sendMessageAtFrontOfQueue`; Looper-sız konstruktor deprecated-dir; inner class + delayed message = leak.
   - `HandlerThread`; `Looper.setMessageLogging()` ilə yavaş message-ları tapmaq.
3. **`ActivityThread.main()`**
   - `Looper.prepareMainLooper()` → `ActivityThread.attach()` → `Looper.loop()`.
   - Lifecycle callback-ləri (onCreate, onResume) binder-dən gələn ClientTransaction kimi main Looper-a post olunur.

4. **Choreographer və VSYNC**
    - Frame büdcəsi: 60 Hz → 16.6 ms, 90 Hz → 11.1 ms, 120 Hz → 8.3 ms.
    - `doFrame`: input → animation → insets animation → traversal (measure, layout, draw) → commit.
    - UI thread display list yazır, RenderThread onu GPU-ya göndərir (HWUI). Jank = frame deadline-ın qaçırılması.
    - Compose frame fazaları: composition → layout → drawing.
5. **ANR under the hood**
    - Hədlər: input dispatch 5 s, BroadcastReceiver 10 s (foreground) / 60 s (background), Service 20 s (foreground) / 200 s (background), `startForegroundService()`-dən sonra `startForeground()` üçün təxminən 10 s.
    - Sistem (InputDispatcher, ActivityManagerService) timeout-u necə aşkarlayır, SIGQUIT ilə trace necə yığılır.
6. **StrictMode**
    - ThreadPolicy (disk read/write, network, custom slow call), VmPolicy (leak-lər); `penaltyLog`, `penaltyDeath`.
7. **Binder və thread-lər**
    - Binder kernel driver; sinxron transaction (çağıran thread bloklanır) və `oneway`.
    - Hər app-da binder thread pool: `Binder:<pid>_N` thread-ləri, default maksimum 15 əlavə thread.
    - `TransactionTooLargeException` (təxminən 1 MB paylaşılan buffer).
    - Main thread-dən sinxron binder çağırışı (PackageManager, ContentResolver) ANR riskidir.
    - AIDL callback-ləri binder thread-də gəlir, main-də yox.
8. **ART runtime thread-ləri və GC**
    - HeapTaskDaemon (concurrent GC), FinalizerDaemon + FinalizerWatchdogDaemon (`finalize()` 10 s-dən çox çəkərsə crash), ReferenceQueueDaemon, Signal Catcher, JIT thread pool, Profile Saver.
    - GC zamanı thread suspension: checkpoint/suspend point, "suspend all" pauzası; ART thread state-ləri (Runnable, Native, Suspended, WaitingForGc və s.).
9. **Tarixi və müasir alətlər**
    - AsyncTask niyə deprecated oldu: leak, konfiqurasiya dəyişməsi, serial executor qarışıqlığı, xəta idarəsi. IntentService, Loader-lər.
    - Service main thread-də işləyir — "background komponent" background thread demək deyil.
    - WorkManager daxili: WorkerWrapper, TaskExecutor, `Worker`-in default Executor-u, `CoroutineWorker`, JobScheduler/AlarmManager seçimi.
10. **Multi-process app-lar və IPC**
    - `android:process`, Messenger, AIDL, ContentProvider; Room-da multi-instance invalidation.

11. **Compose və concurrency under the hood**
    - Snapshot sistemi: MVCC (multi-version concurrency control) ideyası; global snapshot, `Snapshot.withMutableSnapshot`, `apply` zamanı conflict detection.
    - `mutableStateOf`-a istənilən thread-dən yazmaq təhlükəsizdir, amma dəyişiklik recomposition-a yalnız snapshot apply olunandan sonra düşür.

- Recomposer və `AndroidUiDispatcher`: composition main thread-də, Choreographer frame-i ilə sinxron işləyir; `MonotonicFrameClock`, `withFrameNanos`.
- Effect scope-ları: `LaunchedEffect` və `rememberCoroutineScope` coroutine-ləri composition-ın context-ində — main thread-də işləyir. Orada blocking kod frame-i dayandırır.
- `snapshotFlow` (state → Flow), `derivedStateOf` (lazy hesablama, lazımsız recomposition-un qarşısı), `produceState`.
- Praktika: `LaunchedEffect` içində `Thread.sleep(50)` ilə jank yarat, Perfetto-da gör, sonra `withContext(Dispatchers.Default)` ilə düzəlt.

> Diaqram: **Handler.post-dan r.run()-a qədər · 2 thread, 7 addım** — şəkil canlı sənəddədir: https://claude.ai/artifact/3RfryE9eFVgPsAFo18fY3s

Mesaj yoxdursa `next()` `nativePollOnce()` ilə `epoll_wait`-də yatır; başqa thread `enqueueMessage()` edəndə `nativeWake()` eventfd vasitəsilə onu oyadır. Sonsuz `Looper.loop()` CPU yandırmamasının səbəbi budur.

Mental model: bir frame, 120 Hz (8.3 ms büdcə)

Yuxarıdakı diagram işin main thread-ə necə çatdığını göstərir. Bu cədvəl isə çatdıqdan sonra frame-ə nə olduğunu.

Vaxt (ms)	Main thread	RenderThread	Ekran
0.0	VSYNC → `Choreographer#doFrame`: input, animation	—	frame N göstərilir
0.0–12.0	traversal: `onBindViewHolder` 12 ms, measure, layout, draw	gözləyir	—
8.3	hələ traversal-dadır	—	VSYNC: yeni frame hazır deyil, köhnə frame təkrar göstərilir (jank)
12.0	display list hazırdır, RenderThread-ə sync	`DrawFrame`: GPU komandaları	—
16.7	—	—	frame N+1 nəhayət görünür, bir frame gecikmə ilə

12 sual: Looper və MessageQueue

№	Sual	Qısa cavab
1	Nədir?	Thread-də sonsuz mesaj dövrü və vaxta görə sıralanmış mesaj növbəsi
2	Niyə mövcuddur?	UI toolkit-i tək thread-də saxlamaq və başqa thread-lərdən təhlükəsiz iş ötürmək
3	Hansı problemi həll edir?	UI-ı çox thread-dən yeniləməyin race və deadlock riskləri
4	Əvvəl necə idi?	Klassik GUI-lərdə də eyni model var idi (Win32 message loop, X11 event loop); Android onu ümumi Looper/Handler API-si kimi verdi

№	Sual	Qısa cavab
5	API	`Handler.post`, `postDelayed`, `Looper.getMainLooper()`, `HandlerThread`, `Dispatchers.Main`
6	Runtime	`MessageQueue.next()` vaxtı çatan mesajı götürür; sync barrier-ləri və IdleHandler-ləri işləyir
7	Kernel	`nativePollOnce` → `epoll_wait`; `nativeWake` → eventfd-ə yazı; thread S state-də yatır
8	CPU	Boş main thread CPU işlətmir; oyanma wakeup ilə gəlir
9	Yaddaş	Message pool (50 obyekt), Handler-in Activity-yə referansı ilə leak riski
10	Qiyməti	Post-dan icraya gecikmə növbənin dərinliyindən asılıdır; bir uzun mesaj hamını gecikdirir
11	Edge case-lər	Sızan sync barrier (UI donur), deep sleep-də dayanan `postDelayed`, `quit()`-dan sonra post, `sendMessageAtFrontOfQueue` ilə sıranın pozulması
12	Alternativlər	`Dispatchers.Main` və `Main.immediate` (Handler üzərində), Executor-lar, Compose-da `LaunchedEffect`

Under-the-hood zənciri

Background thread `Handler.post(r)` → `enqueueMessage` → `nativeWake` (eventfd) → main thread `epoll_wait`-dən oyanır → `next()` → `dispatchMessage` → `r.run()` → View invalidate → növbəti VSYNC-də `doFrame` → RenderThread → SurfaceFlinger → ekran

Digər ekosistemlərdə

- iOS: main thread + RunLoop, `DispatchQueue.main` — eyni ideya.
- Desktop: Swing EDT, JavaFX Application Thread, Win32 message loop.
- Flutter: UI thread-də Dart event loop + raster thread — main/RenderThread cütlüyünə bənzər.
- Web: brauzerin main thread-i + event loop; ağır iş üçün Web Worker.

Gözünlə gör

- Perfetto: main thread-də `Choreographer#doFrame`, `traversal`; RenderThread-də `DrawFrame`; "Expected/Actual Timeline" track-ları (jank növləri).
- `adb shell dumpsys gfxinfo <paket> framestats` — frame vaxtları.
- `Looper.getMainLooper().setMessageLogging(...)` — hər mesajın başlanğıc və son log-u.
- `adb shell dumpsys activity processes` — process importance; `adb bugreport` — ANR trace-ləri.

Laboratoriya

Lab 8.1 — Jank lab

- **Hipotez:** main thread-də frame büdcəsindən uzun iş birbaşa frame itkisinə çevrilir.
- **Qurulum:** RecyclerView və ya LazyColumn-da hər item üçün 2, 8, 16 ms süni iş (busy loop); 60 və 120 Hz rejimlərində.
- **Ölçülər:** `gfxinfo`-da janky frame faizi, Perfetto-da Actual Timeline.
- **Gözlənilən nəticə:** 120 Hz-də 8 ms artıq sərhəddir; 16 ms hər iki rejimdə jank verir.

