Faza 0 — Kompüterin təməli: Öz-özünə öyrənmə sənədi

> **Snapshot:** bu fayl Claude Docs-dakı canlı sənədin 2026-11-21 tarixli nüsxəsidir. Canlı versiya (yeni fəsillər orada əvvəl çıxır): https://claude.ai/artifact/B3NWt64tSJnbGDa3poTKaG Cavablar: study-guide-answer-key.md. Kod: ch00-foundations, ch01-cpu, ch02-memory, ch03-build.
> **Şəkillər:** canlı sənəddə 306 şəkil var — Fəsil 0–3-də hər başlıqda ən azı 3 şəkil, hər fəslin sonunda xülasə xəritəsi (mind map) və onun boş nüsxəsi. Burada 11-i PNG kimi göstərilir; qalanlarının yerində 🖼️ işarəsi, adı və altındakı "Burada nəyə bax" izahı qalır. Şəkilləri görmək üçün canlı sənədi aç.

Nov 19, 2026 · @Atesh

Bu sənədi necə oxumalı

Bu sənəd planın Faza 0-ını öyrədir: kompüterin içində nə baş verdiyini, bir bitdən başlayıb kernel-in interrupt-a cavab verməsinə qədər. Heç nə bilmədən başlaya bilərsən: hər söz istifadə olunmazdan əvvəl izah olunur.

Vəd və dürüstlük

- Sənədi sona qədər işləsən, Faza 0-ın hər anlayışını 5 səviyyədə izah edə biləcəksən: sadə izah, texniki tərif, under the hood, real kod və production nümunəsi.
- İntervyuer bir sualın ardınca 4–5 dərinləşən sual versə, dayanmadan cavab verə biləcəksən.
- Bu sənəd 10 illik təcrübəni əvəz etmir. Senior intervyusundan keçmək üçün planın digər fazaları, real layihədə praktika və səsli cavab məşqi də lazımdır. Sondakı mock interview sonuncunu məşq etdirir.
- Bütün komanda çıxışları real maşında alınıb: Linux 6.18, x86-64 Intel Xeon 2.1 GHz (virtual maşın), gcc 13.3, clang 18.1, Python 3.13, OpenJDK 21, Kotlin 2.1.0. Sənin maşınında rəqəmlər fərqli ola bilər, prinsip isə eyni qalır.
- Kotlin nümunələri: hər kod nümunəsinin yanında Kotlin versiyası var və kotlinc 2.1.0 ilə işlədilib. C, assembly, shell və Java aşağı qatları göstərdiyi yerdə qalır, Kotlin isə eyni fikri sənin öz stack-ində göstərir.

Nə qədər vaxt

Təxminən 35 saat: həftədə 8–10 saatla 4 həftə. Plandakı "1–2 həftə" bu mövzuları qismən bilən oxucu üçündür; sıfırdan başlayırsansa, tələsmə.

Nə lazımdır

- Linux terminalı: Ubuntu, Windows-da WSL2 və ya macOS terminalı (macOS-da fərqli olan komandalar qeyd olunub).

- `gcc` və ya `clang`, `python3`, `java`. Bunların nə olduğunu hələ bilmirsənsə, narahat olma: Fəsil 3-ə qədər hər komandanı olduğu kimi kopyala və çıxışı sənəddəkilə müqayisə et.
- Kağız, qələm və bir qeyd dəftəri.

Oxu vərdişləri

Bunlar sənədin ən vacib hissəsidir. Onlarsız sənəd sadəcə oxunan mətn olur, onlarla isə bilik olur.

1. Hər fəslin "Əvvəlcə təxmin et" suallarına oxumazdan əvvəl kağızda cavab yaz. Səhv təxmin normaldır: cəhd etmək yadda saxlamanı gücləndirir.
2. "Dayan və düşün" sualını görəndə dayan, cavabı yaz, sonra davam et.
3. Hər eksperimenti öz kompüterində işlət. Oxumaq başqa şeydir, gözlə görmək başqa.
4. Cavab açarını (Cavab açarı) yalnız öz cavabını yazandan sonra aç.
5. Hər fəslin sonunda onu bir dostuna və ya səsli olaraq özünə izah et.
6. Bitirdiyin fəsli 1, 3, 7 və 21 gün sonra "Keçid qapısı" sualları ilə təkrarla.

Hər fəsil necə qurulub

Hər mərkəzi anlayış dörd pillə ilə öyrədilir. Hansı pillədə dayansan da, bildiyin şey doğru qalır.

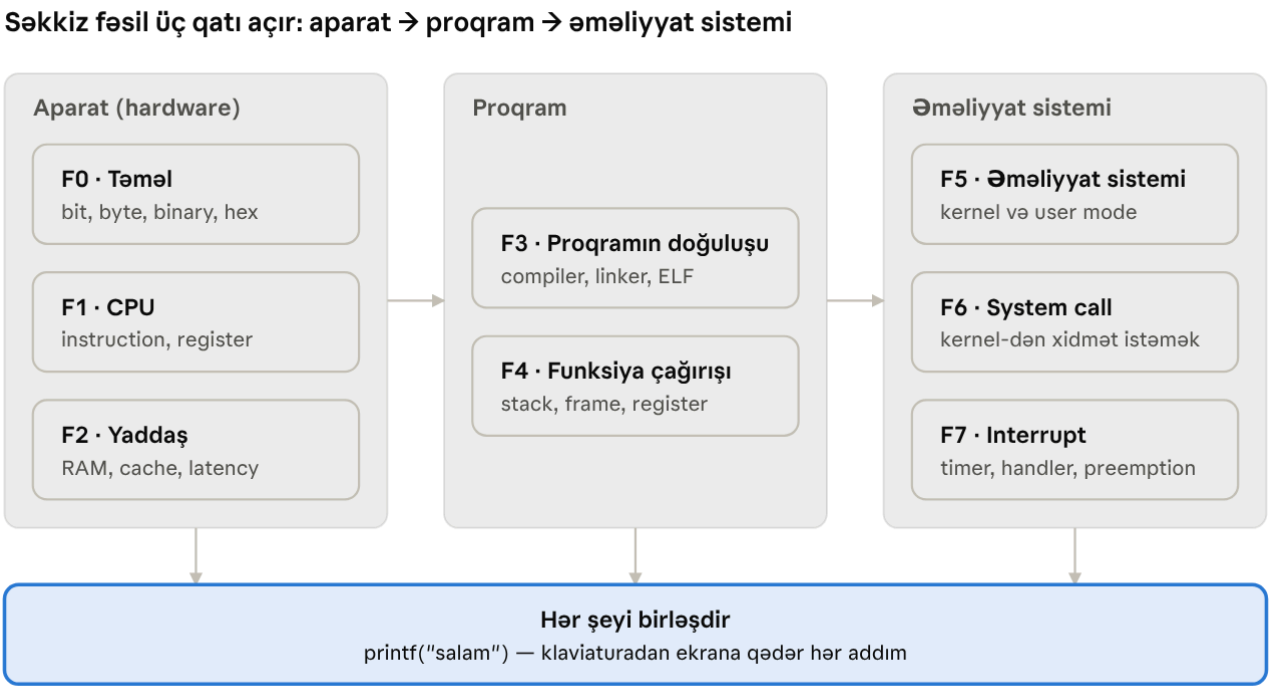
1. **Uşaq səviyyəsi** — gündəlik həyatdan analogiya və onun harada sındığı.
2. **Mühəndis səviyyəsi** — dəqiq tərif və mexanizm, addım-addım.
3. **Senior səviyyəsi** — ən alt qat, rəqəmlər, tarix, trade-off-lar və Android-dakı qarşılığı.
4. **İntervyu səviyyəsi** — bunu intervyuda necə danışmalı və ardınca gələn suallara necə cavab verməli.

Fəsil xəritəsi

Fəsil	Nə öyrənəcəksən	Təxmini vaxt	Plandakı mövzu
0 · Təməl	Bit, byte, binary, hex; mətn və ədədlər yaddaşda necə saxlanılır; fayl və proqram nədir	4 saat	Planın fərz etdiyi təməl
1 · CPU	Instruction, register, clock, core; CPU bir proqramı necə icra edir	4 saat	0.1
2 · Yaddaş	RAM, ünvan, cache, locality, latency rəqəmləri	4 saat	0.2
3 · Proqramın doğuluşu	Mətndən işləyən proqrama: compiler, linker, ELF, loader	4 saat	0.3
4 · Funksiya çağırışı	Stack, frame, return address, calling convention	4 saat	0.4
5 · Əməliyyat sistemi	Kernel, user mode, privilege səviyyələri	3 saat	0.5
6 · System call	Proqram kernel-dən necə xidmət istəyir və bu niyə bahalıdır	4 saat	0.6
7 · Interrupt	Timer, handler, preemption-un əsası	3 saat	0.7
Birləşdir, mock interview, yoxlama	Bütün zənciri bir hekayə kimi danışmaq, intervyu məşqi	5 saat	Mənimsəmə meyarı

Bilik xəritəsi

Fəsillər aşağıdan yuxarı qurulur: hər fəsil yalnız əvvəlki fəsillərdə öyrəndiyin sözlərdən istifadə edir. Ona görə sıra ilə oxu.



Əvvəlcə kompüterin "bədənini" öyrənirsən (aparat), sonra onun icra etdiyi proqramın necə yarandığını, sonda isə hər şeyi idarə edən əməliyyat sistemini. Son bölmədə üç qat bir hekayədə birləşir.

Fəsil 0 — Təməl: kompüter informasiyanı necə saxlayır

Bu fəsildən sonra bacaracaqsan

- Kompüterin niyə yalnız 0 və 1 ilə işlədiyini izah etmək.
- Bit və byte-ı tanımaq, 8 bitin neçə fərqli vəziyyət verdiyini hesablamaq.
- İstənilən kiçik ədədi binary və hex yazısına çevirmək və geri.
- Hərfin, rəqəmin və emojinin yaddaşda hansı byte-lar olduğunu göstərmək.
- Mənfi ədədlərin necə saxlandığını və overflow-un niyə baş verdiyini izah etmək.
- Fayl və proqramın əslində nə olduğunu bir cümlə ilə demək.

Əvvəlcə təxmin et

Kağızda cavab yaz, sonra davam et. Cavablar fəslin içində və Cavab açarındadır.

1. **T0.1** Kompüter "A" hərfini yaddaşda nə şəkildə saxlayır?
2. **T0.2** 8 işıq açarı ilə neçə fərqli "açıq/qapalı" kombinasiyası qurmaq olar?
3. **T0.3** Ən böyük `Int` ədədinə (2 147 483 647) 1 əlavə etsən, nə alarsan?

Sirr

Telefonundakı şəkil, mahnı, mesaj və oyun — hamısı eyni şeydən ibarətdir: milyardlarca 0 və 1-dən. Bəs necə olur ki, eyni 0 və 1-lər bir yerdə səs, başqa yerdə rəng, başqa yerdə proqram olur? Bu fəslin sonunda cavabı biləcəksən.

> 🖼️ **Şəkil 1:** Sirr · telefondakı hər şey bitlərdən ibarətdir

Burada nəyə bax: dörd fərqli şey eyni xammala gedir. Fəslin sonunda eyni bitlərin necə fərqli məna aldığını biləcəksən.

Yeni sözlər

informasiya · tranzistor · bit · byte · decimal · binary · hexadecimal (hex) · kilobyte və kibibyte · kodlaşdırma (encoding) · ASCII · Unicode · UTF-8 · unsigned və signed · two's complement · overflow · endianness · fayl · proqram

Uşaq səviyyəsi: işıq açarları və kod kitabı

Təsəvvür et ki, divarda bir sıra işıq açarı var. Hər açar ya açıqdır, ya qapalı. Üçüncü vəziyyət yoxdur.

Sən və dostun razılaşırsınız: "Əgər 2-ci və 8-ci açar açıqdırsa, bu 'A' hərfidir". Bu razılaşmaya **kod kitabı** deyək.

Dostun otağa girib açarlara baxır və mesajı oxuyur. Açarların özü heç nə bilmir. Mənanı kod kitabı verir.

Kompüter də tam belə işləyir: milyardlarca kiçik "açar" və onların necə oxunacağını deyən razılaşmalar.

> 🖼️ **Şəkil 2:** Uşaq səviyyəsi · 8 açar və kod kitabı

Burada nəyə bax: yalnız 2-ci və 8-ci açar açıqdır. Kod kitabı bu nümunəni 'A' kimi oxuyur.

Analogiyada	Kompüterdə
Bir işıq açarı: açıq və ya qapalı	Bir **bit**: 1 və ya 0
8 açardan ibarət bir sıra	Bir **byte**
Divardakı açar sırasının nömrəsi	Yaddaşda byte-ın **ünvanı** (Fəsil 2-də)
Dostlar arasındakı kod kitabı	Kodlaşdırma qaydası: ASCII, UTF-8, two's complement
Eyni açarlar, başqa kod kitabı — başqa mesaj	Eyni byte-lar, başqa proqram — başqa məna

Analogiya harada sınır: real bit işıq açarı deyil. O, tranzistordakı elektrik gərginliyi, yaddaş hüceyrəsindəki elektrik yükü və ya diskdəki maqnit sahəsidir. Bir də: açarı əllə çevirirsən, kompüter isə saniyədə milyardlarla biti dəyişir.

> 🖼️ **Şəkil 3:** Yazan və oxuyan · eyni kod kitabı lazımdır

Burada nəyə bax: kitab eyni olanda mesaj çatır, fərqli olanda məna dəyişir. Bu, "kodlaşdırma" fikrinin üzüdür.

> 🖼️ **Şəkil 4:** Real bit · tranzistor, kondensator, flash

Burada nəyə bax: üç fərqli fizika, eyni fikir: iki vəziyyət. Bitin harada saxlanması dəyişir, oxunuş qaydası dəyişmir.

Mühəndis səviyyəsi

Niyə yalnız 0 və 1?

Kompüter elektriklə işləyir. Elektrik dövrəsində iki vəziyyəti ayırmaq çox asandır: gərginlik var, ya da yoxdur.

Onluq sistemdəki kimi 10 fərqli səviyyəni ayırmaq çətin olardı. Kiçik bir elektrik "səs-küyü" 6-nı 7-yə çevirərdi. İki səviyyədə isə səhv etmək çox çətindir.

Bu iki vəziyyəti saxlayan və dəyişən kiçik hissəyə **tranzistor** deyilir. Tranzistor elektriklə idarə olunan kiçik açardır. Müasir telefon prosessorunda onlardan milyardlarla var.

> 🖼️ **Şəkil 5:** Niyə ikilik · lampa və 10 pilləli parlaqlıq

Burada nəyə bax: lampanın iki halını uzaqdan da ayırmaq olur, 10 pillənin qonşularını isə çətin.

> 🖼️ **Şəkil 6:** Səs-küy · 2 səviyyə və 10 səviyyə

Burada nəyə bax: eyni səs-küy geniş "1" zolağından çıxmır, dar zolaqlarda isə rəqəmi dəyişir.

> 🖼️ **Şəkil 7:** Tranzistor · elektriklə idarə olunan açar

Burada nəyə bax: gate-ə verilən gərginlik cərəyanı açıb-bağlayır. Bu, bitin ən aşağı fiziki qədir.

Bit və byte

Bit (ingiliscə *binary digit* — "ikilik rəqəm") informasiyanın ən kiçik vahididir: 0 və ya 1. **İnformasiya** isə bir şey haqqında bildiyimiz hər şeydir: ədəd, hərf, rəng, səs.

Bir bit yalnız iki şey deyə bilər: hə və ya yox. Daha çox şey demək üçün bitləri birləşdiririk:

- 1 bit → 2 vəziyyət: 0, 1
- 2 bit → 4 vəziyyət: 00, 01, 10, 11
- 3 bit → 8 vəziyyət: 000, 001, 010, 011, 100, 101, 110, 111

Hər yeni bit vəziyyət sayını iki dəfə artırır. n bit ilə vəziyyət sayı N belə hesablanır:

```
N = 2^{n}
```

Byte 8 bitdən ibarət qrupdur. Müasir kompüterlərdə yaddaşda ünvanı olan ən kiçik hissə byte-dır. Bir byte $2^8 = 256$ fərqli vəziyyət saxlayır: 0-dan 255-ə qədər.

Bit sayı	Vəziyyət sayı	Harada rast gələcəksən
8	256	Bir byte, şəkildə bir rəng kanalı
16	65 536	Java və Kotlin-də `Char`
32	4 294 967 296	Kotlin `Int`
64	təxminən 1.8×10^{19}	Kotlin `Long`, müasir CPU-nun register-i (Fəsil 1-də)

> 🖼️ **Şəkil 8:** Bit · hər yeni lampa halları ikiyə vurur

Burada nəyə bax: lampa sayı bir artınca hallar iki dəfə çoxalır.

> 🖼️ **Şəkil 9:** Bit ağacı · 3 bit = 8 hal

Burada nəyə bax: hər səviyyədə hər budaq ikiyə bölünür; 3 səviyyədən sonra 8 yarpaq qalır. Bu, $N = 2^n$ düsturunun şəklidir.

> 🖼️ **Şəkil 10:** Byte və Kotlin tipləri · 1, 2, 4, 8 byte

Burada nəyə bax: Kotlin-in tam ədəd tipləri 1, 2, 4 və 8 byte-lıq qutulardır. Qutu böyüdükcə sığan ədədlər də çoxalır.

Dayan və düşün D0.1: 10 bit ilə neçə fərqli vəziyyət var?

Ədədləri bitlərlə yazmaq: binary

Gündəlik həyatda **decimal** (onluq) sistemdən istifadə edirik. Orada 10 rəqəm var (0–9) və hər mövqe sağındakından 10 dəfə böyükdür. 372 əslində 3 × 100 + 7 × 10 + 2 × 1 deməkdir.

Binary (ikilik sistem) eyni ideyadır. Sadəcə cəmi 2 rəqəm var (0 və 1), hər mövqe isə sağındakından 2 dəfə böyükdür: 1, 2, 4, 8, 16, 32, 64, 128 ...

Nümunə: binary 1101 neçədir?

Mövqenin qiyməti	**8**	**4**	**2**	**1**
Bit	1	1	0	1
Töhfə	8	4	0	1

Cəm: 8 + 4 + 0 + 1 = 13. Deməli, binary 1101 = decimal 13.

Tərsinə çevirmək üçün ədədi təkrar-təkrar 2-yə böl və qalıqları yaz. Sonra qalıqları aşağıdan yuxarı oxu.

Addım	**Ədəd**	**2-yə böləndə**	**Qalıq**
1	13	6	1
2	6	3	0
3	3	1	1
4	1	0	1

Qalıqlar aşağıdan yuxarı: 1, 1, 0, 1 → 1101. Bu üsul niyə işləyir? Hər bölmədə ən sağdakı bit qalıq kimi "düşür", ədəd isə bir mövqe sağa sürüşür.

İkinci nümunə: 200 = 128 + 64 + 8 → 11001000. Proqramçılar binary ədədi `0b` ilə başlayır: `0b1101` = 13.

> 🖼️ **Şəkil 11:** Binary hekayəsi · 8, 4, 2, 1 manatlıq əskinaslar

Burada nəyə bax: 13 manat 8, 4 və 1 manatlıqlarla yığılır, 2 manatlıq lazım olmur. Seçimi 1 və 0 ilə yazanda 1101 alınır. Yuxarıdakı iki cədvəl bu şəklin addım-addım hesabıdır.

> 🖼️ **Şəkil 12:** Binary mövqeləri · 13 və 200

Burada nəyə bax: 8 xanalıq sırada vurulmuş xanaların qiymətlərini topla, ədəd alınır.

Dayan və düşün D0.2: `0b101010` decimal-da neçədir?

Hexadecimal: bitlərin qısa yazısı

Binary yazı uzun olur: 200 = 11001000. İnsan üçün bunu oxumaq çətindir. Ona görə proqramçılar **hexadecimal** (qısaca hex, on altılıq sistem) istifadə edir.

Hex-də 16 rəqəm var: 0–9 və A, B, C, D, E, F. Burada A = 10, B = 11, ..., F = 15.

Əsas fakt belədir: 4 bit tam olaraq 16 vəziyyət verir. Deməli, bir hex rəqəmi düz 4 bitə uyğundur, bir byte isə düz 2 hex rəqəminə.

Decimal	Binary	Hex		Decimal	Binary	Hex
0	0000	0		8	1000	8
1	0001	1		9	1001	9
2	0010	2		10	1010	A
3	0011	3		11	1011	B
4	0100	4		12	1100	C
5	0101	5		13	1101	D
6	0110	6		14	1110	E
7	0111	7		15	1111	F

Binary-ni hex-ə çevirmək üçün sağdan 4-4 qrupla və hər qrupu cədvəldən tap: 1100 1000 → C 8 → `0xC8` = 200. Proqramçılar hex-i `0x` ilə başlayır: `0xFF` = 255, `0x41` = 65.

Hex-i hər yerdə görəcəksən: yaddaş ünvanlarında (`0x7ffd5a3c`), veb rənglərində (`#FF0000` — qırmızı), Android-da rənglərdə (`0xFFFF0000` — tam görünən qırmızı, ARGB qaydası ilə) və crash log-larında.

> 🖼️ **Şəkil 13:** Hex · 4 bit bir qutuya, bir rəqəmə

Burada nəyə bax: 8 bit iki dördlüyə bölünür, hər dördlük bir hex rəqəmi olur.

> 🖼️ **Şəkil 14:** Decimal → binary → hex · dörd addım

Burada nəyə bax: çevirmə heç bir bölmə tələb etmir, yalnız qruplama və cədvəl.

> 🖼️ **Şəkil 15:** Hex real həyatda · rənglər və ünvanlar

Burada nəyə bax: Android rəngində 4 byte var: alpha, qırmızı, yaşıl, göy. Hər byte 2 hex rəqəmidir.

Dayan və düşün D0.3: `0x2A` decimal-da neçədir?

Böyük ölçülər: KB və KiB

- 1 **kilobyte** (KB) = 1000 byte — beynəlxalq SI standartı.
- 1 **kibibyte** (KiB) = 1024 byte = 2^{10}.
- MB/MiB, GB/GiB, TB/TiB eyni qaydayla gedir.

İki variant niyə var? RAM təbii olaraq 2-nin qüvvətləri ilə qurulur, disk istehsalçıları isə onluq vahidlərlə satır. Ona görə 1 TB disk sistemdə təxminən 931 GiB görünür: 10^{12} / 2^{30} ≈ 931.3. Disk "yalan" demir — sadəcə vahidlər fərqlidir.

> 🖼️ **Şəkil 16:** KB və KiB · 1000 və 1024

Burada nəyə bax: "Ki" hərfləri vahidin ikilik olduğunu bildirir; fərq 1 KB-da cəmi 24 byte-dır.

> 🖼️ **Şəkil 17:** Vahid pillələri · fərq hər pillədə böyüyür

Burada nəyə bax: KB-dan TB-ya qədər fərq 2,4%-dən 10%-ə qədər böyüyür.

> 🖼️ **Şəkil 18:** 1 TB disk · sistemdə 931 GiB

Burada nəyə bax: disk heç nə itirmir. Eyni byte sayı ikilik vahidlə sayılanda kiçik ədəd alınır.

Mətn: hərflər necə ədədə çevrilir

Kompüter yalnız ədədləri saxlaya bilir. Hərfi saxlamaq üçün hər hərfə bir ədəd veririk. Bu razılaşmaya **kodlaşdırma** (*encoding*) deyilir — analogiyadakı kod kitabı məhz budur.

ASCII 1963-cü ildə qəbul olunmuş köhnə və ən geniş yayılmış kod kitabıdır. O, 128 simvolu 0–127 ədədlərinə bağlayır: ingilis hərfləri, rəqəmlər, durğu işarələri.

Simvol	Decimal	Hex	Binary
A	65	0x41	0100 0001
a	97	0x61	0110 0001
0 (rəqəm simvolu)	48	0x30	0011 0000
boşluq	32	0x20	0010 0000

Diqqət: "0" simvolu 0 ədədi deyil, o 48-dir. Klaviaturada "7" yazanda kompüter 55 ədədini alır.

Bir şeyə də fikir ver: 'A' ilə 'a' arasında fərq düz 32-dir, yəni yalnız bir bit. Bu təsadüf deyil — Senior səviyyəsində səbəbini görəcəksən.

ASCII-də "ə", "ş", Çin yazısı və emoji yoxdur. **Unicode** dünyanın bütün simvollarına bir nömrə verir. Bu nömrəyə **code point** deyilir və `U+` ilə hex şəklində yazılır: 'A' = U+0041, 'ə' = U+0259, '😀' = U+1F600. Unicode-da 1 114 112 yer var (U+0000-dan U+10FFFF-ə qədər).

Bəs code point-i byte-lara necə yazmalı? Ən populyar üsul **UTF-8**-dir: simvoldan asılı olaraq 1, 2, 3 və ya 4 byte işlədir.

Simvol	Code point	UTF-8 byte-ları (hex)	Byte sayı
A	U+0041	41	1
ə	U+0259	C9 99	2

Simvol	Code point	UTF-8 byte-ları (hex)	Byte sayı
😀	U+1F600	F0 9F 98 80	4

UTF-8-in gözəlliyi: ASCII mətni UTF-8-də eyni byte-lardan ibarətdir. Ona görə köhnə fayllar sınmadı. UTF-8-i 1992-ci ildə Ken Thompson və Rob Pike hazırlayıb.

> 🖼️ **Şəkil 19:** Kod kitabı · hər simvolun nömrəsi

Burada nəyə bax: kompüter yaddaşda hərfi yox, onun nömrəsini saxlayır.

> 🖼️ **Şəkil 20:** Simvol → code point → UTF-8 · 1, 2 və 4 byte

Burada nəyə bax: 'A' 1 byte, 'ə' 2 byte, emoji 4 byte tutur. Haması eyni UTF-8 qaydası ilə yazılır.

> 🖼️ **Şəkil 21:** 'A' və 'a' · 1 bit fərq

Burada nəyə bax: 32-lik bit yanınca kiçik hərf alınır. Bu, ASCII-nin xüsusi quruluşudur; Azərbaycan hərflərində belə qayda yoxdur.

Mənfi ədədlər: two's complement

Bir byte 0–255 aralığında ədəd saxlayırsa, buna **unsigned** (işarəsiz) deyilir. Mənfi ədədlər də lazımdırsa, **signed** (işarəli) ədəd işlədirik.

Sadə fikir belə olardı: ən soldakı bit işarə olsun (0 = müsbət, 1 = mənfi). Amma belədə iki sıfır yaranır (+0 və −0), toplama dövrəsi də mürəkkəbləşir.

Həll **two's complement**-dir ("ikiyə tamamlama"). Qayda: −x almaq üçün x-in bütün bitlərini çevir (0 ↔ 1) və 1 əlavə et.

Addım	Bitlər (8 bit)	İzah
1	0000 0101	5
2	1111 1010	bütün bitlər çevrildi
3	1111 1011	1 əlavə olundu — bu, −5-dir

Yoxlayaq: 5 + (−5) = 0000 0101 + 1111 1011 = 1 0000 0000. Doqquzuncu bit 8 bitlik qutuya sığmır və atılır. Qalır 0000 0000, yəni 0. Eyni toplama dövrəsi həm müsbət, həm mənfi ədədlər üçün işləyir — two's complement-in bütün sırrı budur.

n bitlik signed ədədin aralığı:

```
-2^{n-1} \le x \le 2^{n-1} - 1
```

8 bit üçün bu −128 ilə 127 arasıdır. Kotlin `Int` (32 bit) üçün −2 147 483 648 ilə 2 147 483 647 arasıdır.

> 🖼️ **Şəkil 22:** Two's complement · 8 bitlik ədədlərin dairəsi

Burada nəyə bax: dairənin sağ yarısı müsbət, sol yarısı mənfi ədədlərdir. İlk bit hansı yarımda olduğunu deyir.

> 🖼️ **Şəkil 23:** −5 almaq · çevir və 1 əlavə et

Burada nəyə bax: ikinci addımda hər bit dəyişir, üçüncüdə yalnız sonuncu xana.

> 🖼️ **Şəkil 24:** 5 + (−5) · artıq bit atılır

Burada nəyə bax: toplamanın artıq biti 8 bitlik qutudan kənarda qalır, qutunun içi isə düz 0-dır.

Dayan və düşün D0.4: −1 ədədi 8 bitlik two's complement-də necə yazılır?

Overflow: ədəd qutuya sığmayanda

Saatın əqrəbi 12-dən sonra yenidən 1-ə qayıdır. Bit qutusu da belədir: ən böyük ədədə 1 əlavə edəndə dövrə yenidən başlayır. Buna **overflow** (daşma) deyilir.

8 bitlik signed ədəddə: 127 = 0111 1111. Buna 1 əlavə edəndə 1000 0000 alınır, bu isə −128-dir. Kotlin və Java-da `Int.MAX_VALUE + 1` = −2 147 483 648 olur: proqram xəta vermir, səssizcə dövr edir. "Özün gör" bölməsində bunu real çıxışda görəcəksən.

> 🖼️ **Şəkil 25:** Overflow hekayəsi · odometr 9999 → 0000

Burada nəyə bax: odometr qutusu 4 rəqəmlidir, beşinci rəqəmə yer yoxdur.

> 🖼️ **Şəkil 26:** 127 + 1 · işarə biti dəyişir

Burada nəyə bax: köçürmə sağdan sola axıb işarə bitinə çatır və ədəd mənfi oxunur.

> 🖼️ **Şəkil 27:** Kotlin-də overflow · səssiz və yoxlamalı

Burada nəyə bax: adi `+` səssizcə dövr edir, `Math.addExact` isə daşmada exception atır.

Endianness: byte-ların sırası

Bir byte-dan böyük ədəd (məsələn, 4 byte-lıq `Int`) yaddaşa yazılanda sual yaranır: hansı byte əvvəl gəlsin? Tarixi "2026-11-19" və ya "19.11.2026" yazmaq kimidir: məlumat eynidir, sıra fərqlidir. Byte-ların sırasına **endianness** deyilir.

- **Little-endian:** ən kiçik dəyərli byte əvvəl gəlir. 0x12345678 yaddaşda 78 56 34 12 kimi düzülür. x86-64 kompüterlər və Android telefonlarındakı ARM64 prosessorlar belə işləyir.

- **Big-endian:** ən böyük dəyərli byte əvvəl gəlir: 12 34 56 78. Şəbəkə protokolları (TCP/IP) belə yazır, ona görə buna "network byte order" də deyilir.

> 🖼️ **Şəkil 28:** Endianness hekayəsi · tarixin iki yazılışı

Burada nəyə bax: üç hissə eynidir, fərq yalnız onların sırasındadır.

> 🖼️ **Şəkil 29:** 0x12345678 yaddaşda · little-endian və big-endian

Burada nəyə bax: 0x100 ünvanında little-endian-da 78, big-endian-da 12 dayanır.

> 🖼️ **Şəkil 30:** Telefon, JVM və şəbəkə · kim hansı sıra ilə yazır

Burada nəyə bax: eyni ədəd CPU yaddaşında bir sıra ilə, ByteBuffer və şəbəkədə əks sıra ilə yazılır. Alt sətirdəki dəyərlər bu sistemdə Kotlin ilə yoxlanıb.

Fayl

Fayl diskdə adı olan byte ardıcıllığıdır. Faylın "növü" byte-ların içində yazılmayıb: onu hansı proqramın necə oxuduğu müəyyən edir.

Bir çox format ilk byte-larda "imza" saxlayır, buna **magic number** deyilir. PNG şəkli 89 50 4E 47 ilə başlayır. Linux proqramı (ELF formatı) isə 7F 45 4C 46 ilə: 45, 4C, 46 ASCII-də "E", "L", "F" hərfləridir.

"Mətn faylı" ilə "binary fayl" arasında fiziki fərq yoxdur: hər ikisi byte-lardır. Fərq yalnız onların necə oxunduğundadır.

> 🖼️ **Şəkil 31:** Fayl · ad etiketdir, içi byte-lardır

Burada nəyə bax: adı dəyişmək faylın içindəki byte-lara toxunmur.

> 🖼️ **Şəkil 32:** Magic number · PNG, ELF, ZIP/APK, PDF

Burada nəyə bax: hər formatın ilk byte-ları onun imzasıdır. APK faylı da ZIP imzası ilə başlayır.

> 🖼️ **Şəkil 33:** Mətn və binary fayl · fərq oxuyandadır

Burada nəyə bax: hər iki fayl byte-dan ibarətdir. Mətn redaktoru və loader onları fərqli kod kitabı ilə oxuyur.

Proqram

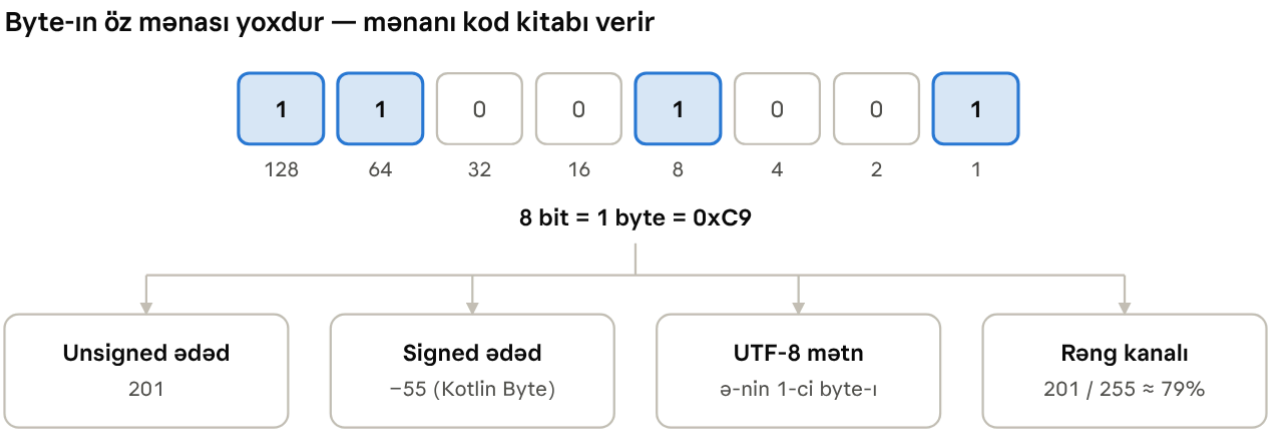
Proqram da bir fayldır. Onun byte-larının bir hissəsi CPU üçün əmrlərdir, bir hissəsi isə data. CPU bu byte-ları oxuyur və əmr kimi yerinə yetirir. Bunun necə baş verdiyini Fəsil 1 və Fəsil 3 göstərəcək.

> 🖼️ **Şəkil 34:** Proqram hekayəsi · resept, aşpaz və CPU

Burada nəyə bax: resept vərəqində həm addımlar, həm inqrediyentlər var. Proqram faylında da həm kod, həm data var.

> 🖼️ **Şəkil 35:** ELF faylının içi · header, kod, data

Burada nəyə bax: header faylın nə olduğunu və haradan başlamaq lazım olduğunu deyir. Kod və data ayrı-ayrı hissələrdə dayanır.



Sirr-in cavabı: eyni 0 və 1-lər fərqli şeylərdir, çünki onları oxuyan proqram fərqli kod kitabından istifadə edir. Şəkil proqramı byte-ı rəng, musiqi proqramı səs, CPU isə əmr kimi oxuyur. Yuxarıdakı 0xC9 byte-ı bunu göstərir: dörd fərqli kod kitabı — dörd fərqli mənа.

Özün gör

Bu komandaları terminalda işlət. Aşağıdakı çıxışlar test maşınından olduqca kopyalanıb (Linux, x86-64). `$` işarəsi komandanın başladığı yeri göstərir, onu yazma.

1. Hərflər hansı byte-lardır?

```
$ printf 'Aa0' | od -A d -t x1 -t u1
0000000  41  61  30
         65  97  48
0000003
```

- `printf 'Aa0'` üç simvolu sonda yeni sətir olmadan çıxarır.
- Ortadakı şaşquli xətt (pipe) bir proqramın çıxışını digərinin girişinə ötürür.
- `od` byte-ları göstərir: `-t x1` hər byte-ı hex, `-t u1` unsigned decimal kimi yazır, `-A d` soldakı sıra nömrəsini decimal edir.
- Birinci sətir 41 61 30 — hex; ikinci sətir 65 97 48 — eyni byte-lar decimal-da. Sonuncu 0000003 — cəmi 3 byte.

> 🖼️ **Şəkil 37:** od çıxışı · sütunlar nə deyir

Burada nəyə bax: eyni üç byte iki sətrdə iki fərqli say sistemi ilə yazılıb.

2. Azərbaycan hərfləri və emoji

```
$ printf 'ə' | od -A d -t x1
0000000 c9 99
0000002
$ printf '😀' | od -A d -t x1
0000000 f0 9f 98 80
0000004
$ printf 'Bakı' | od -A d -t x1
0000000 42 61 6b c4 b1
0000005
```

"ə" 2 byte, emoji 4 byte tutur — UTF-8 cədvəli ilə eyni. "Bakı" 4 hərfdir, amma 5 byte-dır: "ı" hərfi (U+0131) ASCII-də yoxdur və 2 byte tutur (c4 b1).

> 🖼️ **Şəkil 38:** "Bakı" · 4 hərf, 5 byte

Burada nəyə bax: "ı" hərfi iki byte-a yayılır, qalan hərflər bir byte-dır.

3. Proqram faylının imzası

```
$ head -c 4 /bin/ls | od -A d -t x1z
0000000 7f 45 4c 46                                     >.ELF<
0000004
```

`head -c 4` faylın ilk 4 byte-ını götürür. `z` seçimi byte-ları mətn kimi də göstərir: `.ELF`. 7F çap olunan simvol deyil, ona görə onun yerində nöqtə var.

> 🖼️ **Şəkil 39:** /bin/ls · ilk 4 byte

Burada nəyə bax: byte-ları mətn kimi oxuyanda proqramın imzası — "ELF" sözü — görünür.

4. Endianness

```
$ python3 -c "import struct, sys; print(struct.pack('<I', 0x12345678).hex(), struct.pack('>I',
0x12345678).hex(), sys.byteorder)"
78563412 12345678 little
$ lscpu | grep 'Byte Order'
Byte Order:                         Little Endian
```

`'<I'` 4 byte-lıq ədədi little-endian, `'>I'` big-endian yazır. Bu maşının özü little-endian-dır. macOS-da `lscpu` yoxdur, Python sətri isə orada da işləyir.

5. Overflow, emoji uzunluğu və byte-ın işarəsi (JVM)

Bu Java proqramı Kotlin-in də istifadə etdiyi JVM tiplərini yoxlayır. `Bits.java` adı ilə saxla, `javac Bits.java && java Bits` ilə işlət.

```java
        import java.nio.ByteBuffer;
        import java.nio.ByteOrder;
        import java.nio.charset.StandardCharsets;

        public class Bits {
            public static void main(String[] args) {
                System.out.println("MAX+1 = " + (Integer.MAX_VALUE + 1));
                System.out.println("abs(MIN) = " + Math.abs(Integer.MIN_VALUE));
                String s = "😀";
                System.out.println("length = " + s.length() + ", codePoints = " +
s.codePointCount(0, s.length()));
                byte[] b = "ə".getBytes(StandardCharsets.UTF_8);
                System.out.println("bytes = " + b[0] + ", " + b[1]);
                System.out.println("b[0] == 0xC9 ? " + (b[0] == 0xC9) + "; masked: " + ((b[0] &
0xFF) == 0xC9));
                System.out.println("0.1 + 0.2 = " + (0.1 + 0.2));
                System.out.println("ByteBuffer default = " + ByteBuffer.allocate(4).order() + ",
native = " + ByteOrder.nativeOrder());
                int days = 25;
                int ms = days * 24 * 60 * 60 * 1000;
                long msOk = days * 24L * 60 * 60 * 1000;
                System.out.println("25 gun ms (Int) = " + ms + ", (Long) = " + msOk);
                System.out.println("-5 binary = " + Integer.toBinaryString(-5 & 0xFF));
                System.out.println("(byte)127+1 = " + (byte)(127 + 1));
                String t = "Salam 😀";
                System.out.println("take(7) = [" + t.substring(0, 7) + "] len=" + t.length());
            }
        }
```

Real çıxış (OpenJDK 21):

```
MAX+1 = -2147483648
abs(MIN) = -2147483648
length = 2, codePoints = 1
bytes = -55, -103
b[0] == 0xC9 ? false; masked: true
0.1 + 0.2 = 0.30000000000000004
ByteBuffer default = BIG_ENDIAN, native = LITTLE_ENDIAN
25 gun ms (Int) = -2134967296, (Long) = 2160000000
-5 binary = 11111011
(byte)127+1 = -128
take(7) = [Salam ?] len=8
```

Hər sətir bu fəsildən bir faktı sübut edir. Overflow səssizdir. Emoji 2 `char` tutur. "ə"-nin byte-ları mənfi görünür, çünki Java və Kotlin-də `Byte` signed-dir. Sonuncu sətirdə emoji yarıya bölünüb və yerində "?" qalıb. Bunların hər birini Senior səviyyəsində açacağıq.

Kotlin versiyası (`Bits.kt`, kotlinc 2.1.0, eyni JVM):

```kotlin
        import java.nio.ByteBuffer
        import java.nio.ByteOrder
        import kotlin.math.abs

        fun main() {
            println("MAX+1 = ${Int.MAX_VALUE + 1}")
            println("abs(MIN) = ${abs(Int.MIN_VALUE)}")
            val s = "😀"
            println("length = ${s.length}, codePoints = ${s.codePointCount(0, s.length)}")
            val b = "ə".toByteArray(Charsets.UTF_8)
            println("bytes = ${b[0]}, ${b[1]}")
            println("b[0] == 0xC9 ? ${b[0].toInt() == 0xC9}; masked: ${(b[0].toInt() and 0xFF) ==
```

```
0xC9}; toUByte = ${b[0].toUByte()}")
            println("0.1 + 0.2 = ${0.1 + 0.2}")
            println("ByteBuffer default = ${ByteBuffer.allocate(4).order()}, native =
${ByteOrder.nativeOrder()}")
            val days = 25
            val ms = days * 24 * 60 * 60 * 1000
            val msOk = days * 24L * 60 * 60 * 1000
            println("25 gun ms (Int) = $ms, (Long) = $msOk")
            println("-5 binary = ${(-5 and 0xFF).toString(2)}")
            println("(127 + 1).toByte() = ${(127 + 1).toByte()}")
            val t = "Salam 😀"
            println("take(7) = [${t.take(7)}] len=${t.length}")
        }
```

```
$ kotlinc Bits.kt -include-runtime -d Bits.jar && java -jar Bits.jar
MAX+1 = -2147483648
abs(MIN) = -2147483648
length = 2, codePoints = 1
bytes = -55, -103
b[0] == 0xC9 ? false; masked: true; toUByte = 201
0.1 + 0.2 = 0.30000000000000004
ByteBuffer default = BIG_ENDIAN, native = LITTLE_ENDIAN
25 gun ms (Int) = -2134967296, (Long) = 2160000000
-5 binary = 11111011
(127 + 1).toByte() = -128
take(7) = [Salam ?] len=8
```

Nəticələr Java ilə eynidir, çünki Kotlin JVM-də eyni tipləri işlədir: `Int` 32 bitlik two's complement-dir, `Byte` signed-dir, `String` UTF-16-dır. Kotlin-ə məxsus bir əlavə: `toUByte()` byte-ı 0–255 aralığında oxuyur və 201 verir.

> 🖼️ **Şəkil 40:** Bits proqramı · hər sətir bir sübut

Burada nəyə bax: qırmızı sətirlər proqramı sakitcə sındıran hallardır: xəta yoxdur, amma nəticə səhvdir.

Variasiya: öz adını `printf 'Adın' | od -A d -t x1` ilə yoxla. Neçə byte-dır və niyə?

Kod

Əvvəlcə alət: bit əməliyyatları

Koddan əvvəl bitlərlə işləyən beş əməliyyatı tanı. Aşağıda C və Java işarəsi, sonra Kotlin adı verilib.

- **AND** (C/Java `&`, Kotlin `and`): yalnız hər iki bit 1 olanda 1. `1100 1001 AND 1111 0000 = 1100 0000`. Buna "maska" deyilir: lazım olan bitləri saxlayır, qalanını sıfırlayır.
- **OR** (C/Java şaşquli xətt, Kotlin `or`): ən azı biri 1 olanda 1. `1100 1001 OR 0000 0110 = 1100 1111`. Bitləri "yandırır".
- **XOR** (C/Java `^`, Kotlin `xor`): bitlər fərqli olanda 1. `0100 0001 XOR 0010 0000 = 0110 0001` — bu, 'A'-nı 'a'-ya çevirir.
- **NOT** (C/Java `~`, Kotlin `inv()`): bütün bitləri çevirir. Two's complement-in birinci addımı məhz budur.
- **Shift** (C/Java `<<` və `>>`, Kotlin `shl`, `shr`, `ushr`): bitləri sola və ya sağa sürüşdürür. `0000 0101 << 1 = 0000 1010`: 5 → 10. Sola bir sürüşdürmə 2-yə vurmaqdır, sağa bir sürüşdürmə 2-yə bölmək.

> 🖼️ **Şəkil 41:** AND maskası · dəlikli trafaret

Burada nəyə bax: maskanın 1 olan yerləri dəlikdir; byte-ın yalnız o hissəsi nəticəyə keçir.

Nümunə 1 — tam izahlı: ədədi binary-yə çevirmək (C)

```c
#include <stdio.h>

/* n ədədini ikilik (binary) yazıya çevirir və ekrana çıxarır */
void print_binary(unsigned int n) {
    char digits[33];          /* 32 bit + sonda '\0' */
    int count = 0;
    if (n == 0) { printf("0\n"); return; }
    while (n > 0) {
        digits[count] = (n % 2) ? '1' : '0';  /* qalıq: 0 və ya 1 */
        n = n / 2;                            /* ədədi 2-yə böl */
        count++;
    }
    for (int i = count - 1; i >= 0; i--) {    /* tərsinə oxu */
        putchar(digits[i]);
    }
    putchar('\n');
}

int is_power_of_two(unsigned int n) {
    return n != 0 && (n & (n - 1)) == 0;
}

int main(void) {
    print_binary(13);
    print_binary(200);
    print_binary(255);
    printf("%x %X %d\n", 255, 200, 0x41);
    for (unsigned int n = 0; n <= 9; n++) {
        printf("%u:%d ", n, is_power_of_two(n));
    }
    printf("\n");
    return 0;
}
```

Sətir-sətir:

1. `unsigned int n` — işarəsiz 32 bitlik ədəd. Mənfi ədəd gəlməyəcək.
2. `char digits[33]` — 32 simvolluq yer və bir əlavə yer. C-də mətnin sonunu `'\0'` simvolu bildirir.
3. `n % 2` — 2-yə bölmədən qalıq, yəni ən sağdakı bit.
4. `n = n / 2` — ədəd bir bit sağa sürüşür (`n >> 1` ilə eynidir).
5. Tərs dövr — qalıqlar sağdan sola yaranıb, ona görə tərsinə çap edirik.
6. `printf("%x %X %d", ...)` — eyni ədədi kiçik hərfli hex, böyük hərfli hex və decimal kimi çap edir.

Kompilyasiya və işə salma (gcc 13.3):

```
$ gcc -Wall -O2 -o binary binary.c && ./binary
1101
11001000
11111111
```

```
ff C8 65
0:0 1:1 2:1 3:0 4:1 5:0 6:0 7:0 8:1 9:0
```

Kotlin versiyası (`Binary.kt`): eyni alqoritm, amma `UInt` ilə. Mənfi ədəd gəlmir, `%` və `/` unsigned işləyir.

```
        // n ədədini ikilik yazıya çevirir (C-dəki print_binary-nin Kotlin versiyası)
        fun toBinary(n: UInt): String {
            if (n == 0u) return "0"
            val digits = StringBuilder()
            var x = n
            while (x > 0u) {
                digits.append(if (x % 2u == 1u) '1' else '0')   // qalıq: 0 və ya 1
                x /= 2u                                          // ədədi 2-yə böl
            }
            return digits.reverse().toString()                   // tərsinə oxu
        }

        fun isPowerOfTwo(n: UInt): Boolean = n != 0u && (n and (n - 1u)) == 0u

        fun main() {
            println(toBinary(13u))
            println(toBinary(200u))
            println(toBinary(255u))
            println("${255.toString(16)} ${200.toString(16).uppercase()} ${0x41}")
            println((0u..9u).joinToString(" ") { "$it:${if (isPowerOfTwo(it)) 1 else 0}" })
            println("hazir funksiyalar: ${13.toString(2)} ${Integer.toBinaryString(-5)}")
        }
```

```
$ kotlinc Binary.kt -include-runtime -d Binary.jar && java -jar Binary.jar
1101
11001000
11111111
ff C8 65
0:0 1:1 2:1 3:0 4:1 5:0 6:0 7:0 8:1 9:0
hazir funksiyalar: 1101 11111111111111111111111111111011
```

Son sətir Kotlin-in hazır funksiyalarıdır: `13.toString(2)` binary yazılışı verir, `Integer.toBinaryString(-5)` isə −5-in 32 bitlik two's complement yazılışını.

> 🖼️ **Şəkil 42:** toBinary(13) · dövr addım-addım

Burada nəyə bax: hər addımda bir bit qalıq kimi çıxır, ədəd isə yarıya enir.

Nümunə 2 — sən tamamla: ədədi hex-ə çevirmək

Nümunə 1-dəki fikir eynidir, sadəcə 2 əvəzinə 16 işlənir. (1) və (2) yerlərini doldur.

```
        void print_hex(unsigned int n) {
            const char *symbols = "0123456789ABCDEF";
            char digits[9];
            int count = 0;
            if (n == 0) { printf("0\n"); return; }
            while (n > 0) {
                digits[count] = symbols[ ____ ];   /* (1) */
                n = ____;                          /* (2) */
                count++;
```

```
        }
        for (int i = count - 1; i >= 0; i--) putchar(digits[i]);
        putchar('\n');
    }
```

`print_hex(200)`, `print_hex(255)`, `print_hex(4096)` üçün gözlənilən çıxış: `C8`, `FF`, `1000`.

Kotlin-də eyni tapşırıq — (1) və (2)-ni doldur:

```
fun toHex(n: UInt): String {
    val symbols = "0123456789ABCDEF"
    if (n == 0u) return "0"
    val digits = StringBuilder()
    var x = n
    while (x > 0u) {
        digits.append(symbols[ ____ ])   // (1) String-in indeksi Int istəyir
        x = ____                          // (2)
    }
    return digits.reverse().toString()
}
```

`toHex(200u)`, `toHex(255u)`, `toHex(4096u)` üçün gözlənilən çıxış: `C8`, `FF`, `1000`.

> 🖼️ **Şəkil 43:** İpucu · 200 → C8

Burada nəyə bax: Nümunə 1-dəki dövr eynidir, yalnız 2 əvəzinə 16 işlənir və qalıq cədvəldən simvola çevrilir.

Nümunə 3 — yalnız tapşırıq

`is_power_of_two(n)` funksiyasını dövrsüz, bir sətirlə yaz. 0–9 üçün gözlənilən çıxış: `0:0 1:1 2:1 3:0 4:1 5:0 6:0 7:0 8:1 9:0`. İpucu: 8 = 1000, 7 = 0111. Nümunə 1-in kodu cavabı artıq göstərir — ona baxmadan yazmağa çalış.

Sonra Kotlin-də də yaz: `fun isPowerOfTwo(n: UInt): Boolean` — bir sətir, `and` ilə. Gözlənilən çıxış eynidir.

Tipik səhv: Kotlin-də byte oxumaq

❌ **Səhv:**

```
val bytes = "ə".toByteArray(Charsets.UTF_8)
if (bytes[0].toInt() == 0xC9) println("UTF-8 ə tapıldı")   // heç vaxt çap olunmur
```

Nə baş verir: Kotlin `Byte`-ı signed-dir, 0xC9 byte kimi −55-dir. `toInt()` ədədi 32 bitə genişləndəndə işarə bitini yuxarı bitlərə köçürür: 1100 1001 → 1111 1111 1111 1111 1111 1111 1100 1001 = −55. Buna **sign extension** deyilir. −55 isə 201-ə bərabər deyil.

✅ **Düzgün:**

```
if ((bytes[0].toInt() and 0xFF) == 0xC9) println("UTF-8 ə tapıldı")
```

Niyə işləyir: `and 0xFF` yuxarı 24 biti sıfırlayır və 201 qalır. Sübut — yuxarıdakı Java çıxışı: `b[0] == 0xC9 ? false; masked: true`.

> 🖼️ **Şəkil 44:** Sign extension · Byte → Int və and 0xFF

Burada nəyə bax: `toInt()` yuxarı 24 biti 1 ilə doldurur; `and 0xFF` onları sıfırlayır və 201 qalır.

Senior səviyyəsi

Bu hissə Fəsil 0-ın ən dərin qatıdır. Burada sonrakı fəsillərdən bir neçə söz gəlir. Hər birini yerində bir cümlə ilə izah edirəm və harada tam öyrənəcəyini deyirəm.