Lab 8.2 — Öz Looper-in

- **Qurulum:** Kotlin-də Looper/Handler klonu; 3 producer thread hər biri 10 000 mesaj göndərir, onların 10%-i `postDelayed(100 ms)`.
- **Ölçülər:** sıranın düzgünlüyü (vaxt + FIFO), post-dan icraya gecikmənin p50 və p99-u.
- **Sındır:** gözləməni busy loop ilə əvəz et və CPU istifadəsini müqayisə et — `epoll_wait`-in niyə vacib olduğunu gör.

Öz layihəndə

Debug build-də StrictMode-u `detectAll().penaltyLog()` ilə aktiv et. Main thread-dəki disk və network pozuntularının siyahısını çıxar və hər birini düzəlt.

Praktika

- [ ] Kotlin-də öz Looper/Handler/MessageQueue klonunu yaz: zamana görə sıralanmış queue, delayed message, `quit()`, idle handler.
- [ ] cs.android.com-da `Looper.java`, `MessageQueue.java`, `Handler.java`, `Choreographer.java`, `ActivityThread.java` fayllarını oxu, axını kağıza çək.
- [ ] Main Looper-a `Printer` qoşub 16 ms-dən uzun message-ları log et (BlockCanary ideyası).
- [ ] Perfetto ilə app trace et: `Choreographer#doFrame`, RenderThread `DrawFrame`, binder transaction-ları, main thread state-ləri.
- [ ] StrictMode-u aktiv et, main thread-də disk oxuma ilə tətiklə.
- [ ] Bilərəkdən 3 növ ANR yarat (input, broadcast, service) və trace-ləri müqayisə et.
- [ ] AIDL service yaz, callback-in hansı thread-də gəldiyini log et.

Özünü yoxla

- Main thread `Looper.loop()`-da sonsuz dövrdədirsə, CPU niyə 100% olmur?
- Sync barrier nədir və Choreographer onu niyə işlədir?
- AIDL callback-dən birbaşa View-u yeniləmək niyə crash verir?
- Service-də ağır iş niyə ANR yaradır?

- (İ) Handler, Looper, MessageQueue və Message arasındakı əlaqəni izah et.
- (İ) Jank-ın səbəbini Perfetto-da addım-addım necə tapırsan?

Mənbələr

- cs.android.com — frameworks/base (`android.os`, `android.view`), art/runtime.
- Android Developers: Processes and threads overview, ANRs, "Slow rendering" bələdçisi.
- source.android.com — Binder və IPC bölməsi.
- Perfetto docs — Android app tracing.

Mənimsəmə meyarı: `Handler.post {}` çağırışından lambda-nın main thread-də işləməsinə qədər (epoll daxil) hər addımı və bir frame-in VSYNC-dən ekrana qədər yolunu izah edirsən.

Faza 9 — Kotlin Coroutines under the hood (3 həftə) · Advanced

Coroutine thread deyil: compiler-in suspend funksiyanı state machine-ə çevirməsi və dispatcher-in onu thread-lər üzərində davam etdirməsidir. Thread bloklanmır, yalnız coroutine dayanır (suspend) və sonra istənilən thread-də davam edə bilər.

Problem: 10 000 eyni vaxtlı network sorğusu lazımdır. 10 000 thread təxminən 10 GB virtual stack deməkdir və telefon bunu verməz. Gözləyən hər sorğu thread tutmasın deyə nə etməli?

Niyə vacibdir: müasir Android kodunun concurrency-si demək olar tamamilə coroutines üzərindədir. Under the hood-u bilməyən developer cancellation, exception və dispatcher bug-larını izah edə bilmir.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 1, 3, 6, 10, 11, 12 · [Dərin] 2, 4, 5, 7, 8, 9 · [Ekspert] CoroutineScheduler və `JobSupport` mənbə kodu.

1. **Niyə coroutines**
   - Thread bahalıdır (~1 MB stack, kernel obyekti, context switch); callback hell.
   - Cooperative multitasking; stackless (Kotlin, C#) və stackful (Go, Loom) coroutine-lər; suspend ≠ block.
2. **suspend funksiyalar under the hood**
   - CPS transformasiyası: `suspend fun foo(): T` → `fun foo(cont: Continuation<T>): Any?`.
   - `COROUTINE_SUSPENDED` marker-i — funksiya "dayandım" deyir.
   - State machine: `label` sahəsi, lokal dəyişənlər continuation obyektinin sahələrinə köçürülür (spilling).
   - `ContinuationImpl`, `BaseContinuationImpl.resumeWith` dövrü.
   - `suspendCoroutine` / `suspendCancellableCoroutine` ilə callback API-ni suspend-ə çevirmək.
3. **Builder-lər**
   - `launch`, `async`, `runBlocking` (thread-i bloklayır!), `coroutineScope`, `supervisorScope`, `withContext`.
   - stdlib səviyyəsi: `createCoroutine`, `startCoroutine`, intrinsics.
4. **CoroutineContext**
   - Indexed set: Element, Key, `+` operatoru, CombinedContext.
   - Job, CoroutineDispatcher (ContinuationInterceptor), CoroutineName, CoroutineExceptionHandler.
5. **Dispatcher-lər under the hood**
   - `dispatch()` və `isDispatchNeeded()`; DispatchedContinuation.
   - `Dispatchers.Main` → HandlerContext (`Handler.post`); `Main.immediate` artıq main-dədirsə dispatch etmir.
   - `Dispatchers.Default` → CoroutineScheduler: core sayı qədər thread, hər worker-in local queue-su, work-stealing.
   - `Dispatchers.IO` eyni scheduler-in thread-lərini paylaşır; limit 64 və ya core sayı (hansı böyükdürsə).
   - `limitedParallelism(n)`, `Dispatchers.Unconfined`, `Executor.asCoroutineDispatcher()`.

6. **Structured concurrency**
    - Job iyerarxiyası (parent-child); Job state-ləri: New, Active, Completing, Cancelling, Cancelled, Completed.
    - Cancellation yayılması; cooperative cancellation (`isActive`, `ensureActive()`, `yield()`, suspend nöqtələri).
    - `CancellationException` niyə udulmamalıdır (`catch (e: Exception)` tələsi); `NonCancellable`, `withTimeout`.
    - Exception yayılması: `launch` və `async` fərqi, SupervisorJob, CoroutineExceptionHandler yalnız root coroutine-də işləyir.
7. **Coroutines-də thread-safety**
    - Coroutine fərqli thread-lərdə davam edə bilər — paylaşılan mutable state yenə problemdir.
    - `Mutex` (suspend edir, bloklamır, reentrant deyil), `Semaphore`, atomics, thread confinement (`limitedParallelism(1)`), actor pattern.
    - ThreadLocal və coroutines: `asContextElement()`, ThreadContextElement. `synchronized` blok daxilində suspend etmək olmaz.
8. **Channels**
    - Rendezvous, buffered, unlimited, conflated; `send`/`receive`-in suspend mexanizmi; fan-in, fan-out, `produce`.
9. **Flow under the hood**
    - Cold stream; `collect` zənciri sadəcə iç-içə funksiya çağırışlarıdır.
    - Context preservation qaydası; `flowOn` upstream-i başqa dispatcher-ə köçürür (daxildə channel yaradır).
    - `buffer`, `conflate`, `collectLatest`; `callbackFlow`, `channelFlow`.
    - StateFlow (conflation, equality, thread-safe `value`), SharedFlow (replay, extraBufferCapacity, onBufferOverflow); `stateIn`, `shareIn`, `SharingStarted.WhileSubscribed`.
10. **Android inteqrasiyası**
    - `viewModelScope` (SupervisorJob + `Dispatchers.Main.immediate`), `lifecycleScope`, `repeatOnLifecycle`, `collectAsStateWithLifecycle`.
    - Main-safety konvensiyası; dispatcher-ləri Hilt ilə inject etmək.
11. **Debug**
    - `-Dkotlinx.coroutines.debug` (thread adında coroutine adı), `kotlinx-coroutines-debug` (`DebugProbes.dumpCoroutines()`), Android Studio coroutine debugger.

12. **Populyar kitabxanaların thread modeli** — hər gün işlətdiyin kitabxanaların içində hansı thread-in nə etdiyini bil; mənbə kodunda təsdiqlə.
    - **OkHttp:** `Dispatcher` asinxron çağırışları öz ExecutorService-ində işlədir (default `maxRequests` 64, `maxRequestsPerHost` 5); `ConnectionPool` (default 5 idle connection, 5 dəqiqə keep-alive); interceptor-lar çağıran thread-də işləyir.
    - **Retrofit suspend:** daxildə `Call.enqueue` + `suspendCancellableCoroutine` — main-safe-dir, əlavə `withContext(Dispatchers.IO)` lazım deyil; coroutine ləğv olunanda HTTP çağırışı da ləğv olunur.
    - **Room:** suspend DAO və Flow sorğuları main-safe-dir, sorğuları öz executor-unda (yeni versiyalarda coroutine context-ində) icra edir; `withTransaction` və InvalidationTracker-in Flow-ları necə yenilədiyi.