1. Bit fiziki olaraq nədir

- RAM-da (Fəsil 2-də) bir bit kiçik bir kondensatordakı elektrik yüküdür. Kondensator — yük saxlayan mikroskopik "qab"dır. Yük yavaş-yavaş axıb gedir, ona görə yaddaş çipi hər sətri təxminən hər 64 millisaniyədə bir yenidən doldurur. Buna **refresh** (təzələmə) deyilir.
- Telefonun daimi yaddaşında (flash) bit izolyasiya olunmuş qatda "tələyə salınmış" elektronlardır. Elektrik kəsəndə də qalır.
- CPU-nun içindəki ən sürətli yaddaşda — register-lərdə (Fəsil 1-də) — bit bir neçə transistordan qurulmuş dövrənin vəziyyətidir.
- Bit təsadüfən çevrilə bilər: kosmik şüa və ya elektrik səs-küyü bunu edə bilər. Server-lərdə **ECC** (Error-Correcting Code, xəta düzəldən kod) yaddaş hər söz üçün əlavə yoxlama bitləri saxlayır. Bir bitlik səhvi düzəldir, iki bitlik səhvi aşkar edir.

Nəticə: "0" və "1" fiziki dünyada iki fərqli vəziyyətin adıdır. Proqramçı üçün bu bir abstraksiyadır. Abstraksiya da bəzən sınır, və senior bunu bilir.

> 🖼️ **Şəkil 45:** DRAM · sızan vedrə və refresh

Burada nəyə bax: yük sızır, amma "1" tanınmaz hala düşmədən refresh onu bərpa edir.

> 🖼️ **Şəkil 46:** Bitin üç fiziki forması · DRAM, flash, register

Burada nəyə bax: yalnız flash elektrik olmadan biti saxlayır. Ona görə telefon sönəndə RAM boşalır, fayllar isə qalır.

> 🖼️ **Şəkil 47:** ECC · 1 səhv düzəlir, 2 səhv görünür

Burada nəyə bax: əlavə yoxlama bitləri bir səhvin yerini tapır, iki səhvdə isə yalnız xəbərdarlıq verir.

2. UTF-8-in daxili quruluşu

UTF-8 hər code point-i 1–4 byte-a bu şablonlarla yerləşdirir. `x` — code point-in öz bitləridir.

Code point aralığı	Byte sayı	Bit şablonu
U+0000 – U+007F	1	0xxxxxxx
U+0080 – U+07FF	2	110xxxxx 10xxxxxx
U+0800 – U+FFFF	3	1110xxxx 10xxxxxx 10xxxxxx
U+10000 – U+10FFFF	4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

"ə" (U+0259) addım-addım:

1. 0x259 binary-də: 10 0101 1001 (10 bit).
2. 2 byte-lıq şablonda 11 boş yer var (5 + 6). Sola bir 0 əlavə edirik: 010 0101 1001.
3. Bölürük: ilk 5 bit 01001, son 6 bit 011001.
4. Şablona qoyuruq: 110 + 01001 = 1100 1001 = C9; 10 + 011001 = 1001 1001 = 99.
5. Nəticə C9 99 — "Özün gör"-dəki `od` çıxışı ilə eynidir.

Bu dizaynın üç gizli üstünlüyü var:

- **ASCII uyğunluğu:** 0–127 arası simvollar dəyişmir. Köhnə ingiliscə mətn faylı artıq düzgün UTF-8 faylıdır.
- **Self-synchronizing** (özünü sinxronlaşdıran): davam byte-ları həmişə 10 ilə başlayır. Mətnin ortasından oxumağa başlasan, ən çoxu 3 byte geri gedib simvolun başlanğıcını taparsan.
- **Sıfır byte yoxdur:** çox-byte-lı simvolların içində 00 byte-ı heç vaxt görünmür. C dilində mətnin sonunu 00 bildirir, ona görə köhnə C kodu UTF-8-i sındırmır.

Tarix: UTF-8-i Ken Thompson 1992-ci ilin sentyabrında Nyu-Cersidə bir yeməkxanada, Rob Pike-ın gözü qarşısında, masanın kağız örtüyünün üstündə layihələndirib. Bir neçə gün sonra onu Plan 9 əməliyyat sistemində tətbiq ediblər. Mənbə: Rob Pike-ın 2003-cü ildə yazdığı "UTF-8 history" məktubu.

Təhlükəsizlik tərəfi — overlong encoding: "/" simvolu (2F) nəzəri olaraq 2 byte-lıq şablonla da yazıla bilər: C0 AF. Buna **overlong encoding** (lazım olandan uzun yazılış) deyilir. UTF-8 standartı (RFC 3629) bunu qadağan edir. 2000-ci ildə Microsoft IIS 4.0 və 5.0 veb-serverləri yolu yoxlayanda bu formadan xəbərsiz idi. Hücumçular URL-də "/" əvəzinə `%c0%af` yazıb server-in veb qovluğundan kənara çıxır və komanda işlədinə bilirdilər (CVE-2000-0884, MS00-078). Bugünkü dekoderlər bu byte-ları rədd edir:

```
$ python3 -c "
try:
    b'\xc0\xaf'.decode('utf-8')
except UnicodeDecodeError as e:
    print('overlong:', e)"
overlong: 'utf-8' codec can't decode byte 0xc0 in position 0: invalid start byte
```

> 🖼️ **Şəkil 48:** UTF-8 qatarı · lokomotiv və vaqonlar

Burada nəyə bax: ilk byte-ın başındakı 1-lərin sayı simvolun neçə byte olduğunu deyir.

> 🖼️ **Şəkil 49:** ə → C9 99 · bitlər şablona

Burada nəyə bax: boz xanalar şablondan gəlir, rəngli xanalar isə 0x259-un öz bitləridir.

> 🖼️ **Şəkil 50:** Self-synchronizing · ortadan oxumaq

Burada nəyə bax: 98 byte-ı "10" ilə başlayır, ona görə oxuyan geri gedir və F0-da simvolun başını tapır.

3. JVM-də mətn: UTF-16 və surrogate cütləri

- Java və Kotlin-də `String` UTF-16 **code unit**-lərindən (16 bitlik vahidlərdən) ibarətdir. `Char` tipi 16 bitdir.
- 16 bitə U+0000–U+FFFF aralığı sığır. Bu aralığa **BMP** (Basic Multilingual Plane) deyilir. Bütün Azərbaycan hərfləri buradadır.
- Daha böyük code point-lər (emoji kimi) iki `Char` tutur. Bu cütə **surrogate pair** deyilir.

Formula, 😀 (U+1F600) üçün:

1. 0x1F600 − 0x10000 = 0xF600. Bu 20 bitə sığan ədəddir.
2. Yuxarı 10 bit: 0xF600 sağa 10 sürüşdürüləndə 0x3D alınır. 0xD800 + 0x3D = **D83D** (high surrogate).
3. Aşağı 10 bit: 0xF600 AND 0x3FF = 0x200. 0xDC00 + 0x200 = **DE00** (low surrogate).
4. Yoxlama: Python-da `'😀'.encode('utf-16-be').hex()` = d83dde00.

Nəticələr:

- `"😀".length == 2`, amma `codePointCount == 1` (Bits.java çıxışı).
- `substring(0, 7)` cütü ortadan kəsir. Tək qalan D83D heç bir simvol deyil. UTF-8-ə çevirəndə Java onu "?" ilə əvəz edir: `take(7) = [Salam ?]`. Kotlin-in `take(7)` funksiyası da içəridə eyni `substring`-i çağırır.
- Code point saymaq da kifayət deyil. 👍🏽 = 👍 (U+1F44D) + dəri rəngi modifikatoru (U+1F3FD). Bu 2 code point, 4 `Char` və 8 UTF-8 byte-dır, amma ekranda 1 işarədir. İstifadəçinin "bir simvol" saydığı vahidə **grapheme cluster** deyilir. Ailə emojiləri bir neçə emojini ZWJ (U+200D, "zero width joiner") ilə birləşdirir.
- Android-də mətni istifadəçi gözü ilə kəsmək üçün `android.icu.text.BreakIterator.getCharacterInstance()` işlət (API 24-dən).

Android NDK tərəfi — modified UTF-8: JNI-nin `NewStringUTF` funksiyası standart UTF-8 deyil, "modified UTF-8" gözləyir. Orada U+0000 iki byte ilə (C0 80) yazılır. BMP-dən kənar simvol isə 4 byte əvəzinə iki surrogate kimi, hər biri 3 byte (cəmi 6 byte) yazılır. C++ kodundan real UTF-8 emoji (F0 9F 98 80) `NewStringUTF`-a ötürsən nə olur? CheckJNI aktiv olanda (emulator-da default) tətbiq bu mesajla dayanır: `JNI DETECTED ERROR IN APPLICATION: input is not valid Modified UTF-8: illegal start byte 0xf0`. Daha yeni ART versiyalarında səhv giriş "?" ilə əvəz olunur və log-a yazılır. Hər iki halda nəticə səhvdir. Təhlükəsiz yol: byte massivini Kotlin-ə ötürüb `String(bytes, Charsets.UTF_8)` ilə çevirmək. JNI-ni tam Faza 10-da öyrənəcəksən.

> 🖼️ **Şəkil 51:** UTF-16 · hər Char bir oturacaq

Burada nəyə bax: Azərbaycan hərfləri bir yer tutur, emoji iki yer.

> 🖼️ **Şəkil 52:** Surrogate cütü · U+1F600 → D83D DE00

Burada nəyə bax: 20 bitlik ədəd iki 10 bitlik yarıya bölünür və hər yarı öz əsasına əlavə olunur.

> 🖼️ **Şəkil 53:** 👍🏽 · 1, 2, 4 və 8

Burada nəyə bax: eyni işarə dörd fərqli vahidlə dörd fərqli ədəd verir.

4. Two's complement: hardware niyə onu seçdi

- Toplama dövrəsi (**adder**) signed və unsigned ədədlər üçün eynidir. 1111 1111 + 0000 0001 = 0000 0000 (daşan bit atılır). Unsigned baxışda bu 255 + 1 = 0-dır (wrap). Signed baxışda −1 + 1 = 0-dır (tam düz). Bir dövrə, iki məna.
- Köhnə üsullarda (sign-magnitude, ones' complement) iki sıfır var idi: +0 və −0. Two's complement-də sıfır təkdir.
- CPU ədədin signed olduğunu bilmir. Fərqi instruction seçir. Məsələn, x86-64-də müqayisədən sonra `jl` (signed "kiçikdirsə keç") və `jb` (unsigned "aşağıdırsa keç") fərqli **flag**-lərə baxır. Flag — CPU-nun son əməliyyat haqqında saxladığı bir bitlik qeyddir: `jl` SF və OF flag-lərinə, `jb` CF flag-ına baxır (Fəsil 1-də).
- Aralıq asimmetrikdir: 8 bitdə −128...127. −128-in müsbət cütü yoxdur. Ona görə −(−128) yenə −128 olur (More.java: `-128 neg = -128`), `Math.abs(Int.MIN_VALUE)` isə `Int.MIN_VALUE` qaytarır (Bits.java: `abs(MIN) = -2147483648`).

Overflow-un üç dünyası:

Dil	İşarəli overflow	Nə baş verir
Java, Kotlin (JVM və ART)	Dəqiq təyin olunub	Wraparound: riyazi cəmin aşağı 32 biti qalır (JLS §15.18.2)
C, C++	Undefined behavior	Kompilyator "bu heç vaxt baş vermir" fərz edib kodu dəyişə bilər
Swift; Rust debug rejimində	Xəta	Proqram dayanır

Undefined behavior (təyin olunmamış davranış) — standartın "bu halda nə olacağına söz vermirəm" dediği yerdir. Kompilyator — source kodu maşın koduna çevirən proqramdır (Fəsil 3-də). O, UB-nin heç vaxt baş verməyəcəyini fərz etməyə haqlıdır. Bu C koduna bax:

```c
#include <stdio.h>
#include <limits.h>
int check(int x) { return x + 1 > x; }
int main(void) { printf("check(INT_MAX) = %d\n", check(INT_MAX)); return 0; }
```

Eyni fayl, fərqli kompilyator və bayraqlar (clang 18.1.3, gcc 13.3, x86-64):

Kompilyator və bayraq	Çıxış
clang -O0	check(INT_MAX) = 0

Kompilyator və bayraq	Çıxış
clang -O2	check(INT_MAX) = 1
clang -O2 -fwrapv	check(INT_MAX) = 0
gcc -O0	check(INT_MAX) = 1
gcc -O2	check(INT_MAX) = 1
gcc -O2 -fwrapv	check(INT_MAX) = 0

- `clang -O0` həqiqətən toplayır: INT_MAX + 1 wrap edib INT_MIN olur, INT_MIN > INT_MAX yanlışdır → 0.
- `-O2` (optimizasiya) ilə kompilyator düşünür: "signed overflow ola bilməz, deməli x + 1 həmişə x-dən böyükdür" və funksiyanı `return 1` ilə əvəz edir. gcc bunu hətta `-O0`-da da edir.
- `-fwrapv` kompilyatora "overflow wrap etsin" deyir və cavab yenə 0 olur.
- Real təhlükə: C-də `if (len + offset < len)` kimi overflow yoxlamalarını kompilyator eyni məntiqlə silə bilər. Bu, təhlükəsizlik boşluğu deməkdir.

Kotlin-də eyni funksiya (`Overflow.kt`). JVM-də overflow undefined behavior deyil, həmişə wrap edir. Ona görə JIT `x + 1 > x`-i `true` ilə əvəz edə bilməz:

```
        import java.math.BigDecimal
        import kotlin.math.abs

        // C-dəki ub.c ilə müqayisə: JVM-də overflow həmişə wrap edir
        fun check(x: Int) = x + 1 > x

        fun main() {
            println("check(Int.MAX_VALUE) = ${check(Int.MAX_VALUE)}")
            try {
                Math.addExact(Int.MAX_VALUE, 1)
            } catch (e: ArithmeticException) {
                println("addExact: ${e.message}")
            }
            val h = "polygenelubricants".hashCode()
            println("hash = $h, abs(h) % 10 = ${abs(h) % 10}, floorMod = ${Math.floorMod(h, 10)},
h.mod(10) = ${h.mod(10)}")
            println("UInt.MAX_VALUE + 1u = ${UInt.MAX_VALUE + 1u}")
            println("(100 + 50).toByte() = ${(100 + 50).toByte()}")
            println("BigDecimal(0.1) = ${BigDecimal(0.1)}")
            println("BigDecimal of text 0.1 = " + BigDecimal("0.1"))
        }
```

```
$ kotlinc Overflow.kt -include-runtime -d Overflow.jar && java -jar Overflow.jar
check(Int.MAX_VALUE) = false
addExact: integer overflow
hash = -2147483648, abs(h) % 10 = -8, floorMod = 2, h.mod(10) = 2
UInt.MAX_VALUE + 1u = 0
(100 + 50).toByte() = -106
BigDecimal(0.1) = 0.1000000000000000055511151231257827021181583404541015625
BigDecimal of text 0.1 = 0.1
```

- Eyni funksiya C-də optimizasiyadan asılı olaraq 0 və ya 1 verirdi, Kotlin-də isə həmişə `false`-dur.
- Kotlin-ə məxsus detal: `%` qalığı bölünənin işarəsini saxlayır (−8). Kotlin 1.5-dən gələn `mod()` isə bölənin işarəsini götürür (2), ona görə massiv indeksi üçün təhlükəsizdir.

- `UInt` da wrap edir: ən böyük dəyər + 1 = 0.

JVM-də overflow-u tutmaq üçün `Math.addExact`, `multiplyExact`, `toIntExact` var. Onlar wrap etmir, `ArithmeticException` atır. Real çıxış (More3.java): `addExact: integer overflow`. Kotlin-də `UInt` və `ULong` (Kotlin 1.5-dən stabil) unsigned riyaziyyat verir. JVM-də ayrıca unsigned tip yoxdur: Kotlin `UInt`-i adi `Int`-in üstündə value class kimi qurur, müqayisə və bölmə üçün xüsusi kod yaradır.

> 🖼️ **Şəkil 54:** Bir adder · iki oxunuş

Burada nəyə bax: dövrə eyni bitləri qaytarır; onların mənasını hansı "eynəklə" oxuduğun müəyyən edir.

> 🖼️ **Şəkil 55:** Mənfi ədədin üç üsulu · iki sıfır problemi

Burada nəyə bax: −0 sətri. Köhnə üsullarda iki sıfır var, two's complement-də yoxdur.

> 🖼️ **Şəkil 56:** Overflow-un dünyaları · JVM, C, Swift/Rust

Burada nəyə bax: eyni ifadə C-də optimizasiyadan asılı olaraq dəyişir, JVM-də isə dəyişmir.

5. Overflow-un real tarixdə izləri

Hadisə	Hesab	Nə oldu
Kotlin `Int` millisaniyə	2^31 ms ≈ 24,86 gün	`days * 24 * 60 * 60 * 1000` 25 gündə mənfi olur: −2134967296 (Bits.java)
Windows 95 və 98	2^32 ms ≈ 49,7 gün	Microsoft KB216641: 49,7 gün fasiləsiz işləyəndən sonra kompüter dona bilər
Boeing 787	248 gün; 2^31 × 0,01 s ≈ 248,55 gün	FAA 2015-ci ildə direktiv verdi: generator idarəetmə bloklarının sayğacı 248 gün fasiləsiz enerjidən sonra daşır; təyyarələr vaxtaşırı tam söndürülməli idi
"Gangnam Style"	2^31 − 1 = 2 147 483 647	YouTube 1 dekabr 2014-də baxış sayğacının 64 bitə keçdiyini elan etdi
2038 problemi	2^31 − 1 saniyə, 1970-dən	Signed 32-bit `time_t` 2038-01-19 03:14:07 UTC-də daşır

Senior bu hekayələri dəqiq danışır:

- **Boeing:** FAA direktivi sayğacın vahidini demir. "Saniyənin yüzdə biri ilə sayan signed 32-bit sayğac" — 248 günün riyazi izahıdır, təsdiq edilmiş fakt yox. Bunu "güman" kimi de.
- **YouTube:** bir YouTube nümayəndəsi sonra Variety-yə sayğacın aylar əvvəl 64 bitə keçirildiyini, sayğacın üstündəki animasiyanın isə "easter egg" olduğunu deyib. Yəni sayğac "sınmayıb", mühəndislər əvvəlcədən düzəldib.
- **2038 və Android:** bionic (Android-in C kitabxanası, Fəsil 6-da) sənədlərinə görə 32-bit Android-də `time_t` 32 bitdir və 2038-də daşacaq; 64-bit ABI-də problem yoxdur. 64-bit tələbəsinin səbəblərindən biri də budur.

> 🖼️ **Şəkil 57:** Dolan sayğaclar · real hadisələr

Burada nəyə bax: beş fərqli sahə, eyni səbəb — sonlu sayğac.

> 🖼️ **Şəkil 58:** Sayğac nə vaxt dolur · maksimum × addım

Burada nəyə bax: hər məşhur rəqəm sadə vurmadan çıxır. Boeing sətri riyazi izahdır, təsdiq edilmiş fakt yox.

> 🖼️ **Şəkil 59:** 2038 · bir saniyədə 1901-ə

Burada nəyə bax: ən böyük dəyərə 1 saniyə əlavə olunanda tarix təxminən 136 il geri qayıdır.

6. Endianness praktikada

- x86-64 və Android-in ARM64 cihazları little-endian işləyir. Bu maşında: `lscpu` → Little Endian; Java → `native = LITTLE_ENDIAN`.
- Şəbəkə protokolları (TCP/IP başlıqları) big-endian istifadə edir. Buna **network byte order** deyilir. C-də `htons` və `htonl` ("host to network") funksiyaları çevirir.
- Java dünyası big-endian-dır: `DataOutputStream` və `ByteBuffer` default olaraq `BIG_ENDIAN` işləyir (Bits.java).
- Klassik Android bug-ı: OpenGL ES-ə və ya native kitabxanaya buffer ötürəndə `.order(ByteOrder.nativeOrder())` unudulur. Native kod byte-ları little-endian oxuyur və `Float`-lar zibilə çevrilir. Android-in rəsmi OpenGL nümunələri məhz bu səbəbdən `ByteBuffer.allocateDirect(...).order(ByteOrder.nativeOrder())` yazır.
- Endianness byte-ların sırasıdır, bitlərin yox. UTF-8 mətndə endianness problemi yoxdur, çünki vahid 1 byte-dır. UTF-16 mətndə var, ona görə UTF-16 faylları başında **BOM** (byte order mark: FE FF və ya FF FE) daşıya bilər.

> 🖼️ **Şəkil 60:** Byte sırası xəritəsi · CPU, şəbəkə, Java, mətn

Burada nəyə bax: telefonun özü little-endian-dır, amma Java və şəbəkə big-endian danışır.

> 🖼️ **Şəkil 61:** BOM · UTF-16 faylının ilk iki byte-ı

Burada nəyə bax: BOM eyni simvoldur (U+FEFF), sadəcə iki sıra ilə yazılıb.

> 🖼️ **Şəkil 62:** OpenGL bug-ı · 1.0f zibilə çevrilir

Burada nəyə bax: eyni dörd byte əks sıra ilə oxunanda 1.0 əvəzinə sıfıra yaxın bir ədəd alınır. Bu ədəd Python-da yoxlanıb: $4{,}6006 \times 10^{-41}$.

7. Fayl imzaları: dizaynın arxasındakı düşüncə

PNG imzası 8 byte-dır: 89 50 4E 47 0D 0A 1A 0A. PNG spesifikasiyasının izah hissəsinə görə hər byte-ın vəzifəsi var:

- **89** — yuxarı biti 1 olan byte. Faylı yalnız 7 bit ötürən köhnə kanal korlayıbsa, bu byte dəyişir və səhv dərhal görünür. Həm də fayl mətn faylı ilə qarışdırılmır.
- **50 4E 47** — ASCII-də "PNG". İnsan gözü ilə tanınır.
- **0D 0A** — CR LF. Windows və Unix arasında sətir sonu çevrilməsi faylı korlayıbsa, aşkar edilir.
- **1A** — DOS-da `type` komandasını dayandıran simvol. Faylı ekrana çıxaranda zibil görünmür.
- **0A** — LF. İstənilməyən əks çevrilməni tutur.

ELF imzası: 7F 45 4C 46 (7F + "ELF"). ELF faylının qalanını Fəsil 3-də açacağıq.

> 🖼️ **Şəkil 63:** PNG imzası · 8 keşikçi

Burada nəyə bax: imzanın hər hissəsi müəyyən bir korlanma növünü tutmaq üçündür.

> 🖼️ **Şəkil 64:** Korlanmış imza · nə dəyişir

Burada nəyə bax: qırmızı byte-lar korlanmanın izidir; hər halda imza orijinala uyğun gəlmir.

> 🖼️ **Şəkil 65:** ELF pasportu · ilk 16 byte

Burada nəyə bax: magic number-dan sonra gələn byte-lar faylın bit enini və byte sırasını deyir. Detallar Fəsil 3-də.

8. Kəsr ədədlər: niyə 0.1 + 0.2 ≠ 0.3

- `Double` 64 bitdir: 1 işarə biti, 11 bit eksponent, 52 bit kəsr hissə. Bu quruluşu **IEEE 754** standartı təyin edir.
- 0.1 binary-də sonsuz təkrarlanan kəsrdir. Decimal-da 1/3 = 0.333... kimi. 52 bitdə kəsilir.
- Real saxlanan dəyər (More3.java): `new BigDecimal(0.1)` = 0.1000000000000000055511151231257827021181583404541015625. Bitləri hex-də: 3fb999999999999a.
- Ona görə 0.1 + 0.2 = 0.30000000000000004 (Bits.java).
- Pul üçün: `Long` (qəpiklə) və ya `BigDecimal("0.1")` — String-dən. `BigDecimal(0.1)` artıq səhv dəyəri götürür.
- Floating point-in tam mexanizmi bu fazanın mövzusu deyil. Bilməli olduğun: kəsr ədəd də sonlu bitdir, və sonlu bitə hər şey sığmır.

> 🖼️ **Şəkil 66:** Sonsuz kəsr · 1/3 və 0,1

Burada nəyə bax: onluqda 1/3 ilə ikilikdə 0,1 eyni problemlə üzləşir.

> 🖼️ **Şəkil 67:** Double-ın quruluşu · 0,1 bitlərdə

Burada nəyə bax: kəsr hissədə 1001 təkrarlanır və 52-ci bitdə yuvarlaqlaşır.

> 🖼️ **Şəkil 68:** Pul və double · BigDecimal və Long

Burada nəyə bax: BigDecimal(0.1) səhv dəyəri dəqiq göstərir; mətndən yaradılan BigDecimal isə düz 0,1-dir.

Bunu az adam bilir: "I" hərfi Azərbaycan dilində fərqli kiçilir

Fakt: `"TITLE".toLowerCase(Locale("az"))` "title" qaytarmır. "tıtle" qaytarır — U+0131, nöqtəsiz ı.

Sübut (More2.java, OpenJDK 21):

```
az lower codepoints: [U+0074, U+0131, U+0074, U+006C, U+0065]
equals title? false
```

Tərsinə də eyni: `"video".toUpperCase(Locale("az"))` = "VİDEO" (U+0130), `equals("VIDEO")` → false (More4.java).

Kotlin-də yoxlama (`Locale.kt`, kotlinc 2.1.0):

```
import java.util.Locale

fun cps(s: String) = s.codePoints().toArray().joinToString(" ") { "U+%04X".format(it) }

fun main() {
    val az = Locale.forLanguageTag("az")
    println("TITLE lowercase(az): " + cps("TITLE".lowercase(az)))
    println("TITLE lowercase():   " + "TITLE".lowercase())
    println("video uppercase(az) == VIDEO ? " + ("video".uppercase(az) == "VIDEO"))
    println("video uppercase()   == VIDEO ? " + ("video".uppercase() == "VIDEO"))
    val dotted = "İ".lowercase()
    println("U+0130 lowercase(): length = ${dotted.length}, " + cps(dotted))
    println("U+0130 lowercase(az): " + cps("İ".lowercase(az)))
}
```

```
$ kotlinc Locale.kt -include-runtime -d Locale.jar && java -jar Locale.jar
TITLE lowercase(az): U+0074 U+0131 U+0074 U+006C U+0065
TITLE lowercase():   title
video uppercase(az) == VIDEO ? false
video uppercase()   == VIDEO ? true
U+0130 lowercase(): length = 2, U+0069 U+0307
U+0130 lowercase(az): U+0069
```

Bu çıxış iki iddianı sübut edir. Arqumentsiz Kotlin `lowercase()` və `uppercase()` cihazın locale-ından asılı deyil. "İ" (U+0130) invariant qaydada kiçiləndə 2 simvol olur (i + U+0307), Azərbaycan locale-ı ilə isə düz "i" olur.

Niyə: Azərbaycan və türk əlifbasında iki cüt var: I ↔ ı və İ ↔ i. Java bu qaydaya düzgün əməl edir.

Real işdə niyə vacibdir:

- Java-nın arqumentsiz `toLowerCase()` və `toUpperCase()` metodları cihazın default locale-ını işlədir. Telefonun dili Azərbaycan dili olanda `"ID".toLowerCase()` "ıd" olur. JSON açarları, `enum.valueOf`, HTTP başlıqları və fayl uzantıları ilə müqayisələr səssizcə sınır — yalnız az və tr istifadəçilərində. Test cihazın ingilis dilindədirsə, bug-ı heç vaxt görməzsən.
- Kotlin 1.5-dən `lowercase()` və `uppercase()` invariant locale ilə işləyir, ona görə texniki açarlar üçün təhlükəsizdir. Köhnə `toLowerCase()` default locale-ı işlədirdi; o, Kotlin 1.5-də deprecated olub, Kotlin 2.1-dən isə istifadəsi xətadır.
- İstifadəçiyə göstərilən mətn üçün locale-lı versiya lazımdır: `lowercase(Locale("az"))`. Texniki açarlar üçün `Locale.ROOT` və ya arqumentsiz Kotlin `lowercase()`.
- Daha bir tələ: Kotlin sənədlərinə görə `'İ'.lowercase()` invariant locale-da "i" + U+0307 (birləşən nöqtə) qaytarır. 1 simvol 2 simvol olur: kiçiltmə mətnin uzunluğunu dəyişə bilər.

> 🖼️ **Şəkil 69:** "I" hərfinin iki ailəsi · ingilis və Azərbaycan

Burada nəyə bax: Azərbaycan əlifbasında nöqtəli və nöqtəsiz "i" ayrı hərflərdir və hər birinin öz cütü var.

> 🖼️ **Şəkil 70:** Locale ilə kiçiltmə · real çıxış

Burada nəyə bax: qırmızı sətirlər locale-dan asılıdır; göy sətir Kotlin-in invariant `lowercase()`-ıdır.

> 🖼️ **Şəkil 71:** az istifadəçilərinə xüsusi bug · və qayda

Burada nəyə bax: bug kodda deyil, telefonun dilində başlayır. Qayda sadədir: texniki açar üçün invariant, istifadəçi mətni üçün istifadəçinin locale-ı.

Yanlış təsəvvürlər

Yanlış təsəvvür	Həqiqət	Özün necə sübut edərsən
Bir hərf bir byte-dır	Yalnız ASCII simvollar üçün. "ə" 2, emoji 4 byte-dır	`printf 'ə'` çıxışını `od -t x1`-ə ötür: c9 99
`String.length` hərflərin sayıdır	UTF-16 code unit-lərin sayıdır	Bits.java: `length = 2, codePoints = 1`
Code point sayı = ekrandakı simvol sayı	👍🏽 2 code point-dir, ekranda 1 işarə	`codePointCount` ilə yoxla, sonra BreakIterator ilə müqayisə et
Overflow olanda proqram xəta verir	JVM-də səssiz wraparound, C-də undefined behavior	`Int.MAX_VALUE + 1` çap et: −2147483648
`Math.abs` həmişə müsbət qaytarır	`abs(Int.MIN_VALUE)` mənfidir	Bits.java: `abs(MIN) = -2147483648`
Byte 0–255 arası ədəddir	Java və Kotlin-də `Byte` −128…127-dir; 0–255 üçün `UByte` var	`"ə".toByteArray()[0]` çap et: −55

Yanlış təsəvvür	Həqiqət	Özün necə sübut edərsən
1 KB həmişə 1024 byte-dır	KB = 1000, KiB = 1024. 1 TB disk OS-də təxminən 931 GiB görünür	10^12 / 2^30 = 931,32
Hex, binary və decimal fərqli ədədlərdir	Eyni ədədin fərqli yazılışıdır	`printf("%x %d", 255, 0xFF)` → ff 255
Endianness bitlərin sırasıdır	Byte-ların sırasıdır	`struct.pack('<I', 0x12345678).hex()` → 78563412
UTF-8 mətndə endianness problemi var	Yoxdur, vahid 1 byte-dır; UTF-16-da var	UTF-16 faylının ilk 2 byte-ına bax: FE FF və ya FF FE
`0.1 + 0.2 == 0.3`	false; 0.1 binary-də dəqiq yazılmır	`println(0.1 + 0.2)` → 0.30000000000000004
C-də overflow da wraparound-dur	Undefined behavior; nəticə kompilyatordan və bayraqdan asılıdır	`ub.c`: clang -O0 → 0, clang -O2 → 1
`toLowerCase()` hər yerdə eyni nəticə verir	Default locale-dan asılıdır; az və tr-də I → ı	More2.java: `equals title? false`

İntervyuda

Hər model cavab altı hissədən ibarətdir: **Qısa cavab → Mexanizm → Rəqəm və sübut → Trade-off → Real nümunə → Fərqləndirici detal**. Bu formanı Mock interview-da ayrıca öyrənəcəksən. Burada əvvəlcə cavabı özün yaz, sonra modellə müqayisə et. Sonda "follow-up nərdivanı" var: interviewer-in növbəti sualları, hər biri bir qat dərin.

İ0.1 — Proqramçılar niyə hex istifadə edir? `FLAG_ACTIVITY_NEW_TASK = 0x10000000` nəyi bildirir?

- **Qısa cavab:** Bir hex rəqəm düz 4 bitə uyğun gəlir, ona görə hex bit naxışlarını yığcam və oxunaqlı göstərir. 0x10000000 — yalnız 28-ci biti (0-dan saymaqla) 1 olan 32 bitlik ədəddir; flag kimi işlənir.
- **Mexanizm:** 16 = 2^4, deməli 1 hex rəqəm = 4 bit, 2 hex rəqəm = 1 byte. 0x10000000-da 8 hex rəqəm var = 32 bit. Ən soldakı rəqəm 1 = 0001, yəni bitlər 31, 30, 29 sıfır, bit 28 bir. Intent flag-ləri bir `Int`-in içində 32 müstəqil açar kimi saxlanır. Birləşdirmək `or`, yoxlamaq `and` ilədir.
- **Rəqəm və sübut:** 0x10000000 = 268435456 = 2^28 (More.java: `FLAG_ACTIVITY_NEW_TASK = 268435456, bit = 28`). `Integer.numberOfTrailingZeros(0x10000000)` = 28, `Integer.bitCount` = 1 (More4.java).
- **Trade-off:** Bitmask yığcamdır (32 flag 4 byte-da) və sürətlidir (bir `and` instruction-ı). Amma tip təhlükəsizliyi zəifdir: funksiyaya istənilən `Int` ötürmək olar. Android bunu `@IntDef(flag = true)` annotasiyası ilə lint səviyyəsində qoruyur. Kotlin-də oxunaqlı alternativ `EnumSet`-dir, amma o, obyektdir və daha çox yer tutur.
- **Real nümunə:** `intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK` yazmaq əvvəlki flag-ləri silir. Düzgünü `intent.addFlags(...)` və ya `intent.flags = intent.flags or ...`-dır. Notification-dan Activity açanda NEW_TASK və CLEAR_TOP birlikdə lazım olur və biri digərini silməməlidir.
- **Fərqləndirici detal:** Hex byte sərhədlərini də göstərir: C9 99 — hər iki rəqəm bir byte. Unix fayl icazələri isə octal-dır (0755): hər rəqəm 3 bit (rwx). Tələ: Java-da `010` octal-dır və 8-ə bərabərdir (More4.java: `java octal 010 = 8`). Kotlin-də isə octal literal ümumiyyətlə yoxdur — bu tələni qəsdən bağlayıblar.

Follow-up nərdivanı:

1. *"Bir flag-i necə söndürərsən?"* → `flags and FLAG.inv()`: FLAG-ın biti 0, qalanları 1 olan maska ilə AND.
2. *"Bir ədəddə neçə 1 bit olduğunu necə taparsan?"* → `Integer.bitCount` (Kotlin: `countOneBits()`). x86-64-də bunu bir `popcnt` instruction-ı edir.
3. *"`x and (x - 1)` nə edir?"* → Ən sağdakı 1 biti söndürür. Nəticə 0-dırsa, x 2-nin qüvvətidir (Nümunə 3).
4. *"HashMap-in ölçüsü niyə 2-nin qüvvətidir?"* → İndeks `(n - 1) and hash` ilə bir instruction-la alınır və heç vaxt mənfi olmur. `%` üçün bölmə instruction-ı lazımdır və o, xeyli yavaşdır (Fəsil 1-də). Java HashMap əlavə olaraq `hash ^ (hash >>> 16)` ilə yuxarı bitləri aşağı bitlərə qarışdırır, çünki maska yalnız aşağı bitlərə baxır.
5. *"Bu flag-lər başqa prosesdəki system_server-ə necə gedir?"* → Parcel-ə 4 byte-lıq `Int` kimi yazılır və Binder ilə ötürülür (Faza 8-də).

İ0.2 — Two's complement nədir və niyə istifadə olunur?

- **Qısa cavab:** Mənfi ədədi bitlərlə yazma üsuludur: −x = (x-in bitlərini çevir) + 1. Toplama və çıxma signed və unsigned ədədlər üçün eyni dövrə ilə işləyir, sıfır isə təkdir.
- **Mexanizm:** 8 bitdə 5 = 0000 0101 → NOT → 1111 1010 → +1 → 1111 1011 = −5 (Bits.java: `-5 binary = 11111011`). Başqa baxış: ən böyük bitin çəkisi −128-dir. 1111 1011 = −128 + 64 + 32 + 16 + 8 + 2 + 1 = −128 + 123 = −5. n bitin aralığı: −2^(n−1) ... 2^(n−1) − 1.
- **Rəqəm və sübut:** `(byte)(127 + 1)` = −128; `Int.MAX_VALUE + 1` = −2147483648; 8 bitdə 100 + 50 = −106 (More.java: `100+50 as byte = -106`).
- **Trade-off:** Aralıq asimmetrikdir, ona görə `abs(MIN)` və `-MIN` overflow edir. Alternativ: unsigned tiplər (Kotlin `UInt`) — mənfi olmayan sayğaclar üçün yaxşıdır, amma Java API-ləri ilə işləmək çətinləşir. Çox böyük ədədlər üçün `Long` və ya `BigInteger`.
- **Real nümunə:** `Math.abs(key.hashCode()) % shards` ilə shard seçilir. `"polygenelubricants".hashCode()` dəqiq `Int.MIN_VALUE`-dur. `abs` mənfi qalır, `% 10` = −8 olur və massivdə `ArrayIndexOutOfBoundsException` atılır. Düzəliş: `Math.floorMod(h, n)` (= 2) və ya `(h and Int.MAX_VALUE) % n`. Sübut (More3.java): `hash = -2147483648, abs = -2147483648, abs % 10 = -8, floorMod = 2`.
- **Fərqləndirici detal:** Ölçü 16 olsa, bug görünməz: `abs(MIN) % 16 = 0` (More.java), çünki −2^31 16-ya qısıqsız bölünür. 2-nin qüvvəti ölçülü cədvəllər bug-ı gizlədir, ölçü 10 olanda partlayır. Bu, testdə tutulmayan, production-da bir gün çıxan bug-dır.

Follow-up nərdivanı:

1. *"C-də signed overflow nə olur?"* → Undefined behavior. `ub.c`: clang -O0 → 0, clang -O2 → 1; `-fwrapv` ilə həmişə 0.
2. *"Overflow-u necə aşkar edərsən?"* → JVM-də `Math.addExact` (`ArithmeticException: integer overflow`). CPU-da toplamadan sonra overflow flag-ı (OF) qalxır (Fəsil 1-də).
3. *"`shr` və `ushr` fərqi nədir?"* → `shr` (Java `>>`) işarə bitini kopyalayır, `ushr` (Java `>>>`) sola 0 qoyur. −8 shr 1 = −4; −8 ushr 1 = 2147483644 (More4.java).
4. *"Byte-ı Int-ə çevirəndə nə olur?"* → Sign extension: işarə biti yuxarı 24 bitə kopyalanır. 0..255 almaq üçün `and 0xFF`.

5. *"Kotlin `UInt` JVM-də necə işləyir?"* → Value class-dır, içində adi `Int` saxlanır. Toplama eynidir (two's complement sayəsində!). Müqayisə və bölmə fərqlidir, ona görə kompilyator xüsusi funksiyalar çağırır: məsələn, müqayisədən əvvəl hər iki ədədin işarə bitini çevirir.

İ0.3 — `"😀".length` niyə 2-dir? Mətni düzgün necə kəsərsən?

- **Qısa cavab:** Kotlin və Java `String`-i UTF-16 code unit-lərindən ibarətdir. 😀 (U+1F600) BMP-dən kənardadır və iki unit tutur — surrogate pair. `length` code unit sayar.
- **Mexanizm:** 0x1F600 − 0x10000 = 0xF600; yuxarı 10 bit + 0xD800 = D83D, aşağı 10 bit + 0xDC00 = DE00. Tarixçə: əvvəlcə Unicode-un 16 bitə sığacağı güman edilirdi, Java da buna görə 16 bitlik `char` seçdi. Unicode 2.0 (1996) 16 biti aşdı və surrogate-ləri gətirdi. Java 5 (2004) `codePointAt` kimi code point API-lərini əlavə etdi, amma `length` artıq dəyişilə bilməzdi.
- **Rəqəm və sübut:** `"😀".length` = 2, code point = 1; 👍🏽 — length 4, code point 2, UTF-8-də 8 byte; `"😀👍🏽"` — length 6, code point 3 (More4.java); "Bakı" — 4 hərf, 5 byte.
- **Trade-off:** UTF-16 BMP daxilində indeksləməni sadə edir, amma bu yanlış təhlükəsizlik hissi yaradır. UTF-8 ASCII-ağır mətnlərdə yaddaşa qənaət edir və şəbəkə ilə faylların standartıdır. JDK 9-dan Compact Strings (JEP 254) Latin-1 mətnləri daxildə simvol başına 1 byte saxlayır, amma API UTF-16 olaraq qalır.
- **Real nümunə:** Bildiriş mətnini və ya username-i `take(20)` ilə kəsmək yarım emoji və "?" verir. Server "20 simvol" limitini code point ilə, client `length` ilə sayanda validasiya bir tərəfdə keçir, o biri tərəfdə keçmir. Düzəliş: grapheme sərhədinə görə kəsmək (`BreakIterator`) və limitin vahidini API müqaviləsində yazmaq.
- **Fərqləndirici detal:** NDK-da `NewStringUTF` "modified UTF-8" gözləyir. Standart UTF-8 emoji (F0 9F 98 80) CheckJNI ilə `illegal start byte 0xf0` xətası verir.

Follow-up nərdivanı:

1. *"Code point-ləri necə sayarsən?"* → `codePointCount(0, length)` və ya `codePoints().count()`.
2. *"Server 255 byte limit qoyub. Client nəyi saymalıdır?"* → `toByteArray(Charsets.UTF_8).size`. "Bakı" 4 deyil, 5-dir.
3. *"Mətni byte limitinə görə kəsəndə nəyə diqqət edərsən?"* → 10xxxxxx (davam byte-ı) üzərində kəsmə. Geri qayıdıb simvolun ilk byte-ında kəs, sonra grapheme sərhədini də yoxla.
4. *"Unicode normalization nədir?"* → "é" iki cür yazıla bilər: U+00E9 və ya "e" + U+0301. Ekranda eynidir, `equals` isə false-dur. Müqayisədən əvvəl `Normalizer.normalize(s, Normalizer.Form.NFC)`.
5. *"Regex-də `.` emojini tutur?"* → Java regex code point ilə işləyir: `.` bir code point tutur, grapheme cluster-i yox. Grapheme cluster üçün `\X` var (Java 9-dan).

İ0.4 — Endianness nədir və Android developer-ə nə vaxt lazımdır?

- **Qısa cavab:** Çox-byte-lı ədədin byte-larının yaddaşdakı sırasıdır. Little-endian kiçik byte-ı əvvəl, big-endian böyük byte-ı əvvəl yazır. x86-64 və Android ARM64 little-endian-dır; şəbəkə və Java stream-ləri big-endian.
- **Mexanizm:** 0x12345678 little-endian-da 78 56 34 12, big-endian-da 12 34 56 78 kimi yazılır. Register-in içində "sıra" yoxdur: byte-ları yaddaşdan register-ə yükləyən instruction onları doğru yerə qoyur. Ad Gülliverin səyahətlərindən gəlir: yumurtanı hansı ucundan sındırmaq üstündə müharibə. Kompüterlərə bu adı Danny Cohen 1980-ci ildə "On Holy Wars and a Plea for Peace" məqaləsində gətirib.

- **Rəqəm və sübut:** Python `struct.pack` çıxışı: `78563412 12345678 little`; Java: `ByteBuffer default = BIG_ENDIAN, native = LITTLE_ENDIAN`.
- **Trade-off:** Heç biri "sürətli" deyil. Little-endian-da eyni ünvandan həm 32 bitlik, həm 8 bitlik dəyəri oxumaq olur (ilk byte ən kiçik hissədir). Big-endian hex dump-da insan üçün düz oxunur. Əsas qayda: fayl və şəbəkə formatının endianness-ini həmişə açıq yaz.
- **Real nümunə:** OpenGL ES və TFLite üçün `ByteBuffer.allocateDirect(n).order(ByteOrder.nativeOrder())`. Binary format parse edəndə hər formatın öz qaydısı var: WAV little-endian-dır, PNG-nin chunk uzunluqları big-endian.
- **Fərqləndirici detal:** ARM64 hardware-i iki rejimi də dəstəkləyə bilər, amma Android yalnız little-endian işlədir. Byte sırasını çevirmək bir instruction-dır: x86-64-də `bswap`, ARM64-də `rev`. Java-da `Integer.reverseBytes` JIT tərəfindən bu instruction-a çevrilə bilər.

Follow-up nərdivanı:

1. *"Maşının endianness-ini kodla necə yoxlayarsən?"* → `ByteOrder.nativeOrder()`. C-də: `int x = 1;` yaz, ilk byte-ına bax: 1-dirsə little-endian.
2. *"UTF-8 faylında BOM lazımdır?"* → Lazım deyil. EF BB BF ola bilər, amma endianness bildirmir və bəzi parser-ləri sındırır (məsələn, JSON-un ilk simvolu gözlənilən `{` olmur).
3. *"Network byte order niyə big-endian-dır?"* → Tarixi qərardır: ilk İnternet protokolları belə təyin edib. İndi dəyişmək mümkün deyil, çünki bütün cihazlar bu formada danışır.
4. *"C struct-ını fayla birbaşa yazmaq niyə təhlükəlidir?"* → Endianness və padding (sahələr arasındakı boş byte-lar, Fəsil 2-də) platformadan asılıdır. Formatı sahə-sahə açıq yazmaq lazımdır.

Bug-ı tap

Hər kodda gizli bir səhv var. Əvvəl nə baş verdiyini yaz, sonra düzəlişi. Cavablar Cavab açarındadır.

B0.1 — PNG yoxlaması heç vaxt true qaytarmır:

```
fun isPng(header: ByteArray): Boolean =
    header.size >= 4 &&
    header[0].toInt() == 0x89 &&
    header[1] == 'P'.code.toByte() &&
    header[2] == 'N'.code.toByte() &&
    header[3] == 'G'.code.toByte()
```

B0.2 — "25 gündən sonra xatırlat" dərhal işləyir:

```
val days = 25
val delayMs = days * 24 * 60 * 60 * 1000
handler.postDelayed(reminder, delayMs.toLong())
```

B0.3 — Yalnız Azərbaycan dilli telefonlarda crash:

```
enum class MediaType { IMAGE, VIDEO, AUDIO }

fun parse(fromServer: String): MediaType =
    MediaType.valueOf(fromServer.uppercase(Locale.getDefault()))
// server "video" göndərir
```

Öz sözlərinlə izah et

Bu fəsli bir dostuna 5 cümlə ilə izah et — yaz və ya səslə danış. Yaxşı izahda bunlar var:

- [ ] Bit nədir və niyə iki vəziyyət kifayətdir.
- [ ] n bit 2^n vəziyyət verir; byte = 8 bit = 256 vəziyyət.
- [ ] Hex niyə rahatdır: 1 rəqəm = 4 bit.
- [ ] Mətn zənciri: simvol → code point → encoding (UTF-8 və ya UTF-16) → byte-lar.
- [ ] Mənfi ədəd two's complement ilə yazılır; aralıq asimmetrikdir.
- [ ] Overflow JVM-də wrap edir, C-də undefined behavior-dur.
- [ ] Endianness byte-ların sırasıdır.
- [ ] Bir konkret nümunə: "ə" = C9 99.

Boş xəritə: aşağıdakı xəritəni kağıza köçür. Hər budaq üçün qeydə baxmadan bir cümlə yaz. Sonra fəslin sonundakı Xülasə şəkli ilə müqayisə et.

> 🖼️ **Şəkil 72:** Boş xəritə · öz sözlərinlə doldur

Keçid qapısı

Qeydə baxmadan cavabla, sonra Cavab açarı ilə yoxla. 8 sualdan ən azı 7-sini rubrikanın 3-cü səviyyəsində (mexanizmi ilə) cavablaya bilirsənsə, Fəsil 1-ə keç. Bu ilk fəsil olduğu üçün əvvəlki fəsildən sual yoxdur.

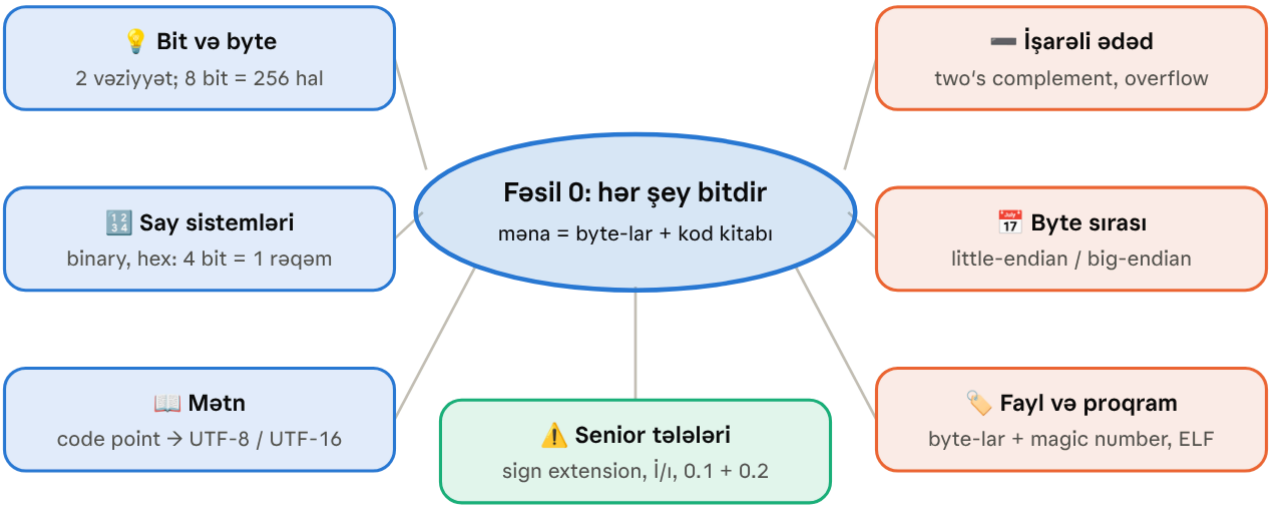
1. **K0.1 (hesabla):** 156-nı binary-yə və hex-ə çevir.
2. **K0.2 (nəticəni təxmin et):** `"Bakı".length` və `"Bakı".toByteArray(Charsets.UTF_8).size` nə qaytarır? Niyə fərqlidir?
3. **K0.3 (nəticəni təxmin et):** 8 bitlik signed ədəddə 100 + 50 nə verir? Hesabı bitlərlə göstər.
4. **K0.4 (niyə):** `Math.abs(Int.MIN_VALUE)` niyə mənfidir?
5. **K0.5 (bug-ı tap):** UTF-8 BOM yoxlaması `if (bytes[0].toInt() == 0xEF)` niyə işləmir? Düzəlt.
6. **K0.6 (nəticəni təxmin et):** 0x12345678 ədədi Android telefonun yaddaşında yazılanda ilk byte hansıdır?
7. **K0.7 (dizayn qərarı):** Server username üçün "maksimum 32 byte" limit qoyur. Client-də nəyi sayarsən və kəsmək lazım olsa necə kəsərsən?
8. **K0.8 (nəticəni təxmin et):** `"😀👍".length` və code point sayı nədir?

Yadda saxla

- Bit iki vəziyyətdir; n bit 2^n vəziyyət verir; byte 8 bitdir.
- Binary, decimal və hex eyni ədədin fərqli yazılışıdır; 1 hex rəqəm = 4 bit.
- Mətn = code point-lər + encoding. UTF-8 1–4 byte işlədir; JVM `String`-i UTF-16-dır, emoji 2 `Char` tutur.
- Mənfi ədədlər two's complement-dir; overflow JVM-də səssiz wrap, C-də undefined behavior. Kotlin `Byte` signed-dir: `and 0xFF`.
- Endianness byte sırasıdır: telefon little-endian, şəbəkə və Java stream-ləri big-endian.

Bir sətirlik mental model: Kompüterdə hər şey byte-dır; məna byte-da deyil, ona baxış qaydasındadır.

Xülasə şəkli



Bir cümlə: kompüterdə hər şey bitdir, mənanı isə bitləri oxuyanın kod kitabı verir. Bu xəritəni yaddaşdan çəkə bilirsənsə, Fəsil 1-ə hazırsan.

Fəsil 1 — CPU: proqramı icra edən maşın

Bu fəsildən sonra bacaracaqsan

- CPU-nun bir instruction-ı necə götürüb icra etdiyini addım-addım izah etmək: fetch → decode → execute.
- Register, PC, SP və flag-lərin nə olduğunu demək və onları debugger-də görmək.
- Sadə C funksiyasının x86-64 və ARM64 assembly-sini oxumaq.
- `count++`-ın CPU üçün niyə bir addım olmadığını sübut etmək.
- Core, hardware thread (SMT) və clock tezliyini rəqəmlərlə izah etmək.

Əvvəlcə təxmin et

Cavabı kağıza yaz, sonra oxumağa davam et. Cavablar Cavab açarındadır.

1. **T1.1** CPU bir saniyədə təxminən neçə sadə əməliyyat edə bilər: min, milyon, yoxsa milyard?
2. **T1.2** Kotlin-də `count++` CPU üçün neçə addımdır: bir, yoxsa bir neçə?
3. **T1.3** Telefonunda 8 core var. Bu, hər tətbiqin 8 dəfə sürətli işləməsi deməkdir?

Sirr

Bu maşında kiçik bir `add(2, 3)` funksiyasını debugger-də dayandırdıq. CPU-nun "barmağı" (PC) 0x555555555129 ünvanında idi. CPU bir instruction icra etdi və barmaq 0x55555555512d-yə keçdi — düz 4 byte irəli. Bir register-də 5 yarandı. Sonra `ret` instruction-ı barmağı tamam başqa yerə, `main` funksiyasına atdı.

Bu barmaq nədir? Niyə məhz 4 byte? CPU hara qayıtmalı olduğunu haradan bilir? Bu fəsil bu sualları cavablayır.

> 🖼️ **Şəkil 74:** Sirr · PC barmağı və ret

Burada nəyə bax: barmaq bir dəfə 4 byte irəliləyir, sonra tamam başqa yerə atılır.

Yeni sözlər

CPU, instruction, machine code, assembly, ISA, register, PC, SP, flag, ALU, LR, clock, cycle, GHz, IPC, fetch, decode, execute, core, SMT, hardware thread, pipeline, x86-64, ARM64.

Uşaq səviyyəsi

Təsəvvür et ki, mətbəxdə çox sürətli, amma çox itaətkar bir aşpaz var. O, resept kitabını sətir-sətir oxuyur və hər sətirdə yazılanı dəqiq edir. Barmağı həmişə oxuduğu sətirin üstündədir.

Aşpazın əlinin altında bir neçə kiçik qab var. Onlar azdır, amma dərhal əlçatandır. Məhsulların çoxu isə uzaqdakı anbardadır. Bir metronom tiq-tiq edir və aşpaz hər tiqə bir hərəkət edir.

> 🖼️ **Şəkil 75:** Uşaq səviyyəsi · mətbəx, aşpaz, anbar

Burada nəyə bax: qablar əlin altındadır, anbar uzaqdadır. Bu məsafə Fəsil 2-nin əsas mövzusudur.

> 🖼️ **Şəkil 76:** Bir neçə aşpaz · core və SMT

Burada nəyə bax: dörd aşpaz dörd iş görür; iki barmaqlı aşpaz isə eyni alətləri iki resept arasında bölür.

Analogiyada	Real sistemdə
Aşpaz	CPU core
Resept kitabı	Proqram: yaddaşdakı instruction-lar
Kitabdakı bir sətir	Bir instruction
Oxunan sətri göstərən barmaq	PC (program counter)
Əlin altındakı kiçik qablar	Register-lər
Uzaqdakı anbar	RAM (Fəsil 2-də)
Bıçaq və ocaq	ALU: hesablayan dövrə
Kitabın kənarında "son nəticə sıfır idi" qeydi	Flag-lər
Metronomun bir tiqqiltısı	Bir clock cycle
Mətbəxdə bir neçə aşpaz	Bir neçə core
Bir aşpazın iki barmaqla iki resepti növbəli oxuması	SMT (Hyper-Threading)

Analogiya harada sınır:

- Aşpaz reseptin mənasını anlayır. CPU heç nə "anlamır": bit naxışları elektrik dövrəsində müəyyən yolları açıb-bağlayır, vəssalam.
- Aşpaz bir anda bir iş görür. Real CPU eyni anda bir neçə instruction-ı müxtəlif mərhələlərdə saxlayır (pipeline, Senior səviyyəsində).
- Qabların adları sabitdir: x0, x1, ... Resept hər addımda hansı qabı işlətdiyini adı ilə deyir.

> 🖼️ **Şəkil 77:** Analogiyanın sərhədi · CPU anlamır

Burada nəyə bax: CPU üçün "məna" yoxdur, yalnız bit naxişı və dövrə var.

Mühəndis səviyyəsi

1. Instruction, machine code, assembly və ISA

- **CPU** (Central Processing Unit, mərkəzi prosessor) — proqramı icra edən çipdir.
- **Instruction** — CPU-nun bir sadə əmri: "iki ədədi topla", "yaddaşdan oxu", "başqa ünvana keç". Hər instruction byte-larla kodlanır. Fəsil 0-dan bilirsən: kompüterdə hər şey byte-dır, proqram da.

- **Machine code** — bu byte-ların özüdür. ARM64-də `add x0, x1, x0` instruction-ı 32 bitlik 0x8b000020 sözüdür.
- **Assembly** — machine code-un insan üçün oxunan yazılışıdır. Bir assembly sətri = bir instruction.
- **ISA** (Instruction Set Architecture) — CPU-nun başa düşdüyü instruction-ların tam siyahısı və qaydalarıdır. Proqram ilə hardware arasında müqavilədir. x86-64 (noutbuk və server-lər) və ARM64 (Android telefonlar) iki fərqli ISA-dır. Birinin machine code-u o birində işləmir.

> 🖼️ **Şəkil 78:** ISA · eyni fikir, iki dil

Burada nəyə bax: eyni toplama x86-64-də 4 byte-lıq `lea`, ARM64-də 4 byte-lıq `add` olur, amma byte-lar heç oxşamır. Bu byte-lar "Özün gör" bölməsindəki real `objdump` çıxışındandır.

> 🖼️ **Şəkil 79:** Koddan CPU-ya · dörd qat

Burada nəyə bax: assembly ilə machine code eyni instruction-ın iki yazılışıdır.

> 🖼️ **Şəkil 80:** 0x8b000020 · instruction-ın bit sahələri

Burada nəyə bax: 32 bit ayrı-ayrı sahələrə bölünür; decode hər sahəni oxuyub nə ediləcəyini müəyyən edir.

Dayan və düşün D1.1: Telefon üçün kompilyasiya olunmuş `.so` faylı noutbukun x86-64 prosessorunda niyə birbaşa işləmir?

2. Register-lər

- **Register** — CPU-nun içində, hesablayan dövrənin yanında olan kiçik və ən sürətli yaddaş xanasıdır. ARM64-də və x86-64-də hər biri 64 bitdir.
- Sayı çox azdır: ARM64-də 31 ümumi təyinatlı register var (x0–x30), x86-64-də 16 (rax, rbx, ..., r15).
- Hesablama register-lərdə aparılır. Tipik ritm: yaddaşdan register-ə oxu → hesabla → nəticəni yaddaşa yaz.
- **ALU** (Arithmetic Logic Unit) — toplama, çıxma, AND, OR, müqayisə edən dövrədir. Girişini register-lərdən alır, nəticəni register-ə yazır.

Xüsusi register-lər:

- **PC** (program counter; x86-64-də adı RIP) — icra olunacaq instruction-ın ünvanı. "Sirr"dəki barmaq məhz budur.
- **SP** (stack pointer; x86-64-də RSP) — stack-in ucunun ünvanı. Stack funksiya çağırışları üçün ayrılmış yaddaş sahəsidir (Fəsil 4-də tam).
- **Flag-lər** — son əməliyyatın nəticəsi haqqında bir bitlik qeydlər. ARM64-də dördü var: N (negative, mənfi), Z (zero, sıfır), C (carry, daşma), V (overflow). x86-64-də adları SF, ZF, CF, OF-dur. Fəsil 0-dakı `jl` və `jb` məhz bunlara baxır.

- **LR** (link register) — ARM64-də x30-un ikinci adı. Funksiya çağırılanda qayıdış ünvanı bura yazılır. x86-64-də belə register yoxdur: `call` qayıdış ünvanını stack-ə qoyur.

> 🖼️ **Şəkil 81:** Hesablama ritmi · load → ALU → store

Burada nəyə bax: ALU-ya çatmaq üçün data əvvəlcə register-ə gətirilməlidir.

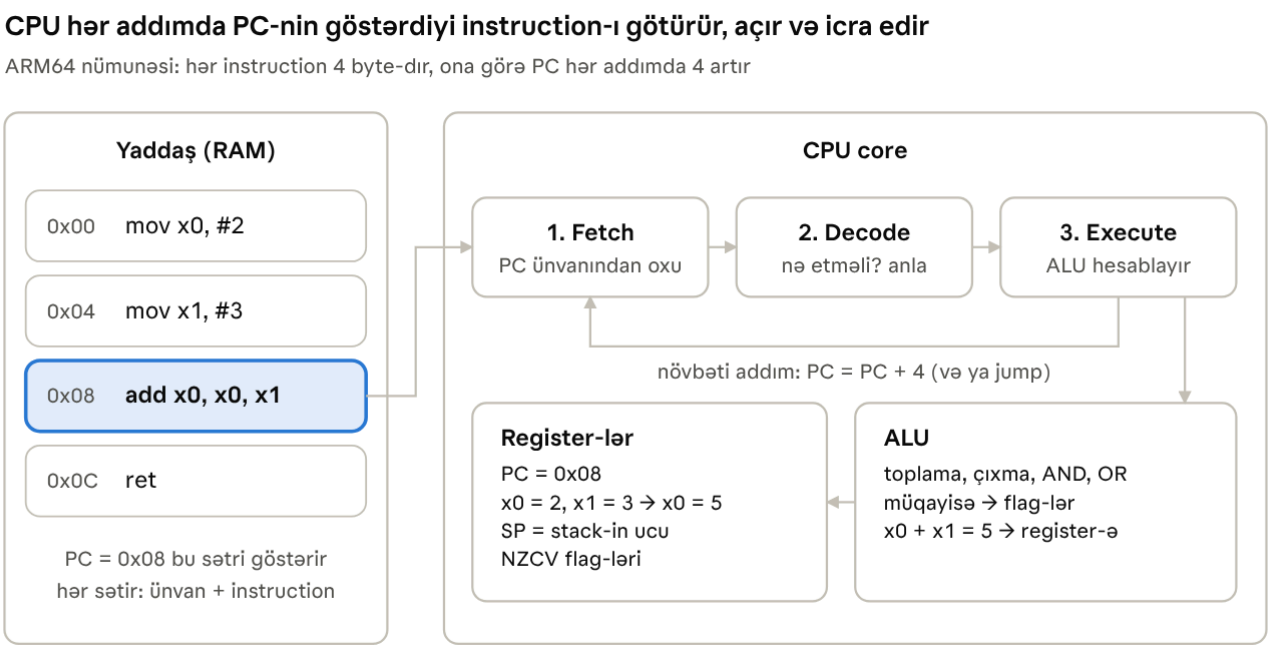
> 🖼️ **Şəkil 82:** Register xəritəsi · ARM64 və x86-64

Burada nəyə bax: ARM64-də register çoxdur və qayıdış ünvanı üçün xüsusi LR var; x86-64-də yoxdur.

> 🖼️ **Şəkil 83:** Flag-lər · cmp sonrası N və Z

Burada nəyə bax: eyni cmp fərqli ədədlərlə fərqli flag-ləri yandırır.

3. Fetch → decode → execute



CPU işlədikcə bu dövrü dayanmadan təkrarlayır:

1. **Fetch** (götür): PC-dəki ünvandan instruction-ın byte-larını oxu.
2. **Decode** (aç): byte-lara bax və müəyyən et — "bu toplama əmridir, x0 və x1 ilə".
3. **Execute** (icra et): ALU hesablayır. Nəticə register-ə, yaddaşa və ya PC-yə yazılır.
4. **PC növbəti instruction-a keçir.** ARM64-də hər instruction 4 byte-dır, ona görə PC = PC + 4. Instruction jump (keçid) idisə, PC yeni ünvanı alır.

Bu kiçik proqramı addım-addım izlə:

Addım	PC (əvvəl → sonra)	Instruction	x0	x1	Nə oldu
0	0x00	—	?	?	Başlanğıc
1	0x00 → 0x04	`mov x0, #2`	2	?	x0-a 2 yazıldı
2	0x04 → 0x08	`mov x1, #3`	2	3	x1-ə 3 yazıldı
3	0x08 → 0x0C	`add x0, x0, x1`	5	3	ALU topladı, nəticə x0-a
4	0x0C → LR	`ret`	5	3	PC = x30-dakı qayıdış ünvanı

> 🖼️ **Şəkil 85:** Dörd addım · kino lenti

Burada nəyə bax: cədvəldəki hər sətir bir kadrdır; rəngli sətir həmin addımda dəyişən yerdir.

> 🖼️ **Şəkil 86:** PC-nin yolu · +4 və ret

Burada nəyə bax: ardıcıl kodda PC 4-4 artır, ret isə onu LR-dəki ünvana atır.

Dayan və düşün D1.2: 3-cü addımdan sonra x1 niyə hələ də 3-dür?

4. Clock, cycle və GHz

- **Clock** — CPU-ya saniyədə milyardlarla "tiq" verən elektrik siqnalıdır. İki tiq arasındakı vaxta **cycle** deyilir.
- **GHz** (giqahers) — saniyədə bir milyard cycle. 2,1 GHz-də bir cycle = 1 / 2 100 000 000 saniyə ≈ 0,48 ns. İşıq bu müddətdə cəmi təxminən 14 sm gedir.
- Bir instruction həmişə bir cycle demək deyil. Vurma təxminən 3 cycle çəkir. Müasir CPU isə bir cycle-da bir neçə instruction bitirə bilir. Ortalamaya **IPC** (instructions per cycle) deyilir.
- Müqayisə üçün: ilk mikroprosessor Intel 4004 (1971) 740 kHz-də işləyirdi. Bu maşının nominal tezliyi (2,1 GHz) ondan təxminən 2800 dəfə çoxdur.
- Tezlik sabit deyil: CPU yüklənəndə qalxır (turbo), qızınanda enir. Telefonda bu daha kəskindir (Senior səviyyəsində).

> 🖼️ **Şəkil 87:** Clock · metronom və cycle

Burada nəyə bax: dəlgənin bir "dişi" bir cycle-dır.

> 🖼️ **Şəkil 88:** Instruction və cycle · IPC

Burada nəyə bax: vurma bir neçə cycle çəkir, amma bir cycle-da bir neçə sadə instruction bitə bilər.

> 🖼️ **Şəkil 89:** Bir cycle · işıq 14 sm

Burada nəyə bax: bir cycle o qədər qısadır ki, işıq bu müddətdə telefonun boyu qədər yol gedir.

Dayan və düşün D1.3: 3 GHz CPU bir saniyədə neçə cycle edir? Bir cycle neçə nanosaniyədir?

5. Core və SMT

- **Core** — fetch-decode-execute maşınının tam bir nüsxəsidir: öz PC-si, öz register-ləri, öz ALU-ları. 8 core = eyni anda 8 müstəqil instruction axını.
- **SMT** (Simultaneous Multithreading; Intel-də adı Hyper-Threading) — bir core-un içində iki dəst register və iki PC. Core iki instruction axınını qarışdırıb icra edir, ALU-ları paylaşır. Hər axına **hardware thread** deyilir.
- Diqqət: hardware thread ilə əməliyyat sisteminin thread-i başqa şeylərdir. OS thread-ini Faza 3-də öyrənəcəksən. Orada görəcəksən ki, OS minlərlə thread-i bir neçə hardware thread üzərində növbə ilə işlədir (Faza 2).
- Bu maşın: `lscpu` → `Thread(s) per core: 1`, `Core(s) per socket: 2`. Telefon core-larında adətən SMT yoxdur.
- Core çox olsa da, tətbiq onlardan yalnız işini paralel bölə bildiyi qədər faydalanır. Bunun qanununu (Amdahl) Faza 11-də görəcəksən.

> 🖼️ **Şəkil 90:** Çip və core-lar · bina və mətbəxlər

Burada nəyə bax: hər mətbəxin öz aşpazı var; SMT-li mətbəxdə bir aşpaz iki resepti növbə ilə oxuyur.

> 🖼️ **Şəkil 91:** SMT-nin içi · iki PC, bir icra hissəsi

Burada nəyə bax: SMT-də yalnız PC və register-lər ikiqatdır; icra hissəsi birdir.

> 🖼️ **Şəkil 92:** Üç mərtəbə · OS thread, hardware thread, core

Burada nəyə bax: proqramın onlarla thread-i CPU-nun bir neçə hardware thread-inə növbə ilə düşür.

Özün gör

Mühit: Linux 6.18, x86-64 virtual maşın, gcc 13.3, clang 18.1.3, gdb. Çıxışlar real-dır.

1. CPU-nu tanı

```
$ lscpu | grep -E '^(Architecture|Byte Order|CPU.s.:|Model
name|Thread|Core|Hypervisor|L1d|L2|L3)'
Architecture:                            x86_64
Byte Order:                              Little Endian
CPU(s):                                  2
Model name:                              Intel(R) Xeon(R) Processor @ 2.10GHz
Thread(s) per core:                      1
Core(s) per socket:                      2
Hypervisor vendor:                       KVM
L1d cache:                               96 KiB (2 instances)
L2 cache:                                4 MiB (2 instances)
L3 cache:                                260 MiB (1 instance)
```

- `Architecture: x86_64` — ISA.
- `CPU(s): 2` — OS-in gördüyü hardware thread sayı. `Thread(s) per core: 1` — SMT yoxdur, yəni 2 core = 2 hardware thread.

- `Hypervisor vendor: KVM` — bu, virtual maşındır: böyük bir server CPU-sunun 2 core-luq payı.
- L1d, L2, L3 — cache-lər. Onları Fəsil 2-də açacağıq.

> 🖼️ **Şəkil 93:** lscpu · bu maşının çipi

Burada nəyə bax: "2 instances" hər core-un öz L1d və L2-si olduğunu deyir; L3 isə birdir və paylaşılır.

Telefonda (burada işlədilməyib — cihazda yoxla): `adb shell cat /proc/cpuinfo` və hər core-un maksimum tezliyi üçün `adb shell cat /sys/devices/system/cpu/cpu*/cpufreq/cpuinfo_max_freq`. Gözlə: core-ların maksimum tezliyi eyni olmayacaq. Səbəbini Senior səviyyəsində görəcəksən.

2. PC və SP-ni canlı gör (gdb)

`step.c`:

```
        long add(long a, long b) { return a + b; }
        int main(void) { long r = add(2, 3); return (int)r; }
```

```
$ gcc -O1 -g -fcf-protection=none -fno-inline -o step step.c
$ gdb -q -batch -ex 'break add' -ex 'run' -ex 'info registers rip rsp rdi rsi rax' \
      -ex 'x/2i $pc' -ex 'stepi' -ex 'info registers rip rax' -ex 'x/1i $pc' \
      -ex 'stepi' -ex 'info registers rip rsp' ./step
Breakpoint 1, add (a=a@entry=2, b=b@entry=3) at step.c:1
rip            0x555555555129      0x555555555129 <add>
rsp            0x7fffffff6c90      0x7fffffff6c90
rdi            0x2                 2
rsi            0x3                 3
rax            0x55555555512e      93824992235822
=> 0x555555555129 <add>:    lea    (%rdi,%rsi,1),%rax
   0x55555555512d <add+4>:  ret
rip            0x55555555512d      0x55555555512d <add+4>
rax            0x5                 5
=> 0x55555555512d <add+4>:  ret
0x000055555555513d in main () at step.c:2
rip            0x55555555513d      0x55555555513d <main+15>
rsp            0x7fffffff6c98      0x7fffffff6c98
$ ./step; echo "exit code = $?"
exit code = 5
```

(gdb-nin bir neçə xəbərdarlıq sətri ixtisar edilib.) Sətir-sətir:

- `rip` — x86-64-də PC-nin adıdır. CPU `add` funksiyasının ilk instruction-ında dayanıb.
- `rdi = 2`, `rsi = 3` — arqumentlər register-lərdə gəlib. Bu qayda **calling convention**-dır (Fəsil 4-də).
- `rax` hələ köhnə "zibil" dəyərdir: register-i heç kim təmizləmir.
- `lea (%rdi,%rsi,1),%rax` — "rdi + rsi-ni hesabla, rax-a yaz". Bir `stepi` (bir instruction irəli) sonra `rip` 0x...129-dan 0x...12d-yə keçdi: +4, çünki bu instruction 4 byte-dır. `rax = 5`.
- `ret`-dən sonra `rip` `main`-in ortasına keçdi, `rsp` isə 8 artdı (...c90 → ...c98). `ret` qayıdış ünvanını stack-dən götürdü. "Sirr"in cavabı budur: CPU hara qayıdacağını stack-dəki 8 byte-dan bilir.
- Proqramın exit code-u 5-dir: `main` nəticəni qaytardı.

> 🖼️ **Şəkil 94:** gdb stepi · register-lərin üç fotoşəkli

Burada nəyə bax: birinci stepi-də rip 4 byte irəliləyir və rax 5 olur; ikincidə rip main-ə atılır.

3. Eyni funksiya, iki ISA

`add.c`:

```
long add(long a, long b) { return a + b; }
long max2(long a, long b) { return a > b ? a : b; }
long counter;
void inc(void) { counter++; }
```

x86-64 (gcc 13.3):

```
$ gcc -O1 -fcf-protection=none -c -o add_x86.o add.c && objdump -d -M intel add_x86.o
0000000000000000 <add>:
   0:   48 8d 04 37             lea    rax,[rdi+rsi*1]
   4:   c3                      ret
0000000000000005 <max2>:
   5:   48 39 fe                cmp    rsi,rdi
   8:   48 89 f8                mov    rax,rdi
   b:   48 0f 4d c6             cmovge rax,rsi
   f:   c3                      ret
0000000000000010 <inc>:
  10:   48 83 05 00 00 00 00    add    QWORD PTR [rip+0x0],0x1
  17:   01
  18:   c3                      ret
```

ARM64 (clang 18.1.3):

```
$ clang --target=aarch64-linux-gnu -O1 -c -o add_arm.o add.c && llvm-objdump -d add_arm.o
0000000000000000 <add>:
       0: 8b000020      add x0, x1, x0
       4: d65f03c0      ret
0000000000000008 <max2>:
       8: eb01001f      cmp x0, x1
       c: 9a81c000      csel    x0, x0, x1, gt
      10: d65f03c0      ret
0000000000000014 <inc>:
      14: 90000008      adrp    x8, 0x0 <add>
      18: f9400109      ldr x9, [x8]
      1c: 91000529      add x9, x9, #0x1
      20: f9000109      str x9, [x8]
      24: d65f03c0      ret
```

Nə görürsən:

- x86-64 instruction-larının uzunluğu fərqlidir: `ret` 1 byte, `cmp` 3, `lea` 4, `add [mem]` 8 byte. ARM64-də hər instruction düz 4 byte-dır: ünvanlar 0, 4, 8, c, 10...
- x86-64 `counter++`-ı bir instruction-la yazır: yaddaşdakı ədədə birbaşa 1 əlavə edir. ARM64 üç addım yazır: `ldr` (yaddaşdan oxu), `add`, `str` (yaddaşa yaz). Bunun nə üçün vacib olduğunu "Kod" hissəsində görəcəksən.

- Fəsil 0 ilə əlaqə: ARM64-ün `add` instruction-ı 32 bitlik 0x8b000020 sözüdür. Faylın byte-larına baxsan (`llvm-objdump -s -j .text add_arm.o`), `2000008b` görərsən. Bu, little-endian-dır: ən kiçik byte (20) əvvəl.
- Ubuntu-da gcc default olaraq hər funksiyanın əvvəlinə `endbr64` qoyur. Bu, təhlükəsizlik üçün əlavə instruction-dır; burada `-fcf-protection=none` ilə onu çıxarmışıq ki, kod sadə görünsün.

Kotlin versiyası (`Add.kt`). Kotlin kodu birbaşa machine code-a yox, əvvəlcə JVM **bytecode**-una çevrilir. Bytecode — JVM-in (Android-da ART-ın) öz instruction dilidir; telefonda ART onu ARM64 machine code-a çevirir (Fəsil 3-də). Bytecode-a `javap` ilə baxmaq olar:

```
fun add(a: Long, b: Long): Long = a + b
fun max2(a: Long, b: Long): Long = if (a > b) a else b
fun min2(a: Long, b: Long): Long = if (a < b) a else b
fun absDiff(a: Long, b: Long): Long = if (a > b) a - b else b - a

class Counter {
    var count = 0
    fun inc() { count++ }
}
```

```
$ kotlinc Add.kt -d cls && javap -c -cp cls AddKt
  public static final long add(long, long);
    Code:
       0: lload_0
       1: lload_2
       2: ladd
       3: lreturn
```

Top-level funksiyalar `AddKt` adında class-ın static metodlarına çevrilir. `lload_0` və `lload_2` iki `Long` arqumentini stack-ə qoyur (hər `Long` iki yer tutur, ona görə 0 və 2), `ladd` toplayır, `lreturn` qaytarır. Bytecode register-lərlə yox, stack ilə işləyən virtual maşının dilidir.

> 🖼️ **Şəkil 95:** add üç dildə · x86-64, ARM64, bytecode

Burada nəyə bax: iki CPU register-lərlə işləyir, JVM bytecode-u isə dəyərləri stack-ə qoyub götürür.

Variasiya: `add`-ı `a * b` ilə əvəz et. x86-64-də `imul`, ARM64-də `mul` görəcəksən.

Kod

Calling convention qısaca (Fəsil 4-də tam): x86-64 Linux-da ilk arqument `rdi`, ikinci `rsi` register-ində gəlir, nəticə `rax`-da qaytarılır. ARM64-də arqumentlər `x0`, `x1`, ... register-lərində gəlir, nəticə `x0`-da qaytarılır.

Nümunə 1 — tam izahlı: `max2` iki ISA-da

```
long max2(long a, long b) { return a > b ? a : b; }
```

x86-64 (gcc -O1, yuxarıdakı objdump-dan):

```
cmp    rsi, rdi     ; b − a hesablanır, nəticə atılır, yalnız flag-lər yenilənir
mov    rax, rdi     ; nəticə = a (hələlik)
cmovge rax, rsi     ; flag-lərə görə "b >= a" doğrudursa, nəticə = b
ret                 ; qayıt; nəticə rax-dadır
```

ARM64 (clang -O1):

```
cmp  x0, x1          ; a − b, yalnız NZCV flag-ləri yenilənir
csel x0, x0, x1, gt  ; "a > b" doğrudursa x0 = a, yoxsa x0 = b
ret                  ; qayıt; nəticə x0-dadır
```

Sətir-sətir:

1. `cmp` əslində çıxmadır, amma nəticəni saxlamır, yalnız flag-ləri yeniləyir. Fəsil 0-dakı two's complement və flag-lər burada işə düşür.
2. `cmovge` və `csel` — **conditional select** (şərtli seçim). İki dəyərdən birini flag-lərə görə seçir.
3. Kodda jump yoxdur. Kompilyator bilir ki, jump CPU üçün bahalı ola bilər (Senior səviyyəsində: branch prediction). Belə koda **branchless** (budaqsız) kod deyilir.
4. `gt` — "greater than": signed müqayisə. Unsigned üçün `hi` olardı. CPU signed-liği bilmir, instruction seçir (Fəsil 0).

Kotlin versiyası: `fun max2(a: Long, b: Long): Long = if (a > b) a else b` (Kotlin-in hazır funksiyası `maxOf(a, b)`-dir). Bytecode-u:

```
$ javap -c -cp cls AddKt
  public static final long max2(long, long);
    Code:
       0: lload_0
       1: lload_2
       2: lcmp
       3: ifle          10
       6: lload_0
       7: goto          11
      10: lload_2
      11: lreturn
```

Bytecode-da jump var. `lcmp` iki `Long`-u müqayisə edib −1, 0 və ya 1 qaytarır; `ifle` "≤ 0-dırsa 10-a keç" deyir. Bu jump-ın machine code-da qalıb-qalmayacağını JIT və ya ART-ın AOT kompilyatoru həll edir: o, `max2`-ni `cmov` və ya `csel` ilə budaqsız edə bilər.

> 🖼️ **Şəkil 96:** max2 · budaq və seçici

Burada nəyə bax: cmov və csel hər iki dəyəri hazır saxlayıb flag-ə görə birini seçir; təxminə ehtiyac qalmır.

Nümunə 2 — sən tamamla: `min2`

```
        long min2(long a, long b) { return a < b ? a : b; }
```

ARM64 (clang -O2) çıxışı, bir yer boşdur:

```
cmp  x0, x1
csel x0, x0, x1, ____
ret
```

Boşluğu doldur. Sonra yoxla: `clang --target=aarch64-linux-gnu -O2 -S -o - min.c`.

Kotlin-də eyni tapşırıq: `fun min2(a: Long, b: Long): Long = if (a ____ b) a else b`. Doldurandan sonra `javap -c` ilə bytecode-una bax: `max2`-dəki `ifle` burada hansı instruction-a çevrilir?

Nümunə 3 — yalnız tapşırıq

`long abs_diff(long a, long b)` funksiyasını C-də yaz: |a − b| qaytarsın. Əvvəl təxmin et: clang ARM64 üçün `-O2` ilə onu jump ilə, yoxsa jump-sız yazacaq? Neçə instruction olacaq? Sonra kompilyasiya edib yoxla. Real çıxış Cavab açarındadır.

Sonra Kotlin-də də yaz: `fun absDiff(a: Long, b: Long): Long`. `javap -c` ilə bytecode-una bax və jump-ları say. C-dəki ARM64 kodu ilə müqayisə et: jump-ları machine code-dan kim çıxarır?

Tipik səhv: "`count++` bir addımdır"

❌ **Səhv düşüncə:** "`count++` bir sətirdir, deməli CPU-da bir addımdır. İki core eyni anda artırsa, heç nə itməz."

Nə baş verir: Java və Kotlin kodu əvvəlcə **bytecode**-a çevrilir — JVM-in (Android-da ART-ın) öz instruction dili (Fəsil 3-də). Real çıxış:

```
        public class Counter {
            int count;
            void inc() { count++; }
        }
```

```
$ javac Counter.java && javap -c Counter
  void inc();
    Code:
       0: aload_0
       1: dup
       2: getfield      #7                  // Field count:I
       5: iconst_1
       6: iadd
       7: putfield      #7                  // Field count:I
      10: return
```

Üç addım açıq görünür: oxu (`getfield`), topla (`iadd`), yaz (`putfield`). ARM64-də də üç instruction-dır: `ldr`, `add`, `str` (yuxarıdakı `inc`). x86-64-də bir instruction görünür (`add QWORD PTR [rip+0x0],0x1`), amma o da içəridə oxu-topla-yaz edir və başqa core-ları gözləmir.

Kotlin-də eyni class (`Add.kt`-dəki `Counter`):

```
        class Counter {
            var count = 0
            fun inc() { count++ }
        }
```

```
$ javap -c -p -cp cls Counter
  public final void inc();
    Code:
       0: aload_0
       1: getfield      #16                 // Field count:I
       4: istore_1
       5: aload_0
       6: iload_1
       7: iconst_1
       8: iadd
       9: putfield      #16                 // Field count:I
      12: return
```

Kotlin də eyni üç addımı yaradır: `getfield` (oxu), `iadd` (artır), `putfield` (yaz). Fərq yalnız köhnə dəyəri müvəqqəti `istore_1`-də saxlamasıdır.

> 🖼️ **Şəkil 97:** count++ · üç addım, üç dildə

Burada nəyə bax: hər dildə eyni üçlük var — oxu, artır, yaz.

İki core eyni anda `count++` edəndə (başlanğıcda count = 0). Hər core-un öz register-ləri var: A-nın x9-u ilə B-nin x9-u ayrı şeylərdir.

Addım	Core A	Core B	Yaddaşda count
1	`ldr`: x9 = 0		0
2		`ldr`: x9 = 0	0
3	`add`: x9 = 1		0
4		`add`: x9 = 1	0
5	`str`: count = 1		1
6		`str`: count = 1	1

İki artırma oldu, nəticə isə 1-dir. Bir artırma itdi. Buna **lost update** deyilir; bu, **race condition**-un ən sadə formasıdır (Faza 4-də tam).

✅ **Düzgün:** atomic əməliyyat. Kotlin-də `AtomicInteger.incrementAndGet()`. CPU-da nə olur? C-də eyni əməliyyatın (`__atomic_fetch_add`) real çıxışı:

x86-64 (gcc 13.3 -O2):

```
lock add    QWORD PTR acounter[rip], 1
```

ARM64, ARMv8.0 (clang 18 -O2):

```
.LBB2_1:
    ldaxr   x9, [x8]
    add x9, x9, #1
    stlxr   w10, x9, [x8]
    cbnz    w10, .LBB2_1
```

ARM64, ARMv8.1 (`-march=armv8.1-a`):

```
ldaddal x8, x8, [x9]
```

Niyə işləyir: x86-64-də `lock` prefiksi əməliyyat bitənə qədər həmin yaddaş xanasına başqa core-ların toxunmasına imkan vermir. ARM64-ün `ldaxr` və `stlxr` cütü demək olar ki, belə danışır: "oxu; sonra yalnız arada heç kim toxunmayıbsa yaz". Kimsə toxunubsa, `stlxr` uğursuz olur (w10 ≠ 0), `cbnz` isə döngünü təkrarlayır. ARMv8.1-dən bunu bir instruction (`ldaddal`) edir. Mexanizmləri Faza 4 və 6-da açacağıq. Burada bilməli olduğun: düzgün paralel artırma CPU-dan xüsusi instruction tələb edir.

Kotlin-də özün gör (`Race.kt`): iki thread hər biri 10 milyon dəfə həm adi `Int`-i, həm `AtomicInteger`-i artırır. Thread-i Faza 3-də öyrənəcəksən; burada bilməli olduğun: `thread { }` bloku kodu ayrıca işə salır və iki blok iki core-da eyni anda işləyə bilər.

```
import java.util.concurrent.atomic.AtomicInteger
import kotlin.concurrent.thread

var plain = 0
val atomic = AtomicInteger()

fun main() {
    val work = {
        repeat(10_000_000) {
            plain++                     // oxu, artır, yaz: üç addım
            atomic.incrementAndGet()    // CPU-nun atomic instruction-ı
        }
    }
    val a = thread { work() }
    val b = thread { work() }
    a.join(); b.join()
    println("gozlenilen = 20000000, plain = $plain, atomic = ${atomic.get()}")
}
```

```
$ kotlinc Race.kt -include-runtime -d Race.jar && java -jar Race.jar
gozlenilen = 20000000, plain = 18749662, atomic = 20000000
$ java -jar Race.jar
gozlenilen = 20000000, plain = 18346206, atomic = 20000000
```

Adi sayğac artırmaların 6–8%-ini itirdi, `AtomicInteger` heç birini itirmədi.

> 🖼️ **Şəkil 98:** Lost update · iki core, bir itmiş artırma

Burada nəyə bax: iki core eyni 5-i oxuyur, ikisi də 6 yazır — bir artırma iz qoymadan yox olur.

Senior səviyyəsi

1. Pipeline: CPU bir anda bir neçə instruction üzərində işləyir

- Paltaryuyan maşını düşün: yuma, qurutma, ütüləmə. Birinci yük qurudularkən ikincisi artıq yuyulur. Hər mərhələ eyni anda başqa yük üzərində işləyir. CPU-da da fetch, decode və execute eyni anda fərqli instruction-larla məşğuldur. Buna **pipeline** deyilir. Müasir yüksək sürətli core-larda mərhələ sayı 10-dan çoxdur.
- **Latency** (gecikmə) — bir instruction-ın başlayıb bitməsinə qədər keçən vaxt. **Throughput** — vahid vaxtda neçə instruction-ın bitdiyi. Pipeline latency-ni azaltmır, throughput-u artırır.

- Pipeline-ın düşməni **asılılıqdır**: instruction əvvəlkinin nəticəsini gözləyirsə, növbəti mərhələ boş qalır.

Bunu **llvm-mca** ilə gör. Bu alət LLVM-in CPU modellərindən istifadə edərək instruction-ların pipeline-da necə irəlilədiyini simulyasiya edir. Aşağıda Skylake modeli: əvvəl 4 asılı `imul` (hər biri əvvəlkinin nəticəsini istifadə edir), sonra 4 müstəqil `imul` (rax, rbx, rcx, rdx):

```
$ llvm-mca-18 -mtriple=x86_64 -mcpu=skylake -x86-asm-syntax=intel -iterations=1 -timeline dep.s
[0,0]     DeeeER    .    .   imul    rax, rax
[0,1]     D===eeeER .    .   imul    rax, rax
[0,2]     D======eeeER  .   imul    rax, rax
[0,3]     D=========eeeER   imul    rax, rax
$ llvm-mca-18 -mtriple=x86_64 -mcpu=skylake -x86-asm-syntax=intel -iterations=1 -timeline ind.s
[0,0]     DeeeER  .    imul  rax, rax
[0,1]     D=eeeER .    imul  rbx, rbx
[0,2]     D==eeeER.    imul  rcx, rcx
[0,3]     D===eeeER    imul  rdx, rdx
```

(Cədvəlin başlıq sətirləri ixtisar edilib.) Hər simvol bir cycle-dır: **D** — instruction növbəyə qoyuldu, **=** — gözləyir, **e** — icra olunur, **E** — icra bitdi, **R** — nəticə rəsmiləşdi (retire). Hər `imul` 3 cycle icra olunur. Asılı zəncirdə hər biri əvvəlkini gözləyir və iş 15 cycle çəkir. Müstəqil zəncirdə hər cycle yeni `imul` başlayır və iş 9 cycle çəkir. 100 iterasiyada model 1203 və 405 cycle verir: IPC 0,33 və 0,99.

Simulyasiya bir şeydir, real CPU başqa. Bu maşında ölçdük (`ilp2.c`, 200 milyon dövr, 3 təkrarın ən yaxşısı). Dövrün içi belədir:

```
        /* 4 imul, hər biri əvvəlkinin nəticəsini gözləyir */
        __asm__ volatile("imul %0,%0\n\timul %0,%0\n\timul %0,%0\n\timul %0,%0" : "+r"(a));
        /* 4 imul, 4 ayrı zəncir: bir-birini gözləmir */
        __asm__ volatile("imul %0,%0\n\timul %1,%1\n\timul %2,%2\n\timul %3,%3"
                         : "+r"(a), "+r"(b), "+r"(c), "+r"(d));
```

```
$ gcc -O2 -o ilp2 ilp2.c && ./ilp2
asili zencir:    3.03 ns / 4 imul = 0.757 ns bir imul
musteqil zencir: 0.98 ns / 4 imul = 0.245 ns bir imul
ferq: 3.1 defe
```

Başqa işə salmada fərq 3,5 dəfə oldu: virtual maşında dəyər bir az oynayır. Bundan senior nəticələr:

Kotlin versiyası (`Ilp.kt`): eyni iki zəncir, inline assembly olmadan. JIT kodu tam kompilyasiya etsə deyə, ölçmə 5 dəfə təkrarlanır və ən yaxşı nəticə götürülür.

```
        // ilp2.c-nin Kotlin versiyası: asılı və müstəqil vurma zəncirləri (JIT-dən sonra
ölçülür)
        fun dependent(n: Int, seed: Long): Long {
            var a = seed
            repeat(n) { a *= a; a *= a; a *= a; a *= a }      // hər vurma əvvəlkini gözləyir
            return a
        }

        fun independent(n: Int, seed: Long): Long {
            var a = seed; var b = seed + 2; var c = seed + 4; var d = seed + 6
            repeat(n) { a *= a; b *= b; c *= c; d *= d }      // 4 ayrı zəncir
            return a xor b xor c xor d
        }
```

```
        fun main() {
            val n = 200_000_000
            var sink = 0L
            var bestDep = Long.MAX_VALUE; var bestInd = Long.MAX_VALUE
            repeat(5) {                                          // ilk turlar JIT üçün isinmədir
                var t = System.nanoTime(); sink += dependent(n, 3L + it); bestDep =
minOf(bestDep, System.nanoTime() - t)
                t = System.nanoTime(); sink += independent(n, 3L + it); bestInd = minOf(bestInd,
System.nanoTime() - t)
            }
            println("asili zencir:    %.2f ns / 4 vurma".format(bestDep.toDouble() / n))
            println("musteqil zencir: %.2f ns / 4 vurma".format(bestInd.toDouble() / n))
            println("ferq: %.1f defe   (sink=${sink and 1})".format(bestDep.toDouble() /
bestInd))
        }
```

```
$ kotlinc Ilp.kt -include-runtime -d Ilp.jar && java -jar Ilp.jar
asili zencir:    3.72 ns / 4 vurma
musteqil zencir: 1.24 ns / 4 vurma
ferq: 3.0 defe   (sink=1)
```

Kotlin-də də fərq 3 dəfədir: JIT-in yaratdığı machine code-da da vurmalar bir-birini gözləyir. `sink` nəticəni çap etmək üçündür ki, JIT hesablamanı "lazımsız" sayıb silməsin.

- Eyni say instruction, 3 dəfə fərqli sürət. Sürəti instruction sayı yox, **asılılıq zənciri** təyin edir.
- `imul`-un latency-si bu CPU ailəsində 3 cycle-dır (Agner Fog-un instruction cədvəlləri). 0,757 ns / 3 ≈ 0,25 ns bir cycle, yəni təxminən 4 GHz — nominal 2,1 GHz-dən yuxarı. Səbəb turbo rejimidir. Bu, təxmini hesabdır.
- Kotlin praktikası: kompilyator və JIT tam ədədlərin cəmini bir neçə akkumulyatora özü bölə bilər, `Double` cəmini isə adətən bölmür. Səbəb: floating point toplaması assosiativ deyil. Python-da yoxla: `(0.1 + 0.2) + 0.3` = 0.6000000000000001, `0.1 + (0.2 + 0.3)` = 0.6. Sıranı dəyişmək nəticəni dəyişir, ona görə kompilyator buna icazə vermir.

> 🖼️ **Şəkil 99:** Pipeline · camaşırxana

Burada nəyə bax: pipeline bir yükü tezləşdirmir, amma eyni vaxtda daha çox yük bitirir.

> 🖼️ **Şəkil 100:** llvm-mca · asılı və müstəqil imul

Burada nəyə bax: asılı zəncirdə göy zolaqlar pilləkən kimi düzülür, müstəqildə isə üst-üstə düşür.

> 🖼️ **Şəkil 101:** Ilp.kt · 3 dəfə fərq

Burada nəyə bax: simulyasiyanın proqnozu Kotlin-də real ölçülə təsdiqlənir.

2. Branch prediction: CPU gələcəyi təxmin edir

- Pipeline dolu qalmaq üçün CPU `if`-in nəticəsini bilmədən növbəti instruction-ları götürməlidir. **Branch predictor** keçmişə baxıb təxmin edir. Səhv təxminə **misprediction** deyilir: pipeline-dakı iş atılır və təxminən 15–20 cycle itir.

- Təcrübə (`branch.c`): 32 768 təsadüfi ədəd (0–255), 128-dən böyük olanları topla, 20 000 dəfə. Əvvəl qarışıq massivdə, sonra sıralanıb. Kodun özü: `if (data[i] >= 128) sum += data[i];`.

```
$ gcc -O1 -fno-tree-vectorize -fno-if-conversion -fno-if-conversion2 -o branch_O1 branch.c &&
./branch_O1
qarisiq: 2.40 s, sirali: 0.18 s, cem eyni? beli
$ gcc -O2 -o branch_O2 branch.c && ./branch_O2
qarisiq: 0.19 s, sirali: 0.19 s, cem eyni? beli
```

- Eyni iş, eyni nəticə, 13 dəfə fərq. Qarışıq massivdə `if` təxminən yarı ehtimalla doğrudur və predictor tez-tez yanılır. Sıralı massivdə əvvəl həmişə "yox", sonra həmişə "hə" gəlir və predictor demək olar ki, yanılmır.
- Hesab: 20 000 × 32 768 ≈ 655 milyon `if`. Təxminən yarısı səhv təxmin olunur: ~328 milyon misprediction. Əlavə vaxt 2,22 s → bir misprediction ≈ 6,8 ns, yəni təxminən 20 cycle (4 GHz-ə yaxın tezlikdə).
- `-O2`-də fərq yoxdur: kompilyator `if`-i budaqsız koda və SIMD-ə (bir instruction-la bir neçə ədəd) çevirdi. `-O1` sətirindəki bayraqlar bu çevrilməni qəsdən söndürür.
- Senior dərsi: performansı təxminlə müzakirə etmə. Ölç və yaradılan koda bax.

Kotlin versiyası (`Branch.kt`, HotSpot JIT, OpenJDK 21):

```
        import kotlin.random.Random

        // branch.c-nin Kotlin versiyası: qarışıq və sıralı massivdə eyni if
        fun run(data: IntArray, reps: Int): Long {
            var sum = 0L
            repeat(reps) {
                for (v in data) if (v >= 128) sum += v
            }
            return sum
        }

        fun main() {
            val data = IntArray(32_768)
            val r = Random(1)
            for (i in data.indices) data[i] = r.nextInt(256)
            repeat(3) { run(data, 200) }                        // JIT isinməsi
            var t = System.nanoTime(); val s1 = run(data, 20_000); val mixed = (System.nanoTime()
- t) / 1e9
            data.sort()
            repeat(3) { run(data, 200) }
            t = System.nanoTime(); val s2 = run(data, 20_000); val sorted = (System.nanoTime() -
t) / 1e9
            println("qarisiq: %.2f s, sirali: %.2f s, cem eyni? %s".format(mixed, sorted, if (s1
== s2) "beli" else "xeyr"))
        }
```

```
$ kotlinc Branch.kt -include-runtime -d Branch.jar && java -jar Branch.jar
qarisiq: 2.22 s, sirali: 0.28 s, cem eyni? beli
```

Bu maşında JIT `if`-i saxladı və fərq təxminən 8 dəfə oldu. Hansı machine code-un yaranacağını JIT özü həll edir: başqa JVM versiyası və ya telefonda ART bu `if`-i budaqsız edə bilər. Ona görə telefonda nəticəni Jetpack Microbenchmark ilə yoxla (burada işlədilməyib — cihazda yoxla).

> 🖼️ **Şəkil 102:** Branch prediction · yol ayrıcı

Burada nəyə bax: CPU dayanmamaq üçün yolu əvvəlcədən seçir; səhv seçim bahadır.

> 🖼️ **Şəkil 103:** Qarışıq və sıralı data · təxminlər

Burada nəyə bax: sıralı sırada predictor yalnız sərhəddə yanılır.

> 🖼️ **Şəkil 104:** Ölçülər · 13×, 8× və fərqsiz

Burada nəyə bax: if budaqsız koda çevriləndə (-O2) data-nın sırası artıq heç nəyi dəyişmir.

3. Out-of-order və speculative execution (Faza 6-nın toxumu)

- Müasir böyük core-lar instruction-ları proqramdakı sıra ilə deyil, hazır olduqları sıra ilə icra edir. Buna **out-of-order execution** deyilir. Nəticələr isə yenə proqram sırası ilə rəsmiləşir (llvm-mca-dakı R). Ona görə bir thread özünü həmişə proqram sırasında görür.
- Amma başqa core yaddaşa baxanda yazıların sırası fərqli görünə bilər. Bu, **memory model** mövzusudur və thread-lərin ən çətin bug-larının mənbəyidir (Faza 6).
- **Speculative execution** — təxmin edilmiş yolu "ehtimalla" icra etmək. Təxmin səhv çıxsa, nəticələr atılır, amma cache-də iz qalır. 2018-ci ildə açıqlanan Spectre və Meltdown hücumları bu izdən gizli məlumat oxuyurdu. Bu maşında müdafiə aktivdir:

```
$ lscpu | grep -E 'Spectre|Meltdown'
Vulnerability Meltdown:                 Not affected
Vulnerability Spectre v1:               Mitigation; usercopy/swapgs barriers and __user pointer
sanitization
Vulnerability Spectre v2:               Mitigation; Enhanced / Automatic IBRS; IBPB conditional;
PBRSB-eIBRS SW sequence; BHI Vulnerable
```

Bu müdafiələr kernel-ə hər girişi bahalaşdırır. Fəsil 6-da syscall-ın qiymətini ölçəndə bunu yadına sal.

> 🖼️ **Şəkil 105:** Out-of-order hekayəsi · restoran

Burada nəyə bax: mətbəx sıranı dəyişir, müştəri isə bunu görmür.

> 🖼️ **Şəkil 106:** İcra qarışıq, retire sıralı

Burada nəyə bax: B A-dan əvvəl bitir, amma R-lər proqram sırası ilə düzülür.

> 🖼️ **Şəkil 107:** Speculative execution · Spectre izi

Burada nəyə bax: nəticələr atılsa da, cache-dəki iz ölçülə bilən vaxt fərqi yaradır.

4. x86-64 və ARM64: senior üçün vacib fərqlər

Xüsusiyyət	x86-64	ARM64 (AArch64)
Harada	Noutbuk, server, Android emulator	Android telefonlar, Apple M-seriyası, çox sayıda server
Instruction uzunluğu	1–15 byte, dəyişkən	Həmişə 4 byte
Ümumi register sayı	16	31 (x0–x30)
Yaddaşla iş	Hesablama instruction-ı yaddaşa birbaşa toxuna bilər (`add [mem], 1`)	Load/store: yaddaşa yalnız `ldr` və `str` toxunur
Qayıdış ünvanı	`call` stack-ə yazır	`bl` x30-a (LR) yazır
Syscall instruction-ı	`syscall`	`svc #0`
Privilege səviyyələri	Ring 0–3	EL0–EL3
Yaddaş modeli	Güclü (TSO): yazılar başqa core-lara əsasən proqram sırası ilə görünür	Zəif: sıranı barrier instruction-ları təmin edir (Faza 6)
Endianness	Little	Little (hardware iki rejimi dəstəkləyə bilər, Android LE işlədir)

- **CISC və RISC:** x86-64 mürəkkəb, dəyişkən uzunluqlu instruction-larla CISC ənənəsindəndir. ARM64 sadə, sabit uzunluqlu instruction-lu RISC-dir. Müasir x86 içəridə instruction-ları sadə **μop**-lara (micro-operation) parçalayır, yəni daxildə RISC-ə bənzəyir.
- **NDK və ABI:** native `.so` faylları hər ABI üçün ayrıca qurulur: `arm64-v8a`, `armeabi-v7a`, `x86_64`, `x86`. Emulator adətən x86_64-dür, telefon arm64-v8a. Google Play 2019-cu ilin avqustundan native kodlu tətbiqlərdən 64-bit versiya tələb edir.
- **Yaddaş modeli:** emulatorda (x86-64, güclü model) "işləyən" data race telefonda (ARM64, zəif model) partlaya bilər. Bu, Faza 6-nın əsas mövzusudur.

> 🖼️ **Şəkil 108:** Instruction uzunluğu · x86-64 və ARM64

Burada nəyə bax: x86-64 xanalarının eni dəyişir, ARM64-ünkü həmişə dörddür.

> 🖼️ **Şəkil 109:** Dörd əsas fərq

Burada nəyə bax: ən vacib fərq son sətirdir — yaddaş modeli thread bug-larını müəyyən edir.

> 🖼️ **Şəkil 110:** APK və ABI · emulator tələsi

Burada nəyə bax: emulator və telefon fərqli ISA və fərqli yaddaş modeli ilə işləyir.

5. Android cihazının CPU-su

- **big.LITTLE** (indiki adı DynamIQ): bir çipdə fərqli ölçülü core-lar var — güclü olanlar və enerjiyə qənaətli olanlar. Eyni kod kiçik core-da bir neçə dəfə yavaş işləyə bilər. Thread-i hansı core-a qoymağı kernel-in scheduler-i həll edir (Faza 2). "Özün gör"dəki `cpuinfo_max_freq` fərqi bundandır.
- **DVFS** (dynamic voltage and frequency scaling): tezlik və gərginlik yükə görə dəyişir. Telefon qızınanda **thermal throttling** tezliyi endirir.
- Nəticə: telefonda performans ölçmək çətindir. Jetpack Microbenchmark kitabxanası bu səbəbdən root-lu cihazda tezlikləri sabitləyən `lockClocks` Gradle tapşırığını təklif edir.
- Telefon core-larında adətən SMT yoxdur: "8 core" = 8 hardware thread.
- Kotlin kodunun yolu (Fəsil 3-də tam): Kotlin → JVM bytecode → DEX → ART-da interpreter, JIT və ya AOT → ARM64 machine code. Yəni `count++` sonda yuxarıdakı `ldr`, `add`, `str`-ə çevrilir.

> 🖼️ **Şəkil 111:** big.LITTLE · böyük və kiçik core-lar

Burada nəyə bax: telefon çipində core-lar eyni deyil; thread-in hansı core-a düşdüyü sürəti dəyişir.

> 🖼️ **Şəkil 112:** DVFS və thermal throttling · sxematik

Burada nəyə bax: eyni kod fərqli anlarda fərqli tezlikdə işləyir.

> 🖼️ **Şəkil 113:** count++-un yolu · Kotlin-dən ARM64-ə

Burada nəyə bax: Kotlin sətri beş mərhələdən keçib CPU-nun üç instruction-ına çevrilir.

6. Saat haqqında iki incə detal

- `rdtsc` instruction-ı TSC (Time Stamp Counter) sayğacını oxuyur. Müasir CPU-larda TSC core-un tezliyindən asılı olmayaraq sabit sürətlə artır: `lscpu`-dakı `constant_tsc` və `nonstop_tsc` flag-ləri bunu deyir. Yəni TSC core cycle-larını saymır, saat kimi işləyir. Bizim ölçmədə effektiv tezlik (~4 GHz) TSC-nin tezliyindən (2,1 GHz) yuxarı idi.
- `/proc/cpuinfo`-dakı BogoMIPS sürət ölçüsü deyil. Kernel açılışda kiçik gözləmə döngülərini kalibrləmək üçün onu hesablayır; adı "bogus" (saxta) sözündəndir. Bu maşında: `bogomips : 4200.00` (= 2 × 2100).

> 🖼️ **Şəkil 114:** TSC və core cycle · iki saat

Burada nəyə bax: TSC xətləri bərabər aralıqlıdır; core-un cycle-ları turbo-da daha sıx olur.

> 🖼️ **Şəkil 115:** Real tezlik · ölçüdən hesab

Burada nəyə bax: dörd addımlı hesab nominal tezliyin real tezlik olmadığını göstərir.

> 🖼️ **Şəkil 116:** BogoMIPS · sürət ölçüsü deyil

Burada nəyə bax: 4200 rəqəmi sadəcə 2 × 2100-dür; CPU-nun gücünü ölçmür.