- **WorkManager:** `Worker.doWork()` Configuration-dakı executor-da, `CoroutineWorker` default olaraq `Dispatchers.Default`-da işləyir.
- **Coil/Glide:** decode və disk cache arxa plan thread-lərində, nəticə main thread-ə; request dedup və lifecycle ilə cancellation.
- **Firebase:** `Task` listener-ləri (`addOnSuccessListener`) default olaraq main thread-də; `FirebaseMessagingService.onMessageReceived` arxa plan thread-ində çağırılır.
- Praktika: hər kitabxananın callback-ində `Thread.currentThread().name` log et və nəticəni cədvələ yaz.

	Platform thread	Kotlin coroutine	Java virtual thread
Kim idarə edir	Kernel scheduler	Dispatcher (kitabxana)	JVM scheduler (ForkJoinPool)
Stack	Ayrı, təxminən 1 MB	Stackless, state heap-də	Stackful, heap-də böyüyən
Dayanma	Thread bloklanır	Yalnız suspend nöqtəsində	Blocking çağırışda unmount
Cancellation	`interrupt()`, əl ilə	Structured, kooperativ	`interrupt()`
Android-da	Hər yerdə	Əsas alət	Əsas alət deyil

Mental model: suspend funksiyanın state machine-i

```
suspend fun loadUser(): User {
    val token = auth()      // suspend nöqtəsi 1
    val user = api(token)   // suspend nöqtəsi 2
    return user
}
```

label	Nə icra olunur	Continuation-da saxlanılan	Qayıdış
0	`auth(cont)` çağırılır	label = 1	`COROUTINE_SUSPENDED` — thread azad olur
1	`resumeWith(token)` ilə davam; `api(token, cont)`	label = 2, `token` sahəsi	`COROUTINE_SUSPENDED`
2	`resumeWith(user)` ilə davam	—	`user` çağırana qaytarılır

Hər `resumeWith` dispatcher-in seçdiyi thread-də baş verir — label 0 ilə label 2 fərqli thread-lərdə işləyə bilər.

12 sual: coroutine və suspend funksiya

№	Sual	Qısa cavab
1	Nədir?	Dayandırılıb sonra davam etdirilə bilən hesablama; thread-i bloklamır
2	Niyə mövcuddur?	Thread-lər bahalıdır, callback-lər oxunmaz olur; ardıcıl görünən asinxron kod
3	Hansı problemi həll edir?	Gözləmə zamanı thread-in boş tutulması, callback hell, çətin ləğvetmə
4	Əvvəl necə idi?	Callback-lər, AsyncTask, RxJava, `Future`/`CompletableFuture`
5	API	`suspend`, `launch`, `async`, `withContext`, `Flow`, `suspendCancellableCoroutine`

№	Sual	Qısa cavab
6	Runtime	CPS + state machine; `BaseContinuationImpl.resumeWith` dövrü; `DispatchedContinuation` thread seçir; Job ağacı
7	Kernel	Kernel coroutine-i görmür; yalnız dispatcher thread-lərini (park/futex, Main üçün epoll)
8	CPU	Suspend və resume adi funksiya çağırışlarıdır; thread dəyişməsə, context switch yoxdur
9	Yaddaş	Hər suspend funksiya üçün heap-də continuation obyekti; lokal dəyişənlər onun sahələridir
10	Qiyməti	Kiçik allocation-lar və dispatch zamanı queue əməliyyatı; thread-dən dəfələrlə ucuz
11	Edge case-lər	Udulan `CancellationException`, main-də `runBlocking`, `Mutex` reentrancy, `GlobalScope` leak, suspend içində blocking kod
12	Alternativlər	Thread + Executor, RxJava, virtual thread-lər (JVM), callback-lər

Under-the-hood zənciri

`viewModelScope.launch { repo.load() }` → `StandaloneCoroutine` Job → `Dispatchers.Main.immediate` → label 0 → Retrofit `suspendCancellableCoroutine` → OkHttp `enqueue` → OkHttp Dispatcher thread → socket I/O → callback → `cont.resume()` → `HandlerContext.dispatch` → `Handler.post` → main Looper → label 1

Digər ekosistemlərdə

- C#: `async/await` də compiler-in qurduğu state machine-dir — Kotlin-ə ən yaxın model.
- JavaScript: `async/await` Promise və event loop-un microtask queue-su üzərindədir.
- Go: goroutine stackful-dur — funksiyalar "rəngli" deyil, `suspend` açar sözü yoxdur.
- Java 21 virtual thread-ləri: stackful; blocking kod olduğu kimi qalır.

Gözünlə gör

- Android Studio: Tools → Kotlin → Show Kotlin Bytecode → Decompile (state machine).
- `-Dkotlinx.coroutines.debug` — thread adında `@coroutine#N`.
- `DebugProbes.dumpCoroutines()` — canlı coroutine-lər və onların suspend nöqtələri.
- Perfetto: `DefaultDispatcher-worker-N` thread-ləri; Android Studio coroutine debugger.

Laboratoriya

Lab 9.1 — 100 000 coroutine və thread

- **Hipotez:** 100 000 coroutine az sayıda thread və kiçik heap ilə işləyir; 100 000 thread isə ya çox resurs tutur, ya da yaradıla bilmir.
- **Qurulum:** hər biri `delay(1000)` və ya `Thread.sleep(1000)` edən 100 000 iş.
- **Ölçülər:** ümumi vaxt, pik RSS, thread sayı (`/proc/<pid>/status`-da Threads).
- **Gözlənilən nəticə:** coroutine variantı təxminən 1 saniyədə bitir; thread variantı ya çox gecikir, ya da OOM verir.

Lab 9.2 — Dispatcher acılığı

- **Qurulum:** `Dispatchers.Default`-da 64 tapşırıq `Thread.sleep(1000)` edir (səhv blocking) və paralel CPU tapşırıqları işləyir; sonra blocking-i `Dispatchers.IO`-ya köçür.
- **Ölçü:** CPU tapşırıqlarının gecikməsi.
- **Gözlənilən nəticə:** birinci variantda Default-un core sayı qədər thread-i yatdığı üçün CPU işləri gözləyir; ikincidə gözləmir.

Lab 9.3 — Cancellation tələsi: `catch (e: Exception)` ilə udulan cancellation-u reproduce et, sonra `ensureActive()` və exception-u yenidən atmaqla düzəlt.

Öz layihəndə

Kod bazanda `GlobalScope`, `runBlocking`, suspend kod içində `catch (e: Exception)` və `Dispatchers.IO`-da CPU-ağır işləri axtar. Hər tapılanı bu fazanın qaydalarına görə qiymətləndir.

Praktika

- [ ] Sadə suspend funksiya yaz, bytecode-a bax (Tools → Kotlin → Show Kotlin Bytecode → Decompile), state machine-i sətir-sətir izah et.
- [ ] Eyni state machine-i əl ilə yaz: `Continuation` interfeysini implement et, `suspend` açar sözü olmadan.
- [ ] `createCoroutine` + öz sadə dispatcher-inlə mini `launch` yaz.
- [ ] 100 000 coroutine və 100 000 thread: yaddaşı və vaxtı ölç.
- [ ] `catch (e: Exception)` ilə `CancellationException`-u udub coroutine-in dayanmadığını gör; `Thread.sleep` və `delay` fərqini göstər.
- [ ] `Mutex` ilə reentrancy deadlock-unu reproduce et.
- [ ] kotlinx.coroutines mənbəsindən `CoroutineScheduler.kt`, `DispatchedContinuation.kt`, `JobSupport.kt` fayllarını oxu.
- [ ] `flowOn`-dan əvvəl və sonra operatorların hansı thread-də işlədiyini log et.

Özünü yoxla

- suspend funksiya thread-i bloklayırmı? `delay()` thread-ə nə edir?
- `Dispatchers.IO` və `Default` eyni thread-ləri necə paylaşır?
- `viewModelScope`-da bir child-ın crash olması digərlərini niyə öldürmür?
- `withContext(Dispatchers.Default)` içində blocking I/O niyə təhlükəlidir?

- (İ) `launch` və `async` exception yayılması baxmından necə fərqlənir?
- (İ) `flowOn` ilə `withContext` arasında fərq nədir?

Mənbələr

- *Kotlin Coroutines: Deep Dive* (Marcin Moskała).
- Kotlin KEEP — coroutines design document (GitHub, Kotlin/KEEP).
- Coroutines guide (kotlinlang.org).
- Roman Elizarov-un KotlinConf çıxışları və Medium məqalələri.

Mənimsəmə meyarı: decompile olunmuş suspend funksiyanı izah edirsən və `launch {}` çağırışından blokun konkret worker thread-də işləməsinə qədər yolu çəkirsən.

Faza 10 — Linux kernel və native səviyyə (2 həftə) · Expert

Bu faza əvvəlki abstraksiyaların kernel mənbə kodunda və NDK-da necə göründüyünü göstərir. Burada artıq kitab yox, real kod və trace sənin əsas mənbəyin olur.