Bunu az adam bilir: ARM64-də `cmp` adında instruction yoxdur

Fakt: ARM64-də `cmp x0, x1` əslində `subs xzr, x0, x1`-dir: "x0 − x1-i hesabla, flag-ləri yenilə, nəticəni XZR-ə yaz". **XZR** — həmişə 0 oxunan və yazılanı atan xüsusi register-dir. `ret` isə əslində `ret x30`-dur.

Sübut — eyni fayl, disassembler-in "alias"ları söndürülüb:

```
$ llvm-objdump -d -M no-aliases add_arm.o
0000000000000008 <max2>:
       8: eb01001f      subs    xzr, x0, x1
       c: 9a81c000      csel    x0, x0, x1, gt
      10: d65f03c0      ret x30
```

Niyə belədir: ARM64-ün 32 bitlik instruction-ında register nömrəsi 5 bitdir: 0–31. Nömrə 31 instruction-dan asılı olaraq ya SP, ya da XZR mənasını verir. Beləliklə ISA ayrıca `cmp` instruction-ı üçün yer xərcləmir: "nəticəsi atılan çıxma" kifayətdir. `mov x0, #2` də əslində `movz`-dur. RISC fəlsəfəsi budur: az, sadə, çoxfunksiyalı instruction.

Real işdə niyə vacibdir: Android-də native crash olanda sistem **tombstone** faylı yazır. Orada x0–x28, x29 (frame pointer, Fəsil 4-də), lr, sp və pc register-lərinin dəyərləri olur. `lr` sənə hansı funksiyadan gəldiyini, `pc` — harada partladığını deyir. Disassembly-də `subs xzr` və ya `ret x30` görəndə çaşmayacaqsan.

> 🖼️ **Şəkil 117:** Ləqəb və əsl ad · cmp, ret, mov

Burada nəyə bax: tanış adların çoxu ləqəbdir; əsl instruction-ı no-aliases göstərir.

> 🖼️ **Şəkil 118:** Register 31 · SP və ya XZR

Burada nəyə bax: eyni 11111 nömrəsi instruction-dan asılı olaraq iki fərqli məna daşıyır.

> 🖼️ **Şəkil 119:** Tombstone · PC və LR

Burada nəyə bax: crash-i araşdırarkən ilk iki sual — harada (pc) və kimdən (lr).

Yanlış təsəvvürlər

Yanlış təsəvvür	Həqiqət	Özün necə sübut edərsən
Bir sətir kod = bir CPU addımı	`count++` bytecode-da 3, ARM64-də 3 instruction-dır	`javap -c Counter`, `llvm-objdump -d add_arm.o`

Yanlış təsəvvür	Həqiqət	Özün necə sübut edərsən
x86-də bir instruction-lıq `add [mem], 1` atomic-dir	Başqa core-lara qarşı deyil; `lock` prefiksi lazımdır	gcc atomic üçün `lock add` yaradır, adi `++` üçün yox
GHz yüksəkdirsə, CPU həmişə daha sürətlidir	IPC, cache və asılılıqlar da həll edir	`ilp2.c`: eyni say `imul`, 3 dəfə fərq
Hər instruction bir cycle çəkir	`imul` 3 cycle latency-dir; bir cycle-da bir neçə instruction da bitə bilər	llvm-mca timeline
CPU instruction-ları yazıldığı sıra ilə icra edir	Out-of-order icra edir, nəticəni sıra ilə rəsmiləşdirir	llvm-mca: D və e sütunları fərqli sıradadır
`if` pulsuzdur	Səhv təxmin təxminən 20 cycle-a başa gəlir	`branch.c`: 2,40 s və 0,18 s
8 core = tətbiq 8 dəfə sürətli	Yalnız paralel bölünən iş sürətlənir; telefonda core-lar da eyni deyil	`cpuinfo_max_freq` fərqi (cihazda)
Hardware thread = OS thread	SMT konteksti ≠ OS-in proqram abstraksiyası	`lscpu`: 2 CPU; `ps -T` minlərlə thread (Faza 3)
Emulatorda işləyən native kod telefonda da işləyəcək	ISA və yaddaş modeli fərqlidir	`.so`-nu `file` ilə yoxla: x86-64 və ya aarch64
BogoMIPS CPU-nun sürətidir	Kernel-in kalibrləmə dəyəridir	Bu maşında 4200 = 2 × 2100

İntervyuda

İ1.1 — CPU bir instruction-ı necə icra edir? `a + b` yazanda nə baş verir?

- **Qısa cavab:** Kompilyator `a + b`-ni machine code-a çevirir. CPU PC-dəki instruction-ı fetch, decode və execute edir, nəticəni register-ə yazır, PC növbəti instruction-a keçir. Pipeline sayəsində eyni anda bir neçə instruction müxtəlif mərhələdə olur.
- **Mexanizm:** ARM64-də `add x0, x1, x0` 32 bitlik 0x8b000020 sözüdür. Decode bu bitlərdən "toplama, girişlər x1 və x0, çıxış x0" oxuyur, ALU toplayır. Arqumentlər calling convention-a görə artıq x0 və x1-dədir.
- **Rəqəm və sübut:** gdb-də `stepi`-dən sonra `rip` 4 artdı, `rax = 5` oldu. 2,1 GHz-də bir cycle ≈ 0,48 ns. `imul`-un latency-si 3 cycle-dır (llvm-mca timeline).
- **Trade-off:** Dərin pipeline yüksək tezlik və throughput verir, amma asılılıqda dayanır və səhv branch təxmininde daha çox iş atır.
- **Real nümunə:** Bitmap-in pixel-lərini emal edən Kotlin dövründə hər pixel üçün `if` qarışıq data ilə yavaş ola bilər. Budaqsız yazılış (`maxOf`, `coerceIn`) və Jetpack Microbenchmark ilə ölçmək lazımdır.
- **Fərqləndirici detal:** ARM64-də `cmp` əslində `subs xzr, …`-dir. Out-of-order core icra sırasını dəyişir, amma nəticəni proqram sırası ilə rəsmiləşdirir.

Follow-up nərdivanı:

1. *"PC və SP fərqi nədir?"* → PC icra olunacaq instruction-ın, SP stack-in ucunun ünvanıdır. x86-64-də `ret` qayıdış ünvanını SP-nin göstərdiyi yerdən götürür (gdb-də rsp +8), ARM64-də isə x30-dan.

2. *"Bir cycle-da neçə instruction bitə bilər?"* → Superscalar core-larda birdən çox. IPC proqramdan asılıdır: müstəqil `imul`-larda model IPC ≈ 1 verdi, asılıda 0,33.
3. *"Branch misprediction nədir və nə qədər bahadır?"* → Səhv yolda götürülən iş atılır. Təxminən 15–20 cycle; bizim ölçmədə ~6,8 ns.
4. *"Out-of-order execution thread-lərə necə təsir edir?"* → Bir thread özünü proqram sırasında görür, amma başqa core yazıları fərqli sırada görə bilər. Sıranı memory barrier-lər təmin edir (Faza 6).
5. *"Spectre nədir?"* → Speculative execution-un cache-də buraxdığı izdən gizli məlumat oxuyan hücum sinfi (2018). Müdafiələr kernel-ə girişi bahalaşdırır.

İ1.2 — `count++` niyə thread-safe deyil?

- **Qısa cavab:** Çünki o, üç addımdır: oxu, artır, yaz. İki core arasında addımlar qarışanda bir artırma itir — lost update.
- **Mexanizm:** Bytecode-da `getfield`, `iadd`, `putfield`; ARM64-də `ldr`, `add`, `str`. Hər core-un öz register-ləri var: ikisi də köhnə dəyəri oxuyur, ikisi də eyni yeni dəyəri yazır.
- **Rəqəm və sübut:** Bu maşında iki Java thread-i hər biri 10 milyon dəfə artırdı (`Race.java`, 3 işə salma):

```
gozlenilen = 20000000, plain = 19050818, atomic = 20000000
gozlenilen = 20000000, plain = 18461663, atomic = 20000000
gozlenilen = 20000000, plain = 19377817, atomic = 20000000
```

Adi `int` artırmaların 3–8%-ini itirdi, `AtomicInteger` heç birini.

- **Trade-off:** `AtomicInteger` zəif contention-da ucuzdur. Yüksək contention-da hər core eyni yaddaş xanasını özünə çəkir və yavaşlayır (Fəsil 2, Faza 6); orada `LongAdder` (sayğacı hissələrə bölür) daha yaxşıdır. Alternativlər: `Mutex`, `synchronized` və ya veriyi bir thread-ə bağlamaq.
- **Real nümunə:** Bir neçə fayl paralel yüklənir və hər biri bitdikdə `downloaded++` edilir. UI-da "9 / 10" ilişib qalır, bar heç vaxt dolmur. Düzəliş: `AtomicInteger` və ya nəticələri `awaitAll()` ilə bir yerdə saymaq.
- **Fərqləndirici detal:** x86-64-də `++` tək instruction olsa da (`add [mem], 1`), `lock` prefiksi olmadan atomic deyil. ARM64-də atomic artırma ARMv8.0-da `ldaxr`/`stlxr` döngüsü, ARMv8.1-dən tək `ldaddal` instruction-ıdır.

Follow-up nərdivanı:

1. *"Sahəyə `@Volatile` əlavə etsəm, düzələr?"* → Xeyr. `@Volatile` yalnız görünürlük və sıra verir (Faza 6); `++` yenə üç addımdır.
2. *"`AtomicInteger` içəridə nə edir?"* → CPU-nun atomic instruction-larını: x86-64-də `lock` prefiksli instruction, ARM64-də LL/SC döngüsü və ya LSE (`ldadd`).
3. *"LL/SC ilə CAS fərqi nədir?"* → LL/SC (`ldaxr`/`stlxr`) ünvana "kimsə toxundumu" sualını yoxlayır. CAS dəyəri müqayisə edir və ABA probleminə düşə bilər (Faza 6).
4. *"Contention-da niyə yavaşlayır?"* → Yaddaş xanası cache line kimi core-dan core-a köçür (Fəsil 2, Faza 6).
5. *"Bu bug-ı testdə necə tutarsən?"* → Çox thread-li stress test. Native kod üçün ThreadSanitizer, JVM üçün jcstress.

İ1.3 — x86-64 və ARM64 fərqi nədir? Android developer üçün niyə vacibdir?

- **Qısa cavab:** İki fərqli ISA-dır. x86-64 CISC ənənəsindən, dəyişkən uzunluqlu instruction-larla; ARM64 RISC-dir, hər instruction 4 byte. Telefonlar ARM64, emulator adətən x86_64-dür. Native kod ABI üzrə ayrı qurulur, yaddaş modelləri isə fərqlidir.
- **Mexanizm:** ARM64 load/store arxitekturasıdır, 31 ümumi register və link register (x30) işlədir. x86-64 yaddaşa birbaşa hesablama edə bilir və qayıdış ünvanını stack-ə yazır.
- **Rəqəm və sübut:** objdump: x86-64-də 1, 3, 4, 8 byte-lıq instruction-lar; ARM64-də hər biri 4 byte. Register sayı 16 və 31.
- **Trade-off:** x86-64 kodu daha sıx ola bilər və yaddaş modeli güclüdür (proqramçı üçün daha bağışlayıcı). ARM64-ün decode-u sadədir və enerjiyə qənaətli dizayn üçün rahatdır; zəif yaddaş modeli hardware-ə sıralama azadlığı verir, amma proqramçıdan barrier tələb edir.
- **Real nümunə:** APK-da yalnız `x86_64` `.so` var: emulatorda hər şey işləyir, telefonda `UnsatisfiedLinkError`. Başqa nümunə: native kodda data race emulatorda görünmür, telefonda crash edir.
- **Fərqləndirici detal:** bionic sənədlərinə görə Android 16-dan kernel yalnız 64-bit işləyəcək, user space isə hələ 32-bit ola bilər. 32-bit Android-də `time_t` 2038-də daşır (Fəsil 0).

Follow-up nərdivanı:

1. *"TSO nə deməkdir?"* → Total Store Order: core-un yazıları başqalarına sıra ilə görünür; yalnız "yazıdan sonrakı oxu" əvvələ keçə bilər.
2. *"32-bit ABI-ni niyə hələ qururlar?"* → Köhnə cihazlar. Amma Google Play 64-bit versiyanı məcburi edib.
3. *"Bütün ABI-ləri bir APK-ya qoymaq nəyə başa gəlir?"* → Ölçü artar. App Bundle hər cihaza yalnız öz ABI-sini çatdırır.
4. *"Bir `.so`-nun hansı ISA üçün olduğunu necə bilərsən?"* → `file libx.so` və ya `readelf -h libx.so`: Machine sahəsi (AArch64 və ya X86-64). ELF başlığını Fəsil 3-də açacağıq.

İ1.4 — Core, hardware thread və OS thread arasında fərq nədir?

- **Qısa cavab:** Core — fiziki icra maşınıdır. Hardware thread — core-un içindəki bir icra konteksti (SMT-də bir core-da iki). OS thread — kernel-in proqram abstraksiyasıdır: scheduler onları hardware thread-lər üzərində növbə ilə işlədir.
- **Mexanizm:** SMT PC və register-ləri iki dəfə saxlayır, ALU və cache-ləri paylaşır. OS thread-in vəziyyəti (register-lər, stack) yaddaşda saxlanır və növbəsi çatanda CPU-ya yüklənir (context switch, Faza 2).
- **Rəqəm və sübut:** Bu maşın: `CPU(s): 2`, `Thread(s) per core: 1`. Telefon core-larında adətən SMT yoxdur.
- **Trade-off:** SMT bir thread gözləyəndə (məsələn, yaddaşdan cavab) core-u boş qoymur. Amma iki thread eyni ALU-lar üçün yarışır, bəzən təhlükəsizlik səbəbindən söndürülür.
- **Real nümunə:** Kotlin-də `Dispatchers.Default` CPU core-ları qədər thread saxlayır (ən azı 2). CPU-ağır işi `Dispatchers.IO`-ya (64 thread-ə qədər) köçürmək onu sürətləndirmir: core sayı dəyişmir, yalnız növbə dəyişməsi artır.
- **Fərqləndirici detal:** `Runtime.getRuntime().availableProcessors()` online core-ları sayar. Telefonda böyük və kiçik core-lar eyni sayılır, amma eyni sürətdə deyil; core-lar enerji üçün söndürülə də bilər.

Follow-up nərdivanı:

1. *"8 core-da 100 thread neca işləyir?"* → Vaxtı hissələrə bölərək, context switch ilə (Faza 2).
2. *"SMT həmişə faydalıdır?"* → Xeyr. Eyni növ hesablama edən iki thread az qazanır, çünki eyni ALU-ları gözləyir.
3. *"CPU-ağır iş üçün neçə thread optimaldır?"* → Təxminən core sayı qədər. I/O gözləyən iş üçün daha çox (Faza 3).
4. *"`availableProcessors()` dəyişə bilər?"* → Bəli: core-lar online və offline ola bilər, konteynerdə limit qoyula bilər.

Bug-ı tap

B1.1 — Yükləmə göstəricisi bir yerdə ilişib qalır. `launch` bloku kodu paralel işə salır (coroutine-ləri Faza 9-da öyrənəcəksən; burada bilməli olduğun: bloklar müxtəlif core-larda eyni anda işləyə bilər).

```
var downloaded = 0

suspend fun downloadAll(urls: List<String>) = coroutineScope {
    urls.forEach { url ->
        launch(Dispatchers.IO) {
            download(url)
            downloaded++                // UI-da "$downloaded / ${urls.size}"
        }
    }
}
```

B1.2 — Emulatorda işləyir, telefonda açılan kimi crash.

```
// build.gradle.kts
android {
    defaultConfig {
        ndk { abiFilters += listOf("x86_64") }
    }
}
// MainActivity
init { System.loadLibrary("imagefilter") }
```

B1.3 — "Optimizasiya filter-i 2 dəfə sürətləndirdi" iddiası.

```
val t0 = System.nanoTime()
val out = applyFilter(pixels)        // bir dəfə çağırılır
val t1 = System.nanoTime()
Log.d("perf", "filter: ${(t1 - t0) / 1_000_000} ms")
// köhnə versiya: 48 ms, yeni versiya: 24 ms — "2 dəfə sürətli!"
```

Bu ölçmədə nə səhvdir? Ən azı üç səbəb tap.

Öz sözlərinlə izah et

Bu fəsli bir dostuna 5 cümlə ilə izah et. Yaxşı izahda bunlar var:

- [ ] CPU PC-nin göstərdiyi instruction-ı fetch, decode, execute edir, sonra PC irəliləyir.

- [ ] Register-lər CPU-nun içindəki az sayıda, ən sürətli xanalardır; PC, SP, LR və flag-lər xüsusi register-lərdir.
- [ ] Clock cycle-ları verir; bir instruction bir cycle demək deyil.
- [ ] Pipeline eyni anda bir neçə instruction-ı işləyir; asılılıq və səhv branch təxmini onu yavaşladır.
- [ ] Core müstəqil icra maşınıdır; hardware thread OS thread-i deyil.
- [ ] x86-64 və ARM64 fərqli ISA-lardır; Android telefonlar ARM64-dür.
- [ ] `count++` üç addımdır və iki core arasında artırma itir.

Boş xəritə: aşağıdakı xəritəni kağıza köçür. Hər budaq üçün qeydə baxmadan bir cümlə yaz. Sonra fəslin sonundakı Xülasə şəkli ilə müqayisə et.

> 🖼️ **Şəkil 120:** Boş xəritə · öz sözlərinlə doldur

Keçid qapısı

Qeydə baxmadan cavabla. 8 sualdan ən azı 7-sini mexanizmi ilə (rubrikanın 3-cü səviyyəsi) cavablaya bilirsənsə, Fəsil 2-yə keç. Son iki sual Fəsil 0-dandır.

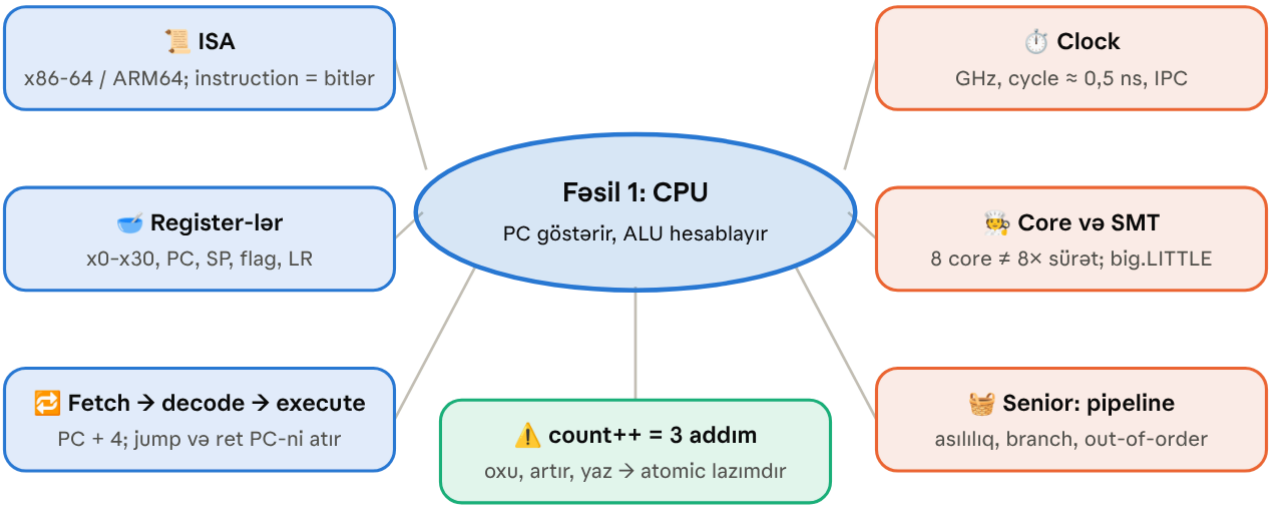
1. **K1.1 (xatırla):** PC, SP və LR nədir? Hər birini bir cümlə ilə de.
2. **K1.2 (hesabla):** 2,5 GHz CPU-da bir cycle neçə nanosaniyədir? İşıq bu müddətdə təxminən neçə santimetr gedir?
3. **K1.3 (nəticəni təxmin et):** ARM64-də 0x1000 ünvanındakı instruction jump deyilsə, ondan sonra PC nə olur? x86-64-də bunu əvvəlcədən niyə deyimək olmur?
4. **K1.4 (niyə):** 4 asılı `imul` 4 müstəqil `imul`-dan niyə təxminən 3 dəfə yavaşdır?
5. **K1.5 (bug-ı tap):** İki core hər biri 1 000 dəfə `count++` edir. Nəticə niyə 2 000-dən az ola bilər? Addım cədvəli ilə göstər və düzəlt.
6. **K1.6 (dizayn qərarı):** CPU-ağır şəkil emalı üçün `Dispatchers.IO`, yoxsa `Dispatchers.Default`? Niyə?
7. **K1.7 (Fəsil 0):** ARM64-ün `ret` instruction-ı 32 bitlik 0xd65f03c0 sözüdür. Telefonun yaddaşında onun byte-ları hansı sıra ilə düzülür?
8. **K1.8 (Fəsil 0):** `csel x0, x0, x1, gt` signed müqayisədir. İki unsigned ədəd (Kotlin `ULong`) üçün hansı şərt lazımdır və niyə fərqli şərt lazımdır?

Yadda saxla

- CPU dövrü: PC-dən fetch → decode → execute → PC irəli (ARM64-də +4) və ya jump.
- Register-lər az və sürətlidir: ARM64-də x0–x30, SP, PC, NZCV; x30 = LR (qayıdış ünvanı).
- Sürəti instruction sayı yox, asılılıqlar və təxminlər təyin edir: pipeline, IPC, branch prediction.
- `count++` üç addımdır (oxu, artır, yaz); paralel artırma üçün atomic instruction lazımdır.
- x86-64 ≠ ARM64: fərqli instruction-lar, fərqli yaddaş modeli; telefonda kod ARM64-də işləyir.

Bir sətirlik mental model: CPU — PC-nin göstərdiyi instruction-ı götürüb icra edən, bunu saniyədə milyardlarla dəfə və eyni anda bir neçə addımda edən maşındır.

Xülasə şəkli



Bir cümlə: CPU barmağı (PC) ilə instruction-ları bir-bir oxuyur, register-lərdə hesablayır, və bu işi saniyədə milyardlarla dəfə təkrarlayır. Bu xəritəni yaddaşdan çəkə bilirsənsə, Fəsil 2-yə hazırsan.

Fəsil 2 — Yaddaş: CPU-nun anbarı və yaxın rəfləri

Bu fəsildən sonra bacaracaqsan

- RAM-ın və ünvanın nə olduğunu, massivin yaddaşda necə düzüldüyünü izah etmək.
- Cache-in niyə lazım olduğunu, cache line-ı, hit və miss-i rəqəmlərlə izah etmək.
- Temporal və spatial locality-ni kodda tanımaq və eyni alqoritmi dəfələrlə sürətləndirmək.
- Yaddaş iyerarxiyasının latency pilləkənini öz maşınında ölçmək: L1, L2, L3, RAM.
- False sharing-i görmək, ölçmək və Kotlin-də düzəltmək.

Əvvəlcə təxmin et

Cavabı kağıza yaz, sonra oxumağa davam et. Cavablar Cavab açarındadır.

1. **T2.1** RAM-dan bir ədəd oxumaq CPU-nun bir toplamasından neçə dəfə uzun çəkir: 2 dəfə, 10 dəfə, yoxsa 100 dəfədən çox?
2. **T2.2** 4096 × 4096 ədədlik cədvəli əvvəl sətir-sətir, sonra sütun-sütun toplayırsan. Hər ikisində eyni say toplama var. Vaxt da eyni olacaq?
3. **T2.3** İki thread iki ayrı dəyişəni artırır, ortaq dəyişən yoxdur. Onlar bir-birini yavaşlada bilərmi?

Sirr

Bu maşında eyni cədvəlin bütün 16 777 216 ədədini iki dəfə topladıq. Birinci dəfə 10 ms çəkdi, ikinci dəfə 150 ms. Kod eynidir, toplama sayı eynidir, nəticə eynidir. Fərq yalnız iki `for` dövrünün sırasındadır.

Sonra cədvəlin hər sətrinə 4 boş xana əlavə etdik: heç kimin oxumadığı 4 ədədlik yer. Yavaş variant 150 ms-dən 89 ms-ə düşdü. Boş yer kodu niyə sürətləndirir? Bu fəsil hər iki suala cavab verir.

> 🖼️ **Şəkil 122:** Sirr · 10, 150 və 89 ms

Burada nəyə bax: iş eynidir, dəyişən yalnız gəzinti sırası və sətrin enidir.

Yeni sözlər

RAM, ünvan, pointer, reference, latency, bandwidth, memory wall, cache, cache hit, cache miss, cache line, temporal locality, spatial locality, L1, L2, L3, working set, row-major, pointer chasing, SRAM, DRAM, set, way, tag, conflict miss, prefetcher, cache coherence, false sharing, padding, TLB, huge page.

Uşaq səviyyəsi: aşpaz, anbar və rəflər

Fəsil 1-dəki aşpazı xatırla. O, çox sürətlidir: saniyədə milyardlarla hərəkət edir. Məhsulların çoxu isə uzaq anbardadır. Anbara gedib-gəlmək təxminən 300–400 hərəkət qədər vaxt aparır.

Hər soğan üçün anbara qaçsa, vaxtının çoxu yolda keçər. Ona görə mətbəxdə rəflər var.

Ən yaxın rəf əlinin altındadır, amma kiçikdir. Bir az aralıda daha böyük şkaf var. Qapının yanında daha böyük soyuducu var, onu bütün aşpazlar paylaşır. Anbar ən böyüyüdür, amma ən uzaqdır.

Aşpaz iki ağıllı vərdiş qazanıb:

- Bir şeyi işlədibsə, onu yaxın rəfdə saxlayır: tezliklə yenə lazım olacaq.
- Anbara gedəndə bir soğan yox, bütöv bir kisə gətirir: yanındakılar da tezliklə lazım olacaq.

> 🖼️ **Şəkil 123:** Uşaq səviyyəsi · aşpazın rəfləri

Burada nəyə bax: hər rəf əvvəlkindən böyükdür, amma aşpazdan daha uzaqdadır.

> 🖼️ **Şəkil 124:** İki vərdiş · temporal və spatial

Burada nəyə bax: bir vərdiş zamanı, o biri məkanı qabaqcadan görür.

Analogiyada	Real sistemdə
Aşpaz	CPU core
Əlin altındakı qablar	Register-lər (Fəsil 1)
Ən yaxın kiçik rəf	L1 cache
Aralıdakı şkaf	L2 cache
Qapı yanındakı ortaq soyuducu	L3 cache
Uzaq anbar	RAM
Şəhər kənarındakı böyük depo	Flash yaddaş, SSD
Anbardakı rəf nömrəsi	Ünvan
Həmişə bütöv gətirilən kisə	Cache line (64 byte)
"İşlətdiyim şey yenə lazım olacaq"	Temporal locality
"Yanındakı da lazım olacaq"	Spatial locality
Rəfdə tapmaq və ya tapmamaq	Cache hit və ya cache miss

Analogiya harada sınır:

- Aşpaz nəyi rəfdə saxlayacağını özü seçir. Real cache-i proqram idarə etmir: hardware özü qərar verir. Proqramçı yalnız data-nı "rəfə yaraşan" şəkildə düzə bilər.
- Mətbəxdə rəfdəki soğan anbardakı soğanın özüdür. Cache isə surətdir: orijinal RAM-da qalır. İki core-un cache-ində eyni data-nın iki surəti ola bilər və onlar uyğunlaşdırılmalıdır (Senior səviyyəsində).
- Kisənin ölçüsü sabitdir: bir byte lazım olsa belə, 64 byte gəlir.

> 🖼️ **Şəkil 125:** Analogiyanın çatışları

Burada nəyə bax: real cache-də seçim hardware-indir və rəfdəki şey orijinal yox, surətdir.

Mühəndis səviyyəsi

1. RAM və ünvan

- **RAM** (Random Access Memory) — proqramın işləyərkən data-sını və instruction-larını saxladığı əsas yaddaş. "Random access" o deməkdir ki, istənilən yerə nömrəsi ilə birbaşa müraciət etmək olar. Kaset kimi əvvəldən sarımaq lazım deyil.
- Elektrik kəsiləndə RAM-dakı hər şey itir. Faylları saxlayan flash yaddaş isə itirmir (Fəsil 0).
- RAM uzun bir byte sırasıdır. Hər byte-ın nömrəsi var: **ünvan** (address). Fəsil 0-dan bilirsən: ünvanı olan ən kiçik vahid byte-dır.
- 64 bitlik CPU-da ünvan 64 bitlik ədəddir və hex ilə yazılır, məsələn `0x5633119cf040`. Proqramın gördüyü ünvanlar əslində "virtual" ünvanlardır, bunu Faza 1-də açacağıq. Bu fəsil üçün ünvanı byte-ın nömrəsi kimi düşün.
- **Pointer** — dəyəri başqa bir şeyin ünvanı olan dəyişən. C-də `int *p` belə dəyişəndir.
- Kotlin-də obyekt saxlayan hər dəyişən də əslində pointer-dir. JVM onu **reference** adlandırır və ünvanı səndən gizlədir.
- Massivin elementləri yaddaşda yan-yana durur. `int a[20]` massivində hər `int` 4 byte-dır. Ona görə `a[i]`-nin ünvanı = `a`-nın başlanğıcı + 4 × i.
- CPU `a[i]`-nin yerini bir vurma və bir toplama ilə tapır. Massivin sürətli olmasının birinci səbəbi budur. İkinci səbəbi aşağıda, cache line-da görəcəksən.

> 🖼️ **Şəkil 126:** RAM · küçə və ev nömrələri

Burada nəyə bax: hər byte-ın öz nömrəsi var və ona birbaşa getmək olur.

> 🖼️ **Şəkil 127:** Massiv · ünvan = başlanğıc + 4 × i

Burada nəyə bax: qonşu elementlərin ünvanları 4-4 artır.

> 🖼️ **Şəkil 128:** Pointer və reference

Burada nəyə bax: C ünvanı göstərir, Kotlin isə eyni şeyi gizlədir.

Dayan və düşün D2.1: `long b[10]` massivi 0x1000 ünvanından başlayır. `long` 8 byte-dır. `b[3]` hansı ünvandadır?

2. Problem: yaddaş CPU-dan çox geri qalır

- **Latency** — bir sorğunun cavabını gözləmə vaxtı. **Bandwidth** — bir saniyədə ötürülən data miqdarı (GB/s).
- Yol analogiyası: latency bir maşının A-dan B-yə getmə vaxtıdır. Bandwidth yolun neçə zolaqlı olduğudur. Zolaq artırmaq bir maşını tezləşdirmir.
- Bu maşında CPU bir sadə toplamaya təxminən 0,5 ns sərf edir (Fəsil 1). RAM-dan bir ədəd oxumaq isə 150–225 ns çəkir (aşağıda ölçəcəyik). Bir RAM gözləməsinin vaxtında yüzlərlə toplama etmək olardı.

- Səbəb on illərlə davam edən bir fərqdir: CPU-lar RAM-ın latency-sindən çox daha sürətlə inkişaf edib. Wulf və McKee 1994-cü ildə bunu **memory wall** (yaddaş divarı) adlandırdılar. Fikirləri belə idi: CPU nə qədər sürətlənsə, proqramın sürətini yaddaş təyin edəcək.
- Həll: CPU-nun yanına kiçik, amma çox sürətli yaddaş qoymaq. Bu yaddaşın adı cache-dir.

> 🖼️ **Şəkil 129:** Latency və bandwidth · yol

Burada nəyə bax: zolaqlar çoxalır, amma hər maşının yolu yenə eyni vaxtı çəkir.

> 🖼️ **Şəkil 130:** Bir RAM gözləməsi · 300–450 toplama

Burada nəyə bax: qırmızı zolağın uzunluğunda yüzlərlə kiçik toplama sığır.

> 🖼️ **Şəkil 131:** Memory wall · sxematik

Burada nəyə bax: iki xətt arasındakı boşluq illər keçdikcə böyüyür — cache bu boşluğu doldurur.

3. Cache: tez-tez lazım olanın yaxındakı surəti

- **Cache** — RAM-dakı data-nın bir hissəsinin CPU-nun içindəki surəti. Kiçikdir (kilobaytlar, meqabaytlar), amma RAM-dan onlarla dəfə sürətlidir.
- CPU bir ünvanı oxumaq istəyəndə əvvəlcə cache-ə baxır.
- **Cache hit** — data cache-dədir. Cavab bir neçə nanosaniyədə gəlir.
- **Cache miss** — data cache-də yoxdur. CPU RAM-a gedir, data-nı həm istifadə edir, həm də cache-ə yazır. Yer açmaq üçün köhnə bir şey cache-dən çıxarılır.
- Proqram cache-i görmür. Eyni `a[i]` kodu hit-də də, miss-də də eyni nəticəni verir, sadəcə fərqli sürətlə. Cache nəticəni yox, yalnız vaxtı dəyişir.
- Cache niyə işləyir? Çünki proqramlar yaddaşı təsadüfi oxumur. Onların iki vərdişi var, bunlara locality deyilir.

> 🖼️ **Şəkil 132:** Cache hekayəsi · masa və arxiv

Burada nəyə bax: masada olan kitab dərhal oxunur, olmayan üçün uzun yol gəlmək lazımdır.

> 🖼️ **Şəkil 133:** Hit və miss · axın

Burada nəyə bax: miss yalnız gecikmə deyil; data cache-ə yazılır və başqa bir line çıxarılır.

> 🖼️ **Şəkil 134:** Cache vaxtı dəyişir, nəticəni yox

Burada nəyə bax: hər iki halda x = 42, fərq yalnız zolağın uzunluğundadır.

4. Locality: proqramların iki vərdişi

- **Temporal locality** (zaman yaxınlığı): indi işlədilən data tezliklə yenə işlədiləcək. Nümunə: dövrdəki `sum` dəyişəni, tez-tez çağırılan funksiyanın kodu.
- **Spatial locality** (məkan yaxınlığı): indi işlədilən ünvanın qonşuları da tezliklə lazım olacaq. Nümunə: massivi ardıcıl gəzmək, instruction-ları ardıcıl icra etmək.
- Cache temporal locality-dən faydalanır, çünki son işlədiləni saxlayır.
- Spatial locality-dən faydalanır, çünki tək byte yox, bütöv bir blok gətirir. Bu bloka cache line deyilir.

> 🖼️ **Şəkil 135:** Temporal locality · sum

Burada nəyə bax: eyni dəyişən hər iterasiyada yenidən lazım olur.

> 🖼️ **Şəkil 136:** Spatial locality · qonşular

Burada nəyə bax: bir miss 16 elementi birlikdə gətirir.

> 🖼️ **Şəkil 137:** Cache-in dostu və düşməni

Burada nəyə bax: sol tərəfdə hər ox yeni yerə sıçrayır, sağ tərəfdə isə bir istiqamətdə axır.

5. Cache line: cache-in "kisəsi"

- **Cache line** — cache ilə RAM arasında daşınan ən kiçik blok. x86-64-də və Android telefonlarındakı ARM64 core-larda 64 byte-dır.
- 64 byte = 16 ədəd `int`. Massiv line sərhədinə düşürsə, `a[0]`-ı oxuyanda CPU `a[0]`–`a[15]`-i birlikdə gətirir. Sonrakı 15 oxunuş pulsuz hit olur.
- Line-lar ünvana görə sabit bölünür. Ünvan / 64 = line nömrəsi, ünvan % 64 = line daxilindəki yer. Özün gör bölməsində bunu real ünvanlarla görəcəksən.
- **Row-major** düzülüş: 2D cədvəl yaddaşda sətir-sətir saxlanılır. Əvvəl 0-cı sətrin bütün elementləri, sonra 1-ci sətir və s. C-də və Kotlin-in `IntArray(n * n)` yazılışında belədir. Element `[i][j]` = `m[i * n + j]`.
- Hesab: 4096 × 4096 cədvəl 16 777 216 `int`-dir. Sətir-sətir gəzəndə hər 16 oxunuşdan biri miss-dir: 16 777 216 / 16 = 1 048 576 miss, yəni oxunuşların 6,25%-i.
- Sütun-sütun gəzəndə hər addım bir sətir, yəni 4096 × 4 = 16 384 byte atlayır. Hər oxunuş yeni line-a düşür: miss təxminən 100%.

> 🖼️ **Şəkil 138:** Cache line · bütöv kisə

Burada nəyə bax: RAM-dan L1-ə həmişə bütöv 64 byte düşür.

> 🖼️ **Şəkil 139:** Ünvan · line nömrəsi və offset

Burada nəyə bax: bölmənin tam hissəsi line-ı, qalığı line daxilindəki yeri verir.

> 🖼️ **Şəkil 140:** Row-major · sətir və sütun gəzintisi

Burada nəyə bax: göy oxlar yaddaşın sırası ilə gedir, qırmızılar hər addımda bir sətir atlayır.

Dayan və düşün D2.2: `long` 8 byte-dır. Bir cache line-da neçə `long` yerləşir? `LongArray`-i ardıcıl gəzəndə oxunuşların neçə faizi miss olar?

6. Cache iyerarxiyası: L1, L2, L3

Bir böyük və eyni zamanda sürətli cache qurmaq olmur. Cache nə qədər böyükdürsə, içində axtarış və siqnalın yolu o qədər uzundur. Ona görə bir neçə səviyyə qurulur, hər biri əvvəlkindən böyük və yavaş.

Ən yuxarıda register-lər durur: onlar core-un içindədir və bir cycle-da oxunur (Fəsil 1). Sonra cache-lər gəlir. Bu maşının rəqəmləri (ölçülər `/sys`-dən, latency-lər Özün gör bölməsindəki ölçmədən):

Səviyyə	Bu maşında ölçü	Kimindir	Ölçülmüş latency
L1d (data)	48 KB, 12-way	hər core-un özü	~1,6–1,8 ns
L1i (instruction)	32 KB, 8-way	hər core-un özü	ölçülməyib
L2	2 MB, 16-way	hər core-un özü	~5–7 ns
L3	260 MB (lscpu); effektiv pay xeyli az	bütün core-lar ortaq	~30–75 ns
RAM	8 GB	hamısı	~150–225 ns

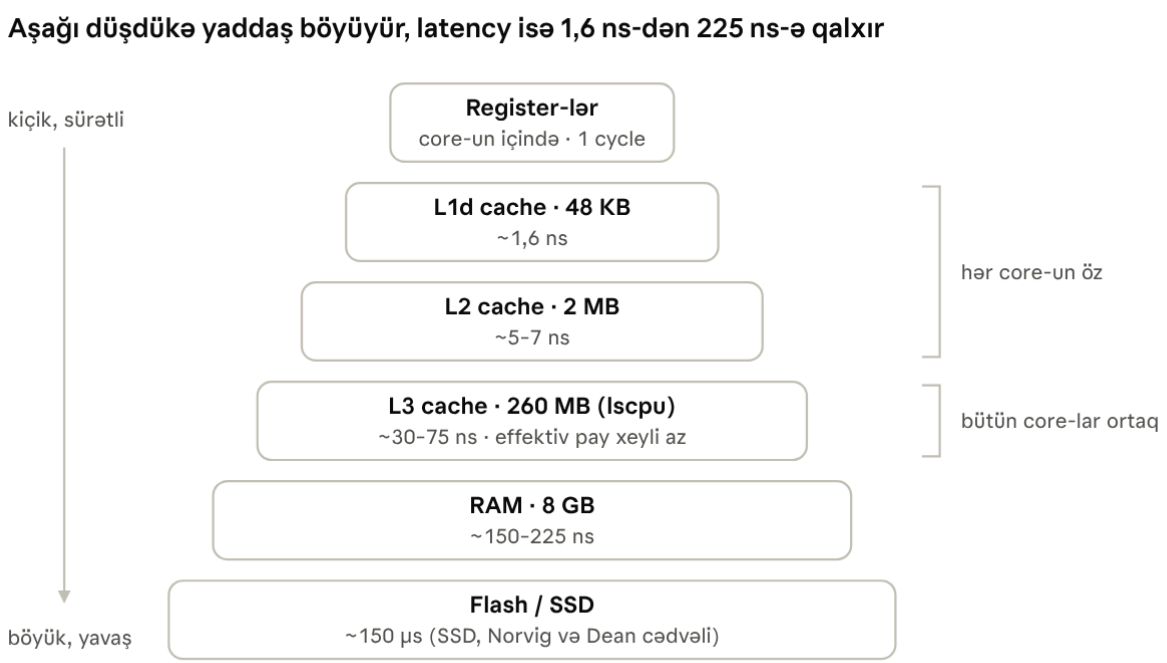
- CPU əvvəl L1-ə baxır. Yoxdursa L2-yə, sonra L3-ə, sonra RAM-a.
- L1 data və instruction üçün ayrıdır (L1d və L1i): CPU eyni anda həm instruction götürür, həm də data oxuyur.
- "12-way" sözünün mənasını Senior səviyyəsində açacağıq. Sirr-in cavabı da orada gizlənib.
- L3 sətri qəribədir: lscpu 260 MB göstərir, amma ölçmədə latency 32 MB-dan sonra RAM səviyyəsinə qalxır. Bu maşın virtual maşındır: L3 böyük server CPU-sunun bütün core-ları və başqa virtual maşınlarla ortaqdır. Bizə düşən effektiv pay xeyli kiçikdir (ehtimal olunan izah; dəqiq bölgünü virtual maşın içindən görmək olmur).
- **Working set** — proqramın müəyyən müddətdə toxunduğu data-nın həcmi. Working set L1-ə sığırsa, proqram uçur. RAM-a çıxırsa, sürünür.

> 🖼️ **Şəkil 141:** Working set · hansı qutuya sığır

Burada nəyə bax: data böyüdükçə bir qutudan çıxıb daha böyük və yavaş qutuya düşür.

> 🖼️ **Şəkil 142:** Kimin cache-i · core-un özü və ortaq

Burada nəyə bax: hər core-un öz L1 və L2-si var, L3 və RAM ortaqdır.



Şəkli yuxarıdan aşağı oxu: hər pillə əvvəlkindən böyükdür, amma yavaşdır. L1 və L2 hər core-un özünündür, L3-ü core-lar paylaşır.

7. Latency rəqəmləri

Peter Norvig "Teach Yourself Programming in Ten Years" esse-sində bir cədvəl verib. Jeff Dean onu genişləndirib və cədvəl "Latency numbers every programmer should know" adı ilə yayılıb (təxminən 2012). Rəqəmlər köhnədir, amma nisbətlər indi də doğrudur. Yanında bu maşının ölçüləri:

Əməliyyat	Norvig və Dean cədvəli	Bu maşında ölçülən	L1-ə nisbətən (bu maşın)
L1 cache-dən oxumaq	0,5 ns	1,6–1,8 ns	1
Branch misprediction	5 ns	təxminən 6,8 ns (Fəsil 1)	~4
L2 cache-dən oxumaq	7 ns	5–7 ns	~3–4
L3 cache-dən oxumaq	cədvəldə yoxdur	30–75 ns	~20–45
Mutex lock/unlock	25 ns	Faza 4-də ölçəcəyik	—
RAM-dan oxumaq	100 ns	150–225 ns	~90–130
SSD-dən təsadüfi 4 KB oxumaq	150 000 ns (150 µs)	ölçülməyib	—
Diskdə yeni yerə keçid (seek)	8 000 000 ns (8 ms)	ölçülməyib	—
Paketin Kaliforniya–Niderland–Kaliforniya yolu	150 000 000 ns (150 ms)	ölçülməyib	—

- Bu maşında RAM cədvəldəkindən yavaşdır. Səbəblər: böyük server çipində yol uzundur, virtual maşın əlavə qat qoyur. Masaüstü kompyuterdə rəqəm çox vaxt 100 ns-ə yaxındır. Öz maşınında ölç, çünki dəqiq dəyər hardware-dən asılıdır.

- İnsan miqyası: L1 oxunuşu 1 saniyə olsaydı, bu maşında RAM oxunuşu təxminən 2 dəqiqə olardı (225 / 1,7 ≈ 132).
- Əzbərlə böyüklük sıralarını: L1 ~1 ns, L2 ~5 ns, L3 ~onlarla ns, RAM ~100 ns, SSD ~100 µs, okean aşırı şəbəkə ~100 ms. Dizayn qərarlarının çoxu bu sıralardan çıxır.

> 🖼️ **Şəkil 144:** İnsan miqyası · L1 = 1 saniyə

Burada nəyə bax: saniyə ilə başlayır, illərlə bitir. RAM artıq iki dəqiqəlik gözləmədir.

> 🖼️ **Şəkil 145:** Loqarifmik şkala · 1 ns-dən 150 ms-ə

Burada nəyə bax: hər qırıq xətt 10 dəfə artımdır; zolağın uzunluğu ədədin neçə rəqəmli olduğunu göstərir.

> 🖼️ **Şəkil 146:** Altı pillə · əzbərlə

Burada nəyə bax: altı rəqəm kifayətdir — qalanını onlardan təxmin etmək olur.

Özün gör

Mühit: Linux 6.18, x86-64 virtual maşın (KVM, 2 core), gcc 13.3, valgrind 3.22, OpenJDK 21, kotlinc 2.1.0. Çıxışlar real-dır.

1. Cache-ləri tanı

```
$ getconf -a | grep -E 'LEVEL1_DCACHE|LEVEL2_CACHE|LEVEL3_CACHE'
LEVEL1_DCACHE_SIZE                 49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  2097152
LEVEL2_CACHE_ASSOC                 16
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  272629760
LEVEL3_CACHE_ASSOC                 20
LEVEL3_CACHE_LINESIZE              64

$ for d in /sys/devices/system/cpu/cpu0/cache/index*; do echo "L$(cat $d/level) $(cat $d/type):
$(cat $d/size), line $(cat $d/coherency_line_size) B, $(cat $d/ways_of_associativity)-way, $(cat
$d/number_of_sets) set, CPU-lar: $(cat $d/shared_cpu_list)"; done
L1 Data: 48K, line 64 B, 12-way, 64 set, CPU-lar: 0
L1 Instruction: 32K, line 64 B, 8-way, 64 set, CPU-lar: 0
L2 Unified: 2048K, line 64 B, 16-way, 2048 set, CPU-lar: 0
L3 Unified: 266240K, line 64 B, 20-way, 212992 set, CPU-lar: 0-1
```

- `LEVEL1_DCACHE_SIZE 49152` — L1d 48 KB-dır. `LINESIZE 64` — bütün səviyyələrdə cache line 64 byte-dır.
- `CPU-lar: 0` — L1 və L2 yalnız 0 nömrəli core-undur. `CPU-lar: 0-1` — L3-ü hər iki core paylaşır.
- `ASSOC`, `way` və `set` sözlərinə Senior səviyyəsində qayıdacağıq.

- Android cihazında eyni qovluğa bax: `adb shell cat /sys/devices/system/cpu/cpu0/cache/index0/size`. Bəzi telefon kernel-ləri bu qovluğu göstərmir. Burada işlədilməyib — cihazda yoxla.

> 🖼️ **Şəkil 147:** /sys · cache-lər qovluq kimi

Burada nəyə bax: hər cache səviyyəsi öz qovluğudur, hər xüsusiyyəti öz faylıdır.

> 🖼️ **Şəkil 148:** Çıxışı oxu · sətir və məna

Burada nəyə bax: hər çıxış sətrinin solunda xam rəqəm, sağında onun mənası var.

> 🖼️ **Şəkil 149:** Yoxlama · set × way × line

Burada nəyə bax: üç müstəqil faylın rəqəmləri bir-birini təsdiqləyir.

2. Massiv yaddaşda necə düzülür

`addr.c` massivin hər elementinin ünvanını, line nömrəsini və line daxilindəki yerini çap edir:

```c
// Massivin elementləri yaddaşda yan-yana durur: ünvanlara və cache line nömrələrinə bax
#include <stdio.h>
#include <stdint.h>

int a[20];

int main(void) {
    for (int i = 0; i < 20; i++) {
        uintptr_t addr = (uintptr_t)&a[i];
        printf("a[%2d]  unvan = 0x%lx  cache line = %lu  line daxilinde = %2lu\n",
               i, addr, addr / 64, addr % 64);
    }
    return 0;
}
```

```
$ gcc -O2 -o addr addr.c && ./addr        (çıxışın bir hissəsi)
a[ 0]  unvan = 0x5633119cf040  cache line = 1480895919041  line daxilinde =  0
a[ 1]  unvan = 0x5633119cf044  cache line = 1480895919041  line daxilinde =  4
a[ 2]  unvan = 0x5633119cf048  cache line = 1480895919041  line daxilinde =  8
…
a[14]  unvan = 0x5633119cf078  cache line = 1480895919041  line daxilinde = 56
a[15]  unvan = 0x5633119cf07c  cache line = 1480895919041  line daxilinde = 60
a[16]  unvan = 0x5633119cf080  cache line = 1480895919042  line daxilinde =  0
a[17]  unvan = 0x5633119cf084  cache line = 1480895919042  line daxilinde =  4
```

- Hər növbəti element 4 byte irəlidədir: `…040`, `…044`, `…048`. Bu, `int`-in ölçüsüdür.
- `a[0]`–`a[15]` eyni line-dadır (...041). `a[16]` yeni line-ı açır (...042). Bir line = 16 `int`.
- Ünvanı `%lu` ilə `unsigned long` kimi çap edirik: x86-64 Linux-da `uintptr_t` 64 bitlikdir.

> 🖼️ **Şəkil 150:** addr.c · 16 int bir line-da

Burada nəyə bax: a[15]-dən a[16]-ya keçəndə line nömrəsi bir artır, yer isə +0-a qayıdır.

Kotlin-də: JVM ünvanı göstərmir, amma massivin düzülüşünü göstərə bilər. `Layout.kt` JDK-nın daxili `Unsafe` sinfindən soruşur: data massivin əvvəlindən neçə byte sonra başlayır, bir element neçə byte-dır.

```
        // JVM massivin yaddaşdakı düzülüşünü göstərir: header neçə byte-dır, element neçə byte-
dır
        import sun.misc.Unsafe

        fun main() {
            val f = Unsafe::class.java.getDeclaredField("theUnsafe")
            f.isAccessible = true
            val u = f.get(null) as Unsafe
            fun show(name: String, c: Class<*>) =
                println("%-16s header (data-nin baslangici) = %2d byte, bir element = %d byte"
                    .format(name, u.arrayBaseOffset(c), u.arrayIndexScale(c)))
            show("ByteArray", ByteArray::class.java)
            show("IntArray", IntArray::class.java)
            show("LongArray", LongArray::class.java)
            show("Array<IntArray>", Array<IntArray>::class.java)
            show("Array<Int>", Array<Int>::class.java)
        }
```

```
$ kotlinc Layout.kt -include-runtime -d Layout.jar && java -jar Layout.jar
ByteArray        header (data-nin baslangici) = 16 byte, bir element = 1 byte
IntArray         header (data-nin baslangici) = 16 byte, bir element = 4 byte
LongArray        header (data-nin baslangici) = 16 byte, bir element = 8 byte
Array<IntArray>  header (data-nin baslangici) = 16 byte, bir element = 4 byte
Array<Int>       header (data-nin baslangici) = 16 byte, bir element = 4 byte
```

- Hər JVM massivinin əvvəlində 16 byte-lıq **header** var: obyektin xidməti məlumatı, o cümlədən massivin uzunluğu. Data ondan sonra, yan-yana gəlir.
- `IntArray`-də element 4 byte-dır: C-dəki `int a[]` kimi. `LongArray`-də 8 byte.
- `Array<IntArray>` və `Array<Int>`-də element 4 byte-dır, çünki orada ədəd yox, **reference** saxlanılır. Ədədlər və ya sətirlər heap-də ayrıca obyektlərdir. Reference-in 4 byte olması JVM-in "compressed references" optimallaşdırmasıdır (heap 32 GB-dan kiçik olanda).
- Android-da ART-ın header ölçüsü fərqli ola bilər. Burada işlədilməyib — cihazda yoxla.

> 🖼️ **Şəkil 151:** JVM IntArray · header və data

Burada nəyə bax: ilk 16 byte xidmət üçündür, data ondan sonra C massivi kimi düzülür.

> 🖼️ **Şəkil 152:** IntArray və Array<Int>

Burada nəyə bax: yuxarıda ədədlər yan-yanadır, aşağıda isə hər ref başqa yerə aparır.

3. Lab 0.1 — Cache təsiri: sətir-sətir və sütun-sütun

Hipotez: 2D massivi sütun-sütun gəzmək sətir-sətir gəzməkdən xeyli yavaşdır. Qurulum: 4096 × 4096 `int` (64 MB), hər element bir dəfə oxunur, 5 təkrarın medianı. İkinci arqument `W` sətrin yaddaşdakı enidir: `W > N` olanda hər sətrin sonunda boş xanalar qalır (Sirr-dəki təcrübə).