Problem: Native crash tombstone-unda thread-in adı "Thread-7"-dir, stack-də isə JNI çağırışı var. Bu thread kimdir, Java-ya callback edəndə niyə crash olur və kernel həmin anda nə görürdü?

Niyə vacibdir: native crash, JNI thread xətaları, "D state"-də qalan thread-lər, scheduling latency — bunları yalnız bu səviyyədə diaqnoz etmək olur.

Mövzular (under the hood)

Dərinlik teqləri: [Əsas] 3, 7, 8 · [Dərin] 1, 4, 6 · [Ekspert] 2, 5.

1. **Task yaradılması kernel-də**
   - `kernel/fork.c` — `copy_process()`; `task_struct` sahələri, `thread_info`.
   - Kernel stack (ARM64-də 16 KB) və user stack fərqi.
2. **Futex daxili**
   - `kernel/futex/` — hash bucket-lər, wait queue, `FUTEX_WAIT`/`FUTEX_WAKE`.
   - PI-futex (priority inheritance, `rt_mutex` üzərində), robust futex (sahibi ölən lock).
3. **Sleep və wakeup mexanizmi**
   - Wait queue-lar; `TASK_INTERRUPTIBLE` və `TASK_UNINTERRUPTIBLE`.
   - "D state"-dəki task niyə `kill -9` ilə də ölmür.
4. **Signal-lar və thread-lər**
   - Process-directed və thread-directed signal; `pthread_sigmask`; async-signal-safe funksiyalar.
   - ART Signal Catcher (SIGQUIT → thread dump); native crash (SIGSEGV) → debuggerd → tombstone.
5. **Kernel-in öz concurrency-si**
   - Kernel preemption, interrupt context və process context, softirq/tasklet, kthread, workqueue.
   - `spinlock_t`, `mutex` kernel-də; RCU (Read-Copy-Update) ideyası.
6. **IPC mexanizmlərinin müqayisəsi**
   - Pipe, UNIX domain socket, shared memory (memfd; Android-da ashmem → memfd), message queue, signal.
   - Binder: tək kopyalı transfer (mmap), çağıranın kimliyi (calling UID/PID), obyekt referansları və death notification.
7. **NDK threading**
   - `std::thread`, `std::mutex`, `std::condition_variable`, `std::atomic` + `memory_order` (Faza 6).
   - JNI qaydaları: `JNIEnv` thread-local-dır; native thread-dən Java çağırmaq üçün `AttachCurrentThread`/`DetachCurrentThread`; `JavaVM*` saxlamaq; local və global reference.
   - `ALooper` (NDK Looper); `pthread_setname_np` ilə thread-ə ad vermək (trace üçün).
8. **Profilləmə**

- Perfetto dərindən: thread state, scheduling latency, wakeup ("woken by"), CPU tezliyi, ftrace event-ləri, atrace kateqoriyaları.
- Öz slice-ların: `Trace.beginSection` / `androidx.tracing`.
- simpleperf (CPU və off-CPU profil), heapprofd; Linux-da perf, ftrace, bpftrace.

Mental model: futex əsaslı mutex-in 3 vəziyyəti (Drepper)

Lock word: 0 = boş, 1 = tutulub və gözləyən yoxdur, 2 = tutulub və gözləyən ola bilər.

Addım	T1	T2	Lock word	Kernel
1	CAS 0 → 1 uğurlu		1	iştirak etmir (fast path)
2		CAS 0 → 1 uğursuz; lock word-u 2 edir	2	—
3		`futex_wait(addr, 2)`	2	T2 wait queue-da yatır
4	unlock: atomic decrement (2 → 1); köhnə dəyər 1 deyil → 0 yazır		0	—
5	`futex_wake(addr, 1)`		0	T2 run queue-ya qayıdır
6		oyanır, `xchg` ilə 2 yazır, köhnə dəyər 0-dır → lock alındı	2	—

3-cü vəziyyət (2) olmasaydı, hər unlock gözləyən olub-olmamasından asılı olmayaraq lazımsız `futex_wake` syscall-ı edərdi.

12 sual: futex

№	Sual	Qısa cavab
1	Nədir?	"Fast userspace mutex": user-space atomik dəyişən + yalnız lazım olanda kernel-də gözləmə
2	Niyə mövcuddur?	Lock-ların çoxu contention-sız alınır; hər dəfə syscall bahalı olardı
3	Hansı problemi həll edir?	Uncontended lock-ların syscall xərci
4	Əvvəl necə idi?	Hər əməliyyatda kernel-ə girən mexanizmlər (məsələn, System V semaphore-lar); futex Linux 2.6 dövründə gəldi
5	API	Birbaşa `syscall(SYS_futex, ...)`; dolayı `pthread_mutex`, `pthread_cond`, Java `LockSupport.park`
6	Runtime	bionic/glibc pthread və ART monitor-ları contention olanda futex çağırır
7	Kernel	Ünvana görə hash bucket və wait queue; `futex_wait` dəyəri yoxlayıb yatır (lost wakeup-un qarşısı), `futex_wake` N thread oyadır
8	CPU	Fast path-da yalnız CAS və ya `xchg` instruksiyası
9	Yaddaş	32-bit lock word adi yaddaşdadır; process-lər arası shared memory-də də ola bilər

№	Sual	Qısa cavab
10	Qiyməti	Uncontended: bir atomik instruksiya; contended: 2 syscall + context switch-lər
11	Edge case-lər	Spurious wakeup, lost wakeup (dəyər yoxlaması ilə həll olunur), sahibi ölən lock (robust futex), priority inversion (PI-futex)
12	Alternativlər	Spinlock (qısa gözləmə), adaptive mutex (spin + futex), lock-free strukturlar

Under-the-hood zənciri

`pthread_mutex_lock` (bionic) → CAS uğursuz → `syscall(futex, FUTEX_WAIT)` → `svc #0` → kernel futex wait → hash bucket wait queue → `schedule()` → başqa core-da sahib `FUTEX_WAKE` edir → task run queue-ya → yenidən CAS

Digər ekosistemlərdə

- Windows: `WaitOnAddress` / `WakeByAddressSingle` — futex-ə bənzər mexanizm.
- macOS: `os_unfair_lock`.
- Go: runtime Linux-da daxili gözləmə üçün futex işlədir.

Gözünlə gör

- `strace -f -e futex -T` — hər futex çağırışının müddəti.
- `cat /proc/<pid>/task/<tid>/wchan`; root ilə `cat /proc/<pid>/task/<tid>/stack`.
- `perf trace -s`, `bpftrace` ilə `sys_enter_futex` tracepoint-i.
- Android: `simpleperf record -g`, bugreport-dakı tombstone-lar, Perfetto-da "woken by".

Laboratoriya

Lab 10.1 — Öz futex mutex-in

- **Qurulum:** Drepper-in 3 vəziyyətli mutex-i C-də; 8 thread × 1 000 000 lock/unlock, qısa critical section.
- **Variantlar:** öz mutex-in, `pthread_mutex`, sadə spinlock.
- **Ölçülər:** vaxt, futex syscall sayı (`strace -c -f`), context switch sayı.
- **Gözlənilən nəticə:** futex yalnız contention olanda çağırılır; 3 vəziyyətli versiya lazımsız `futex_wake`-ləri aradan qaldırır.

Lab 10.2 — JNI thread lab

- **Qurulum:** native `std::thread` yarat, `AttachCurrentThread` ilə Java callback çağır; sonra `DetachCurrentThread`-i sil.
- **Ölçülər:** davranış, logcat-dakı ART xəbərdarlıqları və crash-lər.
- **Gözlənilən nəticə:** attach-sız çağırış crash verir; detach etmədən çıxan thread ART tərəfindən abort-a səbəb olur.

Öz layihəndə

App-ının thread dump-ında (`kill -3`) native thread-ləri tap və hansı `.so` kitabxanasına aid olduqlarını müəyyən et.

Praktika

- [ ] C-də `futex` syscall-ı ilə öz mutex-ini yaz (Drepper məqaləsindəki 3 versiya).
- [ ] elixir.bootlin.com-da `copy_process()` və futex wait funksiyalarını oxu, axını çək.
- [ ] NDK: native thread yarat, `AttachCurrentThread` ilə Java callback çağır; `Detach` unudulanda nə olduğunu gör.
- [ ] Perfetto-da bir thread-in Runnable → Running gecikməsini və onu kimin oyatdığını tap.
- [ ] Öz kodunu `Trace.beginSection` ilə işarələ və Perfetto-da gör.
- [ ] simpleperf ilə app-in CPU profilini çıxar, ən bahalı thread-i tap.

Özünü yoxla

- Uninterruptible sleep-dəki process-i niyə `kill -9` ilə öldürmək olmur?
- `JNIEnv`-i thread-lər arasında niyə paylaşmaq olmaz?
- Binder digər IPC-lərdən niyə daha səmərəlidir?
- PI-futex hansı problemi həll edir (Faza 5 ilə əlaqələndir)?

- (İ) `pthread_mutex_lock` uncontended halda syscall edirmi? İzah et.
- (İ) Native thread-dən Java metodunu necə çağırırsan və niyə məhz belə?

Mənbələr

- *Linux Kernel Development* (Robert Love) — process management, scheduling, synchronization fəsilləri.
- *Understanding the Linux Kernel* (Bovet, Cesati) — köhnədir, amma konseptual dərin.
- LWN.net — futex, scheduler, RCU məqalələri.
- Android NDK sənədləri və "JNI tips" səhifəsi (developer.android.com).
- Futexes Are Tricky (Ulrich Drepper).

Mənimsəmə meyarı: `pthread_mutex_lock`-dan kernel wait queue-ya və geri, wakeup-dan scheduler-ə qədər yolu mənbə kodu ilə göstərə bilirsən.

Faza 11 — Senior səviyyə: dizayn, test, production, intervyu (2 həftə) · Senior

Senior səviyyə mövzuları bilmək deyil, onları qərar vermək üçün işlətməkdir: hansı aləti seçmək, bug-ı trace-dən tapmaq, komandaya izah etmək. Bu faza əvvəlki 11 fazanın biliklərini praktik qaydalara və layihələrə çevirir.

Problem: Yeni feature: offline-first sinxronizasiya, 5 paralel yükləmə və UI-da canlı irəliləyiş. Neçə coroutine, hansı dispatcher, hansı queue, overload olanda nə? Senior bu suallara kod yazmazdan əvvəl cavab verir.

Mövzular

1. **Thread-safe dizayn**
    - Invariantları müəyyən et; hər paylaşılan state üçün "onu hansı lock və ya thread qoruyur" sənədləşdir.
    - Annotasiyalar: `@GuardedBy`, `@ThreadSafe`; AndroidX `@MainThread`, `@WorkerThread`, `@UiThread`, `@AnyThread`.
    - Immutable domain modelləri; state-i bir yerdə saxlamaq (single source of truth, StateFlow).
2. **Clean Architecture-də concurrency**
    - Data layer main-safe suspend funksiyalar təqdim edir və dispatcher-i özü seçir.
    - Dispatcher-ləri Hilt ilə qualifier-lə (`@IoDispatcher`) inject et — testdə əvəz etmək üçün.
    - Repository-də request dedup (eyni anda gələn eyni sorğunu birləşdirmək), cache-in thread-safety-si.
3. **Alət seçimi** (cədvəl aşağıda).
4. **Concurrent kodun test edilməsi**
    - kotlinx-coroutines-test: `runTest`, `StandardTestDispatcher` və `UnconfinedTestDispatcher`, virtual time, `advanceUntilIdle`, `Dispatchers.setMain`.
    - Turbine (Flow testləri); stress testlər (çox təkrar, çox thread); jcstress; native kod üçün ThreadSanitizer.
5. **Performans qaydaları**
    - Əvvəl ölç, sonra optimallaşdır; lock-free yalnız ölçüdən sonra.
    - Lock-un tutulma vaxtını qısalt, thread sayını məhdudlaşdır.
    - Startup-da main thread-i boşalt (lazy init, App Startup); Baseline Profiles JIT işini azaldır.

Ssenari	Alət	Niyə
UI-dan bir dəfəlik asinxron iş	`viewModelScope.launch` + `withContext(Dispatchers.IO)`	Lifecycle-a bağlı cancellation
Dəyişən data axını UI-a	`StateFlow` + `collectAsStateWithLifecycle`	Conflation, lifecycle-aware
App ölsə də tamamlanmalı iş	WorkManager	Constraint, retry, process death-ə dayanıqlı
CPU-ağır hesablama	`Dispatchers.Default`	Core sayı qədər thread
Bir resursa ardıcıl giriş	`limitedParallelism(1)` və ya `Mutex`	Thread confinement
Callback API-ni sarmaq	`suspendCancellableCoroutine` / `callbackFlow`	Cancellation dəstəyi
Native thread-dən Java-ya	`AttachCurrentThread` + global ref	`JNIEnv` thread-local-dır
Çox oxuma, az yazış	`CopyOnWriteArrayList`, `StateFlow`, immutable snapshot	Oxuma lock-suz

Dizayn sualları (hər qat üçün)

Hər komponenti (UI, ViewModel, repository, network, database, native) bu suallardan keçir:

- Neçə thread və ya coroutine lazımdır və niyə məhz bu qədər?
- Blocking çağırış varmı? Hansı dispatcher-də işləyir?

- Queue varmı? Bounded-dırmı? Dolanda nə olur?
- Overload olsa nə baş verir: rədd, gecikmə, yoxsa OOM?
- Backpressure varmı (Flow `buffer`, `conflate`, semaphore, `limitedParallelism`)?
- Lock contention varmı? Critical section nə qədər uzundur?
- İş CPU-bound-dur, yoxsa I/O-bound?
- Ləğvetmə (cancellation) bütün zəncir boyu işləyirmi?

Kəmiyyət qanunları

Amdahl qanunu: p — paralelləşdirilə bilən hissə, N — core sayı.

```
S(N) = \frac{1}{(1 - p) + \frac{p}{N}}
```

Nümunə: p = 0.9 olanda 8 core ilə S(8) = 1 / (0.1 + 0.1125) ≈ 4.7, sonsuz core ilə isə S = 1 / 0.1 = 10. Yəni: "daha çox thread = daha sürətli" fikri ardıcıl hissədə dərhal sınır.

Little qanunu: L — sistemdə eyni anda olan orta sorğu sayı, λ — gəliş sürəti, W — orta gözləmə (latency).

```
L = \lambda W
```

Nümunə 1 (backend): 1000 sorğu/s × 0.1 s = 100 sorğu eyni anda sistemdədir. Nümunə 2 (Android image loader): saniyədə 40 şəkil sorğusu, orta 250 ms → L = 10. Deməli, paralellik limiti təxminən 10 olmalıdır; latency 500 ms-ə qalxsa, eyni throughput üçün 20 lazım olacaq — ya da queue böyüyəcək.

Production-da tez-tez rast gəlinən bug-lar

Simptom	Səbəb	Düzəliş
ANR, donma	Main thread-də disk, network və ya sinxron binder	Dispatcher-ə köçür, StrictMode
ANR `onPause`/`onStop`-da	`SharedPreferences.apply()` (QueuedWork gözlənilir)	DataStore
Main-də deadlock	`runBlocking` və ya `Future.get()` main thread-də	Suspend API
Coroutine dayanmır	`catch (e: Exception)` CancellationException-u udur	Yenidən at, `ensureActive()`
Default dispatcher-də ləngimə	Blocking I/O `Dispatchers.Default`-da	`Dispatchers.IO`
OOM: pthread_create failed	Sınırsız thread yaradılması	Pool, coroutines
Activity leak	Handler inner class + delayed message, GlobalScope	Lifecycle-aware scope
`ConcurrentModificationException`	Iterasiya zamanı başqa thread-in dəyişməsi	Concurrent kolleksiya, snapshot
İtən event-lər	StateFlow conflation-ı event üçün işlətmək	Event-i state-ə çevir və ya Channel
Mutex deadlock	Reentrant olmayan `Mutex`-i iki dəfə almaq	Lock-u bir səviyyədə al

İntervyu sualları (hər birinə "under the hood" ilə cavab ver)

1. Process və thread arasında fərq nədir — Linux kernel səviyyəsində?
2. Context switch zamanı nə baş verir və niyə bahalıdır?
3. Zygote nədir, app process-i necə yaranır?
4. `volatile` nə verir, nə vermir? ARM-da hansı instruksiyalara çevrilir?
5. Happens-before nədir? 3 qayda say.
6. `synchronized` under the hood necə işləyir?
7. CAS və ABA problemi nədir?
8. Deadlock-un 4 şərti və qarşısını alma yolları?
9. Looper.loop() sonsuz dövrdədirsə, niyə CPU yandırmır?
10. Handler, Looper, MessageQueue, Message — əlaqələrini çək.
11. ANR hansı hallarda və hansı müddətdən sonra baş verir?
12. suspend funksiya bytecode-da nəyə çevrilir?
13. `Dispatchers.IO` və `Default` fərqi; niyə thread-ləri paylaşırlar?
14. Structured concurrency nədir? SupervisorJob nə vaxt lazımdır?
15. `flowOn` və `withContext` fərqi?
16. ThreadPoolExecutor-un `execute()` alqoritmini izah et.
17. ConcurrentHashMap Java 8+-da necə işləyir?
18. Priority inversion nədir, Android-da necə görünür?
19. Binder thread pool nədir? AIDL callback hansı thread-də gəlir?
20. Bir image loader kitabxanasının concurrency dizaynını qur (cache, dedup, cancellation).

Capstone layihələr

- [ ] **Scheduler simulyatoru** (Faza 2) — FIFO, RR, MLFQ, CFS-ə bənzər vruntime; metriklərlə hesabat.
- [ ] **Öz thread pool-un + blocking queue** (Faza 4, 7) — rejection policy, shutdown, stress testlər.
- [ ] **Looper/Handler klonu** (Faza 8) — delayed və async message, sync barrier, idle handler.
- [ ] **Mini coroutine kitabxanası** (Faza 9) — `createCoroutine` üzərində launch, delay, öz dispatcher-in.
- [ ] **Image loader** (Faza 7–9, 11) — memory/disk cache, request dedup, cancellation, thread-safety testləri.
- [ ] **Performans case study** (Faza 2, 5, 8, 10) — öz app-ının Perfetto trace-ində jank-ın kök səbəbini tap, düzəlt, əvvəl/sonra rəqəmləri ilə yaz.