> 🖼️ **Şəkil 153:** Lab 0.1 · kitab analogiyası

Burada nəyə bax: solda bir səhifə sona qədər oxunur, sağda hər sözdən sonra səhifə çevrilir.

```c
        // Lab 0.1: eyni massivi sətir-sətir və sütun-sütun gəzmək
        // İstifadə: ./rowcol [N] [W]   N = sətir və sütun sayı, W = sətrin yaddaşdakı eni (W >=
N)
        #include <stdio.h>
        #include <stdlib.h>
        #include <time.h>

        static double now(void) {
            struct timespec t;
            clock_gettime(CLOCK_MONOTONIC, &t);
            return t.tv_sec + t.tv_nsec / 1e9;
        }

        static int cmp(const void *x, const void *y) {
            double a = *(const double *)x, b = *(const double *)y;
            return (a > b) - (a < b);
        }

        int main(int argc, char **argv) {
            int n = argc > 1 ? atoi(argv[1]) : 4096;
            int w = argc > 2 ? atoi(argv[2]) : n;
            int *m = malloc((size_t)n * w * sizeof(int));
            for (size_t i = 0; i < (size_t)n * w; i++) m[i] = 1;   // hər səhifəyə toxun

            double row[5], col[5];
            long sum = 0;
            for (int r = 0; r < 5; r++) {
                double t0 = now();
                for (int i = 0; i < n; i++)          // sətir-sətir: ünvanlar ardıcıl
                    for (int j = 0; j < n; j++)
                        sum += m[(size_t)i * w + j];
                double t1 = now();
                for (int j = 0; j < n; j++)          // sütun-sütun: hər addım w*4 byte atlayır
                    for (int i = 0; i < n; i++)
                        sum += m[(size_t)i * w + j];
                double t2 = now();
                row[r] = t1 - t0;
                col[r] = t2 - t1;
            }
            qsort(row, 5, sizeof(double), cmp);
            qsort(col, 5, sizeof(double), cmp);
            printf("N=%d W=%d (%.0f MB)  setir: %.1f ms  sutun: %.1f ms  ferq: %.1f defe
(sum=%ld)\n",
                   n, w, (double)n * w * 4 / (1 << 20), row[2] * 1e3, col[2] * 1e3, col[2] /
row[2], sum);
            free(m);
            return 0;
        }
```

```
$ gcc -O2 -o rowcol rowcol.c && ./rowcol && ./rowcol
N=4096 W=4096 (64 MB)  setir: 10.4 ms  sutun: 150.5 ms  ferq: 14.5 defe  (sum=167772160)
N=4096 W=4096 (64 MB)  setir: 11.1 ms  sutun: 158.4 ms  ferq: 14.3 defe  (sum=167772160)
$ ./rowcol 4096 4100
N=4096 W=4100 (64 MB)  setir: 12.0 ms  sutun: 89.4 ms  ferq: 7.4 defe  (sum=167772160)
$ ./rowcol 4096 4112
N=4096 W=4112 (64 MB)  setir: 12.3 ms  sutun: 99.8 ms  ferq: 8.1 defe  (sum=167772160)
```

- Eyni sayda toplama, eyni cəm, amma sütun variantı 14,5 dəfə yavaşdır. Hipotez təsdiqləndi.
- `sum` dəyəri hər iki gəzintidə eynidir: cache nəticəni dəyişmir, yalnız vaxtı.
- `W=4100`: sətrə 4 boş xana əlavə etdik və sütun gəzintisi 150 ms-dən 89 ms-ə düşdü. Bunun səbəbini Senior səviyyəsində açacağıq.

> 🖼️ **Şəkil 154:** Lab 0.1 · C nəticələri

Burada nəyə bax: göy zolaqlar demək olar dəyişmir, qırmızı zolaq isə W = 4100-də xeyli qısalır.

Kotlin versiyası (`RowCol.kt`): eyni təcrübə iki düzülüşlə. Düz `IntArray(N * N)` C massivi kimidir. `Array<IntArray>`-də isə hər sətir ayrıca obyektdir.

```
        // Lab 0.1-in Kotlin versiyası: eyni massivi sətir-sətir və sütun-sütun gəzmək
        const val N = 4096

        fun median(xs: DoubleArray): Double = xs.sorted()[xs.size / 2]

        fun time(block: () -> Long): Pair<Double, Long> {
            val t0 = System.nanoTime()
            val s = block()
            return (System.nanoTime() - t0) / 1e6 to s
        }

        fun main() {
            // 1) Düz IntArray: N*N element yaddaşda ardıcıl düzülür (C massivi kimi)
            val flat = IntArray(N * N) { 1 }
            // 2) Array<IntArray>: hər sətir ayrı obyektdir, sətirlər heap-də ayrı-ayrı yerlərdə
ola bilər
            val rows = Array(N) { IntArray(N) { 1 } }

            val fr = DoubleArray(5); val fc = DoubleArray(5)
            val rr = DoubleArray(5); val rc = DoubleArray(5)
            var sink = 0L
            repeat(5) { r ->
                time { var s = 0L; for (i in 0 until N) for (j in 0 until N) s += flat[i * N +
j]; s }
                    .also { fr[r] = it.first; sink += it.second }
                time { var s = 0L; for (j in 0 until N) for (i in 0 until N) s += flat[i * N +
j]; s }
                    .also { fc[r] = it.first; sink += it.second }
                time { var s = 0L; for (i in 0 until N) { val row = rows[i]; for (j in 0 until N)
s += row[j] }; s }
                    .also { rr[r] = it.first; sink += it.second }
                time { var s = 0L; for (j in 0 until N) for (i in 0 until N) s += rows[i][j]; s }
                    .also { rc[r] = it.first; sink += it.second }
            }
            println("IntArray(N*N):    setir %.1f ms, sutun %.1f ms, ferq %.1f
defe".format(median(fr), median(fc), median(fc) / median(fr)))
            println("Array<IntArray>:  setir %.1f ms, sutun %.1f ms, ferq %.1f
defe".format(median(rr), median(rc), median(rc) / median(rr)))
            println("(sink=$sink)")
        }
```

```
$ kotlinc RowCol.kt -include-runtime -d RowCol.jar && java -Xmx3g -jar RowCol.jar
IntArray(N*N):    setir 15.0 ms, sutun 150.2 ms, ferq 10.0 defe
Array<IntArray>:  setir 20.1 ms, sutun 177.0 ms, ferq 8.8 defe
(sink=335544320)
```

- Kotlin-də də sütun gəzintisi təxminən 10 dəfə yavaşdır. JIT dövrün sırasını özü dəyişmir: sıranı proqramçı seçir.
- `Array<IntArray>` hər iki variantda bir az yavaşdır: hər addımda əvvəl sətrin reference-ini oxumaq lazımdır.
- `-Xmx3g` JVM-ə 3 GB-a qədər heap verir: iki 64 MB-lıq cədvəl və ehtiyat üçün.

> 🖼️ **Şəkil 155:** Lab 0.1 · Kotlin nəticələri

Burada nəyə bax: Kotlin-də də sütun zolağı təxminən 10 dəfə uzundur; Array<IntArray> hər addımda əlavə bir ref oxuyur.

Miss-ləri say: cachegrind. `perf stat -e cache-misses` real CPU-nun sayğaclarını oxuyur, amma bu virtual maşında `perf` yoxdur. Valgrind-in `cachegrind` aləti isə cache-i proqram ilə simulyasiya edir. `walk.c` cədvəli bir dəfə doldurur (yazı) və bir dəfə gəzir (oxu). Aşağıda çıxışın əsas sətirləri var, hər sətrin əvvəlindəki `==PID==` prefiksi silinib:

```
$ valgrind --tool=cachegrind --cache-sim=yes ./walk row
D refs:         33,625,136  (16,835,405 rd   + 16,789,731 wr)
D1  misses:      2,098,968  ( 1,049,988 rd   +  1,048,980 wr)
D1  miss rate:         6.2% (       6.2%     +        6.2%  )
$ valgrind --tool=cachegrind --cache-sim=yes ./walk col
D refs:         33,625,136  (16,835,405 rd   + 16,789,731 wr)
D1  misses:     17,827,607  (16,778,627 rd   +  1,048,980 wr)
D1  miss rate:        53.0% (      99.7%     +        6.2%  )
```

- `rd` sütunu gəzintinin oxunuşlarıdır. Sətir variantında 1 049 988 miss: düz hesabladığımız 16 777 216 / 16 = 1 048 576-ya yaxın, yəni 6,2%.
- Sütun variantında oxunuşların 99,7%-i miss-dir: hər oxunuş yeni line.
- `wr` sütunu doldurma dövrüdür. O, hər iki proqramda ardıcıldır, ona görə hər ikisində 6,2%-dir.
- Cachegrind iki səviyyə simulyasiya edir (D1 və son səviyyə) və prefetcher-i nəzərə almır. Rəqəmlər real CPU-nun sayğaclarına yaxın, amma eyni deyil.
- **Variasiya:** `./rowcol 1024` (4 MB) və `./rowcol 256` (256 KB) işlət. Cədvəl L2-yə və ya L1-ə sığanda fərq necə dəyişir? Əvvəlcə təxmin et, sonra ölç.

> 🖼️ **Şəkil 156:** Cachegrind · 16-da 1 və 16-da 16

Burada nəyə bax: üst sırada bir qırmızı nöqtə var, alt sırada isə hamısı qırmızıdır.

4. Latency pilləkəni: hər səviyyəni ayrıca ölç

Sualımız: bir oxunuş nə qədər çəkir və bu, data-nın ölçüsündən necə asılıdır? Adi dövr bunu ölçə bilmir, çünki CPU növbəti oxunuşları qabaqcadan başlayır. Ona görə **pointer chasing** işlədirik: hər oxunuşun nəticəsi növbəti oxunuşun ünvanıdır.

- Hər node ayrıca cache line-dadır (64 byte aralıqla).
- Node-lar təsadüfi sırada bir böyük dövrəyə bağlanır (Sattolo alqoritmi). Prefetcher növbəti ünvanı təxmin edə bilmir.
- CPU oxunuşu bitirmədən növbətinin ünvanını bilmir. Deməli, ölçülən vaxt = bir oxunuşun təmiz latency-si.

> 🖼️ **Şəkil 157:** Pointer chasing · xəzinə oyunu

Burada nəyə bax: növbəti oxun haraya gedəcəyi yalnız indiki qeyd oxunandan sonra məlum olur.

```c
        // Pointer chasing: hər oxunuş əvvəlki oxunuşun nəticəsindən asılıdır,
        // ona görə CPU onları qabaqcadan yükləyə bilmir. Ölçülən vaxt = yaddaşın latency-si.
        // İstifadə: ./chase [huge]   huge = 2 MB-lıq huge page istə (TLB təsirini azaldır)
        #define _GNU_SOURCE
        #include <stdio.h>
        #include <stdlib.h>
        #include <stdint.h>
        #include <string.h>
        #include <time.h>
        #include <sys/mman.h>

        static double now(void) {
            struct timespec t;
            clock_gettime(CLOCK_MONOTONIC, &t);
            return t.tv_sec + t.tv_nsec / 1e9;
        }

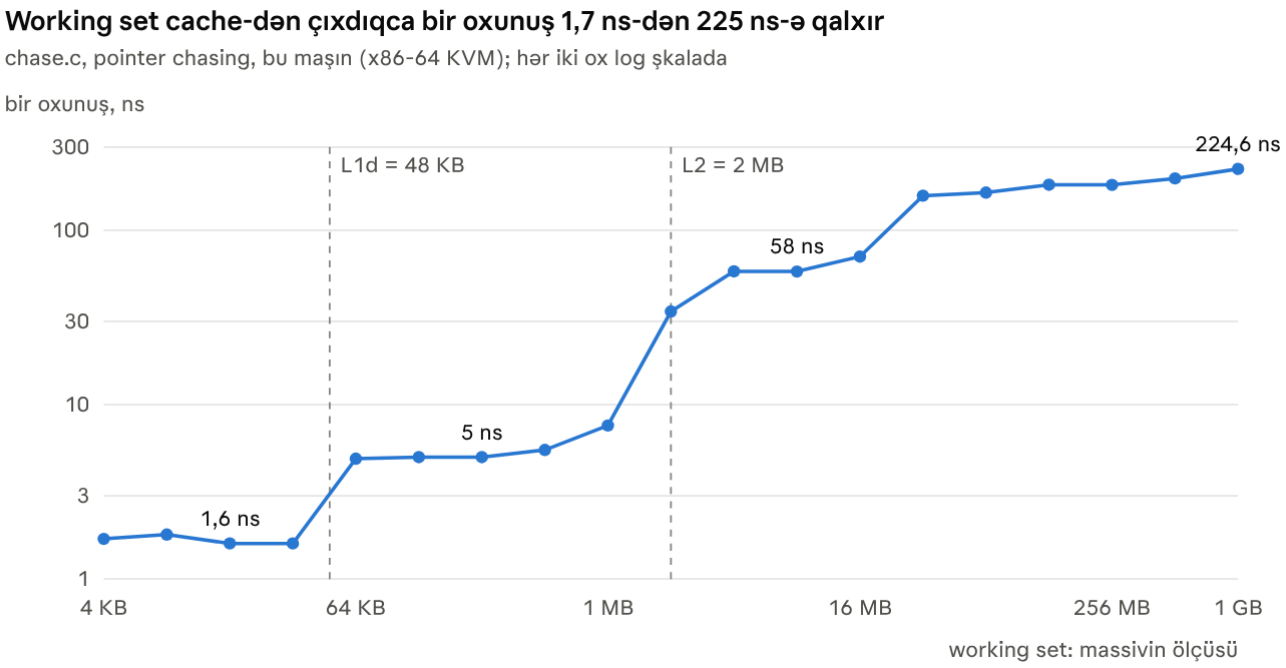
        static uint64_t rng = 88172645463325252ull;
        static uint64_t next_rand(void) {            // xorshift64: sadə təsadüfi ədəd generatoru
            rng ^= rng << 13; rng ^= rng >> 7; rng ^= rng << 17;
            return rng;
        }

        int main(int argc, char **argv) {
            int huge = argc > 1 && strcmp(argv[1], "huge") == 0;
            for (size_t size = 4 << 10; size <= ((size_t)1 << 30); size *= 2) {
                size_t lines = size / 64;            // hər node ayrı cache line-da
                uint64_t *a = aligned_alloc(2 << 20, size < (2 << 20) ? (2 << 20) : size);
                if (huge) madvise(a, size, MADV_HUGEPAGE);
                // Sattolo alqoritmi: bütün line-lardan keçən bir böyük təsadüfi dövrə
                uint64_t *order = malloc(lines * sizeof(uint64_t));
                for (size_t i = 0; i < lines; i++) order[i] = i;
                for (size_t i = lines - 1; i > 0; i--) {
                    size_t j = next_rand() % i;
                    uint64_t t = order[i]; order[i] = order[j]; order[j] = t;
                }
                for (size_t i = 0; i < lines; i++)
                    a[order[i] * 8] = order[(i + 1) % lines] * 8;   // növbəti node-un indeksi
                free(order);

                size_t steps = 20000000, p = 0;
                for (size_t i = 0; i < lines; i++) p = a[p];        // isinmə
                double t0 = now();
                for (size_t i = 0; i < steps; i++) p = a[p];        // asılı oxunuşlar zənciri
                double t1 = now();
                printf("%9zu KB  %6.1f ns\n", size >> 10, (t1 - t0) / steps * 1e9);
                if (p == 12345) puts("");                           // kompilyator dövrü
silməsin
                free(a);
            }
            return 0;
        }
```

```
$ gcc -O2 -o chase chase.c && ./chase
        4 KB     1.7 ns
        8 KB     1.8 ns
       16 KB     1.6 ns
       32 KB     1.6 ns
```

```
    64 KB     4.9 ns
   128 KB     5.0 ns
   256 KB     5.0 ns
   512 KB     5.5 ns
  1024 KB     7.6 ns
  2048 KB    34.2 ns
  4096 KB    58.1 ns
  8192 KB    58.0 ns
 16384 KB    70.6 ns
 32768 KB   157.8 ns
 65536 KB   164.4 ns
131072 KB   182.2 ns
262144 KB   182.0 ns
524288 KB   198.1 ns
1048576 KB  224.6 ns
```



Qrafikdə dörd pillə görünür, hər biri bir yaddaş səviyyəsidir:

- **32 KB-a qədər, ~1,6–1,8 ns:** hər şey L1d-yə (48 KB) sığır.
- **64 KB–1 MB, ~5–7,6 ns:** L1-dən çıxdıq, L2-yə (2 MB) sığırıq.
- **2–16 MB, ~34–71 ns:** L2-dən çıxdıq, L3-dəyik. Pillə düz 2 MB-da başlayır: təsadüfi düzülüşdə cache-in hamısını doldurmaq olmur, bir hissə line-lar eyni set-ə düşüb bir-birini qovur.
- **32 MB-dan yuxarı, ~158–225 ns:** RAM. Iscpu 260 MB L3 desə də, effektiv pay təxminən 16–32 MB-dır.
- 64 MB ilə 1 GB arasında da artım var: 164 → 225 ns. Bu, ünvan tərcüməsinin (TLB) payıdır, Senior səviyyəsində ölçəcəyik.

Kotlin versiyası (`Chase.kt`): eyni alqoritm, `IntArray` ilə. Hər node 16 `int` = 64 byte aralıqla durur.

```kotlin
// chase.c-nin Kotlin versiyası: asılı oxunuşlar zənciri ilə yaddaşın latency-si
import kotlin.random.Random

fun main() {
    val rnd = Random(42)
    for (e in 12..30) {                          // 4 KB-dan 1 GB-a qədər
```

```
                val size = 1 shl e                          // byte
                val lines = size / 64
                val a = IntArray(size / 4)                  // hər node 16 int = 64 byte = bir cache
line
                val order = IntArray(lines) { it }
                for (i in lines - 1 downTo 1) {             // Sattolo: bir böyük təsadüfi dövrə
                    val j = rnd.nextInt(i)
                    val t = order[i]; order[i] = order[j]; order[j] = t
                }
                for (i in 0 until lines) a[order[i] * 16] = order[(i + 1) % lines] * 16
                var p = 0
                repeat(lines) { p = a[p] }                  // isinmə
                val steps = 20_000_000
                val t0 = System.nanoTime()
                for (i in 0 until steps) p = a[p]           // hər oxunuş əvvəlkinin nəticəsini
gözləyir
                val ns = (System.nanoTime() - t0).toDouble() / steps
                println("%9d KB  %6.1f ns".format(size shr 10, ns) + if (p == -1) "!" else "")
            }
        }
```

```
$ kotlinc Chase.kt -include-runtime -d Chase.jar && java -Xmx3g -jar Chase.jar
        4 KB     1.7 ns
        8 KB     2.1 ns
       16 KB     1.6 ns
       32 KB     1.6 ns
       64 KB     5.0 ns
      128 KB     5.0 ns
      256 KB     5.0 ns
      512 KB     5.9 ns
     1024 KB     6.9 ns
     2048 KB    37.4 ns
     4096 KB    54.9 ns
     8192 KB    55.2 ns
    16384 KB    56.9 ns
    32768 KB    96.3 ns
    65536 KB   166.0 ns
   131072 KB   173.1 ns
   262144 KB   180.6 ns
   524288 KB   210.6 ns
  1048576 KB   226.6 ns
```

- Kotlin-in pilləkəni C-ninki ilə demək olar eynidir. Burada dil yox, hardware danışır: hər iki proqram vaxtının çoxunu yaddaşı gözləməyə sərf edir.
- `if (p == -1) "!"` sətri `p`-ni istifadə edir ki, JIT dövrü "lazımsız" sayıb silməsin.
- Telefonda da eyni pilləkən olacaq, amma pillələrin yeri və hündürlüyü fərqli. `Chase.kt`-nin dövrünü instrumentation test-də işlədib ölç. Burada işlədilməyib — cihazda yoxla.

> 🖼️ **Şəkil 159:** C və Kotlin · eyni pilləkən

Burada nəyə bax: iki xətt demək olar üst-üstə düşür; pillələr eyni ölçülərdə qalxır.

Kod

Nümunə 1 — tam izahlı: massiv və göstəricilər massivi

Eyni 10 milyon ədədi üç fərqli düzülüşdə toplayırıq. Ədədlər eynidir, toplama sayı eynidir. Dəyişən yalnız ədədlərin yaddaşda harada durmasıdır.

```c
        // Massiv və göstəricilər massivi: eyni ədədlər, fərqli düzülüş.
        // 1) int a[n]          — ədədlər yan-yana
        // 2) int *p[n]         — hər ədəd ayrıca malloc ilə, p[i] onun ünvanıdır (Kotlin
List<Int> kimi)
        // 3) p qarışdırılıb     — göstəricilər yaddaşda təsadüfi yerlərə baxır
        #include <stdio.h>
        #include <stdlib.h>
        #include <time.h>

        static double now(void) {
            struct timespec t;
            clock_gettime(CLOCK_MONOTONIC, &t);
            return t.tv_sec + t.tv_nsec / 1e9;
        }

        int main(void) {
            int n = 10000000;
            int *a = malloc(n * sizeof(int));
            int **p = malloc(n * sizeof(int *));
            for (int i = 0; i < n; i++) {
                a[i] = i;
                p[i] = malloc(sizeof(int));   // hər ədəd öz kiçik blokunda
                *p[i] = i;
            }
            long s = 0;
            double t0 = now();
            for (int i = 0; i < n; i++) s += a[i];
            double t1 = now();
            for (int i = 0; i < n; i++) s += *p[i];
            double t2 = now();
            srand(1);
            for (int i = n - 1; i > 0; i--) {          // göstəriciləri qarışdır (ədədlər yerində
qalır)
                int j = rand() % (i + 1);
                int *t = p[i]; p[i] = p[j]; p[j] = t;
            }
            double t3 = now();
            for (int i = 0; i < n; i++) s += *p[i];
            double t4 = now();
            printf("int massiv:            %6.1f ms\n", (t1 - t0) * 1e3);
            printf("gostericiler (sirali): %6.1f ms\n", (t2 - t1) * 1e3);
            printf("gostericiler (qarisiq):%6.1f ms   (s=%ld)\n", (t4 - t3) * 1e3, s);
            return 0;
        }
```

```
$ gcc -O2 -o boxed boxed.c && ./boxed
int massiv:               6.8 ms
gostericiler (sirali):   35.4 ms
gostericiler (qarisiq): 156.1 ms   (s=149999985000000)
```

Sətir-sətir izah:

- `int *a = malloc(n * sizeof(int))` — 40 MB-lıq bir blok. 10 milyon ədəd yan-yana durur.

- `int **p` — göstəricilər massivi. Hər element 8 byte-lıq ünvandır.
- `p[i] = malloc(sizeof(int))` — hər ədəd üçün ayrıca kiçik blok. malloc hər blokun yanında öz xidməti məlumatını saxlayır, ona görə 4 byte-lıq ədəd yaddaşda 4 byte-dan xeyli çox yer tutur.
- Birinci dövr (`s += a[i]`): ardıcıl oxu. 64 byte-lıq hər line 16 ədəd gətirir. 6,8 ms.
- İkinci dövr (`s += *p[i]`): əvvəl `p[i]`-ni oxu, sonra onun göstərdiyi yerə get. Bloklar ardıcıl ayrıldığı üçün hələ də əsasən ardıcıldır, amma hər ədəd üçün iki oxunuş var və daha çox line lazımdır. 35,4 ms: 5 dəfə yavaş.
- Qarışdırma dövrü yalnız göstəriciləri dəyişir. Ədədlər yerində qalır, amma indi `p[i]` yaddaşda təsadüfi yerə baxır.
- Üçüncü dövr: eyni kod, amma hər `*p[i]` təxminən yeni line-a düşür və prefetcher növbəti ünvanı təxmin edə bilmir. 156 ms: massivdən 23 dəfə yavaş.
- `s`-i çap edirik ki, kompilyator dövrləri lazımsız sayıb silməsin.

> 🖼️ **Şəkil 160:** Nümunə 1 · üç düzülüş

Burada nəyə bax: üçüncü sırada oxlar bir-birini kəsir — hər ədəd üçün başqa yerə getmək lazımdır.

> 🖼️ **Şəkil 161:** boxed.c · 6,8, 35,4 və 156,1 ms

Burada nəyə bax: kod eynidir, dəyişən yalnız oxunuşların yaddaşda hara düşməsidir.

Kotlin versiyası (`Boxed.kt`): `IntArray` C-nin `int a[]`-sıdır. `List<Int>` isə göstəricilər massividir: hər element bir `Integer` obyektinə reference-dir.

```
        // IntArray və List<Int>: eyni ədədlər, fərqli düzülüş
        fun sumArray(a: IntArray): Long { var s = 0L; for (x in a) s += x; return s }
        fun sumList(l: List<Int>): Long { var s = 0L; for (x in l) s += x; return s }  // hər x
bir Integer obyektidir

        fun ms(block: () -> Long): Pair<Long, Double> {
            val t0 = System.nanoTime(); val s = block(); return s to (System.nanoTime() - t0) /
1e6
        }

        fun main() {
            val n = 10_000_000
            val array = IntArray(n) { it }                  // 4 byte × n, yan-yana
            val list = ArrayList<Int>(n)
            for (i in 0 until n) list.add(i)                // hər ədəd ayrıca Integer obyekti
(header + 4 byte)
            val shuffled = ArrayList(list).apply { shuffle(java.util.Random(1)) }  // istinadlar
qarışıq

            repeat(3) {                                     // ilk dövrlər JIT üçün isinmədir;
sonuncunu çap edirik
                val r1 = ms { sumArray(array) }
                val r2 = ms { sumList(list) }
                val r3 = ms { sumList(shuffled) }
                if (it == 2) {
                    println("IntArray:                %6.1f ms".format(r1.second))
                    println("List<Int> (sirali):      %6.1f ms".format(r2.second))
                    println("List<Int> (qarisiq):     %6.1f ms   (s=${r1.first + r2.first +
r3.first})".format(r3.second))
                }
```

```
        }
    }
```

```
$ kotlinc Boxed.kt -include-runtime -d Boxed.jar && java -Xmx3g -jar Boxed.jar
IntArray:                   4.1 ms
List<Int> (sirali):        30.1 ms
List<Int> (qarisiq):      143.6 ms   (s=149999985000000)
$ java -Xmx3g -jar Boxed.jar
IntArray:                   3.4 ms
List<Int> (sirali):        23.6 ms
List<Int> (qarisiq):      133.8 ms   (s=149999985000000)
```

- `List<Int>` sıralı halda `IntArray`-dən 7 dəfə, qarışıq halda 25–40 dəfə yavaşdır (üçüncü işlətmədə 3,3 və 88,5 ms alındı: fərq ölçmədən ölçməyə dəyişir, sırası dəyişmir). Kotlin-in rəqəmləri C-ninkinə çox oxşayır, çünki səbəb eynidir: yaddaş düzülüşü.
- `List<Int>`-də hər ədəd **boxing** olunur: `Integer` obyektinə qoyulur. -128...127 aralığından kənarda hər dəyər ayrıca obyektdir.
- `shuffle` yalnız reference-lərin sırasını dəyişir. Obyektlər heap-də yarandıqları yerdə qalır, ona görə gəzinti təsadüfi ünvanlara tərəf qaçır.
- Real həyatda "qarışıq" hal tez-tez olur: siyahı çeşidlənəndə, elementlər fərqli vaxtlarda yarananda.

> 🖼️ **Şəkil 162:** Boxed.kt · IntArray və List<Int>

Burada nəyə bax: Kotlin-in zolaqları C-nin zolaqlarına oxşayır, çünki səbəb eynidir.

Nümunə 2 — sən tamamla: şəkli düzgün sırada gəzmək

Boz şəkil `pixels` massivində row-major saxlanılır: `height` sətir, hər sətrdə `width` piksel. Funksiya hər pikseli açır. (1) və (2)-ni elə doldur ki, yaddaş ardıcıl gəzilsin:

```
// pixels: height sətir, hər sətrdə width byte (row-major)
void brighten(unsigned char *pixels, int width, int height) {
    for (int ____ = 0; ____ < ____; ____++)          // (1) xarici dövr
        for (int ____ = 0; ____ < ____; ____++)      // (2) daxili dövr
            pixels[y * width + x] = pixels[y * width + x] / 2 + 128;
}
```

Kotlin-də eyni tapşırıq — (1) və (2)-ni doldur:

```
// pixels: height sətir, hər sətrdə width piksel (row-major), dəyərlər 0..255
fun brighten(pixels: IntArray, width: Int, height: Int) {
    for (____ in 0 until ____)              // (1) xarici dövr
        for (____ in 0 until ____)          // (2) daxili dövr
            pixels[y * width + x] = pixels[y * width + x] / 2 + 128
}
```

Əlavə sual: 4000 × 3000 şəkildə səhv sıra seçsən, daxili dövrün hər addımı neçə byte atlayar? C və Kotlin versiyaları üçün ayrıca hesabla.

> 🖼️ **Şəkil 163:** Şəkil · ekranda və yaddaşda

Burada nəyə bax: şəklin sətirləri yaddaşda bir-birinin ardınca gəlir.

> 🖼️ **Şəkil 164:** İpucu nərdivanı

Burada nəyə bax: ipucuları cavabı demədən düşüncəni düzgün istiqamətə yönəldir.

> 🖼️ **Şəkil 165:** Ölçərək yoxla

Burada nəyə bax: son qutu qərarı verir — rəqəm, təxmin yox.

Nümunə 3 — yalnız tapşırıq

Öz maşınının latency pilləkənini çək:

- `Chase.kt`-ni (və ya `chase.c`-ni) işlət, nəticəni cədvələ yaz.
- `/sys/devices/system/cpu/cpu0/cache` (Linux) və ya `sysctl -a hw` (macOS) ilə cache ölçülərini tap.
- L1→L2, L2→L3 və L3→RAM keçidlərini cədvəldə göstər. Keçidlər cache ölçüləri ilə üst-üstə düşürmü? Düşmürsə, niyə?
- Gözlənilən davranış: 4–5 pillə, pillələr arasında 3–30 dəfə artım, ən yuxarıda 80–250 ns.

> 🖼️ **Şəkil 166:** Nümunə 3 · dörd addım

Burada nəyə bax: dördüncü addım tapşırığın əsas sualıdır.

> 🖼️ **Şəkil 167:** Boş qrafik · öz pilləkənin

Burada nəyə bax: qırıq xətləri öz cache ölçülərinə köçür və nöqtələri qoy.

> 🖼️ **Şəkil 168:** Gözlənilən forma

Burada nəyə bax: nəticən bu formaya oxşamalıdır; oxşamasa, səbəbini axtar.

Tipik səhv: şəklin sütunlarını sütun-sütun hesablamaq

Tapşırıq: şəklin hər sütununun orta parlaqlığını tapmaq. Android-da `Bitmap.getPixels` pikseli sətir-sətir `IntArray`-ə köçürür.

❌ **Səhv:**

```
        val pixels = IntArray(width * height)
        bitmap.getPixels(pixels, 0, width, 0, 0, width, height)   // sətir-sətir düzülür
        val columnAvg = LongArray(width)
        for (x in 0 until width) {                                 // hər sütun üçün
            var s = 0L
            for (y in 0 until height) s += (pixels[y * width + x] and 0xFF).toLong()
            columnAvg[x] = s / height
        }
```

Nə baş verir: daxili dövr hər addımda bir sətir atlayır: `width × 4` byte. 4000 piksel enli şəkildə bu, 16 000 byte-dır. Hər oxunuş yeni cache line-a düşür. 4000 × 3000 şəkil 48 MB-dır: Lab 0.1-dəki vəziyyətin eynisi.

> 🖼️ **Şəkil 169:** Tipik səhv · sütun-sütun

Burada nəyə bax: qırmızı oxun hər addımı bir sətir, yaxud 16 000 byte atlayır.

✅ **Düzgün:**

```
        val columnSum = LongArray(width)
        for (y in 0 until height) {                              // sətir-sətir
            val row = y * width
            for (x in 0 until width) columnSum[x] += (pixels[row + x] and 0xFF).toLong()
        }
        val columnAvg = LongArray(width) { columnSum[it] / height }
```

Niyə: hər iki massiv ardıcıl gəzilir. `columnSum` 4000 × 8 = 32 KB-dır və bütün iş boyu L1-də qalır. Alqoritm eynidir, dövrlərin sırası və aralıq nəticənin yeri dəyişib. Bitmap kodu burada işlədilməyib — cihazda yoxla. Eyni fərqi Lab 0.1-də C və Kotlin-də ölçmüşdük.

> 🖼️ **Şəkil 170:** Düzgün · sətir-sətir və columnSum

Burada nəyə bax: hər piksel öz sütununun xanasına düşür, columnSum isə kiçikdir və L1-də qalır.

> 🖼️ **Şəkil 171:** İki sıra · müqayisə

Burada nəyə bax: oxunuş sayı eynidir, miss sayı 16 dəfə fərqlənir.

Senior səviyyəsi

1. SRAM və DRAM: niyə cache kiçik, RAM böyükdür

- Cache **SRAM**-dan (static RAM) qurulur. Hər bit təxminən 6 tranzistorluq dövrədə saxlanılır. Refresh lazım deyil və çox sürətlidir, amma çipdə çox yer tutur.
- RAM **DRAM**-dan (dynamic RAM) qurulur. Hər bit 1 tranzistor və 1 kondensatordur. Sıx və ucuzdur, amma kondensatorun yükü axıb gedir. Ona görə çip sətirləri dövri olaraq təzələyir (Fəsil 0-dakı refresh).
- DRAM-dan oxumaq bir neçə mərhələdir. Əvvəl lazım olan sətir açılır və "row buffer"-ə köçürülür. Sonra oradan sütun seçilir. Oxumaq kondensatorları boşaldır, ona görə sətir geri yazılmalıdır.
- Bizim 150–225 ns-in hamısı DRAM çipində keçmir. Bir hissəsi yoldur: cache-lərdə miss axtarışı, çipin daxili şəbəkəsi, yaddaş kontrolleri. Server çipində bu yol uzundur.
- Nəticə: eyni sahədə DRAM SRAM-dan xeyli çox bit saxlayır. Ona görə cache meqabaytlarla, RAM isə gigabaytlarla ölçülür. Bu mövzunun klassik mənbəyi Ulrich Drepper-in "What Every Programmer Should Know About Memory" (2007) məqaləsidir.

> 🖼️ **Şəkil 172:** SRAM və DRAM · iki cür bit

Burada nəyə bax: solda bir bit üçün altı qutu, sağda bir qutu və bir stəkan var.

> 🖼️ **Şəkil 173:** DRAM-dan oxunuş · üç mərhələ

Burada nəyə bax: bir neçə byte üçün bütöv sətir açılır və sonra geri yazılır.

> 🖼️ **Şəkil 174:** SRAM və DRAM yan-yana

Burada nəyə bax: hər sətirdə bir üstünlük o birinin zəifliyidir.

2. Cache-in içi: set, way, tag və Sirr-in cavabı

Cache 64 byte-lıq yerlərdən ibarət cədvəldir. Sual: gələn line hansı yerə qoyulsun?

- Hər line istənilən yerə qoyula bilsə (**fully associative**), axtarış hər yeri yoxlamalıdır. Bu, böyük cache-də bahalıdır.
- Hər line yalnız bir yerə düşə bilsə (**direct-mapped**), tez-tez işlədilən iki line eyni yerə düşəndə bir-birini qovur.
- Kompromis: **set-associative** cache. Cache set-lərə bölünür, hər set-də N yer var. Bu yerlərə **way** deyilir. Line öz set-inin istənilən way-inə düşə bilər.

> 🖼️ **Şəkil 175:** Line hara qoyulsun · üç qayda

Burada nəyə bax: göy yerlər maşının dayana biləcəyi yerlərdir.

Ünvan üç hissəyə bölünür:

Ünvanın hissəsi	Bitlər (bu maşının L1d-si)	Nə üçündür
Offset	0–5 (6 bit, $2^6 = 64$)	Line daxilində hansı byte
Index	6–11 (6 bit, 64 set)	Hansı set
Tag	12 və yuxarı	Set daxilində line-ın kimliyi, müqayisə üçün

> 🖼️ **Şəkil 176:** Ünvan · tag, index, offset

Burada nəyə bax: aşağı 12 bit set-i və byte-ı seçir, qalanı yalnız kimlikdir.

- Bu maşının L1d-si: 48 KB / 64 byte = 768 line, 12-way → 768 / 12 = 64 set. Index 6 bitdir.
- Nəticə: bir-birindən 4096 byte-ın misli qədər uzaq ünvanlar həmişə eyni L1 set-inə düşür. O set-də isə cəmi 12 yer var.
- L2: 2 MB / 64 = 32 768 line, 16-way → 2048 set, index 11 bit. Period 2048 × 64 = 128 KB-dır.
- Cache boş olsa belə, eyni set-ə düşən line-lar bir-birini qovur. Buna **conflict miss** deyilir.

Sirr-in cavabı. Lab 0.1-in sütun variantını addım-addım izləyək:

1. Bir sütun 4096 sətrdən keçir və 4096 müxtəlif line-a toxunur: 4096 × 64 = 256 KB.
2. Hər line-da 16 qonşu sütun var. Deməli, növbəti 15 sütun eyni 4096 line-ı yenə oxuyacaq.
3. 256 KB L2-yə (2 MB) rahat sığır. Line-lar L2-də qalsa, növbəti 15 sütun L2-dən oxunar.
4. Amma `W = 4096`-da addım 16 384 byte = 256 line-dır. L2 set nömrəsi hər sətrdə 256 artır və 2048-ə görə qalıq alınır. Cəmi 2048 / 256 = 8 fərqli set alınır.
5. 8 set × 16 way = 128 yer. 4096 line 128 yerə sığmır. Line-lar növbəti sütun gəlməmiş qovulur və hər oxunuş L3-ə gedir.
6. `W = 4100`-də addım 16 400 byte = 256,25 line-dır. Set nömrəsi hər dəfə bir az sürüşür və line-lar bütün 2048 set-ə paylanır. 4096 line L2-yə sığır, növbəti 15 sütun L2-dən oxunur.

Yoxlama: 89 ms / 16,7 milyon oxunuş ≈ 5,3 ns. Bu, L2-nin ölçülən latency-sidir. 4 boş xana conflict miss-i aradan qaldırdı. Sətir variantı isə yenə də sürətlidir, çünki orada oxunuşların 94%-i L1 hit-dir.

> 🖼️ **Şəkil 177:** Sirr-in cavabı · 8 set və 2048 set

Burada nəyə bax: üst zolaqda yalnız səkkiz qırmızı xətt var, alt zolaq isə bütünlüklə doludur.

Praktik dərs: ölçüsü 2-nin qüvvəti olan (1024, 2048, 4096) cədvəlləri sütun-sütun gəzmək conflict miss yaradır. Ədədi kitabxanalarda sətrin yaddaşdakı enini (leading dimension) bu səbəbdən bəzən fərqli seçirlər.

3. Prefetcher: CPU yaddaşda da gələcəyi təxmin edir

Prefetcher — giriş naxışını görüb data-nı istənməmişdən əvvəl gətirən hardware. Fəsil 1-dəki branch prediction kimi, bu da təxmindir. Aşağıdakı təcrübə 1 GB massivdə hər k-cı elementə toxunur:

> 🖼️ **Şəkil 178:** Prefetcher · diqqətli ofisiant

Burada nəyə bax: qırıq xətli boşqablar hələ istənməyib, amma yoldadır.

```c
// Stride testi: 1 GB massivdə hər k-cı elementə toxunmaq.
// k = 1..16 olanda toxunulan cache line sayı eynidir (hamısı), ona görə vaxt az dəyişir.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

static double now(void) {
    struct timespec t;
    clock_gettime(CLOCK_MONOTONIC, &t);
    return t.tv_sec + t.tv_nsec / 1e9;
}

int main(void) {
    size_t n = (size_t)1 << 28;                  // 2^28 int = 1 GB
    int *a = malloc(n * sizeof(int));
    for (size_t i = 0; i < n; i++) a[i] = 1;
    for (size_t k = 1; k <= 256; k *= 2) {
        double best = 1e9;
        for (int r = 0; r < 3; r++) {
            double t0 = now();
            for (size_t i = 0; i < n; i += k) a[i] *= 3;
            double t = now() - t0;
            if (t < best) best = t;
        }
```

```
            size_t touches = n / k;
            size_t lines = k <= 16 ? n / 16 : touches;   // toxunulan cache line sayı
            printf("k=%3zu (%4zu byte)  toxunus: %9zu  line: %9zu  vaxt: %6.1f ms  bir
toxunus: %5.2f ns\n",
                   k, k * 4, touches, lines, best * 1e3, best / touches * 1e9);
        }
        free(a);
        return 0;
    }
```

```
$ gcc -O2 -o stride stride.c && ./stride
k=  1 (   4 byte)  toxunus: 268435456  line:  16777216  vaxt:  198.1 ms  bir toxunus:  0.74 ns
k=  2 (   8 byte)  toxunus: 134217728  line:  16777216  vaxt:  120.1 ms  bir toxunus:  0.89 ns
k=  4 (  16 byte)  toxunus:  67108864  line:  16777216  vaxt:   86.4 ms  bir toxunus:  1.29 ns
k=  8 (  32 byte)  toxunus:  33554432  line:  16777216  vaxt:   82.7 ms  bir toxunus:  2.47 ns
k= 16 (  64 byte)  toxunus:  16777216  line:  16777216  vaxt:   76.4 ms  bir toxunus:  4.55 ns
k= 32 ( 128 byte)  toxunus:   8388608  line:   8388608  vaxt:   76.8 ms  bir toxunus:  9.16 ns
k= 64 ( 256 byte)  toxunus:   4194304  line:   4194304  vaxt:   70.3 ms  bir toxunus: 16.75 ns
k=128 ( 512 byte)  toxunus:   2097152  line:   2097152  vaxt:   45.3 ms  bir toxunus: 21.58 ns
k=256 (1024 byte)  toxunus:   1048576  line:   1048576  vaxt:   16.2 ms  bir toxunus: 15.50 ns
```

Kotlin versiyası (`Stride.kt`):

```
    // stride.c-nin Kotlin versiyası: 1 GB massivdə hər k-cı elementə toxunmaq
    fun main() {
        val n = 1 shl 28                            // 2^28 int = 1 GB
        val a = IntArray(n) { 1 }
        var k = 1
        while (k <= 256) {
            var best = Long.MAX_VALUE
            repeat(3) {
                val t0 = System.nanoTime()
                var i = 0
                while (i < n) { a[i] *= 3; i += k }
                best = minOf(best, System.nanoTime() - t0)
            }
            val touches = n / k
            println("k=%3d (%4d byte)  vaxt: %6.1f ms  bir toxunus: %5.2f ns"
                .format(k, k * 4, best / 1e6, best.toDouble() / touches))
            k *= 2
        }
    }
```

```
$ kotlinc Stride.kt -include-runtime -d Stride.jar && java -Xmx3g -jar Stride.jar
k=  1 (   4 byte)  vaxt:  282.4 ms  bir toxunus:  1.05 ns
k=  2 (   8 byte)  vaxt:  252.6 ms  bir toxunus:  1.88 ns
k=  4 (  16 byte)  vaxt:  109.4 ms  bir toxunus:  1.63 ns
k=  8 (  32 byte)  vaxt:   87.8 ms  bir toxunus:  2.62 ns
k= 16 (  64 byte)  vaxt:   81.1 ms  bir toxunus:  4.84 ns
k= 32 ( 128 byte)  vaxt:   80.0 ms  bir toxunus:  9.53 ns
k= 64 ( 256 byte)  vaxt:   69.5 ms  bir toxunus: 16.57 ns
k=128 ( 512 byte)  vaxt:   44.0 ms  bir toxunus: 20.97 ns
k=256 (1024 byte)  vaxt:   18.0 ms  bir toxunus: 17.16 ns
```

Nəticəni oxuyaq:

- k = 4–16 arasında toxunuş sayı 4 dəfə azalır, amma vaxt demək olar dəyişmir (86 → 76 ms). Səbəb: toxunulan line sayı eynidir. Vaxtı toxunuş yox, line daşınması təyin edir.
- k = 1–2-də vaxt daha çoxdur: burada 268 milyon instruction-ın özü də iş görür.

- Dərslik deyərdi: k = 32-də line sayı yarıya düşür, vaxt da yarıya düşməlidir. Amma 76,4 → 76,8 ms, sonra 70,3 ms. Vaxt yalnız 512 byte-dan sonra kəskin düşür.
- Səbəb prefetcher-dir. Intel-in optimallaşdırma əl kitabı L2-yə data gətirən iki prefetcher təsvir edir. Biri (streamer) ardıcıl addımları görüb qabaqdakı line-ları gətirir. Digəri (spatial prefetcher) hər line-ı cütü ilə 128 byte-lıq blok halına tamamlayır. Biz toxunmasaq da, qonşu line-lar da yaddaşdan gəlir.
- Dərs: dərslikdəki model ("vaxt = toxunulan line sayı") ilk yaxınlaşmadır. Real hardware daha çox gətirir. Təxmin etmə, ölç.
- Pointer chasing prefetcher-i aldadır: növbəti ünvan oxunuş bitənə qədər məlum deyil. Linked list, ağac və `List<Int>` kimi strukturlar ona görə yavaşdır.

> 🖼️ **Şəkil 179:** Stride testi · real və model

Burada nəyə bax: qırmızı xətt sürətlə düşür, göy sütunlar isə 512 B-a qədər demək olar eyni qalır.

> 🖼️ **Şəkil 180:** İki prefetcher · streamer və spatial

Burada nəyə bax: bir line-a toxunmaq qırıq xətli qonşu line-ları da yola salır. Qabaqdakı line-ların sayı sxematikdir.

4. Bir neçə core: cache coherence və false sharing (Faza 6-nın toxumu)

- Hər core-un öz L1 və L2-si var. İki core eyni line-ın surətini saxlaya bilər. Biri yazanda o birinin surəti köhnəlir.
- Hardware bunu özü həll edir: yazan core line-ı "özününkü" edir, digər core-lardakı surətlər etibarsız sayılır. Bu mexanizmə **cache coherence** deyilir. Protokolları (MESI ailəsi) Faza 6-da açacağıq.
- Vacib detal: coherence-in vahidi dəyişən yox, cache line-dır.
- Nəticə: iki thread eyni line-dakı **fərqli** dəyişənləri yazanda line core-lar arasında qaçır. Proqramın məntiqində ortaq data yoxdur, amma hardware üçün var. Buna **false sharing** deyilir.

> 🖼️ **Şəkil 181:** False sharing · bir dəftər, iki nəfər

Burada nəyə bax: sətirlər fərqlidir, dəftər isə bir dənədir.

> 🖼️ **Şəkil 182:** Line ping-pong · addım-addım

Burada nəyə bax: qırıq xətli line etibarsız surətdir; qırmızı ox line-ın core-lar arasında qaçmasıdır.

```
        // False sharing: iki thread FƏRQLİ sayğacları atomic artırır.
        // Sayğaclar eyni cache line-dadırsa, core-lar line-ı bir-birinin əlindən alır.
        // İstifadə: ./false_sharing D   D = iki sayğac arasındakı məsafə (byte)
        #include <stdio.h>
        #include <stdlib.h>
        #include <pthread.h>
        #include <stdatomic.h>
        #include <time.h>
```

```c
#define ITERS 100000000L

static char buf[4096] __attribute__((aligned(128)));   // 128 byte sərhədindən başlayır
static _Atomic long *counter[2];

static void *work(void *arg) {
    _Atomic long *c = counter[(long)arg];
    for (long i = 0; i < ITERS; i++)
        atomic_fetch_add_explicit(c, 1, memory_order_relaxed);   // x86-64: lock add
    return NULL;
}

int main(int argc, char **argv) {
    int d = argc > 1 ? atoi(argv[1]) : 8;
    counter[0] = (_Atomic long *)buf;
    counter[1] = (_Atomic long *)(buf + d);
    struct timespec t0, t1;
    clock_gettime(CLOCK_MONOTONIC, &t0);
    pthread_t a, b;
    pthread_create(&a, NULL, work, (void *)0);
    pthread_create(&b, NULL, work, (void *)1);
    pthread_join(a, NULL);
    pthread_join(b, NULL);
    clock_gettime(CLOCK_MONOTONIC, &t1);
    double ms = (t1.tv_sec - t0.tv_sec) * 1e3 + (t1.tv_nsec - t0.tv_nsec) / 1e6;
    printf("mesafe %3d byte: %6.0f ms\n", d, ms);
    return 0;
}
```

```
$ gcc -O2 -pthread -o false_sharing false_sharing.c
$ for d in 8 32 56 64 120 128 256; do ./false_sharing $d; done
mesafe   8 byte:   6372 ms
mesafe  32 byte:   4585 ms
mesafe  56 byte:   5928 ms
mesafe  64 byte:    690 ms
mesafe 120 byte:    641 ms
mesafe 128 byte:    595 ms
mesafe 256 byte:    531 ms
```

- Sayğaclar arası məsafə 64 byte-dan azdırsa, onlar eyni line-dadır: 4,6–6,4 saniyə.
- 64 byte və artıq: fərqli line-lar, 0,5–0,7 saniyə. Fərq 7–10 dəfədir.
- Tək thread eyni 100 milyon atomic artırmanı bu maşında 617 ms-də edir. Deməli, fərqli line-larda iki thread həqiqətən paralel işləyir.
- False sharing-də bir artırma təxminən 50–60 ns çəkir. Bu, L3 latency-sinə yaxındır: line hər dəfə o biri core-dan gətirilir.
- Bu maşında `volatile` ilə adi artırma (atomic olmadan) fərq göstərmədi (35 ms, bütün məsafələrdə). Atomic əməliyyat line-ı mütləq özünə almalıdır, ona görə effekt onda açıq görünür. Real sayğaclar da (`AtomicLong`) atomic-dir.

Kotlin versiyası (`FalseSharing.kt`): iki thread bir `AtomicLongArray`-in iki elementini artırır. Sayğacları massivin əvvəlindən uzaqda (64-cü indeksdən) başladırıq. Niyə belə etdiyimizi "Bunu az adam bilir" bölməsində görəcəksən.

```kotlin
// false_sharing.c-nin Kotlin versiyası: iki thread fərqli sayğacları artırır
import java.util.concurrent.atomic.AtomicLongArray
import kotlin.concurrent.thread

const val ITERS = 100_000_000
```

```
        fun run(distanceBytes: Int): Long {
            val counters = AtomicLongArray(128)          // 128 long = 1 KB, ardıcıl
            val first = 64                               // massivin header-indən 512 byte uzaqda
            val second = first + distanceBytes / 8       // ikinci sayğacın indeksi
            val t0 = System.nanoTime()
            val a = thread { repeat(ITERS) { counters.incrementAndGet(first) } }
            val b = thread { repeat(ITERS) { counters.incrementAndGet(second) } }
            a.join(); b.join()
            return (System.nanoTime() - t0) / 1_000_000
        }

        fun main() {
            run(8)                                       // isinmə: JIT kodu kompilyasiya etsin
            for (d in listOf(8, 64, 128, 256)) {
                val times = List(5) { run(d) }.sorted()
                println("mesafe %3d byte: median %5d ms  (5 olcu: %s)".format(d, times[2],
times))
            }
        }
```

```
$ kotlinc FalseSharing.kt -include-runtime -d FalseSharing.jar && java -jar FalseSharing.jar
mesafe   8 byte: median  4269 ms  (5 olcu: [611, 3989, 4269, 4844, 5377])
mesafe  64 byte: median   619 ms  (5 olcu: [592, 598, 619, 658, 666])
mesafe 128 byte: median   604 ms  (5 olcu: [591, 602, 604, 612, 696])
mesafe 256 byte: median   581 ms  (5 olcu: [513, 559, 581, 601, 694])
```

- Kotlin-də də eyni line: təxminən 7 dəfə yavaş (median 4,3 s və 0,6 s).
- 8 byte-lıq 5 ölçüdən biri 611 ms çıxıb. Ehtimal olunan izah: OS həmin ölçüdə iki thread-i bir core-da növbə ilə işlədib və line core-lar arasında qaçmayıb. Ona görə tək ölçüyə yox, medianına baxırıq.

> 🖼️ **Şəkil 183:** False sharing · C və Kotlin ölçüləri

Burada nəyə bax: qırmızı fondakı sütunlarda məsafə 64 byte-dan azdır və hər iki dildə onlar eyni cür hündürdür.

Düzəliş: padding. Dəyişənləri fərqli line-lara ayırmaq üçün arasına boş yer qoyulur.

- JDK-nın daxili `@Contended` annotasiyası sahənin ətrafına avtomatik padding qoyur. Default ölçü (`-XX:ContendedPaddingWidth`) 128 byte-dır. İstifadəçi kodunda işləməsi üçün `-XX:-RestrictContended` lazımdır.
- Niyə 64 yox, 128? Intel-in spatial prefetcher-i line-ları 128 byte-lıq cütlərə tamamlayır. Bu, qonşu line-lar arasında da yalançı paylaşma yarada bilər. OpenJDK müzakirəsində Intel hətta çox işlək ortaq data üçün 256 byte tövsiyə edib.
- Apple Silicon-da macOS cache line-ı 128 byte kimi bildirir (`hw.cachelinesize: 128`). Go kitabxanası da ona görə arm64 üçün padding-i 128 byte-a qaldırıb.
- Bizim ölçmədə 64 byte kifayət etdi (690 və 595 ms). 128 byte isə platformalar arasında daha təhlükəsiz seçimdir.
- JDK-nın `LongAdder` sinfi bu fikri hazır verir. Çox thread eyni sayğacı artıranda o, hər birinə ayrıca hüceyrə (`Cell`) açır. Hüceyrələr `@Contended` ilə bir-birindən ayrılır. `sum()` onları toplayır. Android-da `LongAdder` API 24-dən var.