Final mənimsəmə meyarı: intervyuda istənilən concurrency qərarını "niyə" ilə əsaslandıra bilirsən, Perfetto trace-dən jank və ANR-in kök səbəbini tapırsan və komandaya bu mövzuda daxili təqdimat keçə bilirsən.

Layihə nərdivanı

24 laboratoriya bir-birinin üzərində qurulur: hər biri əvvəlkinin kodunu və nəticəsini istifadə edir. Hər birini `labs/NN-ad/` qovluğunda saxla, README-də hipotezi, ölçmə nəticəsini və izahı yaz.

- [ ] 01 — Tək thread-li proqram və `strace` ilə syscall xəritəsi (Faza 0)
- [ ] 02 — fork/exec/wait ilə mini shell (Faza 1)
- [ ] 03 — Copy-on-write ölçümü (Faza 1)
- [ ] 04 — Pipe ping-pong ilə context switch xərci (Faza 2)

- [ ] 05 — İki thread: `gettid()` və `/proc/<pid>/task` (Faza 3)
- [ ] 06 — Paylaşılan counter-də race: 100 × 100 000 (Faza 3)
- [ ] 07 — Mutex ilə counter (Faza 4)
- [ ] 08 — Atomic və `LongAdder` ilə counter (Faza 4, 7)
- [ ] 09 — Producer-consumer, `wait`/`notify` (Faza 4)
- [ ] 10 — Bounded blocking queue, 3 versiya (Faza 4)
- [ ] 11 — Semaphore ilə 10 icazəli connection pool (Faza 4)
- [ ] 12 — Reader-writer lock və writer starvation (Faza 4)
- [ ] 13 — Qəsdən deadlock, sonra lock ordering (Faza 5)
- [ ] 14 — `volatile`-sız stop flag (Faza 6)
- [ ] 15 — False sharing benchmark-ı (Faza 6)
- [ ] 16 — jcstress ilə MP litmus testi (Faza 6)
- [ ] 17 — Öz spinlock-un: TAS → TTAS → backoff (Faza 4, 6)
- [ ] 18 — Treiber lock-free stack (Faza 6)
- [ ] 19 — Michael-Scott lock-free queue (Faza 6)
- [ ] 20 — Öz thread pool-un: submit, shutdown, rejection, Future (Faza 7)
- [ ] 21 — Öz Looper/Handler-in (Faza 8)
- [ ] 22 — Mini coroutine: `createCoroutine` + öz dispatcher-in (Faza 9)
- [ ] 23 — Öz futex mutex-in (Faza 10)
- [ ] 24 — Image loader capstone (Faza 7–9, 11)

Təkamül layihəsi: eyni server, 5 arxitektura

Bir echo server-i 5 dəfə yenidən yaz və hər addımda eyni yük testini et: 100, 1 000 və 10 000 eyni vaxtlı connection. Bu layihə thread, I/O modeli və coroutine arasındakı fərqi rəqəmlərlə göstərir.

Addım	Arxitektura	Nəyi öyrədir	Nəyi ölç	Faza
1	Blocking, bir thread	Bir connection digərlərini bloklayır	Throughput, latency	3
2	Thread-per-connection	Thread-in qiyməti, context switch	Thread sayı, RSS, p99 latency	2, 3
3	Thread pool + bounded queue	Backpressure və rejection	Queue dərinliyi, rədd olunan sorğular	7
4	`epoll` event loop (C)	I/O multiplexing: bir thread, minlərlə connection	CPU, connection sayı	3, 10
5	Kotlin coroutines (Ktor və ya Java NIO `Selector` + coroutine)	Suspend + event loop birləşməsi	Eyni metriklər və kodun sadəliyi	9

Sonda beş addımın nəticələrini bir cədvəldə yan-yana qoy və hər fərqi müvafiq fazanın mexanizmi ilə izah et.

Qarışdırılan anlayışlar

Bu cütlər eyni şey deyil. Hər birini bir cümlə ilə fərqləndirə bilməlisən; intervyuda ən çox bunlar soruşulur.

Anlayışlar	Fərq bir cümlədə	Faza
Program və process	Program diskdəki passiv fayldır, process isə onun işləyən nüsxəsi	1
Process və thread	Process resurs konteyneridir, thread isə icra vahidi	1, 3
Concurrency və parallelism	Concurrency işlərin zamanca üst-üstə düşməsidir, parallelism eyni anda fiziki icradır	3
Blocking və non-blocking	Çağırış nəticə hazır olana qədər thread-i saxlayırmı — bu, birinci ox	3
Sync və async	Nəticə çağırışın özündə qayıdır, yoxsa sonra (callback, future, suspend) gəlir — bu, blocking-dən ayrı ikinci ox	3, 9
Race condition və data race	Biri məntiqi xətadır, digəri sinxronizasiyasız eyni vaxtlı müraciət; biri digəri olmadan ola bilər	4
Atomicity, visibility, ordering	Bölünməzlik, görünürlük və sıra — üç ayrı zəmanətdir; `volatile` yalnız son ikisini verir	4, 6
Mutex və semaphore	Mutex-in sahibi var və bir icazədir; semaphore sahibsiz sayğac olub N icazə verir	4
Mutex və spinlock	Mutex gözləyəni yatırır, spinlock isə dövrə vurdurur	4
Latch və barrier	Latch bir dəfəlik sayğacdır (`CountDownLatch`), barrier təkrar istifadə olunan görüş nöqtəsidir (`CyclicBarrier`)	7
Deadlock, livelock, starvation	Dayanıb gözləmək; işləyib irəliləməmək; heç vaxt növbə almamaq	5
Lock-free və wait-free	Lock-free-də ən azı bir thread irəliləyir, wait-free-də hər thread məhdud addımda bitirir	6
Coherence və consistency	Coherence bir ünvan üzrə razılıqdır, consistency ünvanlar arası sıra qaydalarıdır	6
Thread, coroutine, virtual thread	Kernel task-ı; kitabxana səviyyəsində stackless hesablama; JVM-in idarə etdiyi stackful yüngül thread	3, 9
`launch` və `async`	`launch` nəticəsiz Job, `async` isə `Deferred` nəticə qaytarır; hər ikisində xəta parent-i ləğv edir, root `async`-də isə `await()`-ə qədər görünmür	9
`withContext` və `flowOn`	Biri blokun context-ini dəyişir, digəri yalnız Flow-un upstream hissəsinin	9
`Dispatchers.Main` və `Main.immediate`	`immediate` artıq main thread-dədirsə dispatch etmədən dərhal icra edir	9

Yanlış təsəvvürlər

Aşağıdakı fikirlər hətta təcrübəli developer-lərdə də rast gəlinir. Hər fazanı bitirəndə uyğun sətirləri oxu və "niyə yanlışdır"ı öz sözlərinlə izah et.

Yanlış təsəvvür	Həqiqət	Faza
`fork()` bütün yaddaşı kopyalayır	Copy-on-write: səhifələr yalnız yazış zamanı kopyalanır	1
Singleton bütün app üçün təkdir	Hər process-in öz heap-i var; multi-process app-da hər process-də ayrı singleton	1
Thread çox olduqca proqram sürətli olur	Core sayından çox CPU-bound thread context switch və cache itkisi gətirir; tavanı Amdahl qanunu qoyur	2, 3

Yanlış təsəvvür	Həqiqət	Faza
Java-da RUNNABLE thread CPU-da işləyir	RUNNABLE thread I/O-da bloklanmış və ya Ready növbəsində gözləyir ola bilər	3
Data race yoxdursa, race condition da yoxdur	Check-then-act hər əməliyyat atomik olsa belə səhv nəticə verir	4
`synchronized` həmişə yavaşdır	Contention olmayanda thin lock bir CAS-dır; bahalı olan contention-dır	4
Deadlock yalnız iki lock ilə olur	Main thread-də `Future.get()`, `runBlocking`, reentrant olmayan `Mutex`, class initialization da deadlock yaradır	5, 9
`volatile` dəyişəni thread-safe edir	Yalnız görünürlük və sıra verir; `counter++` yenə atomik deyil	6
Emulator-da işləyirsə, hər cihazda işləyəcək	x86 TSO-dur, ARM isə zəif ordering-ə malikdir; memory model bug-ları real cihazda çıxır	6
Lock-free həmişə lock-dan sürətlidir	Yüksək contention-da CAS retry-ları bahalıdır; qərar yalnız ölçüdən sonra	6
Main thread `Looper.loop()`-da CPU-nu yeyir	Mesaj yoxdursa `epoll_wait`-də kernel-də yatır	8
Service arxa plan thread-idir	Service callback-ləri main thread-də işləyir	8
`postDelayed(r, 1000)` dəqiq 1 saniyə sonra işləyir	`uptimeMillis` deep sleep-də dayanır; queue məşquldursa gecikir	8
Coroutine yüngül thread-dir	Coroutine thread deyil: thread üzərində işləyən, dayandırıla bilən hesablamadır, state-i heap-də saxlanır	9
suspend funksiya avtomatik arxa planda işləyir	`suspend` thread dəyişmir; blocking kod suspend funksiyanın içində də thread-i bloklayır	9
`Thread.sleep` və `delay` eynidir	`sleep` thread-i bloklayır; `delay` coroutine-i dayandırır, thread başqa iş görür	9
`catch (e: Exception)` coroutine-də təhlükəsizdir	CancellationException-u udur, cancellation işləmir	9
Hər şeyi `Dispatchers.IO`-da etmək olar	CPU-ağır iş üçün `Default`; IO-nun paralellik limiti var	9

"Bilirəm" yoxlama siyahısı

Mövzunu bitmiş say yalnız bu 40 sualın hər birinə 5 səviyyədə cavab verə biləndə: sadə izah, texniki tərif, under the hood, real kod və production nümunəsi. Hər sualı 5 səviyyədə cavablayanda qutucuğu işarələ.

- [ ] Process nədir?
- [ ] Thread nədir?
- [ ] Process və thread kernel səviyyəsində necə fərqlənir?
- [ ] Thread-lər nəyi paylaşır, nəyi paylaşmır?
- [ ] Thread stack nədir və overflow necə tutulur?
- [ ] Virtual memory və page fault nədir?
- [ ] Copy-on-write nədir və Zygote onu necə istifadə edir?
- [ ] Context switch nədir və niyə bahalıdır?

- [ ] Scheduler nə edir? Preemption nədir?
- [ ] Concurrency və parallelism fərqi nədir?
- [ ] Race condition və data race fərqi nədir?
- [ ] Critical section nədir?
- [ ] Atomik əməliyyat və CAS nədir?
- [ ] Mutex daxildə necə işləyir?
- [ ] Futex nədir?
- [ ] Semaphore və mutex fərqi nədir?
- [ ] Spinlock nə vaxt doğru seçimdir?
- [ ] Condition variable-da niyə `while` istifadə olunur?
- [ ] Deadlock-un 4 şərti hansılardır?
- [ ] Livelock və starvation nədir?
- [ ] Priority inversion nədir?
- [ ] Cache line və false sharing nədir?
- [ ] Cache coherence və MESI necə işləyir?
- [ ] Memory barrier nədir?
- [ ] Visibility və ordering nədir?
- [ ] Happens-before nədir?
- [ ] `volatile` nə verir, nə vermir?
- [ ] Lock-free və wait-free nə deməkdir?
- [ ] ABA problemi nədir?
- [ ] Thread pool niyə lazımdır və ölçüsü necə seçilir?
- [ ] Work-stealing nədir?
- [ ] Blocking/non-blocking və sync/async necə fərqlənir?
- [ ] `epoll` və event loop necə işləyir?
- [ ] Looper və MessageQueue necə işləyir?
- [ ] ANR necə yaranır və necə diaqnoz olunur?
- [ ] Binder thread pool nədir?
- [ ] Coroutine ilə thread arasında fərq nədir?
- [ ] Suspend funksiya daxildə nəyə çevrilir?
- [ ] `Dispatchers.Default`, `IO` və `Main` necə fərqlənir?
- [ ] Structured concurrency və cancellation necə işləyir?

Həftəlik cədvəl və əsas resurslar

12 faza 26 həftəyə yayılır; ən ağır fazalar (sinxronizasiya, memory model, Android, coroutines) 3 həftə çəkir. Faza 0-a 2 həftə lazım olsa, qalanı bir həftə sürüşür.

> Diaqram: **26 həftəlik yol xəritəsi · 12 faza, 4 blok** — şəkil canlı sənəddədir: https://claude.ai/artifact/3RfryE9eFVgPsAFo18fY3s

Faza 0–5 OS təməlini qurur; 6–7 memory model və JVM alətlərini, 8–9 Android və coroutines-i, 10–11 isə native səviyyəni və senior təcrübəni əhatə edir.

Həftəlik ritm (8–10 saat)

- **4 saat** — oxu və mənbə kodu.
- **4 saat** — praktika: kod yaz, sındır, ölç.
- **1 saat** — izah: öyrəndiyini səsli və ya yazılı izah et, diagram çək.
- **1 saat** — həftənin sonunda "Özünü yoxla" sualları və keçən fazalardan 3 təsadüfi sual (spaced repetition).

Yoxlama nöqtələri

Hər blokun sonunda bir günü yoxlamaya ayır: qeydlərsəz, kitabsız, internetsız. Keçməsən, növbəti bloku bir həftə təxirə sal və zəif fazaları təkrarla.

1. **Yoxlama 1 — 11-ci həftənin sonu (OS təməli)**
   - [ ] 60 dəqiqədə: fork-exec-wait mini shell və bounded blocking queue yaz.
   - [ ] Hazır deadlock-lu proqramın thread dump-ından kök səbəbi tap.
   - [ ] 15 dəqiqəlik izah: "`Thread.start()`-dan kernel-dəki task-a qədər" (səsini yaz, sonra dinlə).
2. **Yoxlama 2 — 16-cı həftənin sonu (memory model və JVM)**
   - [ ] MP litmus testinin x86 və ARM nəticələrini kağızda yaz və jcstress ilə təsdiqlə.
   - [ ] Öz thread pool-unu yaz və `ThreadPoolExecutor`-un `execute()` alqoritmi ilə fərqlərini izah et.
3. **Yoxlama 3 — 22-ci həftənin sonu (Android və Kotlin)**
   - [ ] Looper/Handler klonunu yaz və decompile olunmuş suspend funksiyanı sətir-sətir izah et.
   - [ ] Real app-in Perfetto trace-ində bir jank frame-in səbəbini tap.
4. **Yoxlama 4 — 26-cı həftənin sonu (final)**
   - [ ] Capstone layihələrdən ən azı 3-ü tamamdır.
   - [ ] 20 intervyu sualının hər birinə 2 dəqiqəlik "under the hood" cavabı səsli ver.

Əsas resurslar

Mənbə	Növ	Fazalar	Nə üçün
OSTEP (Arpaci-Dusseau)	Kitab, pulsuz	0–5	OS-in əsası, ən sadə dil
Computer Systems: A Programmer's Perspective	Kitab	0	Hardware, yaddaş, linking
The Linux Programming Interface (Kerrisk)	Kitab	1, 3, 10	Syscall və pthreads üçün reference
Linux Kernel Development (Love)	Kitab	2, 10	Kernel daxili
The Little Book of Semaphores	Kitab, pulsuz	4	Sinxronizasiya tapşırıqları
Java Concurrency in Practice (Goetz)	Kitab	3–5, 7, 11	JVM concurrency-nin əsas kitabı
Rust Atomics and Locks (Bos)	Kitab	4, 6, 10	Atomics və memory ordering
The Art of Multiprocessor Programming (Herlihy, Shavit)	Kitab	6	Lock-free nəzəriyyə
Is Parallel Programming Hard...? (McKenney)	Kitab, pulsuz	6, 10	Memory barriers, RCU

Mənbə	Növ	Fazalar	Nə üçün
Kotlin Coroutines: Deep Dive (Moskała)	Kitab	9	Coroutines daxili
Preshing, Shipilëv, JSR-133 Cookbook	Blog, məqalə	6	Memory model praktikada
cs.android.com	Mənbə kodu	3, 4, 8	Framework və ART
elixir.bootlin.com	Mənbə kodu	10	Linux kernel
kotlinx.coroutines	Mənbə kodu	9	Dispatcher, Job, Flow daxili
Perfetto docs	Sənəd	2, 5, 8, 10, 11	Trace analizi

Kitabların hamısını başdan-sona oxumaq lazım deyil: hər fazada göstərilən fəsillər kifayətdir. JCIP və OSTEP istisnadır — onları tam oxu.

Aralıqlı təkrar cədvəli

Hər faza bitəndən sonra onu 4 dəfə qısa təkrarla: hər dəfə 15–20 dəqiqə, kitabsız. Tarixləri təqvimə dərhal yaz.

Təkrar	Nə vaxt	Nə et
1	1 gün sonra	Fazanın "Özünü yoxla" suallarına yaddaşdan cavab ver
2	3 gün sonra	12 sual cədvəlini boş vərəqdə bərpa et
3	7 gün sonra	Mental model cədvəlini və ya diagramı sıfırdan çək
4	21 gün sonra	Lüğətdən fazanın terminlərini flashcard kimi keç, bir laboratoriyanı yaddaşdan yenidən yaz

Hər sessiyanın ilk 5 dəqiqəsi: əvvəlki fazalardan 3 sual (interleaving).

Mənbə kodu oxuma ardıcıllığı

Bu faylları göstərilən sıra ilə oxu. Hər faylda əvvəlcə public API-ni, sonra bir əsas metodun yolunu izlə.