- Qayda: çox yazılan, az oxunan sayğac (statistika, metrika) üçün `AtomicLong` yox, `LongAdder`. Tez-tez dəqiq dəyər lazımdırsa, `AtomicLong`.

> 🖼️ **Şəkil 184:** Üç düzəliş · padding, @Contended, LongAdder

Burada nəyə bax: üç yol eyni fikrə xidmət edir — işlək sayğacları fərqli line-lara ayırmaq.

5. TLB: ünvan tərcüməsinin cache-i (Faza 1-in toxumu)

- Proqramın ünvanları virtual-dır. CPU hər oxunuşda virtual ünvanı fiziki ünvana tərcümə etməlidir. Tərcümə cədvəlləri (page table) də yaddaşdadır.
- Yaddaş **səhifələrə** (page) bölünür. Bu maşında adi səhifə 4096 byte-dır (`getconf PAGESIZE`).
- Son tərcümələr kiçik bir cache-də saxlanılır: **TLB** (Translation Lookaside Buffer). TLB-də tapılmayan tərcümə üçün CPU page table-ı gəzir: bu, əlavə yaddaş oxunuşlarıdır.
- 1 GB massiv 262 144 adi səhifədir, TLB isə bunların kiçik bir hissəsini saxlaya bilir. **Huge page** ilə (2 MB) 1 GB cəmi 512 səhifədir.
- `chase.c`-ni `huge` arqumenti ilə işlətdik (`madvise(MADV_HUGEPAGE)`). Bu maşında transparent huge page `madvise` rejimindədir, yəni proqram istəyəndə verilir:

> 🖼️ **Şəkil 185:** TLB · tərcümə dəftərçəsi

Burada nəyə bax: TLB miss-də data-nı oxumazdan əvvəl ünvanın özünü tapmaq lazımdır.

> 🖼️ **Şəkil 186:** 1 GB · 262 144 və 512 səhifə

Burada nəyə bax: eyni sahə solda yüzlərlə kiçik, sağda bir neçə iri parçaya bölünüb.

Working set	Adi səhifə (4 KB)	Huge page (2 MB)
64 MB	164,4 ns	156,3 ns
256 MB	182,0 ns	159,7 ns
1 GB	224,6 ns	159,5 ns

- Huge page ilə RAM pilləsi 64 MB-dan 1 GB-a qədər düz qalır. Adi səhifədəki 65 ns-lik fərq TLB miss-lərin qiymətidir.
- Virtual yaddaşı, page table-ı və page fault-u Faza 1-də açacağıq. Bu fəsildən yadda saxla: ünvanın özü də cache-lənir.

> 🖼️ **Şəkil 187:** Huge page · RAM pilləsi düz qalır

Burada nəyə bax: göy sütunlar düz qalır, narıncılar isə 1 GB-a doğru böyüyür.

6. Android cihazında yaddaş

- Flaqman telefonların böyük core-u, məsələn ARM Cortex-X4 (2023): L1d 64 KB, L2 hər core üçün 512 KB–2 MB, ortaq L3 isə istəyə bağlı olaraq 512 KB–32 MB. Dəqiq ölçüləri çip istehsalçısı seçir.
- Müasir ARM core-larda cache line demək olar həmişə 64 byte-dır.
- big.LITTLE (Fəsil 1): kiçik core-ların cache-i kiçikdir. Thread bir core-dan digərinə köçəndə isti cache-ini itirir və bir müddət yavaş işləyir. Bunu scheduling (Faza 2) ilə birlikdə görəcəyik.
- ART-da da `IntArray` ardıcıldır, `List<Int>` isə boxing edir. Qaynar yolda (şəkil emalı, audio buffer, böyük siyahıların hesablanması) primitiv massivlər və ardıcıl gəzinti seç.
- Telefonda latency pilləkənini ölçmək üçün `Chase.kt`-nin dövrünü instrumentation test-də işlət. Burada işlədilməyib — cihazda yoxla.

> 🖼️ **Şəkil 188:** Telefon çipi · cache-lər

Burada nəyə bax: böyük core-un öz L1 və L2-si var, L3 hamı üçün ortaqdır.

> 🖼️ **Şəkil 189:** Core dəyişməsi · soyuq cache

Burada nəyə bax: köçmədən sonra sürət dərhal düşür və cache isinəndə qalxır.

> 🖼️ **Şəkil 190:** Android qaynar yol · et və etmə

Burada nəyə bax: sol sütun bu fəslin ölçülərindən çıxan vərdişlərdir.

7. Tarix: cache necə yarandı

- İlk kommersiya cache-i IBM System/360 Model 85-də idi (1968-ci ilin yanvarında elan olunub). Cache 16 və ya 32 KB idi. Yaddaş buferi tezliklə "cache" adı ilə tanındı.
- 1994-cü ildə Wulf və McKee "Hitting the Memory Wall: Implications of the Obvious" qeydini yazdılar (Computer Architecture News, mart 1995). "Memory wall" termini oradan yayıldı.
- O vaxtdan bəri cavab eynidir: daha çox cache səviyyəsi, daha ağıllı prefetcher-lər və locality-yə hörmət edən proqramlar.

> 🖼️ **Şəkil 191:** Tarix · 1968, 1994, bu gün

Burada nəyə bax: problem 1968-də göründü, 1994-də adını aldı.

> 🖼️ **Şəkil 192:** Sözün mənası · cacher

Burada nəyə bax: ad cache-in əsas xüsusiyyətini deyir — görünməzlik.

> 🖼️ **Şəkil 193:** Üç sütun · eyni cavab

Burada nəyə bax: üçüncü sütun proqramçının payıdır.

Bunu az adam bilir: JVM massivinin header-i sayğacı yavaşladır

Fakt. `AtomicLongArray`-in ilk elementlərindəki sayğac başqa thread-lər həmin massivlə işləyəndə yavaşlayır. Bu, sayğaclar fərqli line-larda olsa belə baş verir.

> 🖼️ **Şəkil 194:** Ad səhifəsi · analogiya

Burada nəyə bax: ikinci nəfər yalnız adı oxuyur, amma səhifə eyni səhifədir.

Sübut (`HeaderLine.kt`): iki sayğac hər dəfə 256 byte aralıqdadır. Dəyişən yalnız birinci sayğacın massivdəki yeridir.

```
        // Massivin header-i ilə eyni cache line-da olan sayğac: başqa thread-in yoxlaması onu yavaşladır
        import java.util.concurrent.atomic.AtomicLongArray
        import kotlin.concurrent.thread

        fun run(first: Int, second: Int): Long {
            val c = AtomicLongArray(128)
            val t0 = System.nanoTime()
            val a = thread { repeat(100_000_000) { c.incrementAndGet(first) } }
            val b = thread { repeat(100_000_000) { c.incrementAndGet(second) } }
            a.join(); b.join()
            return (System.nanoTime() - t0) / 1_000_000
        }

        fun main() {
            run(64, 96)                                     // isinmə
            for ((f, s) in listOf(0 to 32, 64 to 96, 0 to 64, 1 to 64, 2 to 64)) {
                val t = List(5) { run(f, s) }.sorted()
                println("first=$f second=$s median ${t[2]} ms $t")
            }
        }
```

```
$ kotlinc HeaderLine.kt -include-runtime -d HeaderLine.jar && java -jar HeaderLine.jar
first=0 second=32 median 2050 ms [1791, 1948, 2050, 2094, 2208]
first=64 second=96 median 561 ms [554, 560, 561, 592, 648]
first=0 second=64 median 1911 ms [617, 1768, 1911, 1936, 1988]
first=1 second=64 median 1939 ms [619, 1915, 1939, 1954, 1986]
first=2 second=64 median 1806 ms [576, 578, 1806, 1919, 2054]
```

- 0-cı və 32-ci element: 2,05 s. 64-cü və 96-cı element: 0,56 s. Məsafə eynidir, fərq 3,7 dəfədir.
- Birinci sayğac 0, 1 və ya 2-ci elementdədirsə, nəticə həmişə pisdir: bu elementlər massivin ilk 64 byte-ındadır.

Niyə. `Layout.kt` göstərdi: JVM massivinin ilk 16 byte-ı header-dir, uzunluq da oradadır. Ona görə 0-cı element header ilə eyni cache line-dadır. `incrementAndGet(i)` hər dəfə indeksi massivin uzunluğu ilə yoxlayır. Ehtimal olunan mexanizm: ikinci thread hər artırmada header-i oxuyur, birinci thread isə eyni line-a yazır. Line core-lar arasında qaçır. Bu, false sharing-in ən gizli formasıdır: sayğaclar bir-biri ilə yox, massivin öz xidməti məlumatı ilə toqquşur.

> 🖼️ **Şəkil 195:** Header və c[0 · eyni 64 byte]

Burada nəyə bax: B-nin qırıq oxu A-nın yazdığı yerə çox yaxındır.

Niyə vacibdir. Hər thread-ə bir yuva verən sayğac massivləri, ring buffer-lər, thread-lər arası növbələr tez-tez "0-cı element"ə yazır. Profilə yalnız "atomic yavaşdır" görünür. İşlək yuvaları header-dən və bir-birindən ən azı 128 byte aralı saxla və ya `LongAdder` işlət. Bu maşında ölçülmüşdür. ART-da header-in quruluşu fərqlidir — cihazda yoxla.

> 🖼️ **Şəkil 196:** HeaderLine.kt · 2050 və 561 ms

Burada nəyə bax: birinci sayğac ilk 64 byte-dadırsa, zolaq həmişə uzundur.

Yanlış təsəvvürlər

Nə düşünürlər	Həqiqət	Özün necə sübut edərsən
Big-O eynidirsə, sürət də təxminən eynidir	İki O(n²) gəzinti arasında 14,5 dəfə fərq ola bilər	`rowcol.c` və `RowCol.kt`
RAM "random access"-dir, istənilən ünvan eyni sürətlə oxunur	Data-nın harada olduğuna görə 1,6 ns ilə 225 ns arası	`chase.c` və `Chase.kt`
CPU yaddaşdan lazım olan byte-ı oxuyur	Həmişə bütöv 64 byte-lıq line gəlir, prefetcher çox vaxt qonşularını da gətirir	cachegrind (6,2% miss), `stride.c`
Cache-i proqram idarə edir	Hardware idarə edir; proqram yalnız data-nın düzülüşü və gəzinti sırası ilə təsir edir	Eyni kod, fərqli sıra → fərqli vaxt
İki thread fərqli dəyişənlərə yazırsa, bir-birinə mane olmur	Eyni line-dadırlarsa, 7–10 dəfə yavaşlayırlar (false sharing)	`false_sharing.c`, `FalseSharing.kt`
JVM-də cache məsələsi yoxdur, JIT hər şeyi həll edir	JIT dövr sırasını və obyektlərin düzülüşünü dəyişmir	`RowCol.kt` (10 dəfə), `Boxed.kt` (25–40 dəfə)
`List<Int>` ilə `IntArray` eyni şeydir, fərq sintaksisdədir	`List<Int>`-də hər ədəd ayrıca obyektdir, gəzinti pointer chasing-ə çevrilir	`Boxed.kt`
lscpu-dakı L3 ölçüsü tam sənindir	Virtual maşında L3 ortaqdır; effektiv pay kiçik ola bilər (bizdə 260 MB əvəzinə təxminən 16–32 MB)	`chase.c`-nin pillələri
Cache line hər yerdə 64 byte-dır, padding üçün 64 kifayətdir	Apple Silicon 128 bildirir, Intel prefetcher-i line-ları cüt gətirir; JDK 128 byte padding qoyur	`@Contended`, `ContendedPaddingWidth`
Artıq yaddaş işlətmək həmişə yavaşdır	4 boş xana conflict miss-i aradan qaldırdı: 150 → 89 ms	`./rowcol 4096 4100`

İntervyuda

İ2.1 — Cache nədir və niyə lazımdır? Latency rəqəmlərini de.

- **Qısa cavab:** Cache CPU-nun yanında, RAM-dakı data-nın bir hissəsinin kiçik və sürətli surətidir. RAM CPU-dan yüzlərlə dəfə yavaş cavab verir, cache isə locality sayəsində oxunuşların çoxunu bir neçə nanosaniyədə verir.
- **Mexanizm:** CPU əvvəl L1-ə, sonra L2-yə, L3-ə, sonda RAM-a baxır. Miss-də 64 byte-lıq line gətirilir və set-associative cədvəldə köhnə bir line-ın yerinə qoyulur. Temporal locality son işlədilənin, spatial locality qonşuların tezliklə lazım olacağını deyir.
- **Rəqəm və sübut:** Norvig və Dean cədvəli: L1 0,5 ns, L2 7 ns, RAM 100 ns. Bu maşında pointer chasing ilə ölçdüm: L1 1,6 ns, L2 5 ns, L3 30–75 ns, RAM 150–225 ns.
- **Trade-off:** Böyük cache daha çox data saxlayır, amma daha yavaşdır və çipdə çox yer tutur. Ona görə bir neçə səviyyə var. Yüksək associativity conflict miss-i azaldır, amma axtarışı bahalaşdırır.
- **Real nümunə:** Android-da şəkil filtri pikselləri sütun-sütun gəzəndə kadr vaxtı aşır və jank yaranır. Sətir-sətir gəzinti eyni alqoritmi bir neçə dəfə sürətləndirir.
- **Fərqləndirici detal:** Cache coherence-in vahidi də line-dır. Ona görə multithread kodda fərqli dəyişənlər eyni line-dadırsa, false sharing yaranır.

Follow-up nərdivanı:

1. *"Cache line neçə byte-dır və bunu necə bilərsən?"* → x86-64-də və Android ARM64-də 64 byte. Linux-da `getconf LEVEL1_DCACHE_LINESIZE` və ya `/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size`.
2. *"Set-associative nə deməkdir?"* → Ünvanın index bitləri set-i seçir, line həmin set-in N way-indən birinə düşür. 48 KB, 12-way L1-də 64 set var: 4 KB aralıqlı ünvanlar eyni set üçün döyüşür.
3. *"Cache miss-in növləri?"* → Compulsory (ilk toxunuş), capacity (working set sığmır), conflict (eyni set-ə çox line). Lab 0.1-də padding conflict miss-i aradan qaldırdı.
4. *"Prefetcher nə edir və nə vaxt işləmir?"* → Ardıcıl və sabit addımlı naxışları görüb line-ları qabaqcadan gətirir. Pointer chasing-də növbəti ünvan məlum deyil, ona görə kömək etmir.
5. *"TLB nədir?"* → Virtual ünvanların fiziki ünvana tərcüməsinin cache-i. Böyük working set-də TLB miss əlavə gecikmə verir; huge page bunu azaldır (bizdə 1 GB-da 225 → 160 ns).

İ2.2 — 2D massivi sətir-sətir gəzmək niyə sütun-sütun gəzməkdən sürətlidir?

- **Qısa cavab:** Massiv row-major saxlanılır. Sətir-sətir gəzinti ardıcıl ünvanları oxuyur və hər line-dan 16 `int` istifadə edir. Sütun-sütun gəzinti hər addımda bir sətir atlayır və hər oxunuşda yeni line gətirir.
- **Mexanizm:** Sətir variantında 16 oxunuşdan biri miss-dir, prefetcher növbəti line-ları qabaqcadan gətirir. Sütun variantında L1 miss təxminən 100%-dir. 4096 × 4 byte addımı 2-nin qüvvəti olduğu üçün line-lar L2-də yalnız 8 set-ə düşür və conflict miss yaranır.
- **Rəqəm və sübut:** 4096 × 4096 `int`: 10,4 ms və 150,5 ms (14,5 dəfə). Cachegrind: D1 read miss 6,2% və 99,7%. Kotlin `IntArray`-də 10 dəfə.
- **Trade-off:** Sütun əməliyyatları çoxdursa, data-nı column-major saxlamaq (və ya transpoze etmək) olar. Bu, sətir əməliyyatlarını yavaşladır. Blocking (tiling) hər ikisini balanslaşdırır.

- **Real nümunə:** `Bitmap.getPixels` ilə alınan massivdə sütun ortalamalarını sütun-sütun hesablamaq. Düzəliş: sətir-sətir gəzib `columnSum[x]`-ə toplamaq.
- **Fərqləndirici detal:** Sətrə 4 boş xana əlavə edəndə sütun gəzintisi 150-dən 89 ms-ə düşdü: line-lar bütün L2 set-lərinə paylanır və növbəti 15 sütun L2-dən oxunur.

Follow-up nərdivanı:

1. *"Kotlin-in `Array<IntArray>`-i nə dəyişir?"* → Hər sətir ayrı obyektdir. Sətir daxilində data ardıcıldır, amma sətirlər arasında reference oxunuşu var. Bizdə hər iki gəzinti bir az yavaş oldu.
2. *"Matris vurmasında bu necə görünür?"* → Klassik i-j-k sırasında `b[k][j]` sütunla oxunur. i-k-j sırası daxili dövrü sətir boyunca aparır.
3. *"JIT dövrlərin sırasını dəyişə bilərmi?"* → Ölçməmizdə dəyişmədi. Loop interchange-i kompilyatorlar yalnız təhlükəsiz sübut edə bildikləri hallarda edir.
4. *"Cache-oblivious alqoritm nədir?"* → Cache ölçüsünü bilmədən, məsələn rekursiv bölmə ilə hər səviyyədə locality veren alqoritm.

İ2.3 — False sharing nədir? Necə aşkar edib düzəldirsən?

- **Qısa cavab:** İki thread eyni cache line-dakı fərqli dəyişənlərə yazanda line core-lar arasında qaçır. Proqramda ortaq data yoxdur, amma performans ortaq data kimi düşür.
- **Mexanizm:** Cache coherence line səviyyəsində işləyir. Yazmaq üçün core line-ı müstəsna alır, o biri core-un surəti etibarsız olur. Növbəti yazıda hər şey tərsinə təkrarlanır.
- **Rəqəm və sübut:** 2 thread × 100 milyon atomic artırma: eyni line 4,6–6,4 s, fərqli line 0,5–0,7 s (C). Kotlin `AtomicLongArray`: median 4,3 s və 0,6 s.
- **Trade-off:** Padding yaddaş sərf edir: hər sayğac 8 byte yox, 128 byte. Minlərlə obyektdə bu, cache-in özünü doldurur. Ona görə padding yalnız çox yazılan, thread-lər arası qaynar sahələrə qoyulur.
- **Real nümunə:** Analitika SDK-sında hər thread `counts[threadIndex]++` edir. CPU profili atomic əməliyyatları göstərir, amma səbəb alqoritm deyil, düzülüşdür. Düzəliş: `LongAdder`, thread-local sayğac və ya indeksi 16-ya vurmaq.
- **Fərqləndirici detal:** JVM massivinin ilk elementləri header ilə eyni line-dadır. Bizim ölçmədə 0-cı elementdəki sayğac, məsafə 256 byte olsa da, 3,7 dəfə yavaş idi.

Follow-up nərdivanı:

1. *"Necə aşkar edərsən?"* → Linux-da `perf c2c` cache-to-cache ötürmələri və "HITM" hadisələrini göstərir. JVM-də əvvəl thread sayını artırıb miqyaslanmaya bax: thread artır, vaxt da artırsa, şübhə var.
2. *"`@Contended` niyə 128 byte qoyur?"* → Prefetcher line-ları cüt gətirə bilər və bəzi platformalarda line 128 byte-dır. Default `ContendedPaddingWidth` 128-dir.
3. *"`LongAdder` daxildə necə işləyir?"* → Toqquşma olanda thread-lər ayrı-ayrı `Cell`-lərə yazır. `Cell`-lər `@Contended`-dir. `sum()` hamısını toplayır, ona görə oxu bahalı və dəqiq anı əks etməyə bilər.
4. *"False sharing yalnız yazıda olur?"* → Yazı lazımdır. Bir thread yazır, digəri eyni line-dan yalnız oxusa da, oxuyan tərəf hər dəfə line-ı yenidən gətirir. Header nümunəsi məhz belədir.

İ2.4 — `IntArray` ilə `List<Int>` arasındakı performans fərqi haradan gəlir?

- **Qısa cavab:** `IntArray` ədədləri yan-yana saxlayır. `List<Int>` isə `Integer` obyektlərinə reference-lər saxlayır. Gəzinti hər ədəd üçün reference-i, sonra obyekti oxuyur.
- **Mexanizm:** Bir `int` 4 byte-dır. Boxing olunmuş dəyər 4 byte-lıq reference və header-li obyektdir. Bir line-a daha az dəyər düşür. Obyektlər heap-də səpələnibsə, gəzinti pointer chasing olur və prefetcher kömək etmir.
- **Rəqəm və sübut:** 10 milyon ədəd: `IntArray` 3,4–4,1 ms, `List<Int>` 24–30 ms, qarışdırılmış `List<Int>` 134–144 ms. C analoqu da eyni nisbətləri verir.
- **Trade-off:** `List<Int>` ümumi API, null və generic-lərlə rahatdır. Qaynar yolda (çox element, tez-tez gəzinti) primitiv massiv daha yaxşıdır.
- **Real nümunə:** Audio buffer-i və ya qrafik nöqtələrini `List<Float>`-da saxlayıb hər kadrda hesablamaq. `FloatArray`-ə keçmək həm GC yükünü, həm cache miss-ləri azaldır.
- **Fərqləndirici detal:** Boxing üçün `Integer` cache-i yalnız -128...127 aralığını əhatə edir. Ondan kənarda hər dəyər yeni obyektdir.

Follow-up nərdivanı:

1. *"Linked list niyə çox vaxt `ArrayList`-dən yavaşdır, əlavə və silmə O(1) olsa da?"* → Gəzinti pointer chasing-dir: hər node üçün bir RAM latency-si ola bilər. `ArrayList` ardıcıl gəzilir.
2. *"GC bu vəziyyəti necə dəyişir?"* → Obyektləri köçürən (copying) GC onları reference-lərin sırası ilə yan-yana qoya bilər və locality bəzən yaxşılaşır. Amma bu, GC-nin növündən və vaxtından asılıdır, ona arxalanmaq olmaz. Ölçməmizdə qarışıq siyahı yavaş qaldı.
3. *"Kotlin-də `value class` kömək edirmi?"* → Tək dəyərli `value class` çox yerdə boxing-siz saxlanılır, amma generic kolleksiyaya düşəndə yenə boxing olur.

Bug-ı tap

B2.1 — 4 worker 1 worker-dən az qədər sürətlidir. Hər worker öz sayğacını artırır, ortaq dəyişən yoxdur. Coroutine-ləri Faza 9-da öyrənəcəksən; burada bilməli olduğun: `launch(Dispatchers.Default)` blokları müxtəlif core-larda eyni anda işləyə bilər.

```
class Stats(workers: Int) {
    val processed = AtomicLongArray(workers)      // hər worker-ə bir yuva
}

suspend fun processAll(items: List<Item>, stats: Stats) = coroutineScope {
    items.chunked(items.size / 4 + 1).forEachIndexed { w, chunk ->
        launch(Dispatchers.Default) {
            for (item in chunk) {
                handle(item)                      // ucuz, CPU-ağır iş
                stats.processed.incrementAndGet(w)
            }
        }
    }
}
```

B2.2 — Matris vurması düzgün nəticə verir, amma böyük `n`-də çox yavaşdır. Matrislər row-major `DoubleArray`-lərdə saxlanılır.

```
fun multiply(a: DoubleArray, b: DoubleArray, n: Int): DoubleArray {
    val c = DoubleArray(n * n)
```

```
            for (i in 0 until n)
                for (j in 0 until n) {
                    var s = 0.0
                    for (k in 0 until n) s += a[i * n + k] * b[k * n + j]
                    c[i * n + j] = s
                }
            return c
        }
```

Öz sözlərinlə izah et

Bu fəsli bir dostuna 5 cümlə ilə izah et. Yaxşı izahda bunlar var:

- [ ] RAM byte-ların uzun sırasıdır, hər byte-ın ünvanı var; massivin elementləri yan-yana durur.
- [ ] RAM CPU-dan yüzlərlə dəfə yavaş cavab verir; cache bu fərqi locality sayəsində gizlədir.
- [ ] Cache həmişə 64 byte-lıq line gətirir; ardıcıl giriş line-ın hamısından istifadə edir.
- [ ] Pillələr: L1 ~1 ns, L2 ~5 ns, L3 ~onlarla ns, RAM ~100–200 ns; öz maşınında ölçmək olar.
- [ ] Eyni alqoritm dövr sırasına görə 10–15 dəfə fərqli sürətdə işləyə bilər.
- [ ] Pointer chasing (linked list, `List<Int>`) prefetcher-i aldadır.
- [ ] Coherence-in vahidi line-dır; fərqli thread-lərin yazdığı dəyişənlər eyni line-dadırsa, false sharing yaranır.

Boş xəritə: aşağıdakı xəritəni kağıza köçür. Hər budaq üçün qeydə baxmadan bir cümlə yaz. Sonra fəslin sonundakı Xülasə şəkli ilə müqayisə et.

> 🖼️ **Şəkil 197:** Fəsil 2 · boş xəritə

Keçid qapısı

Qeydə baxmadan cavabla. 8 sualdan ən azı 7-sini mexanizmi ilə (rubrikanın 3-cü səviyyəsi) cavablaya bilirsənsə, Fəsil 3-ə keç. Son iki sual əvvəlki fəsillərdəndir.

1. **K2.1 (xatırla):** L1, L2, L3 və RAM-ın latency-sinin böyüklük sırasını de. Hansı ikisi arasında fərq ən böyükdür?
2. **K2.2 (hesabla):** L1d 32 KB, 8-way, line 64 byte-dır. Neçə set var? Bir-birindən neçə byte uzaq ünvanlar həmişə eyni set-ə düşür?
3. **K2.3 (nəticəni təxmin et):** 1 milyonluq `IntArray`-i (4 MB) əvvəl ardıcıl, sonra eyni indekslərlə, amma təsadüfi sırada toplayırsan. Hansı yavaşdır və niyə? Bu maşının pilləkəninə əsasən təxmini fərqi de.
4. **K2.4 (niyə):** Lab 0.1-də sətrə 4 boş xana əlavə etmək sütun gəzintisini niyə 150 ms-dən 89 ms-ə endirdi?
5. **K2.5 (bug-ı tap):** `val hits = LongArray(8)`, 8 thread-in hər biri yalnız `hits[id]++` edir. Nəticə düzgündür, amma proqram gözləniləndən yavaşdır. Niyə? İki düzəliş təklif et.
6. **K2.6 (dizayn qərarı):** 8 thread-in saniyədə milyonlarla dəfə artırdığı, saniyədə bir dəfə oxunan metrika sayğacı üçün `AtomicLong`, yoxsa `LongAdder`? Niyə?
7. **K2.7 (Fəsil 1):** Fəsil 1-də `count++`-ın üç addım olduğunu gördük. Atomic artırma x86-64-də hansı instruction-dur və o, niyə cache line-ı mütləq öz core-una almalıdır?
8. **K2.8 (Fəsil 0):** `0x5633119cf07c` ünvanı hansı cache line-dadır və line daxilində neçənci byte-dır? Cavabı hex ilə ver.

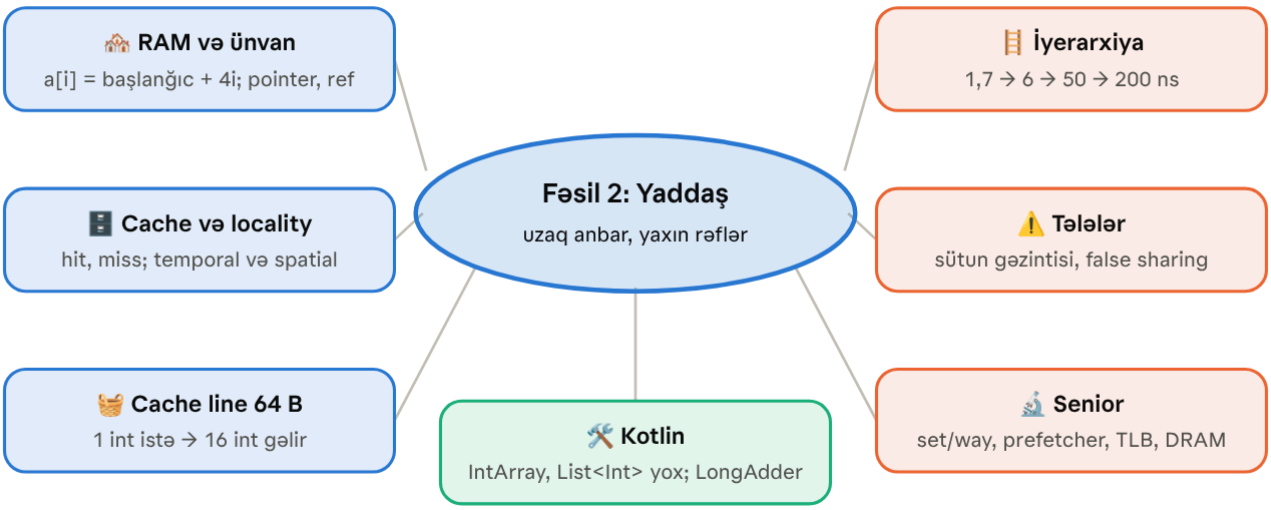
Yadda saxla

- Pillələr: register → L1 (~1 ns) → L2 (~5 ns) → L3 (~onlarla ns) → RAM (~100–200 ns). Hər pillə böyük, amma yavaşdır.
- CPU yaddaşdan həmişə 64 byte-lıq cache line gətirir. Ardıcıl giriş line-ın hamısını işlədir.
- Locality: son işlədilən yenə lazım olur, qonşu da lazım olur. Sürət çox vaxt alqoritmdən yox, düzülüşdən və gəzinti sırasından asılıdır.
- Pointer chasing hər addımda latency ödəyir. Qaynar yolda `IntArray` kimi primitiv massivlər seç.
- Coherence-in vahidi line-dır. Fərqli thread-lərin tez-tez yazdığı dəyişənləri ən azı 128 byte ayır və ya `LongAdder` işlət.

Bir sətirlik mental model: RAM uzaq anbardır, cache onun yaxın rəfləridir; sürətli proqram data-nı rəfə yaraşan şəkildə düzür və ardıcıl gəzir.

Xülasə şəkli

Bütün fəsil bir şəkildə. Burada nəyə bax: mərkəzdə mövzu, budaqlarda mexanizm, rəqəmlər, tələlər və Kotlin baxışı.



Bir cümlə: RAM uzaq anbardır, cache onun yaxın rəfləridir; sürətli proqram data-nı rəfə yaraşan şəkildə düzür və ardıcıl gəzir.

Fəsil 3 — Proqramın doğuluşu: mətndən işləyən prosesə

Bu fəsildən sonra bacaracaqsan

- `gcc hello.c` əmrinin içindəki dörd mərhələni (preprocessor, compiler, assembler, linker) ayrı-ayrı işlədib hər birinin çıxışını oxumaq.
- ELF faylının quruluşunu izah etmək: header, section-lar (`.text`, `.rodata`, `.data`, `.bss`) və segment-lər. Dəyişənin hansı section-a düşdüyünü təxmin edib yoxlamaq.
- Linker-in iki işini göstərmək: symbol resolution və relocation. "Undefined reference" və "multiple definition" xətalarının səbəbini tapmaq.
- `./hello` yazanda kernel-in, dinamik linker-in və libc-nin `main`-dən əvvəl nə etdiyini addım-addım izləmək.
- Kotlin kodunun JVM-də və Android-da hansı yolla işə düşdüyünü izah etmək: class, DEX, ART, OAT, `.so`. JNI ilə Kotlin-dən C funksiyası çağırmaq.
- `main`-dən əvvəl işləyən constructor-ları, debug məlumatını, SONAME-i və simvol versiyalarını göstərmək; strip olunmuş crash ünvanını sətir nömrəsinə çevirmək. Bu fəslin thread-lərlə görüşdüyü üç yeri tanımaq: TLS, `<clinit>` lock-u, loader lock-u.

Əvvəlcə təxmin et

Cavabı kağıza yaz, sonra oxumağa davam et. Cavablar Cavab açarındadır.

1. **T3.1** 7 sətirlik `hello.c` faylı `#include <stdio.h>` ilə başlayır. Compiler onu oxumazdan əvvəl fayl neçə sətrə çevrilir: 10, 100, yoxsa 800-dən çox?
2. **T3.2** `./hello` yazırsan. CPU-nun icra etdiyi ilk instruction sənin proqramının `main` funksiyasındadırmı?
3. **T3.3** Kotlin-də `val table = intArrayOf(...)` yazırsan və mötərizənin içinə 20 000 ədəd qoyursan. Kod kompilyasiya olunacaqmı?

Sirr

Bu maşında iki kiçik C proqramı yığdıq. Hər ikisində bir milyon `int`-lik qlobal massiv var, yəni 4 MB. Birincidə massiv sıfırlarla doludur. İkincidə ilk element 1-dir, qalanı yenə sıfır.

Birinci proqramın faylı 15 992 byte-dır, ikincinin faylı 4 016 008 byte. Bir rəqəm faylı 250 dəfə böyütdü.

Birinci faylda 4 MB-lıq massiv harada gizlənib? Proqram işləyəndə o 4 MB-ı kim yaradır? Bu fəsil hər iki suala cavab verir.

> 🖼️ **Şəkil 199:** Sirr hekayəsi · iki çamadan

Burada nəyə bax: kiçik çamadan yükü yox, yük haqqında qeydi aparır. Kimsə yolda o qeydi oxuyub yükü verməlidir.

> 🖼️ **Şəkil 200:** Sirr · 15 992 və 4 016 008 byte

Burada nəyə bax: kod demək olar eynidir, fərq yalnız `= {1}` hissəsindədir.

> 🖼️ **Şəkil 201:** Sirr · cavabın üç dayanacağı

Burada nəyə bax: hər dayanacaq bir bölmədir. Sirr-in cavabı onların üçünü birlikdə oxuyanda tamamlanır.

Yeni sözlər

source code, build, preprocessor, compiler, assembly, assembler, object faylı, symbol, symbol table, relocation, linker, static library, shared library (`.so`), static linking, dynamic linking, executable, ELF, ELF header, section, segment, `.text`, `.rodata`, `.data`, `.bss`, entry point, loader, `execve`, dinamik linker (`ld.so`), PLT, GOT, PIE, ASLR, auxiliary vector, bytecode, class faylı, class loading, DEX, ART, dex2oat, OAT, JNI, constructor (.init_array), debug məlumatı (DWARF), strip, Build ID, SONAME, simvol versiyası, name mangling, extern "C", TLS, class initialization lock

Uşaq səviyyəsi: resept, tərcüməçi, cildçi və kitabxanaçı

Fəsil 1-dəki aşpaz robotu xatırla. O, yalnız öz dilində yazılmış kartları oxuyur: hər kartda bir əmr. Sən isə resepti Azərbaycan dilində yazmısan.

Arada kimsə resepti robotun kartlarına çevirməlidir. Bu iş beş nəfər arasında bölünüb.

1. **Hazırlayan** resepti oxuyur. "Bax: ümumi sous resepti" kimi qeydlərin yerinə həmin reseptin mətnini yapışdırır.
2. **Tərcüməçi** resepti robotun dilinə çevirir, amma hələ insanın oxuya biləcəyi qısa əmrlərlə.
3. **Kart yazan** hər əmri robotun anladığı kart nömrəsinə çevirir. Bilinməyən yerlər qalır: "sousu 1-ci fəsildən götür, səhifə ?".
4. **Cildçi** bütün fəsilləri bir kitabda birləşdirir və hər "səhifə ?" yerinə real səhifə nömrəsini yazır. Ümumi lüğətləri kitaba tikmir, yalnız "bu kitab filan lüğətlə oxunur" qeydini qoyur.
5. **Kitabxanaçı** kitabı rəfdən alır, masaya açır, lazım olan lüğəti yanına qoyur və robota deyir: "1-ci səhifədən başla".

> 🖼️ **Şəkil 202:** Uşaq səviyyəsi · reseptdən robota

Burada nəyə bax: hər köməkçi yalnız bir iş görür, robot isə sonda yalnız kartları görür.

Analogiyada	Real sistemdə
Azərbaycanca resept	Source code: `hello.c`
Qeydləri açan hazırlayan	Preprocessor: `#include`, `#define`
Tərcüməçi	Compiler: gcc-də `cc1`, Kotlin-də `kotlinc`
Kart yazan	Assembler: `as`

Analogiyada	Real sistemdə
"Səhifə ?" qeydi olan fəsil	Object faylı (`.o`) və onun relocation-ları
Cildçi	Linker: `ld`
Hazır kitab və onun məzmun səhifəsi	Executable ELF faylı və onun header-i
Hamının oxuduğu ümumi lüğət	Shared library: `libc.so.6`
Kitabxanaçı	Loader: kernel və `ld.so`
"1-ci səhifədən başla"	Entry point

> 🖼️ **Şəkil 203:** İki zəncir · hekayə və fayllar

Burada nəyə bax: üst sırada hekayə, alt sırada bu fəslin real faylları var. Sonuncu addım fayl deyil, işləyən prosesdir.

Analogiya harada sınır:

- Tərcüməçi hər cümləni bir-bir çevirir. Compiler isə kodu dəyişə, silə və yerini dəyişə bilir. Bu fəsildə `printf`-in `puts`-a çevrildiyini görəcəksən.
- Kitabda boş səhifə də kağız tutur. ELF faylında isə sıfırlarla dolu massiv yer tutmur: yalnız "4 MB sıfır lazımdır" qeydi yazılır. Sirr-in cavabı budur.
- Kitabxanaçı kitabı bütöv masaya qoymur. Kernel faylı səhifə-səhifə, yalnız lazım olan hissəni yaddaşa gətirir. Bunu Faza 1-də page fault mövzusunda görəcəksən.

> 🖼️ **Şəkil 204:** Analogiyanın zəif yerləri

Burada nəyə bax: hər zəif yer bu fəsildə ayrıca təcrübə ilə göstəriləcək.

Mühəndis səviyyəsi

1. Proqram iki dildə: insanın dili və CPU-nun dili

Fəsil 1-də gördün: CPU yalnız machine code oxuyur. Machine code byte-lardır, hər instruction bir neçə byte. `hello.c` isə mətndir: 127 byte-lıq UTF-8 mətni. CPU onu oxuya bilməz.

Fəsil boyu iki kiçik proqramı izləyəcəyik. Birincisi C-dədir, `hello.c`:

```c
// ən sadə C proqramı: ekrana bir sətir yazır
#include <stdio.h>

int main(void) {
    printf("salam\n");
    return 0;
}
```

İkincisi eyni işi Kotlin-də görür, `Hello.kt`:

```kotlin
// ən sadə Kotlin proqramı: ekrana bir sətir yazır
fun main() {
```

```
            println("salam")
        }
```

```
$ wc -l -c hello.c Hello.kt
  7 127 hello.c
  4  92 Hello.kt
 11 219 total
```

`hello.c`-də 7 sətir, 127 byte var. Şərhdəki `ə`, `ə`, `ı` hərflərinin hər biri UTF-8-də 2 byte tutur (Fəsil 0). Bu iki faylı bütün təcrübələrdə işlədəcəyik.

Mətni CPU-nun byte-larına çevirən prosesə **build** deyilir. `gcc hello.c -o hello` bir əmr kimi görünür, amma `gcc` əslində dispetçerdir: dörd ayrı aləti növbə ilə çağırır.

Mərhələ	Alət	Giriş	Çıxış	Ayrıca işlətmək
1. Preprocessing	`cc1` -in preprocessor hissəsi	`hello.c`	`hello.i`, C mətni	`gcc -E`
2. Compilation	`cc1`	`hello.i`	`hello.s`, assembly mətni	`gcc -S`
3. Assembly	`as`	`hello.s`	`hello.o`, object faylı	`gcc -c`
4. Linking	`ld`, `collect2` vasitəsilə	`hello.o` və kitabxanalar	`hello`, executable	`gcc`

- `gcc -v hello.c` bu çağırışları göstərir. Bu maşında `cc1`, `as` və `collect2` görünür; `collect2` özü `ld`-ni çağırır.
- Beşinci addım fayl yaratmır. **Loader** faylı yaddaşa açır və işləyən **prosesə** çevirir (Bölmə 7).
- Kotlin-də yol fərqlidir: `kotlinc` machine code yox, JVM bytecode yazır. CPU-nun dilinə çevirməni proqram işləyəndə JVM edir (Bölmə 8).

> 🖼️ **Şəkil 205:** Build konveyeri · dörd stansiya

Burada nəyə bax: hər stansiya bir fayl alır və növbəti faylı verir. Sonuncu ox fayla yox, prosesə gedir.

> 🖼️ **Şəkil 206:** Fayl ölçüləri · hər mərhələ

Burada nəyə bax: 127 byte-lıq mətn 15 960 byte-lıq executable olur. Yolda ən böyük sıçrayış preprocessor-dadır.

> 🖼️ **Şəkil 207:** gcc · dörd alətin dispetçeri

Burada nəyə bax: üç bayraq konveyeri müəyyən stansiyada dayandırır və aralıq faylı göstərir.

2. Preprocessor: mətni mətnə çevirir

Preprocessor C dilini bilmir. O, `#` ilə başlayan sətirləri icra edən mətn redaktorudur.

- `#include <stdio.h>` — "bu faylın bütün məzmununu bura yapışdır".
- `#define N 10` — "N sözünü 10 ilə əvəz et".
- `#ifdef DATA` ... `#endif` — "DATA təyin olunubsa, bu hissəni saxla, yoxsa sil". Sirr-dəki iki proqramı bir fayldan, `-DDATA` bayrağı ilə yığdıq.

Nəticə yenə C mətnidir. `gcc -E hello.c` 7 sətri 820 sətrə çevirdi: `stdio.h` və onun daxil etdiyi fayllar. Bizim `main` ən sondadır.

> 🖼️ **Şəkil 208:** #include · kopyala və yapışdır

Burada nəyə bax: `#include` sətri yox olur, onun yerinə başqa faylın mətni gəlir.

> 🖼️ **Şəkil 209:** gcc -E · 7 sətirdən 820 sətrə

Burada nəyə bax: `hello.i`-nin demək olar hamısı başqa fayldandır. `# 4 "hello.c"` sətri xəta mesajlarının doğru sətri göstərməsi üçündür.

Preprocessor C-ni bilmədiyi üçün tələ qurur:

```
#define SQUARE(x) x * x
printf("%d\n", SQUARE(1 + 2));    // 9 gözləyirsən
```

`gcc -E` göstərir: `1 + 2 * 1 + 2`. Proqram 5 çap edir. Düzəliş: `#define SQUARE(x) ((x) * (x))` və ya adi funksiya.

Kotlin-də preprocessor yoxdur. `import` mətn yapışdırmır, yalnız adı qısaldır. Makronun işini `const val` və adi funksiyalar görür, onlar isə tipləri və hesablama sırasını bilir:

```
fun square(x: Int) = x * x
println(square(1 + 2))            // 9: əvvəl 1 + 2 hesablanır
```

> 🖼️ **Şəkil 210:** Makro tələsi · 5 və 9

Burada nəyə bax: solda mətn dəyişdirilib, sağda dəyər hesablanıb. Fərq mötərizələrdədir.

Dayan və düşün D3.1: `#define MAX(a, b) a > b ? a : b` yazılmışdır. `2 * MAX(3, 4)` hansı mətnə çevrilir və proqram nə çap edir?

3. Compiler: C-dən assembly-yə

Compiler (gcc-də `cc1`) C mətnini oxuyur, səhvləri tapır, tipləri yoxlayır və eyni işi görən assembly yazır. Assembly Fəsil 1-dən tanışdır: hər sətir bir instruction-dur, amma hələ mətn şəklində.

`gcc -S -O0 hello.c` çıxışının əsas hissəsi:

```
    .section    .rodata
.LC0:
    .string "salam"
```

```
    .text
    .globl  main
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
    movl    $0, %eax
    popq    %rbp
    ret
```

`endbr64` — CPU-nun təhlükəsizlik nişanıdır: "bura tullanmaq icazəlidir". Proqramın işinə təsir etmir, sadəcə 4 byte tutur. Bölmə 4-dəki offset-lər onu nəzərə alır.

- `.section .rodata` — sonrakı data "yalnız oxunan" bölməyə düşəcək. `"salam"` mətni orada saxlanılır (Bölmə 6).
- `.LC0` — mətnin adı (label). Ünvanı hələ bilinmir, ona görə adla istinad olunur.
- `leaq .LC0(%rip), %rax` — mətnin ünvanını hesabla. `movq %rax, %rdi` — onu birinci arqument kimi `rdi`-yə qoy. Arqumentlərin register-lərdə necə ötürüldüyünü Fəsil 4-də açacağıq.
- `call puts@PLT` — diqqət: kodda `printf` yazmışdıq, compiler isə `puts` çağırır.

Niyə? Compiler `printf`-in nə etdiyini bilir. Format sətrində `%` yoxdursa və sonu `\n`-dirsə, `printf("salam\n")` ilə `puts("salam")` eyni nəticəni verir. `puts` isə formatı təhlil etmir. Compiler bunu `-O0`-da belə edir. `printf("%d\n", x)` isə `printf` olaraq qalır. Compiler tərcüməçi yox, redaktordur.

> 🖼️ **Şəkil 211:** Compiler · printf yerinə puts

Burada nəyə bax: sol tərəfdəki çağırış sağ tərəfdə başqa funksiyaya çevrilib, nəticə isə eynidir.

> 🖼️ **Şəkil 212:** hello.c və hello.s · sətir-sətir

Burada nəyə bax: bir C sətri bir neçə assembly sətrinə çevrilir. `main`-in əvvəlindəki və sonundakı sətirlər stack frame üçündür (Fəsil 4).

Kotlin compiler-i (`kotlinc`) assembly yox, **JVM bytecode** yazır: real CPU üçün yox, JVM adı virtual maşın üçün instruction-lar. `println("salam")` belə olur:

```
0: ldc           #8      // String salam
2: getstatic     #14     // Field java/lang/System.out
5: swap
6: invokevirtual #20     // Method java/io/PrintStream.println
9: return
```

Kotlin-in `println` funksiyası `inline` funksiyadır: kotlinc onun içini bura yapışdırıb. Ona görə proqram işləyəndə Kotlin kitabxanasından heç bir sinif yüklənmədi (Özün gör, 5).

> 🖼️ **Şəkil 213:** İki compiler · assembly və bytecode

Burada nəyə bax: C-də bu sətir üç instruction olur, bytecode-da dörd. Say fərqlidir, iş eynidir: mətni götür, çap funksiyasını çağır.

4. Assembler və object faylı: boşluqlu maşın kodu

Assembler (`as`) hər assembly sətrini byte-lara çevirir: `push %rbp` → `55`, `mov %rsp,%rbp` → `48 89 e5`, `ret` → `c3`. Nəticə **object faylıdır**: `hello.o`, 1 496 byte.

Amma assembler iki şeyi bilmir:

1. `"salam"` mətni yaddaşda harada olacaq? Bunu proqramın bütün hissələri birləşəndə bilmək olar.
2. `puts` haradadır? O, ümumiyyətlə bu faylda deyil, libc-dədir.

> 🖼️ **Şəkil 214:** Object faylı · boşluqlu blank

Burada nəyə bax: blankın üçüncü sətri tamdır, ilk ikisində isə ünvan yeri boş qalıb.

Bilinməyən yerlərə assembler sıfır yazır və qeyd buraxır. Bu qeydə **relocation** deyilir. `objdump -dr hello.o`:

```
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax
            b: R_X86_64_PC32    .rodata-0x4
  12:   e8 00 00 00 00          call   17 <main+0x17>
            13: R_X86_64_PLT32  puts-0x4
```

- `e8` — `call` instruction-ı. Ardınca gələn 4 byte hədəfə qədər olan məsafədir, hələlik `00 00 00 00`.
- `13: R_X86_64_PLT32 puts-0x4` — "0x13-cü byte-dan başlayan 4 byte-a `puts`-a qədər olan məsafəni yaz". Bu, linker üçün tapşırıqdır. `-0x4` düzəlişi məsafənin növbəti instruction-dan hesablandığı üçündür.

> 🖼️ **Şəkil 215:** hello.o · sıfır byte-lar və relocation

Burada nəyə bax: 30 byte-ın 8-i hələ sıfırdır və hər sıfır qrupunun öz qeydi var.

Object faylının **symbol table**-ı da var. `nm hello.o`:

```
0000000000000000 T main
                 U puts
```

- `T` — "bu ad burada təyin olunub, kod bölməsindədir".
- `U` — undefined: "bu adı istifadə edirəm, amma məndə yoxdur".
- Object faylı da ELF formatındadır, amma tipi `REL`-dir (relocatable). Entry point 0-dır, program header yoxdur. Onu işə salmaq olmaz, yalnız linker-ə vermək olar.

> 🖼️ **Şəkil 216:** Symbol table · T və U

Burada nəyə bax: `T` çıxıntıdır, `U` boşluqdur. Linker onları bir-birinə uyğunlaşdırır.

Dayan və düşün D3.2: `hello.o`-da `call` instruction-ının ünvan hissəsi niyə `00 00 00 00`-dır? Onu kim və nə vaxt doldurur? Kotlin-də `invokestatic demo/UtilKt.square` kimi sətirlərdə ünvan yoxdur, ad var. Bu ad nə vaxt ünvana çevrilir?

5. Linker: hissələri birləşdirir, boşluqları doldurur

Linker (`ld`) object fayllarını və kitabxanaları bir executable-da birləşdirir. Onun iki işi var:

1. **Symbol resolution:** hər `U` adı üçün onu təyin edən faylı tapır.
2. **Relocation:** hər hissəyə son ünvan verir və relocation qeydlərinə əsasən sıfır byte-ları real məsafələrlə doldurur.

İki fayldan ibarət proqrama bax (kodu Kod bölməsində, Nümunə 1-dədir):

```
$ nm main.o           $ nm util.o
T main                b calls
U printf              T square
U square
```

- `main.o`-da `square` U-dur, `util.o`-da T. Linker onları cütləyir.
- `calls` kiçik `b` ilə görünür. O, `static`-dir: yalnız `util.c`-nin içində görünür, başqa fayl onu istəyə bilməz.

> 🖼️ **Şəkil 217:** Linker · tapmaca hissələri

Burada nəyə bax: hər qırıq xətt bir U adını onu təyin edən fayla bağlayır.

Linker xətaları bu iki işdən birinin alınmamasıdır:

Xəta	Nə deməkdir	Bu maşında necə aldıq
`undefined reference to 'square'`	U var, heç bir faylda T yoxdur	`gcc main.o -o app`
`multiple definition of 'square'`	Eyni ad üçün iki T	`gcc main.o util.o util.o -o app`

Bağlamadan sonra `main`-də sıfırlar doldurulub:

```
115b:   call   1050 <puts@plt>
```

`1050` — `puts`-un özü deyil, kiçik keçid kodudur: `puts@plt`. Onun nə üçün lazım olduğunu Senior səviyyəsində açacağıq.

> 🖼️ **Şəkil 218:** Linker-in iki işi · resolution və relocation

Burada nəyə bax: sol tərəf adlarla, sağ tərəf ünvanlarla işləyir.

Kitabxana — hazır object fayllarının toplusu. İki növü var:

- **Static library** (`libutil.a`) — object fayllarının arxivi. Linker ondan yalnız lazım olan faylları götürüb executable-a köçürür. Linker faylları soldan sağa oxuyur: `gcc -L. -lutil main.c` xəta verdi, çünki `libutil.a`-ya baxanda hələ heç kim `square` istəməmişdi. `gcc main.c -L. -lutil` işlədi.
- **Shared library** (`libc.so.6`) — linker ondan kod köçürmür. Executable-a yalnız "mənə `libc.so.6` lazımdır" qeydini (`NEEDED`) yazır. Kod proqram işə düşəndə yüklənir (Bölmə 7). Buna **dynamic linking** deyilir.
- `gcc -static hello.c` libc-nin lazım olan hissəsini fayla köçürür: 785 336 byte. Adi yığma 15 960 byte-dır.
- `-l` adlandırma qaydası: `-lutil` → `libutil.a` və ya `libutil.so`, `-lm` → `libm.so`. Kotlin-dəki `System.loadLibrary("square")` da eyni qaydanı işlədir: `libsquare.so` (Tipik səhv).

> 🖼️ **Şəkil 219:** Static və dynamic · 785 336 və 15 960 byte

Burada nəyə bax: solda libc-nin kodu faylın içindədir, sağda onun yerində bir sətirlik qeyd var.

6. ELF: icra olunan faylın quruluşu

Linux-da executable, object faylı və shared library eyni formatdadır: **ELF** (Executable and Linkable Format). Android-da `.so` faylları da ELF-dir.

ELF faylının üç hissəsi var.

1. ELF header — faylın ilk 64 byte-ı. `readelf -h hello`:

```
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Type:                              DYN (Position-Independent Executable file)
Machine:                           Advanced Micro Devices X86-64
Entry point address:               0x1060
```

- `7f 45 4c 46` = `0x7F 'E' 'L' 'F'`: Fəsil 0-dakı "magic number"-in özü.
- `02` — 64 bit, `01` — little-endian (Fəsil 0).
- Type: object faylında `REL`, executable-da `DYN` (PIE, Senior səviyyəsində), shared library-də də `DYN`.
- Entry point: proqramın icraya başlayacağı ünvan. Bu `main` deyil, `_start`-dır: 0x1060.