1. bionic: `libc/bionic/pthread_create.cpp`, `pthread_mutex.cpp` (Faza 3, 4)
2. Linux: `kernel/fork.c` — `copy_process()` (Faza 3, 10)
3. Linux: `kernel/futex/` (Faza 4, 10)
4. ART: `runtime/thread.cc`, `monitor.cc`, `lock_word.h` (Faza 3, 4)
5. JDK: `AbstractQueuedSynchronizer.java`, `ReentrantLock.java` (Faza 4)
6. JDK: `ThreadPoolExecutor.java`, `ForkJoinPool.java` (Faza 7)
7. JDK: `ConcurrentHashMap.java`, `Striped64.java` / `LongAdder.java` (Faza 7)
8. AOSP: `Looper.java`, `MessageQueue.java`, `Handler.java` və native `libutils/Looper.cpp` (Faza 8)
9. AOSP: `Choreographer.java`, `ActivityThread.java` (Faza 8)
10. kotlinx.coroutines: `CoroutineScheduler.kt`, `DispatchedContinuation.kt`, `JobSupport.kt`, `HandlerDispatcher.kt` (Faza 9)

11. Linux: `kernel/sched/fair.c` — EEVDF hissələri (Faza 2, 10)

Bundan sonra hara

Thread və concurrency daha böyük sistem biliyinin köküdür. Bu plan bitəndən sonra təbii davamlar bunlardır; hər biri üçün eyni quruluşda yeni plan qurula bilər.

Növbəti mövzu	Niyə bu plandan sonra	Burada gördüyün anlayışlar
Networking (TCP, HTTP/2 və HTTP/3, TLS, OkHttp daxili)	I/O modelləri və event loop-un real tətbiqidir	`epoll`, backpressure, timeout-lar
Yaddaş və GC (ART GC, heap, leak-lər)	GC thread-ləri və suspend-all pauzaları thread modelinin davamıdır	Thread suspension, false sharing, allocation
Android rendering (HWUI, SurfaceFlinger, Compose runtime)	Choreographer və RenderThread-in dərin davamıdır	VSYNC, frame büdcəsi, Snapshot sistemi
Database daxili (SQLite, Room)	Lock-lar, WAL və transaction izolyasiyası sinxronizasiyanın tətbiqidir	Mutex, reader-writer lock, snapshot
Distributed systems	Sıra, race və razılıq process-lər və maşınlar arası məsələyə çevrilir	Happens-before (Lamport saatları), idempotency, distributed lock-lar
Performance engineering	Ölçmə mədəniyyətini bütün app-a yayır	Perfetto, simpleperf, flame graph, Baseline Profiles

Terminlər lüğəti

Planda işlədilən əsas terminlər əlifba sırası ilə. Bir termini cədvəldəki izahdan daha dəqiq izah edə bilirsənsə, həmin fazanı mənimsəmisən.

Termin	Sadə izah	Faza
ABA problemi	CAS dəyərin A → B → A dəyişdiyini görmür və səhvən uğurlu sayılır	6
Acquire / release	Release-dən əvvəlki yazılar acquire-dən sonra mütləq görünür	6
ANR	Main thread müəyyən müddət cavab verməyəndə sistemin "Application Not Responding" dialoqu	8
Atomik əməliyyat	Yarıda kəsilə bilməyən, digər thread-lərin ya tam, ya heç görmədiyi əməliyyat	4
Binder	Android-in kernel səviyyəli IPC mexanizmi: process-lər arası metod çağırışı	8, 10
Cache coherence (MESI)	Core-ların cache-lərində eyni ünvanın eyni dəyərdə qalmasını təmin edən protokol	6
Cache line	CPU ilə RAM arasında köçürülən ən kiçik blok, adətən 64 byte	0, 6
CAS (compare-and-swap)	"Dəyər hələ A-dırsa, B yaz" — atomik instruksiya, lock-free alqoritmlərin əsası	4
Choreographer	VSYNC siqnalına görə frame işini (input, animation, draw) planlaşdıran Android class-ı	8
Concurrency	Bir neçə işin zamanca üst-üstə düşməsi; bir core-da da mümkündür	3
Condition variable	Şərt ödənənə qədər thread-i yatıran və xəbər gələndə oyadan primitiv	4

Termin	Sadə izah	Faza
Context switch	CPU-nun bir task-dan digərinə keçməsi: register-lərin saxlanması və bərpası	2
Continuation	Coroutine-in "qalan hissəsi": harada dayandığını və lokal dəyişənləri saxlayan obyekt	9
Copy-on-write (COW)	Kopya yalnız kimsə yazanda yaradılır; fork-un və Zygote-un ucuzluğunun səbəbi	1
Coroutine	Thread-i bloklamadan dayandırılıb davam etdirilə bilən hesablama	9
Critical section	Eyni anda yalnız bir thread-in icra etməli olduğu kod hissəsi	4
Data race	İki thread eyni yaddaşa sinxronizasiyasız müraciət edir, ən azı biri yazır	4, 6
Deadlock	Thread-lər bir-birinin tutduğu resursu gözləyir, heç biri irəliləmir	5
Dispatcher	Coroutine-in hansı thread və ya pool-da davam edəcəyini seçən komponent	9
epoll	Linux-da çoxlu fd-ni bir thread-dən gözləməyə imkan verən mexanizm; Looper-in əsası	3, 8
False sharing	Müstəqil dəyişənlərin eyni cache line-da olması səbəbindən yaranan yavaşlama	6
Futex	Contention yoxdursa syscall-sız, varsa kernel-də gözləyən lock primitivi	4, 10
Happens-before	Bir əməliyyatın nəticəsinin digərinə görünəcəyi zəmanəti (JMM)	6
Heap	Dinamik ayrılan yaddaş; process-in bütün thread-ləri paylaşır	1, 3
Interrupt	CPU-nu cari işdən ayırıb kernel handler-ə keçirən siqnal	0
Kernel	Hardware-i idarə edən və process-lərə xidmət göstərən OS nüvəsi	0
Livelock	Thread-lər işləyir, bir-birinə yol verir, amma irəliləyiş yoxdur	5
Lock-free	Ən azı bir thread-in həmişə irəliləməsinə zəmanət verən, lock-suz alqoritm	6
Looper	Thread-də sonsuz mesaj dövrü işlədən Android class-ı	8
Memory barrier (fence)	Müəyyən yaddaş əməliyyatlarının yerini dəyişməyi qadağan edən instruksiya	6
Memory model	Paralel proqramda hansı yaddaş nəticələrinin mümkün olduğunu təyin edən qaydalar	6
MessageQueue	Looper-in emal etdiyi, vaxta görə sıralanmış mesaj növbəsi	8
MMU / TLB	Virtual ünvanı fiziki ünvana çevirən hardware və onun keşi	1
Monitor	Lock + condition variable; Java-da hər obyektin bir monitoru var	4
Mutex	Eyni anda yalnız bir sahibi olan, gözləyəni yatıran lock	4
Page / page fault	Virtual yaddaşın vahidi (4 və ya 16 KB) və ona ilk müraciətdə yaranan kernel hadisəsi	1
Parallelism	Eyni anda bir neçə core-da fiziki icra	3
PCB (`task_struct`)	Kernel-in hər task haqqında saxladığı struktur	1
Preemption	Kernel-in işləyən task-ı onun razılığı olmadan dayandırması	2
Priority inversion	Yüksək prioritetli task aşağı prioritetlinin tutduğu lock-u gözləyir	5
Process	Öz virtual yaddaşı və resursları olan, icra olunan proqram	1
Race condition	Nəticənin thread-lərin icra sırasından asılı olduğu məntiqi xəta	4
Scheduler	Hansı task-ın hansı core-da nə qədər işləyəcəyinə qərar verən kernel hissəsi	2

Termin	Sadə izah	Faza
Semaphore	Sayacı olan sinxronizasiya primitivi; eyni anda N thread-ə icazə verir	4
Spinlock	Gözləyən thread-in yatmadan dövrə vurduğu lock	4
Stack	Funksiya çağırışlarını və lokal dəyişənləri saxlayan, hər thread-in özünə məxsus yaddaşı	0, 3
Starvation	Task-ın uzun müddət resurs ala bilməməsi	5
Structured concurrency	Coroutine-lərin valideyn-uşaq ağacında yaşadığı, ləğv və xətaların ağac üzrə yayıldığı model	9
suspend funksiya	Dayandırıla bilən funksiya; compiler onu state machine-ə çevirir	9
System call	User proqramın kernel-dən xidmət istəməsi (`read`, `write`, `clone`)	0
Thread	Process daxilində öz stack-i və register-ləri olan icra axını	3
Thread pool	Tapşırıqları təkrar istifadə olunan thread-lərdə icra edən mexanizm	7
TLS (`ThreadLocal`)	Hər thread-ə məxsus ayrıca dəyər	3
Virtual memory	Hər process-ə fiziki yaddaşdan asılı olmayan ayrıca ünvan sahəsi	1
`volatile`	Java/Kotlin-də görünürlük və sıra zəmanəti verən sahə modifikatoru	6
VSYNC	Ekranın yenilənmə siqnalı; frame büdcəsini təyin edir	8
Work-stealing	Boş worker-in başqasının növbəsindən iş götürməsi	7, 9
Zombie	Bitmiş, amma parent `wait()` etmədiyi üçün qeydi hələ silinməyən process	1
Zygote	Android-də app process-lərini fork edən ilkin process	1