2. Section-lar — linker üçün bölmələr. `hello`-da 31 section var, ən vacibi dördüdür:

Section	Nə saxlayır	Nümunə	İcazə
`.text`	Machine code	`main`, `_start`	oxu, icra
`.rodata`	Dəyişməz data	`"salam"` mətni	yalnız oxu
`.data`	İlkin dəyəri olan qlobal dəyişənlər	`int counter = 42;`	oxu, yaz
`.bss`	Sıfırla başlayan qlobal dəyişənlər	`int zeros[1024];`	oxu, yaz

3. Segment-lər (program header-lər) — loader üçün. Loader section-lara baxmır. Ona "faylın bu hissəsini bu icazələrlə yaddaşa qoy" demək kifayətdir. `readelf -l hello`-da dörd `LOAD` segmenti var: R, R E, R, RW. Section-lar segment-lərə yığılır: `.text` R E-yə, `.rodata` R-ə, `.data` və `.bss` RW-yə.

> 🖼️ **Şəkil 220:** ELF · kitab kimi

Burada nəyə bax: kitabın iki mündəricatı var — biri cildçi üçün, biri kitabxanaçı üçün.

> 🖼️ **Şəkil 221:** Section-lar və segment-lər · iki görünüş

Burada nəyə bax: onlarca section dörd segmentə yığılır və hər segmentin öz icazəsi var.

Sirr-in cavabı. `.bss` section-unun tipi `NOBITS`-dir: faylda heç bir byte tutmur, yalnız ölçüsü yazılır.

```
$ size bss_zero bss_one
   text    data     bss     dec  filename
   1412     600 4000032 4002044  bss_zero
   1412 4000616       8 4002036  bss_one
```

- `bss_zero`-da massiv `.bss`-dədir: faylda 0 byte, yaddaşda 4 MB.
- `bss_one`-da ilk element 1-dir. Massiv artıq "hamısı sıfır" deyil, ona görə bütövlüklə `.data`-ya düşür: faylda 4 MB.
- 4 MB-ı kim yaradır? Loader. O, `.bss` üçün sıfırlarla dolu yaddaş ayırır (Bölmə 7).

> 🖼️ **Şəkil 222:** Dörd section · Sirr-in cavabı

Burada nəyə bax: dörd sətirdən yalnız sonuncusu faylda yer tutmur.

Dayan və düşün D3.3: `static int cache[256];` funksiyanın içində yazılıb. O, stack-dədir, yoxsa `.bss`-də? Faylın ölçüsünə təsir edirmi?

7. Loader: fayldan prosesə

Fayl diskdə yatır. Onu işləyən prosesə **loader** (yükləyici) çevirir. Linux-da loader-in iki hissəsi var: kernel və dynamic linker.

Terminalda `./hello` yazanda shell yeni proses yaradır (Faza 1-də) və **`execve`** system call-ını çağırır. **System call** — proqramın kernel-dən xidmət istəməsidir (Fəsil 6). `execve` deyir: “bu prosesin içini `./hello` ilə əvəz et”.

> 🖼️ **Şəkil 223:** Loader · main səhnəyə çıxmazdan əvvəl

Burada nəyə bax: `main` beşinci addımdır, birinci yox.

Addım 1 — kernel.

- Faylın `x` (icra) icazəsini yoxlayır. Yoxdursa: `Permission denied`.
- İlk byte-lara baxır: `7f 45 4c 46`, deməli ELF-dir.
- Hər `LOAD` segmentini öz icazəsi ilə yaddaşa qoşur, yəni **map** edir. Fayl bütöv kopyalanmır: hissələr lazım olanda oxunur. Bunu Faza 2-də (virtual yaddaş) açacağıq.
- `.bss` üçün sıfırla dolu yaddaş ayırır. Sirr-dəki 4 MB burada yaranır.

- `INTERP` segmentində adı yazılan proqramı da yükləyir: `/lib64/ld-linux-x86-64.so.2`. Bu, **dynamic linker**-dir, qısa adı **ld.so**.
- Stack-ə proqramın arqumentlərini (`argv`), mühit dəyişənlərini (`envp`) və **auxv**-ni qoyur. auxv — kernel-in ld.so üçün qısa qeydidir: proqram harada yüklənib, entry point haradadır, səhifə ölçüsü neçədir.
- İcranı ld.so-ya verir. `hello`-nun `_start`-ına yox!

Addım 2 — ld.so.

- `hello`-nun `NEEDED` siyahısını oxuyur: `libc.so.6`. Faylı tapır və map edir.
- Relocation-ları tamamlayır: `puts` kimi adların libc-dəki real ünvanlarını cədvələ yazır. Bu cədvəlin adı **GOT**-dur (Senior səviyyəsi). Bu maşında 96 relocation.
- `hello`-nun entry point-inə keçir: `_start`, 0x1060.

Addım 3 — libc başlanğıcı.

- `_start` bir neçə instruction-dır: `main`-in ünvanını `rdi`-yə qoyur və `__libc_start_main`-i çağırır.
- `__libc_start_main` libc-ni hazırlayır və `main`-i çağırır.
- `main` qayıdanda onun dəyəri ilə `exit` çağırılır. `return 0` prosesin **çıxış kodu** olur.

> 🖼️ **Şəkil 224:** execve-dən main-ə · kernel, ld.so, hello

Burada nəyə bax: kernel işini bitirəndə icra `hello`-ya yox, ld.so-ya keçir. `hello`-nun öz kodu ən sonda başlayır.

Bunu gözünlə gör. `hello_g` — `-g` ilə yığılmış eyni proqramdır. gdb-nin `starti` əmri onu birinci instruction-da dayandırır:

```
$ gdb -q -batch -ex starti -ex 'x/i $pc' ./hello_g
Program stopped.
0x00007ffff7fe4530 in _start () from /lib64/ld-linux-x86-64.so.2
=> 0x7ffff7fe4530 <_start>:  mov    %rsp,%rdi
```

Birinci instruction ld.so-nundur. İndi `main`-də dayanıb "məni kim çağırdı?" soruşaq (`bt`, arqumentlər qısaldılıb):

```
#0  main () at hello.c:5
#1  __libc_start_call_main (...) at ../sysdeps/nptl/libc_start_call_main.h:58
#2  __libc_start_main_impl (...) at ../csu/libc-start.c:360
#3  _start ()
```

Zəncir aşağıdan yuxarı oxunur: `_start` → `__libc_start_main` → `main`.

`main`-dən əvvəl daha kim işləyir? Constructor-lar. C-də funksiyanı `__attribute__((constructor))` ilə işarələsən, o, `main`-dən əvvəl işləyir. C++-da qlobal obyektlərin constructor-ları da belədir. Linker onların ünvanlarını `.init_array` section-una yazır. `destructor` isə `main` qayıdandan sonra, `.fini_array`-dən işləyir.

```
        // libinit.c: shared library-nin öz constructor-u
        #include <stdio.h>
```

```c
__attribute__((constructor))
static void lib_init(void) {
    printf("1) libinit.so constructor-u\n");
}

int lib_value(void) { return 42; }
```

```c
// ctor.c: main-dən əvvəl və sonra kim işləyir?
#include <stdio.h>

int lib_value(void);                // libinit.so-dan

__attribute__((constructor))
static void before_main(void) {     // .init_array-ə düşür
    printf("2) proqramın constructor-u\n");
}

__attribute__((destructor))
static void after_main(void) {      // .fini_array-ə düşür
    printf("4) proqramın destructor-u\n");
}

int main(void) {
    printf("3) main, lib_value() = %d\n", lib_value());
    return 0;
}
```

```
$ gcc -O2 -shared -fPIC libinit.c -o libinit.so
$ gcc -O2 ctor.c -L. -linit -Wl,-rpath,'$ORIGIN' -o ctor
$ ./ctor
1) libinit.so constructor-u
2) proqramın constructor-u
3) main, lib_value() = 42
4) proqramın destructor-u
$ readelf -S -W ctor | grep -E 'init_array|fini_array'
  [21] .init_array  INIT_ARRAY  0000000000003d78 002d78 000010 08  WA  0   0  8
  [22] .fini_array  FINI_ARRAY  0000000000003d88 002d88 000010 08  WA  0   0  8
```

- `.init_array` 16 byte-dır, yəni iki göstərici: biri bizim `before_main`, digəri gcc-nin öz köməkçisi `frame_dummy` (`nm ctor` ilə görünür).
- `-Wl,-rpath,'$ORIGIN'` executable-a “kitabxananı öz qovluğumda axtar” qeydini yazır. Onsuz ld.so `libinit.so`-nu tapmaz.

Kim çağırır? `-O0 -g` ilə yığıb gdb-də hər constructor-da dayanaq (iki ayrı işə salma; ünvanlar və arqumentlər qısaltılıb):

```
$ gdb -q -batch -ex 'set breakpoint pending on' -ex 'break lib_init' -ex run -ex bt ./ctor_g
#0  lib_init (...) at libinit.c:6
#1  call_init (...) at ./elf/dl-init.c:74
#2  call_init (...) at ./elf/dl-init.c:120
#3  _dl_init (...) at ./elf/dl-init.c:121
#4  _dl_start_user () from /lib64/ld-linux-x86-64.so.2

$ gdb -q -batch -ex 'break before_main' -ex run -ex bt ./ctor_g
#0  before_main (...) at ctor.c:8
#1  call_init (...) at ../csu/libc-start.c:145
#2  __libc_start_main_impl (...) at ../csu/libc-start.c:347
#3  _start ()
```

- Kitabxananın constructor-unu ld.so özü (`_dl_init`) çağırır, hələ proqramın `_start`-ı işləməmiş.
- Proqramın öz constructor-unu `__libc_start_main` çağırır, `main`-dən bir addım əvvəl.
- Android-də də belədir: `.so`-nun constructor-ları `System.loadLibrary`-nin içində, `dlopen` zamanı işləyir. JVM `JNI_OnLoad`-u yalnız ondan sonra axtarır. Constructor-da ağır iş görmək — yavaş `loadLibrary` deməkdir.

Kotlin-də `main`-dən əvvəl `<clinit>` işləyir. Top-level `val`-lar faylın sinfinin (`InitKt`) static initializer-ində hesablanır. JVM `InitKt.main`-i çağırmazdan əvvəl həmin sinfi initialize edir:

```
// Init.kt: Kotlin-də main-dən əvvəl kim işləyir?
val config = loadConfig()            // top-level val: InitKt.<clinit>-də hesablanır

fun loadConfig(): String {
    println("1) InitKt.<clinit>: config yüklənir")
    return "prod"
}

object Db {                          // object: öz <clinit>-i, yalnız ilk istifadədə
    init { println("3) Db.<clinit>: ilk istifadə") }
    fun query() = "ok"
}

fun main() {
    println("2) main, config = $config")
    println("4) Db.query() = ${Db.query()}")
}
```

```
$ kotlinc Init.kt -d out && java -cp out InitKt
1) InitKt.<clinit>: config yüklənir
2) main, config = prod
3) Db.<clinit>: ilk istifadə
4) Db.query() = ok
$ javap -c -p -cp out InitKt | sed -n '/static {}/,/return/p'
  static {};
    Code:
       0: invokestatic  #63                 // Method loadConfig:()Ljava/lang/String;
       3: putstatic     #11                 // Field config:Ljava/lang/String;
       6: return
```

- `config` `main`-dən əvvəl hazırdır: `InitKt.<clinit>` artıq işləyib. Bu, C-dəki `.init_array`-in JVM qarşılığıdır.
- `Db` `object`-dir: onun `<clinit>`-i yalnız `Db.query()` ilk dəfə çağırılanda işləyir. “3)” “4)”-dən əvvəl çap olunur, çünki string şablonu `println`-dən əvvəl hesablanır.
- Android-də `Application.onCreate`-dən əvvəl də kod işləyir: məsələn, ContentProvider-lərin `onCreate`-i. Onu Faza 8-də görəcəksən.

> 🖼️ **Şəkil 225:** main-dən əvvəl · constructor-ların sırası

Burada nəyə bax: hər iki dünyada `main` sıranın ortasındadır. Ondan əvvəl işləyən kod da sənin proqramındır və başlanğıc vaxtına yazılır.

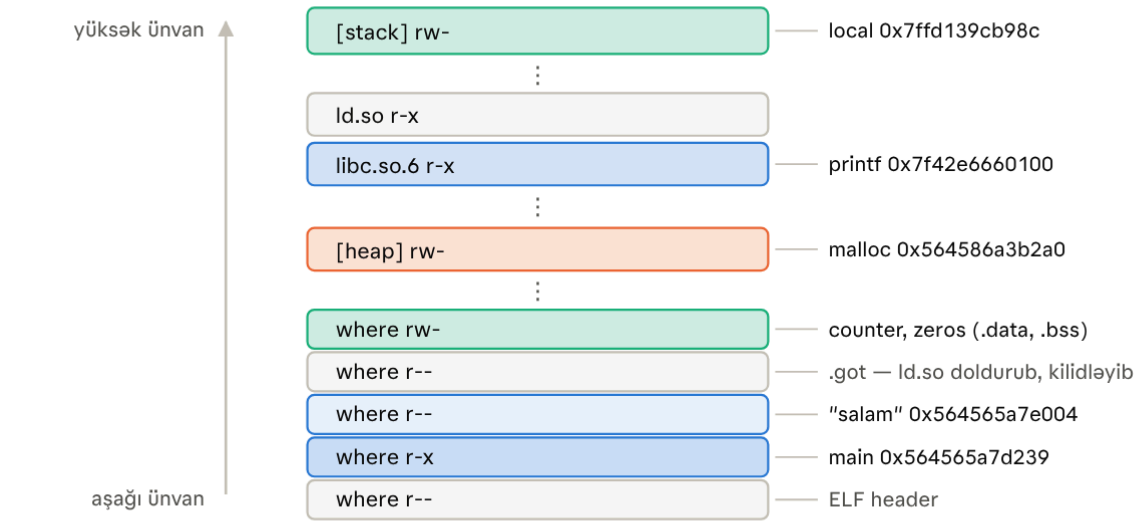
Dayan və düşün D3.5: `libinit.so`-nun constructor-u proqramın öz constructor-undan niyə əvvəl işləyir? Sıranı kim təyin edir?

Prosesin yaddaş xəritəsi. `where.c` hər hissənin ünvanını çap edir, sonra `/proc/self/maps`-i oxuyur (kodu Özün gör, 4-də):

```
main (kod, .text):      0x564565a7d239
"salam" (.rodata):     0x564565a7e004
counter (.data):        0x564565a80010
zeros (.bss):           0x564565a80040
heap (malloc):          0x564586a3b2a0
printf (libc):          0x7f42e6660100
local (stack):          0x7ffd139cb98c
```

- Kod, sabitlər və dəyişənlər yan-yanadır (0x5645...): bunlar `where` faylının segmentləridir.
- **Heap** — proqramın işləyərkən `malloc` ilə istədiyi yaddaş. Onu Faza 2-də açacağıq.
- libc 0x7f...-dadır. Stack ən yuxarıdadır: 0x7ffd....
- `/proc/self/maps`-də `where` üçün 5 sətir var: `r--`, `r-x`, `r--`, `r--`, `rw-`. `readelf -l`-də isə 4 `LOAD` var idi. Artıq `r--` sətri belə yaranır: ld.so GOT-u doldurandan sonra onu yalnız oxunan edir. Buna **RELRO** deyilir (Senior səviyyəsi).

🗺️ **Bir prosesin yaddaş xəritəsi (where.c, bu maşın)**



Yadda saxla: proses = faylın segment-ləri + heap + kitabxanalar + stack, hər biri öz icazəsi ilə.

Miqyas yoxdur: əslində boşluqlar çox böyükdür. ASLR hər işə salmada ünvanları dəyişir.

Burada nəyə bax: hər dəyişənin yeri növündən asılıdır: kod `r-x`-də, sabit `r--`-də, qlobal `rw-`-də, lokal stack-də.

ASLR. Proqramı ikinci dəfə işə salanda `main` 0x55b4adfa2239-da idi. Hər dəfə başqa ünvan. Buna **ASLR** (Address Space Layout Randomization) deyilir: hücumçu kodun harada olduğunu əvvəlcədən bilməsin. Bunun üçün `hello` **PIE**-dir (Position-Independent Executable): istənilən ünvanda işləyə bilir. `setarch -R` bir proses üçün ASLR-i söndürür: onda `main` hər dəfə 0x555555555239-dadır.

Static proqram. `hello_static`-də `INTERP` yoxdur. Kernel icranı birbaşa onun `_start`-ına verir, ld.so və `libc.so.6` yüklənmir. `strace` system call-ları bir-bir göstərir: dynamic `hello` — 36, static — 18. Fərqin çoxu ld.so-nun işidir: `/etc/ld.so.cache`-i açmaq, `libc.so.6`-nı oxumaq, segmentləri map etmək.

Kotlin proqramı necə başlayır? `java -jar hello.jar` də eyni yolu keçir. `java` adi ELF executable-dır: kernel onu yükləyir, ld.so onun kitabxanalarını qoşur. Sonra `java` `libjvm.so`-nu yükləyir (Özün gör, 5-də `/proc/self/maps`-də görəcəksən). Yalnız bundan sonra JVM `HelloKt` sinfini oxuyur. Android-də isə tətbiq prosesi `execve` ilə yaranmır: **zygote** adlı hazır prosesdən `fork` ilə ayrılır. ART və framework artıq yaddaşda olur (Faza 1 və Faza 8).

Dayan və düşün D3.4: `hello_static`-də icranı `_start`-a kim verir? ld.so yoxdursa, `strace` niyə 0 yox, 18 system call göstərir?

8. JVM və Android: eyni mərhələlər, başqa vaxt

Kotlin kodu da compiler, linker və loader mərhələlərindən keçir. Fərq: bəzi mərhələlər build vaxtından **runtime**-a, yəni proqramın işlədiyi vaxta köçür.

> 🖼️ **Şəkil 227:** Üç mətbəx · C, JVM, Android

Burada nəyə bax: üç mətbəxdə də yemək bişirilir. Fərq yalnız nə vaxt bişirilməsindədir.

Class faylı. `kotlinc Hello.kt` → `HelloKt.class`, 628 byte:

```
$ od -A x -t x1 HelloKt.class | head -1
000000 ca fe ba be 00 00 00 34 00 2c 01 00 07 48 65 6c
```

- `ca fe ba be` — class faylının magic number-i. ELF-in `7f 45 4c 46`-sı kimi.
- `00 34` = 52 — formatın versiyası, Java 8 səviyyəsi. kotlinc default olaraq bunu yazır.
- Sonra **constant pool** gəlir: sinfin istifadə etdiyi bütün adlar və sabitlər.

Ad, ünvan yox. C-də `U puts` vardı, linker onu build vaxtı ünvana çevirirdi. Class faylında da adlar var, amma onları build vaxtı heç kim ünvana çevirmir. `javap -c -v MainKt.class`:

```
#12 = Methodref    #8.#11    // demo/UtilKt.square:(I)I
...
invokestatic  #12            // Method demo/UtilKt.square:(I)I
```

JVM bu adı instruction birinci dəfə icra olunanda həll edir: sinfi tapır, yükləyir, metodu tapır və nəticəni yadda saxlayır. Linker-in işini JVM proqram işləyərkən görür.

Lazy class loading. JVM sinfi yalnız birinci dəfə lazım olanda yükləyir. `Main.kt`-də `Rare` sinfi yalnız `RARE` mühit dəyişəni olanda çağırılır:

```
$ java -Xlog:class+load -cp out demo.MainKt | grep -E 'demo|^49|nadir'
[0.106s][info][class,load] demo.MainKt source: file:/home/claude/ch03/ktmulti/out/
[0.107s][info][class,load] demo.UtilKt source: file:/home/claude/ch03/ktmulti/out/
49
$ RARE=1 java -Xlog:class+load -cp out demo.MainKt | grep -E 'demo|^49|nadir'
[0.039s][info][class,load] demo.MainKt source: file:/home/claude/ch03/ktmulti/out/
[0.041s][info][class,load] demo.UtilKt source: file:/home/claude/ch03/ktmulti/out/
49
[0.047s][info][class,load] demo.Rare source: file:/home/claude/ch03/ktmulti/out/
nadir yol
```

`Rare.class` diskdə var, amma birinci işə salmada yaddaşa heç girmədi. İkincidə `49`-dan sonra, ehtiyac yarananda yükləndi.

Başlanğıcın qiyməti. `hello.jar` Kotlin runtime ilə birlikdə 4 793 635 byte, 2 906 fayldır. İşləyəndə 590 sinif yükləndi, 589-u **CDS** (Class Data Sharing) arxivindən. CDS — JDK sinifləri üçün əvvəlcədən hazırlanmış və birbaşa map olunan fayldır. Kotlin kitabxanasından isə 0 sinif yükləndi: `println` inline-dır (Bölmə 3).

Proqram	Orta başlanğıc (bu maşın)
C `hello`	≈1 ms
`java -jar hello.jar`	54 ms
eyni, `-Xshare:off` (CDS-siz)	103 ms

Android. Android-də JVM yoxdur. Yol belədir:

1. `kotlinc` → `.class` faylları, eynilə.
2. **D8** bütün `.class` fayllarını **DEX** formatına çevirir: `classes.dex`. DEX — Android-in bytecode formatıdır. JVM bytecode stack ilə işləyir, DEX isə register-lərlə. Release build-da D8 əvəzinə **R8** işləyir: kodu kiçildir və DEX yazır.
3. APK-ya `classes.dex`, resurslar və `lib/arm64-v8a/*.so` qoyulur. `.so` faylları adi ELF-dir.
4. Cihazda DEX-i **ART** (Android Runtime) icra edir. Android 7-dən ART hibriddir. Əvvəlcə **interpreter** işləyir: bytecode-u bir-bir oxuyub icra edir. Tez-tez işləyən metodları **JIT** (Just-In-Time) proqram işləyərkən machine code-a çevirir. Cihaz boş və şarjda olanda **dex2oat** profilə əsasən isti kodu əvvəlcədən çevirir: **AOT** (Ahead-Of-Time).
5. dex2oat-ın nəticəsi Android 8-dən iki fayldır: `.vdex` (yoxlanılmış DEX) və `.odex` (machine code). `.odex` əslində ELF faylıdır, içində OAT formatı var.
6. `System.loadLibrary("square")` çağırılanda Android-in dynamic linker-i (`linker64`) `libsquare.so`-nu tapıb yükləyir. JNI funksiyanı adla axtarır: `Java_demo_Native_square`.

> 🖼️ **Şəkil 228:** Android · .kt-dən cihazda işləyən koda

Burada nəyə bax: APK-dan iki ayrı yol çıxır. DEX-i ART icra edir, `.so`-nu linker64 yükləyir.

Native kodu bu maşında da yığmaq olar, amma işə salmaq olmaz:

```
$ clang --target=aarch64-linux-android24 -O2 -fPIC -c square.c
$ ld.lld -shared square.o -o libsquare.so
$ file libsquare.so
libsquare.so: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV), dynamically linked,
not stripped
$ llvm-objdump -d --no-show-raw-insn libsquare.so
0000000000010264 <square>:
   10264:       mul     w0, w0, w0
   10268:       ret
```

`x*x` ARM64-də bir instruction-dır: `mul`. Bu fayl x86-64 maşında işləmir — burada işlədilməyib, cihazda yoxla.

> 🖼️ **Şəkil 229:** Ad nə vaxt ünvana çevrilir · üç an

Burada nəyə bax: üç kartda eyni iş görülür — ad ünvana çevrilir. Dəyişən yalnız saat oxunun üstündəki nöqtədir.

Özün gör: doğuluşu addım-addım izlə

Mühit: Linux 6.18, x86-64 (KVM, 2 core), gcc 13.3, GNU ld 2.42, glibc 2.39, clang 18.1, OpenJDK 21.0.13, kotlinc 2.1.0. Bütün çıxışlar bu maşındandır. Sənin ünvanların, ölçülərin və vaxtların bir az fərqli olacaq — bu normaldır.

Ubuntu-da lazım olan alətlər: `sudo apt install gcc gdb strace binutils`. Kotlin üçün JDK və `kotlinc` lazımdır. Faylların hamısı repoda: `phase-00-computer-foundations/ch03-build/`.

> 🖼️ **Şəkil 230:** Alətlər xəritəsi · hansı alət hansı mərhələyə baxır

Burada nəyə bax: hər mərhələnin öz rentgeni var. Aşağıdakı beş təcrübə bu xəritəni soldan sağa gəzir.

Təcrübə 1. Dörd mərhələni ayrı-ayrı gör

```
$ gcc -E hello.c -o hello.i      # 1. preprocessor
$ gcc -S hello.i -o hello.s      # 2. compiler
$ gcc -c hello.s -o hello.o      # 3. assembler
$ gcc hello.o -o hello           # 4. linker
$ ./hello
salam
```

Fayl	Ölçü	İçində nə var
`hello.c`	127 B	sənin 7 sətrin
`hello.i`	21 309 B	`stdio.h` açılıb: 820 sətir
`hello.s`	660 B	assembly mətni
`hello.o`	1 496 B	machine code, relocation-lar, symbol table
`hello`	15 960 B	hazır ELF executable

`gcc` bu işləri özü görmür, o dispetçerdir. `gcc -v` hansı proqramları çağırdığını göstərir (sətirlər qısaldılıb):

```
$ gcc -v hello.c -o hello 2>&1 | grep -E 'cc1 |^ as |collect2 '
 /usr/libexec/gcc/x86_64-linux-gnu/13/cc1 ... hello.c ...
 as -v --64 -o /tmp/cc0BbjzH.o /tmp/ccr4WpwE.s
 /usr/libexec/gcc/x86_64-linux-gnu/13/collect2 ...
```

- `cc1` — compiler. Preprocessor da onun içindədir.
- `as` — assembler. Fayllar `/tmp`-də yaranır və sonra silinir.
- `collect2` — linker-i (`ld`) çağıran köməkçi.

🖼️ **Şəkil 231:** Aralıq fayllar · /tmp-də doğulur, silinir

Burada nəyə bax: adi yığmada `.i` faylı heç yaranmır, `.s` və `.o` isə `/tmp`-də bir an yaşayır. Bayraqlar onları masada saxlamaq üçündür.

Kotlin-də eyni təcrübə:

```
$ kotlinc Hello.kt -d out_hello                      # yalnız compiler
$ kotlinc Hello.kt -include-runtime -d hello.jar     # + Kotlin runtime
$ stat -c '%s %n' Hello.kt out_hello/HelloKt.class hello.jar
92 Hello.kt
628 out_hello/HelloKt.class
4793635 hello.jar
```

Kotlin-də preprocessor və ayrıca assembler yoxdur: `kotlinc` mətndən birbaşa `.class` yazır. Linker də yoxdur: `.jar` sadəcə ZIP arxividir, adları JVM sonra həll edir.

🖼️ **Şəkil 232:** C və Kotlin · alətlərin sayı

Burada nəyə bax: C-nin zənciri ELF faylı ilə bitir, Kotlin-in zənciri isə ZIP arxivi ilə. Kotlin-də adları bağlamaq işi sonraya, JVM-ə qalır.

Variasiya: `gcc -O2 -S hello.c -o hello_o2.s` et və `hello.s` ilə müqayisə et. `pushq %rbp` qalırmı? `.rodata` bölməsinin adı dəyişdimi? (Cavab açarında.)

🖼️ **Şəkil 233:** Fayl ölçüləri · mənbədən jar-a qədər

Burada nəyə bax: şkala loqarifmikdir. `hello.i` mənbədən ≈168 dəfə, static versiya adi executable-dan 49 dəfə böyükdür.

Təcrübə 2. ELF-in içi və Sirr-in yoxlanışı

```
$ file hello.o hello
hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
hello:   ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, ...
$ readelf -h hello.o | grep -E 'Type|Entry|Number of program'
  Type:                              REL (Relocatable file)
  Entry point address:               0x0
  Number of program headers:         0
$ readelf -h hello | grep -E 'Type|Entry|Number of program'
  Type:                              DYN (Position-Independent Executable file)
  Entry point address:               0x1060
  Number of program headers:         13
```

🖼️ **Şəkil 234:** ELF header · hello.o və hello

Burada nəyə bax: magic number eynidir, tip və program header sayı fərqlidir. 0 program header — loader üçün xəritə yoxdur.

Object faylında program header yoxdur: loader onu yükləyə bilməz. Executable-da 13 dənə var. Ən vacibləri (boşluqlar qısaldılıb):

```
$ readelf -l -W hello | grep -E 'INTERP|LOAD|Requesting'
  INTERP  0x000318 ... 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD    0x000000 ... 0x000628 0x000628 R   0x1000
  LOAD    0x001000 ... 0x000175 0x000175 R E 0x1000
  LOAD    0x002000 ... 0x0000ec 0x0000ec R   0x1000
  LOAD    0x002db8 ... 0x000258 0x000260 RW  0x1000
```

Rəqəm cütlükləri `FileSiz` və `MemSiz`-dir. Son `LOAD`-da yaddaş fayldan 8 byte çoxdur (0x260 − 0x258): o 8 byte `.bss`-dir.

> 🖼️ **Şəkil 235:** LOAD segment-ləri · FileSiz və MemSiz

Burada nəyə bax: dörd segment-dən üçündə fayl və yaddaş eynidir. Fərq yalnız RW segment-indədir: o 8 byte faylda yoxdur, loader onu sıfırla verir.

İndi Sirr-i özün yoxla. `bss.c`:

```c
// Sirr: eyni ölçülü massiv, iki fərqli başlanğıc dəyər
#include <stdio.h>

#ifdef DATA
int big[1000000] = {1};   // ilk element 1, qalanı 0: .data
#else
int big[1000000];         // hamısı 0: .bss
#endif

int main(void) {
    printf("%d\n", big[0]);
    return 0;
}
```

```
$ gcc -O2 bss.c -o bss_zero
$ gcc -O2 -DDATA bss.c -o bss_one
$ stat -c '%s %n' bss_zero bss_one
15992 bss_zero
4016008 bss_one
$ readelf -S -W bss_zero | grep -E ' \.data | \.bss '
  [25] .data   PROGBITS  0000000000004000 003000 000010 00  WA  0   0  8
  [26] .bss    NOBITS    0000000000004020 003010 3d0920 00  WA  0   0 32
```

- `.bss`-in ölçüsü 0x3d0920 = 4 000 032 byte, tipi `NOBITS`: faylda yeri yoxdur.
- `bss_one`-da həmin massiv `.data`-dadır, tipi `PROGBITS`: 4 000 000 byte-ın hər biri fayla yazılıb, hamısı (birindən başqa) sıfır olsa belə.

Kotlin-də eyni massiv:

```kotlin
// 1 milyonluq massiv: class faylında data yoxdur, massiv işləyəndə yaranır
val big = IntArray(1_000_000)
```

```
$ kotlinc Zeros.kt -d out_Zeros && ls -l out_Zeros
-rw-r--r-- 1 root root  558 Nov 21 11:51 ZerosKt.class
$ javap -c -p -cp out_Zeros ZerosKt
  static {};
    Code:
       0: ldc           #14                 // int 1000000
       2: newarray       int
       4: putstatic     #11                 // Field big:[I
       7: return
```

Class faylında `.bss` yoxdur, ehtiyac da yoxdur. Massivi `newarray` instruction-ı proqram işləyəndə yaradır, JVM onu sıfırla doldurur. Bəs massivdə sıfır olmayan dəyərlər olsa? Bu, "Bunu az adam bilir" bölməsinin mövzusudur.

> 🖼️ **Şəkil 236:** Sıfır massiv · C və Kotlin

Burada nəyə bax: nə ELF, nə class faylı 4 MB sıfır daşımır. Fərq yalnız sıfırları kimin yaratmasındadır: kernel, ya JVM.

Təcrübə 3. Linker işdə

Faylların kodu Kod bölməsində, Nümunə 1-dədir. `multi/` qovluğunda:

```
$ gcc -c main.c util.c
$ nm main.o
0000000000000000 T main
                 U printf
                 U square
$ nm util.o
0000000000000000 b calls
0000000000000000 T square
```

İndi linker-i qəsdən səhv yola sal:

```
$ gcc main.o -o app
/usr/bin/ld: main.o: in function `main':
main.c:(.text+0xe): undefined reference to `square'
collect2: error: ld returned 1 exit status

$ gcc main.o util.o util.o -o app
/usr/bin/ld: util.o: in function `square':
util.c:(.text+0x0): multiple definition of `square'; util.o:util.c:(.text+0x0): first defined
here
collect2: error: ld returned 1 exit status
```

> 🖼️ **Şəkil 237:** Linker xətaları · çatmayan və artıq hissə

Burada nəyə bax: iki xəta bir qaydanın iki üzüdür. Bir U-ya sıfır T da, iki T da uyğun gəlmir.

`main.c:(.text+0xe)` — xətanın dəqiq yeri: `main`-in 0xe-ci byte-ındakı relocation. Linker səhvi C sətrində yox, relocation-da tapır.

Static library və sıra:

```
$ ar rcs libutil.a util.o
$ gcc -L. -lutil main.c -o app      # ❌ kitabxana əvvəldə
/usr/bin/ld: /tmp/ccarZ0CL.o: in function `main':
main.c:(.text+0xe): undefined reference to `square'
collect2: error: ld returned 1 exit status
$ gcc main.c -L. -lutil -o app      # ✅ kitabxana sonda
$ ./app
49
```

> 🖼️ **Şəkil 238:** Static kitabxana · sıra önəmlidir

Burada nəyə bax: kitabxana əvvəldə olanda linker onu "boş əlli" keçir. `U square` sonra yaranır, amma linker geri qayıtmır.

Kotlin-də `kotlinc` bütün faylları birlikdə görür: ad tapılmasa, xəta compile vaxtı çıxır. Amma JVM-də "linker xətası" runtime-da da ola bilər:

```
$ kotlinc Util.kt Main.kt -d out
$ javap -c -v -cp out demo.MainKt | grep square
   #9 = Utf8               square
  #11 = NameAndType        #9:#10         // square:(I)I
  #12 = Methodref          #8.#11         // demo/UtilKt.square:(I)I
         2: invokestatic  #12                // Method demo/UtilKt.square:(I)I
$ rm out/demo/UtilKt.class          # bir sinfi sil
$ java -cp out demo.MainKt
Exception in thread "main" java.lang.NoClassDefFoundError: demo/UtilKt
    at demo.MainKt.main(Main.kt:5)
```

`NoClassDefFoundError` JVM-in `undefined reference`-idir. Fərq: C-də bu xəta build-da çıxır və proqram heç yaranmır. JVM-də isə proqram işə düşür, xəta adın ilk istifadəsində çıxır. Android-də bunu R8-in səhv sildiyi sinifdə görə bilərsən.

> 🖼️ **Şəkil 239:** Ad tapılmayanda · build və runtime

Burada nəyə bax: qırmızı qutunun yeri. C-də xəta proqram doğulmadan çıxır, JVM-də isə proqram artıq işləyir.

Təcrübə 4. Loader işdə

System call-ları say:

```
$ strace ./hello 2>&1 >/dev/null | wc -l
37
$ strace ./hello_static 2>&1 >/dev/null | wc -l
19
```

Son sətir `+++ exited with 0 +++`-dır, deməli 36 və 18 system call. Dynamic versiyanın başı (qısaldılıb):

```
execve("./hello", ["./hello"], 0x7ffdc9dee110 /* 237 vars */) = 0
brk(NULL)                               = 0x55ebebebd000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fb744c3f000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
```

```
...
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0"..., 832) = 832
```

- ld.so əvvəlcə `/etc/ld.so.cache`-i açır: kitabxanaların harada olduğunun hazır siyahısı.
- Sonra `libc.so.6`-nı açır və ilk 832 byte-ı oxuyur. `\177ELF` — tanış magic number: 177 səkkizlik say sistemində 0x7f-dir.

> 🖼️ **Şəkil 240:** System call-lar · dynamic 36, static 18

Burada nəyə bax: `salam`-ı yazan yalnız bir `write`-dır. Dynamic versiyada system call-ların təxminən üçdə ikisi ld.so-nun işidir.

Kernel-in ld.so-ya buraxdığı qeydi (auxv) gör:

```
$ LD_SHOW_AUXV=1 ./hello | grep -E 'AT_(PHDR|ENTRY|BASE|PAGESZ|RANDOM|EXECFN)'
AT_PAGESZ:            4096
AT_PHDR:              0x558b9d8ab040
AT_BASE:              0x7fb1a4975000
AT_ENTRY:             0x558b9d8ac060
AT_RANDOM:            0x7ffeeb59a319
AT_EXECFN:            ./hello
```

- `AT_ENTRY` = proqramın yükləndiyi baza (0x558b9d8ab000) + 0x1060. readelf-dəki entry point budur.
- `AT_BASE` — ld.so-nun öz ünvanı. `AT_RANDOM` — kernel-in verdiyi 16 təsadüfi byte-ın ünvanı; libc onlardan stack qoruyucusu düzəldir (Fəsil 4).

> 🖼️ **Şəkil 241:** auxv · kernel-in ld.so üçün qeydi

Burada nəyə bax: ld.so hələ heç nə bilmir. Proqramın harada olduğunu, `_start`-ın ünvanını və təsadüfi byte-ları məhz bu qeyddən öyrənir.

ld.so-nun işini gör:

```
$ LD_DEBUG=statistics ./hello 2>&1 | grep relocations
      3621:                 number of relocations: 96
      3621:      number of relocations from cache: 7
$ LD_DEBUG=bindings ./hello 2>&1 | grep -nE "symbol .puts|transferring"
87:      3625: binding file ./hello [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol
`puts' [GLIBC_2.2.5]
104:      3625: transferring control: ./hello
```

`puts` 87-ci sətirdə bağlanır, idarə `hello`-ya 104-cü sətirdə keçir. Yəni bütün adlar `main`-dən əvvəl həll olunur. Bunun adı BIND_NOW-dur; alternativi Senior səviyyəsində.

Prosesin yaddaş xəritəsi, `where.c`:

```c
// Proqramın hissələri yaddaşda harada yaşayır?
#include <stdio.h>
#include <stdlib.h>
```

```c
int counter = 42;            // ilkin dəyərli qlobal: .data
int zeros[1024];             // sıfır qlobal: .bss
const char *greeting = "salam";  // "salam" mətni: .rodata

int main(void) {
    int local = 7;                      // stack
    int *heap = malloc(sizeof(int));    // heap
    printf("main (kod, .text):      %p\n", (void *)main);
    printf("\"salam\" (.rodata):     %p\n", (void *)greeting);
    printf("counter (.data):        %p\n", (void *)&counter);
    printf("zeros (.bss):           %p\n", (void *)zeros);
    printf("heap (malloc):          %p\n", (void *)heap);
    printf("printf (libc):          %p\n", (void *)printf);
    printf("local (stack):          %p\n", (void *)&local);
    fflush(stdout);
    FILE *f = fopen("/proc/self/maps", "r");   // kernel-in bu proses üçün xəritəsi
    char line[512];
    while (fgets(line, sizeof line, f)) fputs(line, stdout);
    fclose(f);
    free(heap);
    return 0;
}
```

```
$ gcc where.c -o where
$ ./where | head -1
main (kod, .text):      0x564565a7d239
$ ./where | head -1
main (kod, .text):      0x55b4adfa2239
$ setarch -R ./where | head -1
main (kod, .text):      0x555555555239
$ setarch -R ./where | head -1
main (kod, .text):      0x555555555239
```

ASLR hər dəfə baza ünvanını dəyişir, amma son 3 hex rəqəm (`239`) dəyişmir. Niyə? Baza həmişə səhifə sərhədinə düşür: 4096 = 0x1000.

> 🖼️ **Şəkil 242:** ASLR · baza dəyişir, offset qalır

Burada nəyə bax: üç işə salmada da son üç rəqəm `239`-dur. Təsadüfi olan yalnız səhifəyə düzlənmiş bazadır.

Kotlin-də ünvanları çap etmək olmur: JVM obyektlərin ünvanını gizlədir. Amma xəritəni oxumaq olar:

```kotlin
// Kotlin proqramı öz yaddaş xəritəsinə baxır (C-dəki where.c-nin qardaşı)
import java.io.File

fun main() {
    val lines = File("/proc/self/maps").readLines()          // kernel-in bu proses üçün
xəritəsi
    println("xəritədə sətir: ${lines.size}")
    val path = { l: String -> l.split(Regex("\\s+")).getOrElse(5) { "" } }
    val perms = { l: String -> l.split(" ")[1] }
    for (name in listOf("bin/java", "libjvm.so", "[heap]", "[stack]")) {
        val hits = lines.filter { path(it).endsWith(name) }
        println("$name: ${hits.size} sətir, icazələr ${hits.map(perms).distinct()}")
    }
}
```

```
$ kotlinc Maps.kt -include-runtime -d maps.jar
$ java -jar maps.jar
xəritədə sətir: 203
bin/java: 5 sətir, icazələr [r--p, r-xp, rw-p]
libjvm.so: 5 sətir, icazələr [r--p, r-xp, rw-p]
[heap]: 1 sətir, icazələr [rw-p]
[stack]: 1 sətir, icazələr [rw-p]
```

- `java` və `libjvm.so` da ELF-dir və eyni icazə nümunəsi ilə yüklənib: `r--`, `r-x`, `rw-`.
- Sətir sayı hər dəfə dəyişir (189, 203...): JVM öz thread-lərini və yaddaş sahələrini işləyərkən yaradır.
- Kotlin obyektləri `[heap]`-də deyil: JVM öz heap-ini ayrıca, adı olmayan sahələrdə saxlayır (Faza 2 və Faza 7).

Təcrübə 5. JVM işdə

```
$ java -Xlog:class+load -jar hello.jar | grep -c 'source:'
590
$ java -Xlog:class+load -jar hello.jar | grep -c 'shared objects file'
589
$ java -Xlog:class+load -jar hello.jar | grep -c ' kotlin\.'
0
```

CDS-dən gəlməyən yeganə sinif `HelloKt`-dur: o, `hello.jar`-dan oxundu.

> 🖼️ **Şəkil 243:** 590 sinif · haradan gəlir

Burada nəyə bax: “salam” çap etmək üçün 590 sinif lazımdır, amma demək olar hamısı hazır arxivdən gəlir. Sənin kodundan yalnız bir sinif oxunur.

Başlanğıc vaxtını ölç (20 dəfə işə sal, ortanı götür):

```
$ s=$(date +%s%N); for i in $(seq 20); do java -jar hello.jar >/dev/null; done; e=$(date +%s%N);
echo $(( (e-s)/20000000 )) ms
```

Variant	1-ci ölçmə	2-ci ölçmə
`java -jar hello.jar`	54 ms	59 ms
`java -Xshare:off -jar hello.jar`	103 ms	89 ms
`./hello` (200 dəfə)	≈1,0–1,2 ms	≈1,3 ms

Rəqəmlər səs-küylüdür, amma nəticə dəyişmir: CDS başlanğıcı xeyli qısaldır, JVM-in doğuluşu C-dən onlarla dəfə uzundur. C-dəki 1 ms-in çoxu `fork` və `execve`-nin özüdür: `/bin/true` da təxminən o qədər çəkir.

> 🖼️ **Şəkil 244:** Başlanğıc vaxtı · C, JVM, CDS

Burada nəyə bax: şkala adi, xətti şkaladır. C-nin zolağı demək olar görünmür — JVM-in işi məhz o boyda fərqdir.

JNI: Kotlin-dən C-yə körpü. Bütün fəsil burada birləşir: C kodu `.so` ELF faylına çevrilir, JVM onu `dlopen` ilə yükləyir və funksiyanı adla tapır.

```c
// native.c: Kotlin-dən çağırılan C funksiyası (JNI)
// Adın quruluşu: Java_<paket>_<sinif>_<metod>
#include <jni.h>

JNIEXPORT jint JNICALL
Java_demo_Native_square(JNIEnv *env, jclass cls, jint x) {
    return x * x;
}
```

```kotlin
// Native.kt: C-də yazılmış funksiyanı .so faylından yükləyib çağırır
package demo

object Native {
    init {
        System.loadLibrary("square")     // libsquare.so faylını axtarır
    }
    @JvmStatic external fun square(x: Int): Int
}

fun main() {
    println(Native.square(7))
    // Bu proses hansı .so fayllarını yükləyib?
    java.io.File("/proc/self/maps").readLines()
        .filter { it.endsWith(".so") }
        .map { it.substringAfterLast(' ') }
        .distinct()
        .filter { "square" in it || "libjvm" in it || "libc.so" in it }
        .forEach { println(it) }
}
```

```
$ J=/usr/lib/jvm/java-21-openjdk-amd64
$ gcc -O2 -shared -fPIC -I$J/include -I$J/include/linux native.c -o libsquare.so
$ nm -D libsquare.so | grep Java
0000000000001100 T Java_demo_Native_square
$ kotlinc Native.kt -include-runtime -d native.jar
$ java -Djava.library.path=. -jar native.jar
49
/usr/lib/jvm/java-21-openjdk-amd64/lib/server/libjvm.so
/home/claude/ch03/jni/libsquare.so
```

- `-shared -fPIC` — shared library yığ; kod istənilən ünvanda işləsin (PIE-nin kitabxana versiyası).
- `nm -D` — dynamic symbol table: başqalarının görə biləcəyi adlar. JVM məhz bu adı axtarır.
- Bir prosesdə həm `libjvm.so`, həm `libsquare.so` var: ikisi də eyni loader qaydaları ilə yüklənib.

> 🖼️ **Şəkil 245:** JNI körpüsü · Kotlin-dən C-yə

Burada nəyə bax: körpünün hər addımı bu fəsildə tanıdığın bir mexanizmdir: kitabxana adı, ELF yüklənməsi, simvol axtarışı.

Variasiya: `-Djava.library.path=.`-ı sil və ya `package demo`-nu `package demo.math` et. Hansı xətaları alırsan? (Tipik səhv və Bug-ı tap B3.2.)

Kod

Nümunə 1 — tam izahlı: iki fayldan bir proqram

C versiyası, üç fayl:

```c
// util.h: başqa fayllara "bu funksiya var" deyən elan (declaration)
#ifndef UTIL_H
#define UTIL_H
int square(int x);
#endif
```

```c
// util.c: funksiyanın özü (definition)
#include "util.h"

static int calls = 0;            // static: yalnız bu faylın içində görünür

int square(int x) {
    calls++;
    return x * x;
}
```

```c
// main.c: square-i çağırır, amma onun kodunu görmür
#include <stdio.h>
#include "util.h"

int main(void) {
    printf("%d\n", square(7));
    return 0;
}
```

```
$ gcc -c main.c util.c && gcc main.o util.o -o app && ./app
49
```

> 🖼️ **Şəkil 246:** Nümunə 1 · üç fayl, üç rol

Burada nəyə bax: `main.c` `util.c`-ni heç görmür. Onları yalnız linker görüşdürür.

Sətir-sətir:

- `int square(int x);` — **declaration** (elan): "belə funksiya var, `int` alır, `int` qaytarır". Kod yoxdur. Compiler `main.c`-ni bu vədə əsasən compile edir.
- `#ifndef UTIL_H … #endif` — **include guard**: header bir faylda iki dəfə daxil edilsə də bir dəfə sayılsın.
- `#include "util.h"` dırnaqla — faylı əvvəl cari qovluqda axtar. `<stdio.h>` bucaqla — sistem qovluqlarında.
- `int square(int x) { … }` — **definition** (tərif): funksiyanın özü. `util.o`-da `T square` olur.
- `static int calls` — `static` qlobal dəyişəni fayla bağlayır. `nm`-də kiçik `b`: başqa fayl onu görə bilməz, eyni adlı başqa `calls` ilə də toqquşmaz.
- `main.c` `square`-in kodunu heç görmür. `main.o`-da `U square` qalır; vədin tutulduğunu linker yoxlayır.

🖼️ **Şəkil 247:** Elan və tərif · vəd və onun yerinə yetirilməsi

Burada nəyə bax: `main.c` yalnız sol kartı görür. Sağ kartı onunla linker birləşdirir.

Kotlin versiyası, iki fayl:

```
// Util.kt: funksiyanın özü
package demo

private var calls = 0            // private top-level: yalnız bu faylda görünür

fun square(x: Int): Int {
    calls++
    return x * x
}
```

```
// Main.kt: square-i çağırır; Kotlin-də header faylı yoxdur
package demo

fun main() {
    println(square(7))
    if (System.getenv("RARE") != null) Rare.hello()   // adətən işləmir
}

object Rare {
    fun hello() = println("nadir yol")
}
```

```
$ kotlinc Util.kt Main.kt -d out && ls out/demo
MainKt.class  Rare.class  UtilKt.class
$ java -cp out demo.MainKt
49
```

- Kotlin-də header yoxdur. `kotlinc` bütün faylları eyni anda oxuyur və elanları özü çıxarır.
- `fun square` heç bir sinfin içində deyil (**top-level**). JVM-də isə hər şey sinifdə olmalıdır: kotlinc `Util.kt` üçün `UtilKt` sinfi yaradır, `square` onun `static` metodu olur.
- `private var calls` — C-dəki `static int calls`-un qarşılığı. JVM-də `UtilKt`-nin `private static` sahəsidir.
- `Rare` ayrıca sinifdir (`Rare.class`) və yalnız lazım olanda yüklənir (Bölmə 8).

🖼️ **Şəkil 248:** İki fayldan proqram · C və Kotlin yanaşı

Burada nəyə bax: C sətrində linker var, Kotlin sətrində yoxdur. Kotlin-də adları JVM sonra, işləyərkən həll edir.

Nümunə 2 — sən tamamla: hər dəyər harada yaşayır?

```
// Nümunə 2: hər dəyişən hansı section-a düşür?
int a = 5;                  // (1)
int b;                      // (2)
static int c = 0;           // (3)
const int d = 7;            // (4)
char *s = "hi";             // (5) s özü və "hi" mətni
```

```
int twice(int x) {          // (6)
    static int calls = 1;   // (7)
    int local = x * 2;      // (8)
    calls++;
    return local;
}
```

🖼️ **Şəkil 249:** Proqram binası · dörd mərtəbə və iki müvəqqəti otaq

Burada nəyə bax: mərtəbələr icazələrinə görə ayrılır. Cədvəli doldurmazdan əvvəl hər dəyər üçün soruş: o, faylda yaşayır, yoxsa yalnız işləyərkən?

Cədvəli doldur, sonra yoxla. Birinci sətir nümunədir:

№	Dəyər	Section	`nm` hərfi
1	`a`	`.data`	`D`
2	`b`	___	___
3	`c`	___	___
4	`d`	___	___
5	`s` və `"hi"`	___	___
6	`twice`	___	___
7	`calls`	___	___
8	`local`	___	___

🖼️ **Şəkil 250:** Qərar ağacı · dəyişən harada yaşayır

Burada nəyə bax: sualların sırası vacibdir. `static` funksiyanın içində olsa belə birinci sualdan keçir. Göstərici ilə onun göstərdiyi mətn ayrı-ayrı dəyərlərdir: ağacı hər birinə ayrıca tətbiq et.

Yoxla: `gcc -c sec.c && nm sec.o`, dəqiq section adı üçün `objdump -t sec.o`. İpucu: böyük hərf — bütün proqrama görünən ad, kiçik hərf — yalnız faylın içində. İki dəyər `nm`-də heç görünməyəcək. Hansılar və niyə?

Kotlin versiyası:

```
// Nümunə 2-nin Kotlin versiyası: hər dəyər JVM-də harada yaşayır?
package demo

var a = 5                   // (1)
var b = 0                   // (2)
const val D = 7             // (4)
val s = "hi"                // (5)
fun twice(x: Int): Int {    // (6)
    val local = x * 2       // (8)
    return local
}
```

Hər biri üçün yaz: `static` sahə (field), getter/setter varmı, yoxsa lokal dəyişən? `D` niyə başqalarından fərqlidir? Yoxla: `kotlinc Sec.kt -d secout && javap -p -c -cp secout demo.SecKt`. Cavablar Cavab açarındadır.

> 🖼️ **Şəkil 251:** Yoxlama açarı · nm hərfləri və JVM evləri

Burada nəyə bax: bu açar cavab deyil, oxuma qaydasıdır. `nm` və `javap` çıxışını görəndə hər hərfi və hər sahəni bu cədvəllə tutuşdur.

Nümunə 3 — tapşırıq: öz proqramının xəritəsi

Yalnız gözlənilən davranış verilir, kodu sən yaz.

> 🖼️ **Şəkil 252:** Nümunə 3 · tapşırığın yolu

Burada nəyə bax: beş addımdan yalnız dördüncüsü fikir tələb edir. Qalanları `where.c`-dən tanıyırsan.

C: `mymap.c`.

1. `main`-in, bir qlobal dəyişənin və bir string sabitinin ünvanını çap etsin.
2. `/proc/self/maps`-dən yalnız öz faylına aid sətirləri çap etsin.
3. Hər ünvanın hansı sətrə düşdüyünü və onun icazəsini yazsın: `main → r-xp`.

Gözlənilən: `main` → `r-x`, sabit → `r--`, qlobal → `rw-`. Sonra `readelf -l -W mymap` ilə müqayisə et: neçə `LOAD` var, `/proc/self/maps`-də neçə sətir var? Fərqi bir cümlə ilə izah et.

> 🖼️ **Şəkil 253:** Ünvan və xəritə · hansı aralığa düşür

Burada nəyə bax: hər xəritə sətri bir aralıqdır. Ünvan hansı aralığa düşürsə, icazəsi də o sətrin icazəsidir.

Kotlin: `MyMap.kt`.

1. `/proc/self/maps`-i oxusun və sətirləri fayl adına görə qruplaşdırsın.
2. Ən çox sətri olan 5 faylı icazələri ilə çap etsin.
3. `.jsa` ilə bitən sətirləri ayrıca saysın.

Gözlənilən: `libjvm.so` siyahıdadır. Adi işə salmada `.jsa` sətirləri var (bu maşında 2: CDS arxivi `classes.jsa`). `java -Xshare:off` ilə onlar 0 olur.

> 🖼️ **Şəkil 254:** Nümunə 3 · Kotlin tapşırığı

Burada nəyə bax: tapşırığın ürəyi bir `groupBy`-dir. Qalan iş sətri sütunlara bölmək və səbətləri saymaqdır.

Tipik səhv: kitabxana adı

❌ Səhv:

```
        System.loadLibrary("libsquare.so")
```

Nə olur:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no libsquare.so in java.library.path:
.
```

> 🖼️ **Şəkil 255:** Tipik səhv · ünvan iki dəfə yazılıb

Burada nəyə bax: sən tam ad yazdın, JVM də öz əlavələrini etdi. Nəticədə adın hər iki ucu iki dəfə yazıldı.

JVM faylı tapmadı. `strace -f -e trace=%file` göstərir ki, o, `./liblibsquare.so.so` adlı fayl axtarıb.

✅ Düzgün:

```
        System.loadLibrary("square")
```

Niyə: JVM adı özü tamamlayır. `System.mapLibraryName("square")` Linux və Android-də `libsquare.so`, Windows-da `square.dll`, macOS-da `libsquare.dylib` verir. gcc-dəki `-lsquare` eyni qaydadır. Tam yol vermək istəsən, başqa metod var: `System.load("/tam/yol/libsquare.so")`.

> 🖼️ **Şəkil 256:** Kitabxana adı · lib + ad + .so

Burada nəyə bax: ortadakı qutu adı hər dəfə “lib” və “.so” ilə əhatə edir. Tam ad versən, iki dəfə əhatə olunur.

> 🖼️ **Şəkil 257:** Kitabxana adı · üç platforma və gcc

Burada nəyə bax: Kotlin kodun dəyişmir, fayl adını platforma özü qurur. Ona görə kodda yalnız `"square"` yazılır.

Senior səviyyəsi

S1. PIE, PLT/GOT və binding: `call puts` əslində necə işləyir

Problem. libc hər prosesdə başqa ünvana düşür (ASLR). `main`-in kodu isə fayldan gəlir və `R E` səhifələrdədir. Hər işə salmada kodu “yamamaq” iki səbəbdən pisdir. Birincisi, kod səhifəsinə yazmaq lazım gələrdi: eyni səhifə həm yazılan, həm icra olunan olardı — bu, hücumçunun arzusudur. İkincisi, eyni faylın səhifələrini yüzlərlə proses bölüşə bilməzdi. Həll: kod sabit qalır, dəyişən ünvanlar ayrıca cədvəldə saxlanılır.

> 🖼️ **Şəkil 258:** PLT və GOT · operator və telefon kitabçası

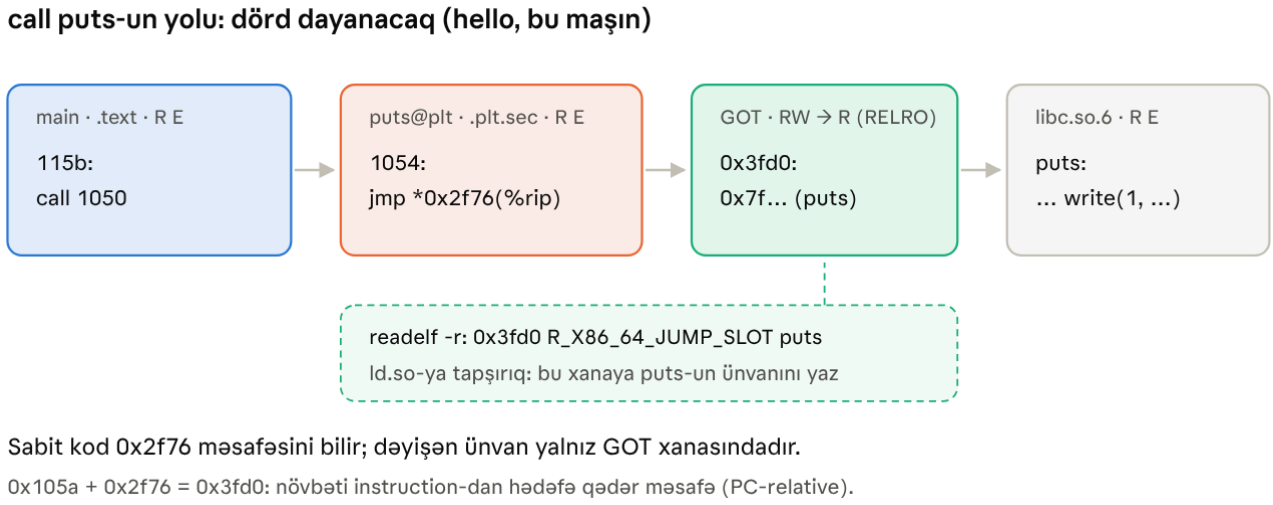
Burada nəyə bax: sənin kodun heç vaxt dəyişmir. Dəyişən yeganə şey sağdakı kitabçadakı nömrədir.

Mexanizm üç hissədir.

1. **PC-relative ünvanlama.** `lea 0x0(%rip),%rax` deməkdir: "növbəti instruction-dan X byte irəlidə". Eyni fayldakı hər şeyə (`"salam"`, öz funksiyaların) belə müraciət olunur. Məsafə fayl harada yüklənsə də dəyişmir, ona görə bu relocation-ları linker build vaxtı bağlayır. **PIE** — bütün kodu belə yazılmış executable.
2. **GOT** (Global Offset Table) — başqa kitabxanalardakı adların ünvan cədvəli. O, `rw-` səhifədədir.
3. **PLT** (Procedure Linkage Table) — hər xarici funksiya üçün kiçik keçid kodu: "GOT-dakı xanaya bax və ora tullan".

```
$ readelf -r -W hello | grep -E 'puts|__libc_start'
0000000000003fd8  0000000100000006 R_X86_64_GLOB_DAT      0000000000000000
__libc_start_main@GLIBC_2.34 + 0
0000000000003fd0  0000000300000007 R_X86_64_JUMP_SLOT     0000000000000000 puts@GLIBC_2.2.5 + 0
$ objdump -d hello | grep -A2 '<puts@plt>:'
0000000000001050 <puts@plt>:
    1050:  f3 0f 1e fa           endbr64
    1054:  ff 25 76 2f 00 00     jmp    *0x2f76(%rip)        # 3fd0 <puts@GLIBC_2.2.5>
```

- `R_X86_64_JUMP_SLOT` — runtime relocation: ld.so-ya "0x3fd0 xanasına `puts`-un ünvanını yaz" tapşırığı.
- `jmp *0x2f76(%rip)` — "0x105a + 0x2f76 = 0x3fd0 ünvanında yazılan ünvana tullan". Ulduz (`*`) dolayı keçid deməkdir.



Burada nəyə bax: dörd dayanacaqdan yalnız üçüncüsü (GOT) işə salmadan-işə salmaya dəyişir.

BIND_NOW və lazy. Ubuntu-da gcc default olaraq `-z now` ilə yığır: `readelf -d hello` → `FLAGS BIND_NOW`, `FLAGS_1 NOW PIE`. ld.so bütün GOT-u `main`-dən əvvəl doldurur (LD_DEBUG-da 87 < 104). Sonra GOT-u `mprotect` ilə yalnız oxunan edir: strace-dəki üç `mprotect(…, PROT_READ)` məhz budur. Bunun adı **full RELRO**-dur: hücumçu GOT xanasını dəyişib `puts` əvəzinə öz kodunu çağıra bilməz.

Lazy variantı gör:

```
$ gcc hello.c -Wl,-z,lazy -Wl,-z,norelro -o hello_lazy
$ objdump -d -j .plt hello_lazy
0000000000001020 <.plt>:
    1020:  ff 35 ea 22 00 00     push   0x22ea(%rip)        # 3310 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:  ff 25 ec 22 00 00     jmp    *0x22ec(%rip)        # 3318 <_GLOBAL_OFFSET_TABLE_+0x10>
    ...
    1034:  68 00 00 00 00        push   $0x0
    1039:  e9 e2 ff ff ff        jmp    1020 <_init+0x20>
```

- Başlanğıcda GOT xanası PLT-nin özünə geri göstərir.
- `push $0x0` — "mən 0 nömrəli relocation-am". Sonra ümumi giriş `.plt[0]` ld.so-nun resolver-inə tullanır.
- Resolver `puts`-u tapır, GOT-a yazır və `puts`-a keçir. İkinci çağırış birbaşa gedir.
- LD_DEBUG: `transferring control` 103-cü sətirdə, `puts`-un bağlanması 105-ci sətirdə — `main` başlayandan sonra.

> 🖼️ **Şəkil 260:** BIND_NOW və lazy · ad nə vaxt həll olunur

Burada nəyə bax: qırmızı qutu — lazy-nin qiyməti. O, başlanğıcdan proqramın ortasına köçür.

	BIND_NOW	**lazy**
Başlanğıc	bütün adlar əvvəlcədən, daha uzun	yalnız işlənən adlar, qısa
İlk çağırış	sabit sürət	resolver gecikməsi
Təhlükəsizlik	GOT yalnız oxunan (full RELRO)	GOT yazıla bilən qalır
Çatmayan simvol	start-da xəta	proqramın ortasında xəta

Android. Bionic-in linker-i lazy binding-i dəstəkləmir: `dlopen(RTLD_LAZY)` da əslində "now" kimi işləyir. NDK clang Android üçün default olaraq `-z now -z relro` işlədir. Yəni hər `.so`-nun bütün simvolları yüklənmə anında həll olunur: böyük native kitabxana = uzun `System.loadLibrary`. Başlanğıcda onu ölç: `System.loadLibrary` ətrafında vaxt və ya Perfetto trace (Faza 8).

S2. Kernel və ld.so-nun içi: binfmt, x icazəsi, interposition

execve kernel-də. `fs/exec.c` faylı açır, ilk byte-larını oxuyur və qeydiyyatlı formatları bir-bir sınayır:

- ELF → `fs/binfmt_elf.c`, funksiya `load_elf_binary()`.
- `#!` ilə başlayan skript → `binfmt_script`: kernel o sətrdəki interpreter-i (məsələn `/bin/sh`) işə salır.
- `binfmt_misc` — admin-in qeyd etdiyi maskalar, məsələn başqa arxitektura binarı üçün emulator.

> 🖼️ **Şəkil 261:** execve · kernel formatı ilk byte-lardan seçir

Burada nəyə bax: kernel faylın adına və genişlənməsinə baxmır, yalnız ilk byte-larına baxır.

`load_elf_binary()` Bölmə 7-dəki işi görür: program header-ləri oxuyur, `PT_INTERP`-i tapıb ld.so-nu da yükləyir, `PT_LOAD`-ları map edir, heap-in başlanğıcını (`brk`) qoyur, stack-ə `argv`, `envp` və auxv yazır. Sonra icranı ld.so-nun entry-sinə verir.

x icazəsi kimin işidir? `chmod -x` edilmiş `hello_nox` ilə yoxla:

```
$ ls -l hello_nox | cut -c1-10
-rw-r--r--
$ ./hello_nox; echo "rc=$?"
/bin/bash: line 1: ./hello_nox: Permission denied
rc=126
$ /lib64/ld-linux-x86-64.so.2 ./hello_nox
salam
```

İkinci halda `execve` ld.so-nun özünə edilir, onun isə `x`-i var. ld.so `hello_nox`-u adi fayl kimi `open` + `mmap` edir. Nəticə: `x` biti təhlükəsizlik sərhədi deyil, sadəcə execve-nin qapısıdır. Sərhəd lazımdırsa, `noexec` mount və ya SELinux işlədilir. Android 10-dan API 29+ hədəfləyən tətbiq öz ev qovluğundakı fayllara `execve` edə bilmir: qayda məhz bu səviyyədə qoyulub.

> 🖼️ **Şəkil 262:** x icazəsi · yalnız execve-nin qapısı

Burada nəyə bax: eyni fayl, eyni icazələr. Fərq yalnız `execve`-yə hansı faylın verildiyindədir.

Interposition. ld.so adı axtaranda müəyyən sıra ilə gedir: əvvəl `LD_PRELOAD`-dakı kitabxanalar, sonra executable, sonra `NEEDED` kitabxanaları. Birinci tapılan qalib gəlir. Yoxla:

```c
// myputs.c: libc-dəki puts-un yerinə keçən funksiya (interposition)
#define _GNU_SOURCE
#include <dlfcn.h>

int puts(const char *s) {
    // əsl puts-u libc-dən tap və ona başqa mətn ver
    int (*real_puts)(const char *) = dlsym(RTLD_NEXT, "puts");
    real_puts("[LD_PRELOAD] puts tutuldu:");
    return real_puts(s);
}
```

```
$ gcc -O2 -shared -fPIC myputs.c -o libmyputs.so
$ LD_PRELOAD=./libmyputs.so ./hello
[LD_PRELOAD] puts tutuldu:
salam
$ LD_PRELOAD=./libmyputs.so ./hello_static
salam
```

- `RTLD_NEXT` — "bu adı məndən sonrakı kitabxanalarda axtar". Beləcə əsl `puts`-u itirmirik.
- Static proqrama təsir etmədi: orada ld.so yoxdur, `puts` artıq faylın içindədir.
- Real istifadə: `malloc` əvəzləri (jemalloc, tcmalloc), profiler-lər, testdə sistem funksiyalarını saxtalaşdırmaq. setuid proqramlarda ld.so `LD_PRELOAD`-u ciddi məhdudlaşdırır.

> 🖼️ **Şəkil 263:** Interposition · LD_PRELOAD birinci gəlir

Burada nəyə bax: `hello` dəyişmədi, `libc` də dəyişmədi. Yalnız axtarış sırasına bir kitabxana əlavə olundu.

JVM-də eyni fikir: classpath sırası. JVM sinfi adla axtaranda classpath-ı soldan sağa gəzir və birinci tapılan qalib gəlir. Eyni paket və adla ikinci sinif yazsan, `LD_PRELOAD` kimi əsl sinfi əvəz edə bilərsən:

```
// lib/Greeter.kt: "kitabxana" — əsl sinif
package demo
object Greeter { fun greet(name: String) = "salam, $name" }
```

```
// fake/Greeter.kt: eyni paket, eyni ad — LD_PRELOAD-un JVM qarşılığı
package demo
object Greeter { fun greet(name: String) = "[saxta Greeter] salam, $name" }
```

```
// App.kt: Greeter-i adla çağırır; hansı sinfin gələcəyini classpath sırası həll edir
package demo
fun main() {
    println(Greeter.greet("Ali"))
    println("yükləndi: " + Greeter::class.java.protectionDomain.codeSource.location)
}
```

```
$ kotlinc lib/Greeter.kt -d lib.jar
$ kotlinc fake/Greeter.kt -d fake.jar
$ kotlinc App.kt -cp lib.jar -include-runtime -d app.jar
$ java -cp app.jar:lib.jar demo.AppKt
salam, Ali
yükləndi: file:/home/claude/ch03/shadow/lib.jar
$ java -cp fake.jar:app.jar:lib.jar demo.AppKt
[saxta Greeter] salam, Ali
yükləndi: file:/home/claude/ch03/shadow/fake.jar
```

- `app.jar` dəyişmədi, `lib.jar` da dəyişmədi. Yalnız axtarış sırasına bir arxiv əlavə olundu — `LD_PRELOAD`-dakı kimi.
- Fərq: `LD_PRELOAD` bir funksiyanı əvəz edir və `RTLD_NEXT` ilə əslinə müraciət edə bilir. JVM-də isə bütöv sinif əvəz olunur, əsl sinifə adı ilə çatmaq olmur.
- Android-də iki asılılıqda eyni sinif olsa, AGP build-i `checkDuplicateClasses` addımında dayandırır: DEX-ə hər sinfin bir nüsxəsi düşür.

Simvol görünməsi (visibility). `.so`-dakı hər qlobal funksiya default olaraq dynamic symbol table-a düşür. Bu üç şey deməkdir: o ixrac olunur, başqası onu əvəz edə bilər, ld.so-nun axtarışı uzanır. `-fvisibility=hidden` hamısını gizlədir, `__attribute__((visibility("default")))` yalnız lazım olanları açır. JNI kitabxanasında ixrac olunmalı olanlar `JNI_OnLoad` və `Java_…` adlarıdır. Daha yaxşı yol: `JNI_OnLoad`-da `RegisterNatives` ilə funksiyaları göstəricilərlə qeyd etmək. Onda `Java_…` adları da gizli qala bilər, paket adı dəyişəndə isə (B3.2) xəta çağırışda yox, yüklənmə anında çıxır.

S3. JVM və Android: linking proqram işləyərkən

JVM spesifikasiyası (fəsil 5) sinfin həyatını belə bölür: **loading** → **linking** (verification, preparation, resolution) → **initialization**.

> 🖼️ **Şəkil 264:** JVM-də sinfin həyatı · loading, linking, initialization

Burada nəyə bax: C-də üç ayrı alətin (linker, ld.so, libc başlanğıcı) işini burada bir JVM görür.

- **Resolution lazy ola bilər.** Spesifikasiya buna icazə verir, HotSpot da belə edir. Simvolik istinad (`demo/UtilKt.square:(I)I`) instruction ilk icra olunanda həll olunur və nəticə constant pool cache-də saxlanır. Sonrakı çağırışlar birbaşa gedir. Bu, JVM-in lazy binding-idir — PLT/GOT-un

qardaşı.
- **Initialization** `<clinit>`-i işə salır, amma yalnız ilk aktiv istifadədə: `new`, `getstatic`, `putstatic`, `invokestatic`. `Rare` obyektinin `<clinit>`-i yalnız `Rare.hello()` çağırılanda işlədi.
- **Verification** bahalıdır: hər metodun bytecode-u tip təhlükəsizliyinə görə yoxlanılır. CDS arxivindəki siniflər artıq yoxlanılmış gəlir — 54 ms ilə 103 ms arasındakı fərqin bir hissəsi buradandır.

Əvvəlcədən hazırla, sonra map et.

- JDK 12-dən default CDS arxivi gəlir (JEP 341): `lib/server/classes.jsa`. O, prosesin xəritəsində görünür: bu maşında 2 sətir, `-Xshare:off` ilə 0.
- **AppCDS** öz siniflərini də arxivə qoyur (dynamic arxiv, JDK 13, JEP 350):

```
$ java -XX:ArchiveClassesAtExit=app.jsa -jar hello.jar
salam
$ java -XX:SharedArchiveFile=app.jsa -Xlog:class+load -jar hello.jar | grep HelloKt
[0.050s][info][class,load] HelloKt source: shared objects file (top)
```

`app.jsa` 126 976 byte-dır. 20 işə salmanın ortası: 60 → 56 ms. Bir sinifli proqramda qazanc kiçikdir; yüzlərlə sinifli server tətbiqində böyük olur. JDK 24-dən (JEP 483) siniflər arxivdə artıq yüklənmiş və link olunmuş halda saxlanıla bilir.

- **Android-də eyni fikir.** Zygote framework siniflərini bir dəfə yükləyir. Hər tətbiq prosesi ondan `fork` ilə yaranır və bu yaddaşı bölüşür. Framework kodu əvvəlcədən AOT kompilyasiya olunmuş **boot image**-dədir. Tətbiqin öz kodu üçün **Baseline Profile** var: başlanğıcda lazım olan sinif və metodların siyahısını APK ilə göndərirsən, dex2oat onları əvvəlcədən kompilyasiya edir.

> 🖼️ **Şəkil 265:** Əvvəlcədən hazırla · CDS, AppCDS, zygote

Burada nəyə bax: üç kart, bir fikir. İşi bir dəfə gör, nəticəni səhifə kimi yaddaşa map et.

Android native: APK-dan birbaşa yükləmə və 16 KB.

- Android 6-dan (API 23) linker64 `.so`-nu APK-nın içindən birbaşa map edə bilir. Şərt: fayl sıxılmamış və səhifə sərhədinə düzülmüş olmalıdır. Müasir AGP minSdk 23+ olanda bunu default edir (`useLegacyPackaging = false`).
- Android 15-dən 16 KB səhifəli cihazlar var. Google Play 1 noyabr 2025-dən Android 15+ hədəfləyən yeni tətbiq və yeniləmələrdən 16 KB dəstəyi tələb edir. Texniki mənası: `.so`-nun `LOAD` segmentləri 16 KB-a düzülməlidir.

```
$ ld.lld -shared -z max-page-size=4096 square.o -o libsquare4k.so
$ readelf -l -W libsquare4k.so | grep LOAD      # Align: 0x1000 0x1000 0x1000
$ ld.lld -shared -z max-page-size=16384 square.o -o libsquare16.so
$ readelf -l -W libsquare16.so | grep LOAD     # Align: 0x4000 0x4000 0x4000
```

Əvvəlki `libsquare.so` (lld default) 0x10000 = 64 KB-a düzülmüşdü: o da uyğundur. Yoxlamaq üçün: APK Analyzer, Lint və ya Google-un `check_elf_alignment.sh` skripti.

> 🖼️ **Şəkil 266:** 16 KB səhifə · segment align

Burada nəyə bax: 4 KB-a düzülmüş üç segment bir 16 KB səhifəyə sıxılır. Hər segmentin icazəsi ayrı olmalıdır, səhifənin isə yalnız bir icazəsi ola bilər.

S4. Tarix: niyə məhz ELF?

- **1971.** İlk Unix-də `a.out` ("assembler output") formatı: header, text, data, simvollar; `.bss` yalnız ölçü kimi. Ad bu gün də yaşayır: `gcc hello.c` çıxışı `a.out`-dur, amma `file a.out` → ELF, ilk byte-lar `7f 45 4c 46`.
- **a.out-un problemi:** bölmələr sabit idi. Shared library üçün hər kitabxanaya əvvəlcədən sabit ünvan ayırmaq lazım gəlirdi. Debug məlumatı və yeni arxitekturalar üçün yer yox idi.

> 🖼️ **Şəkil 267:** a.out və ELF · sabit otaqlar və çevik plan

Burada nəyə bax: ELF-də yeni ehtiyac yeni otaq deməkdir; formatı dəyişmək lazım deyil.

- **ELF-i** AT&T-nin Unix System Laboratories (USL) bölməsi System V Release 4 (SVR4, 1980-lərin sonu) üçün hazırladı: istənilən say section, ayrıca segment planı, dynamic linking üçün `.dynamic` və `INTERP`. Sonra spesifikasiya platformalar arası standart oldu.
- **Linux** ELF dəstəyini 1993-də (0.99.13 kernel-i) aldı, 1995-ə qədər ELF-ə keçdi. a.out dəstəyi Linux 5.1-də (2019) köhnəlmiş elan olundu, 5.18-də Alpha və m68k üçün yığılmadı, 5.19-da x86 kodu silindi, 6.1-də (2022) isə a.out kodunun hamısı silindi.
- **Başqa dünyalar:** Windows — PE/COFF, Apple — Mach-O. Android Linux kernel-i üzərində olduğu üçün ELF-dir.

> 🖼️ **Şəkil 268:** Format ailələri · ELF, PE, Mach-O, class, DEX

Burada nəyə bax: magic number sütunu. Hər format faylın ilk byte-larında özünü tanıdır.

- **C-nin özündə də tarix dəyişir.** GCC 10 (2020) default olaraq `-fno-common` ilə gəlir. Əvvəllər header-dəki `int retries;` kimi ilkin dəyərsiz qlobal hər faylda "common" simvol olurdu və linker onları birləşdirirdi. İndi bu, `multiple definition` xətasıdır (Bug-ı tap, B3.1).

> 🖼️ **Şəkil 269:** Tarix · 1971-dən 2025-ə

Burada nəyə bax: ELF 35 ildən çoxdur ki, yaşayır və hələ də dəyişir: 2025-in yeniliyi — segment align.

S5. Debug məlumatı, strip, Build ID və SONAME: faylın gizli kimliyi

Problem. Release build istifadəçidə çökür. Əlində yalnız ünvan var. Hansı funksiya, hansı sətir? Ünvanı ada çevirmək üçün faylın içində əlavə məlumat lazımdır. Bu bölmə həm də çıxışlarda gördüyün iki sirri açır: `libc.so.6`-dakı "6" və `puts@GLIBC_2.2.5`-dəki versiya.

> 🖼️ **Şəkil 270:** Debug məlumatı · iki xəritə

Burada nəyə bax: proqramın kodu dəyişməyib. `strip` yalnız ünvanı ada çevirən cədvəlləri silib.

Debug məlumatı (DWARF). `gcc -g` fayla `.debug_*` section-ları əlavə edir: hansı ünvan hansı sətrə aiddir, hansı dəyişən haradadır. Bu formata **DWARF** deyilir. `strip` həm onları, həm də simvol cədvəlini (`.symtab`) silir.

```c
// crash.c: qəsdən çökən proqram — debug məlumatı olmadan və olanda
#include <stdio.h>

int parse(const char *s) {
    return *s - '0';           // s == NULL olanda çökür
}

int main(void) {
    const char *input = NULL;  // ❌ bug
    printf("%d\n", parse(input));
    return 0;
}
```

```
$ gcc -O0 -g crash.c -o crash_g
$ cp crash_g crash_stripped && strip crash_stripped
$ stat -c '%s %n' crash_g crash_stripped
17336 crash_g
14472 crash_stripped
$ readelf -S -W crash_g | grep -oE '\.debug_[a-z_]+' | tr '\n' ' '
.debug_aranges .debug_info .debug_abbrev .debug_line .debug_str .debug_line_str
$ nm crash_stripped
nm: crash_stripped: no symbols
$ gdb -q -batch -ex run -ex bt ./crash_g
#0  0x0000555555555159 in parse (s=0x0) at crash.c:5
#1  0x0000555555555184 in main () at crash.c:10
$ gdb -q -batch -ex run -ex bt ./crash_stripped
#0  0x0000555555555159 in ?? ()
#1  0x0000555555555184 in ?? ()
```

Build ID: ünvanı geri ada çevirmək. Linker fayla **Build ID** yazır: faylın məzmunundan hesablanan hash. `strip` onu silmir. Ona görə strip olunmuş faylın crash ünvanını eyni Build ID-li debug faylında axtarmaq olar:

```
$ readelf -n crash_g | grep 'Build ID'
    Build ID: 8181368a672822071237bb9baa5e78bba404c1cb
$ readelf -n crash_stripped | grep 'Build ID'
    Build ID: 8181368a672822071237bb9baa5e78bba404c1cb
$ addr2line -f -e crash_g 0x1159 0x1184
parse
/home/claude/ch03/debug/crash.c:5
main
/home/claude/ch03/debug/crash.c:10 (discriminator 1)
```

- `0x1159` haradan gəldi? gdb ASLR-i söndürür, baza `0x555555554000` olur: `0x555555555159 - 0x555555554000 = 0x1159`. Crash ünvanından bazanı çıxırmaq — simvolların ilk addımıdır.
- Android-də APK-ya strip olunmuş `.so` gedir. Native crash-i oxumaq üçün strip olunmamış fayl lazımdır: onu Play Console-a native debug symbol kimi yükləyirsən, ya da tombstone-u `ndk-stack` və `llvm-symbolizer` ilə özün açırsan. Uyğunluq Build ID ilə yoxlanır.

> 🖼️ **Şəkil 271:** Build ID · crash-dən sətrə

Burada nəyə bax: debug faylı heç vaxt cihaza getmir. İki dünyanı yalnız Build ID birləşdirir.

Kotlin-də eyni sual: stack trace-dəki sətir nömrəsi haradan gəlir?

```
        // Crash.kt: crash.c-nin Kotlin qarşılığı — stack trace-dəki sətir nömrəsi haradan gəlir?
        fun parse(s: String?): Int = s!![0] - '0'     // s == null olanda NullPointerException

        fun main() {
            val input: String? = null                 // ❌ bug
            println(parse(input))
        }
```

```
$ kotlinc Crash.kt -include-runtime -d crash.jar && java -jar crash.jar
Exception in thread "main" java.lang.NullPointerException
    at CrashKt.parse(Crash.kt:2)
    at CrashKt.main(Crash.kt:6)
    at CrashKt.main(Crash.kt)
$ javap -l -cp out CrashKt | sed -n '/void main()/,/LocalVariableTable/p'
  public static final void main();
    LineNumberTable:
      line 5: 0
      line 6: 2
      line 7: 14
```

- `LineNumberTable` class faylının "debug section"-ıdır: bytecode offset → sətir. DWARF-ın `.debug_line`-ı ilə eyni işi görür.
- Release-də R8 sinif və metod adlarını qısaldır. Stack trace-i geri açmaq üçün həmin build-in `mapping.txt` faylı və `retrace` lazımdır. Bu, native-dəki debug faylının JVM qarşılığıdır: o da cihaza getmir, səndə qalır.

SONAME və simvol versiyaları.

```
$ ldd hello
    linux-vdso.so.1 (0x00007f92ea33e000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f92ea000000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f92ea340000)
$ readelf -d /lib/x86_64-linux-gnu/libc.so.6 | grep SONAME
 0x000000000000000e (SONAME)             Library soname: [libc.so.6]
$ tail -1 /usr/lib/x86_64-linux-gnu/libc.so
GROUP ( /lib/x86_64-linux-gnu/libc.so.6 /usr/lib/x86_64-linux-gnu/libc_nonshared.a  AS_NEEDED (
/lib64/ld-linux-x86-64.so.2 ) )
$ objdump -T /lib/x86_64-linux-gnu/libc.so.6 | grep -E ' (puts|memcpy)$'
0000000000087cc0  w   DF .text  0000000000000226  GLIBC_2.2.5 puts
00000000000ba930 g    DF .text  000000000000002c (GLIBC_2.2.5) memcpy
00000000000b1800 g   iD  .text  0000000000000111  GLIBC_2.14  memcpy
$ readelf -V hello | grep 'Name: GLIBC'
  0x0010:   Name: GLIBC_2.2.5  Flags: none  Version: 3
  0x0020:   Name: GLIBC_2.34  Flags: none  Version: 2
```

- **SONAME** — kitabxananın "ABI adı": `libc.so.6`. "6" — uyğun olmayan dəyişikliklər ardınca artırılan nömrədir. Linker `NEEDED`-ə faylın yolunu yox, SONAME-i yazır.
- `libc.so` (nömrəsiz) yalnız build vaxtı üçündür: burada o, hətta ELF deyil, linker skriptidir və `-lc`-ni `libc.so.6`-ya yönləndirir.

- **Simvol versiyası**: `puts@GLIBC_2.2.5` — bu interfeys glibc 2.2.5-dən belədir. `memcpy`-nin iki versiyası var: köhnə proqramlar köhnə davranışı alır, yeni proqramlar yenisini.
- `hello` `GLIBC_2.34` tələb edir (`__libc_start_main`-in yeni versiyası). Daha köhnə glibc-li sistemdə ld.so onu "version `GLIBC_2.34' not found" xətası ilə rədd edər — burada işlədilməyib, köhnə sistemdə yoxla.
- Android-də eyni problem `minSdkVersion` ilə həll olunur: NDK hər API səviyyəsi üçün ayrıca sysroot verir. minSdk-dan yeni olan libc funksiyasını işlətsən, köhnə cihazda `dlopen` "cannot locate symbol" xətası verir.

Kotlin-də versiya class faylının özündədir:

```
        // Ver.kt: class faylının versiyası — JVM-in "simvol versiyası"
        fun main() {
            val bytes = ClassLoader.getSystemResourceAsStream("VerKt.class")!!.readBytes()
            val major = ((bytes[6].toInt() and 0xFF) shl 8) or (bytes[7].toInt() and 0xFF)
            println("VerKt.class major = $major")                        // 52 = Java 8, 65 = Java
21
            println("JVM: ${System.getProperty("java.version")}, " +
                    "class version ${System.getProperty("java.class.version")}")
        }
```

```
$ kotlinc Ver.kt -include-runtime -d ver.jar && java -jar ver.jar
VerKt.class major = 52
JVM: 21.0.13.0.3, class version 65.0
$ kotlinc Ver.kt -jvm-target 21 -include-runtime -d ver21.jar && java -jar ver21.jar
VerKt.class major = 65
JVM: 21.0.13.0.3, class version 65.0
```

Bu JVM daha yeni class faylını qəbul etmir. Yoxlamaq üçün `VerKt.class`-ın 6–7-ci byte-ını əl ilə 69-a (Java 25) qaldırdıq:

```
$ java -cp out:ver.jar VerKt
Error: LinkageError occurred while loading main class VerKt
    java.lang.UnsupportedClassVersionError: VerKt has been compiled by a more recent version of
the Java Runtime (class file version 69.0), this version of the Java Runtime only recognizes
class file versions up to 65.0
```

> 🖼️ **Şəkil 272:** Versiyalı adlar · ELF və JVM

Burada nəyə bax: hər iki dünyada qayda eynidir. Yeni alətlə yığılan fayl köhnə runtime-da işləməyə bilər, ona görə hədəf versiyasını (glibc, minSdk, jvmTarget) şüurlu seç.

S6. Thread-lərə körpü: TLS, `<clinit>` kilidi və loader kilidi

Bu fəsil thread-lərdən danışmırdı, amma üç yerdə onlarla rastlaşdıq: strace-dəki `arch_prctl` və `set_tid_address` (D3.4), `<clinit>` (Bölmə 7) və `dlopen` (JNI). Bütün plan thread-lər haqqında olduğu üçün bu üç yeri indi açırq. Tam mexanizm sonra gələcək: thread — Faza 3, mutex və futex — Faza 4, deadlock — Faza 5.

1. TLS: hər thread-in öz "qlobal" dəyişəni. `__thread` ilə işarələnən dəyişən **TLS**-də (thread-local storage) yaşayır: hər thread onun ayrıca nüsxəsini görür.

```c
// tls.c: hər thread-in öz "qlobal" dəyişəni (thread-local storage)
#include <pthread.h>
#include <stdio.h>

int shared = 0;                 // adi qlobal: .bss, bütün thread-lər üçün bir
__thread int mine = 0;          // TLS: .tbss, hər thread üçün ayrıca nüsxə

void *work(void *name) {
    for (int i = 0; i < 100000; i++) { shared++; mine++; }
    printf("%s: &mine = %p, mine = %d\n", (char *)name, (void *)&mine, mine);
    return NULL;
}

int main(void) {
    pthread_t a, b;
    pthread_create(&a, NULL, work, "A");
    pthread_create(&b, NULL, work, "B");
    pthread_join(a, NULL); pthread_join(b, NULL);
    printf("main: &mine = %p, mine = %d, shared = %d\n", (void *)&mine, mine, shared);
    return 0;
}
```

```
$ gcc -O0 -pthread tls.c -o tls && ./tls
A: &mine = 0x7f9a3edff6bc, mine = 100000
B: &mine = 0x7f9a3e5fe6bc, mine = 100000
main: &mine = 0x7f9a3f1e073c, mine = 0, shared = 114450
$ readelf -S -W tls | grep -E '\.tbss|\.bss '      (boşluqlar qısaltılıb)
 [21] .tbss NOBITS 0000000000003da0 002da0 000004 00 WAT 0 0 4
 [27] .bss NOBITS 0000000000004010 003010 000008 00 WA 0 0 4
$ readelf -l -W tls | grep TLS
 TLS 0x002da0 0x0000000000003da0 0x0000000000003da0 0x000000 0x000004 R 0x4
```

- `&mine` üç thread-də üç fərqli ünvandır. `main`-in nüsxəsinə heç kim toxunmayıb: 0.
- `.tbss` — `.bss`-in TLS versiyasıdır, bayrağında `T` var. ELF-də ayrıca `TLS` program header-i olur: loader hər yeni thread üçün bu "şablon"dan nüsxə açır.
- x86-64-də thread öz TLS sahəsinə `%fs` registri ilə çatır. D3.4-dəki `arch_prctl` libc-nin məhz bu registri qurmasıdır. ARM64-də bu işi `TPIDR_EL0` registri görür.
- `shared` 200 000 olmalı idi, 114 450 çıxdı. Bu, race condition-dur: iki thread eyni dəyəri oxuyub yazır (Faza 4).

Kotlin-də eyni təcrübə `ThreadLocal` ilə:

```kotlin
// Tls.kt: tls.c-nin Kotlin qarşılığı — ThreadLocal hər thread-ə öz nüsxəsini verir
val mine = ThreadLocal.withInitial { 0 }   // hər thread üçün ayrıca dəyər
var shared = 0                             // bütün thread-lər üçün bir sahə

fun work(name: String) {
    repeat(100_000) { shared++; mine.set(mine.get() + 1) }
    println("$name: mine = ${mine.get()}")
}

fun main() {
    val a = Thread { work("A") }; val b = Thread { work("B") }
    a.start(); b.start(); a.join(); b.join()
    println("main: mine = ${mine.get()}, shared = $shared")   // shared 200 000-dən az
ola bilər
}
```

```
$ kotlinc Tls.kt -include-runtime -d tls.jar && java -jar tls.jar
B: mine = 100000
A: mine = 100000
main: mine = 0, shared = 199433
```

- `ThreadLocal` JVM-in öz TLS-idir: dəyər thread obyektinin içindəki cədvəldə saxlanır, ELF-in `.tbss`-i yoxdur.
- Kotlin-də də `shared` itki verdi. Android-də `Looper.myLooper()` məhz `ThreadLocal` üzərində qurulub (Faza 8).

> 🖼️ **Şəkil 273:** TLS · öz cib və ortaq qutu

Burada nəyə bax: TLS dəyərləri düzüdür, ortaq dəyər isə sınıb. Fərq yalnız kimin toxuna bilməsindədir.

2. `<clinit>` kilidi. JVM hər sinfi yalnız bir dəfə initialize edir. Bunun üçün hər sinfin **initialization lock**-u var (JLS §12.4.2): `<clinit>`-i bir thread işlədin, qalanları gözləyir. İki sinif bir-birinə istinad edib iki thread-dən eyni anda başlayanda nə olur?

```
        // InitDeadlock.kt: iki thread, iki object, bir-birini gözləyən <clinit>
        object A {
            init {
                Thread.sleep(100)                 // B-nin də başlamasına vaxt ver
                println("A init: B.value = ${B.value}")
            }
            val value = 1
        }

        object B {
            init {
                Thread.sleep(100)
                println("B init: A.value = ${A.value}")
            }
            val value = 2
        }

        fun main() {
            val t1 = Thread({ println(A.value) }, "t1")    // A-nın <clinit>-ini başladır
            val t2 = Thread({ println(B.value) }, "t2")    // B-nin <clinit>-ini başladır
            t1.start(); t2.start()
            t1.join(2000); t2.join(2000)
            println("2 saniyə keçdi. t1: ${t1.state}, t2: ${t2.state}")
            Runtime.getRuntime().halt(0)                   // asılı thread-ləri gözləmədən çıx
        }
```

```
$ kotlinc InitDeadlock.kt -include-runtime -d dl.jar && java -jar dl.jar
2 saniyə keçdi. t1: RUNNABLE, t2: RUNNABLE
$ jstack <pid>      (qısaltılıb)
"t1" #18 [1022] prio=5 in Object.wait()
   java.lang.Thread.State: RUNNABLE
    at A.<clinit>(InitDeadlock.kt:5)
    - waiting on the Class initialization monitor for B
"t2" #19 [1023] prio=5 in Object.wait()
   java.lang.Thread.State: RUNNABLE
    at B.<clinit>(InitDeadlock.kt:13)
    - waiting on the Class initialization monitor for A
$ jstack <pid> | grep -c 'Found one Java-level deadlock'
0
```

- Heç bir `init` sətri çap olunmadı: iki thread əbədi gözləyir.
- Thread-lərin vəziyyəti `RUNNABLE` görünür, `BLOCKED` yox. jstack-in deadlock axtarışı isə bu kilidi tanımır. Bu deadlock-u yalnız "Class initialization monitor" sətri ilə tanıyırsan.
- Android-də ART eyni qaydanı işlədir. Başlanğıcda iki thread iki böyük `object`-i eyni anda ilk dəfə işlədəndə bu, ANR kimi görünə bilər (Faza 8).

> 🖼️ **Şəkil 274:** <clinit> deadlock-u · iki thread, iki kilid

Burada nəyə bax: dövrə qapanır. Hər thread bir kilidi tutur və digərini gözləyir — bu, deadlock-un klassik şəklidir (Faza 5).

3. Loader kilidi. glibc-də `dlopen` qlobal bir kilid tutur və yüklənən kitabxananın constructor-larını o kilidin altında çağırır. Constructor başqa thread-i gözləsə, o thread isə özü `dlopen` çağırsa, nə olar?

```c
// libwait.c: constructor başqa thread-i gözləyir, o thread isə dlopen çağırır
#include <dlfcn.h>
#include <pthread.h>
#include <stdio.h>

static void *helper(void *arg) {
    fprintf(stderr, "helper: dlopen çağırıram...\n");
    void *h = dlopen("libm.so.6", RTLD_NOW);      // loader lock lazımdır
    fprintf(stderr, "helper: dlopen qayıtdı (%p)\n", h);
    return NULL;
}

__attribute__((constructor))
static void init(void) {
    pthread_t t;
    pthread_create(&t, NULL, helper, NULL);
    fprintf(stderr, "ctor: helper-i gözləyirəm...\n");
    pthread_join(t, NULL);                        // ❌ loader lock bizdədir
    fprintf(stderr, "ctor: bitdi\n");
}
```

```c
// main.c: libwait.so-nu dlopen ilə yükləyir
#include <dlfcn.h>
#include <stdio.h>

int main(void) {
    fprintf(stderr, "main: dlopen(libwait.so)\n");
    void *h = dlopen("./libwait.so", RTLD_NOW);
    fprintf(stderr, "main: yükləndi (%p)\n", h);
    return 0;
}
```

```
$ gcc -O2 -shared -fPIC -pthread libwait.c -o libwait.so
$ gcc -O2 main.c -o main -ldl
$ timeout 3 ./main; echo "çıxış kodu $?"
main: dlopen(libwait.so)
ctor: helper-i gözləyirəm...
helper: dlopen çağırıram...
çıxış kodu 124
$ gdb -q -batch -p <pid> -ex 'thread apply all bt 8'      (qısaltılıb)
Thread 2 (helper):
#0  futex_wait (...) at ../sysdeps/nptl/futex-internal.h:146
#1  __GI___lll_lock_wait (...) at ./nptl/lowlevellock.c:49
```

```
#3  ___pthread_mutex_lock (...) at ./nptl/pthread_mutex_lock.c:128
#4  _dl_open (...) at ./elf/dl-open.c:851
Thread 1 (main):
#3  __pthread_clockjoin_ex (...) at ./nptl/pthread_join_common.c:102
#4  init () from ./libwait.so
#5  call_init (...) at ./elf/dl-init.c:74
#7  _dl_init (...) at ./elf/dl-init.c:121
```

- Çıxış kodu 124 — `timeout` proqramı 3 saniyədən sonra öldürdü. “helper: dlopen qayıtdı” heç çap olunmadı.
- Thread 2 `_dl_open`-da mutex gözləyir, sonda `futex_wait`-də yatır (futex — Faza 4). Thread 1 həmin mutex-i tutub, constructor-un içində `pthread_join`-də gözləyir.
- Android-in linker-i də `dlopen`-u qlobal kilid altında edir. Ona görə `.so` constructor-unda və `JNI_OnLoad`-da başqa thread-in `System.loadLibrary`-sini gözləmək təhlükəlidir.
- Kotlin qarşılığı yuxarıdakı `<clinit>` nümunəsidir: hər iki halda kod “initialize” kilidi altında işləyir və başqa thread-i gözləyir.

> 🖼️ **Şəkil 275:** Fəsil 3-ün thread izləri · üç yer, bir qayda

Burada nəyə bax: üç mövzu üç fərqli qatdadır, amma ikisi eyni qaydaya çıxır: initialize kodu kilid altında işləyir, orada başqa thread-i gözləmə.

Bunu az adam bilir: Kotlin-də massiv literalı data deyil, koddur

Fakt. C-də `const int t[1000] = {…};` faylın `.rodata` hissəsinə hazır rəqəmlər kimi düşür: 4 byte / element. Kotlin-də `val t = intArrayOf(…)` isə data kimi saxlanılmır. Class faylında “data bölməsi” yoxdur. kotlinc sinfin `<clinit>` metoduna hər element üçün ayrıca instruction-lar yazır.

> 🖼️ **Şəkil 276:** Hazır siyahı və diktə · data və kod

Burada nəyə bax: nəticədə eyni massiv alınır, amma Kotlin-də o, proqram işləyəndə element-element qurulur.

Sübut. 1000 elementli cədvəl (dəyərlər 0–99 təkrarlanır):

```
        // 1000 elementli massiv literalı
        val table1000 = intArrayOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, /* … */ 99, 0, 1, /* … */)
```

```
$ kotlinc Table1000.kt -d out_Table1000 && javap -c -p -cp out_Table1000 Table1000Kt
  static {};
    Code:
       0: sipush        1000
       3: newarray       int
       5: astore_0
       6: aload_0
       7: iconst_0
       8: iconst_0
       9: iastore
      10: aload_0
      11: iconst_1
      12: iconst_1
      13: iastore
```

```
  ...
3342: aload_0
3343: sipush        500
3346: iconst_0
3347: iastore
  ...
6816: return
```

- `newarray int` — boş massiv yarat. Sonra hər element üçün dörd instruction: massivi götür, indeksi qoy, dəyəri qoy, yaz (`iastore`).
- `<clinit>`-in kodu 6 817 byte-dır: element başına təxminən 7 byte.

> 🖼️ **Şəkil 277:** Bir element · dörd instruction

Burada nəyə bax: dəyərin özü instruction-ın içindədir. Data "kodun arqumenti" kimi saxlanılır.

Divar. JVM spesifikasiyasına görə bir metodun kodu 65 535 byte-dan böyük ola bilməz. Yoxlayaq:

Massiv	Class faylı	`<clinit>` kodu
`IntArray(1_000_000)`	558 B	`newarray` — data yoxdur
`intArrayOf(1000 element)`	7 382 B	6 817 B
`intArrayOf(5000 element)`	35 142 B	34 577 B
`intArrayOf(20000 element)`	—	`Method too large`

```
$ kotlinc Table20000.kt -d out
exception: java.lang.RuntimeException: Error generating class file Table20000Kt.class (compiled
from [/home/claude/ch03/kt/Table20000.kt]): Method too large: Table20000Kt.<clinit> ()V
```

Eyni 20 000 element C-də problemsizdir: `size -A table20000.o` → `.rodata 80000`.

> 🖼️ **Şəkil 278:** 64 KB divarı · Kotlin və C

Burada nəyə bax: yaşıl zolaq (Kotlin) təxminən 9 000 elementdə divara dəyir. Mavi zolaq (C) divarı heç tanımır.

Niyə vacibdir.

- **Build xətası:** cədvəl böyüyüncə (emoji xəritəsi, CRC cədvəli, lokal data) layihə birdən yığılmamaq üçün səbəb tapır. Bu xəta kotlinc-də olur, D8-dən əvvəl — Android layihəsində də eyni xətanı alırsan.
- **Başlanğıc qiyməti (JVM):** `<clinit>` sinfin ilk istifadəsində işləyir. Minlərlə instruction interpreter-də, hələ JIT olunmamış halda icra olunur.
- **Android-də fərq:** D8 `newarray` və ardıcıl `iastore` zəncirini tanıyır və onu bir `fill-array-data` instruction-ına çevirir. R8-in mənbə kodunda bu çevrilmə var: zəncir başqa instruction-la qırılmayanda massiv bir cədvəllə doldurulur. Yəni DEX-də data yenə element başına 4 byte-dır, ART onu bir addımda köçürür. Amma 64 KB divarı qalır: o, D8-dən əvvəl, kotlinc-də çıxır. d8 bu maşında yoxdur — öz APK-nda `baksmali` və ya APK Analyzer ilə yoxla.

- **Həll:** böyük cədvəli resursa (`res/raw`, `assets`) və ya ayrıca fayla qoy və işə salanda oxu; ya da onu hesablayıb yarat.

Yanlış təsəvvürlər

> 🖼️ **Şəkil 279:** Altı mif · altı həqiqət

Burada nəyə bax: qırmızı sətir — çoxlarının düşündüyü, yaşıl sətir — bu fəsildə ölçdüyün.

Yanlış təsəvvür	Həqiqət	Özün sübut et
gcc C-ni birbaşa machine code-a çevirir	dörd mərhələ var: cc1, as, ld; gcc dispetçerdir	`gcc -v hello.c`
`printf("salam\n")` printf-i çağırır	compiler onu `puts`-a çevirir	`nm hello.o` → `U puts`
Böyük qlobal massiv faylı böyüdür	sıfır massiv `.bss`-dədir, faylda 0 byte	`size bss_zero`
Proqram `main`-dən başlayır	ld.so-nun `_start`-ı, sonra proqramın `_start`-ı, sonra `__libc_start_main`	`gdb -ex starti`
Adlar həmişə lazy həll olunur	Ubuntu-da və Android-də BIND_NOW: `main`-dən əvvəl	`readelf -d`, `LD_DEBUG=bindings`
x icazəsi yoxdursa fayl işləməz	ld.so onu adi fayl kimi açıb işlədir	`ld-linux-x86-64.so.2 ./hello_nox`
Static proqram həmişə sürətlidir	başlanğıc fərqi səs-küydə itir; fayl 49 dəfə böyükdür	`strace`, vaxt dövrü
JVM jar-dakı bütün sinifləri yükləyir	yalnız lazım olanları; Kotlin runtime-dan 0 sinif	`-Xlog:class+load`
Android Java bytecode işlədir	DEX işlədir: interpreter, JIT, AOT	APK Analyzer, `classes.dex`
`loadLibrary`-yə fayl adı verilir	adın ortası verilir: `lib` + ad + `.so`	`strace -f -e trace=%file`
Kotlin massiv literalı data kimi saxlanılır	`<clinit>`-də koddur, 64 KB limiti var	`javap -c -p`
jstack bütün deadlock-ları tapır	<clinit> deadlock-unu tanımır: thread-lər RUNNABLE görünür	jstack, grep 'Class initialization monitor' (S6)
Strip olunmuş faylın crash-i oxunmaz	Build ID ilə debug faylını tap, ünvanı sətrə çevir	readelf -n, addr2line (S5)
main-dən əvvəl sənin kodun işləmir	constructor-lar və <clinit> main-dən əvvəl işləyir	./ctor, java InitKt (Bölmə 7)

> 🖼️ **Şəkil 280:** Mifi sındıran alət · bir əmr

Burada nəyə bax: hər mif bir əmrlə sınır. Mühəndis fərziyyə ilə yox, çıxışla danışır.

> 🖼️ **Şəkil 281:** Static və dynamic · hesab kartı

Burada nəyə bax: yaşıl işarələrin çoxu dynamic sütunundadır. Static-in əsas üstünlüyü sürət yox, müstəqillikdir: bir fayl, asılılıq yoxdur.

İntervyuda

> 🖼️ **Şəkil 282:** Cavabın qatları · qısa cavabdan fərqləndirici detala

Burada nəyə bax: ən alt qat tək başına da düzgün cavabdır. Hər yeni qat cavabı senior səviyyəsinə yaxınlaşdırır.

İ 3.1. "`gcc hello.c && ./hello` əmrindən `main`-ə qədər nə baş verir?"

- **Qısa cavab:** Dörd build mərhələsi (preprocessor, compiler, assembler, linker) ELF executable yaradır. `execve`-də kernel onu yaddaşa map edir, ld.so kitabxanaları qoşur, sonra `_start` → `__libc_start_main` → `main`.
- **Mexanizm:** `cc1` assembly yazır; `as` object faylı yaradır: `REL` tip, sıfır byte-lar və relocation-lar, `T`/`U` simvollar. `ld` onları birləşdirir: PIE, PLT/GOT, `NEEDED libc.so.6`, `INTERP`. Kernel `binfmt_elf`-də `PT_LOAD`-ları map edir, `.bss`-i sıfırlayır, ld.so-nu yükləyir, stack-ə auxv yazır. ld.so libc-ni map edir, relocation-ları tamamlayır, GOT-u kilidləyir.
- **Rəqəm və sübut:** 127 B mənbə → 15 960 B executable; 36 system call; 96 relocation; `gdb starti` birinci instruction-ı ld.so-da göstərir.
- **Trade-off:** static — bir fayl, ld.so yoxdur, amma 785 KB və libc yamağı üçün yenidən yığmaq lazımdır.
- **Real nümunə:** Android-də tətbiq prosesi `execve` ilə yox, zygote-dan `fork` ilə yaranır. Native kitabxanalar `System.loadLibrary`-də linker64 ilə yüklənir və bütün simvollar dərhal həll olunur — böyük `.so` soyuq başlanğıcı uzadır.
- **Fərqləndirici detal:** `_start`-dan əvvəl kernel stack-ə auxv yazır; libc stack qoruyucusunu `AT_RANDOM`-dakı 16 təsadüfi byte-dan düzəldir.

> 🖼️ **Şəkil 283:** İ 3.1 · zəif və güclü cavab

Burada nəyə bax: güclü cavab uzun deyil, dəqiqdir. Hər cümləsi yoxlana bilən bir addımdır.

> 🖼️ **Şəkil 284:** İ 3.1 · cavabın skeleti

Burada nəyə bax: cavabı iki sırada qur: build və run. Interviewer harada dayandırsa, o dayanaqdan dərinləş.

Follow-up nərdivanı:

1. Object faylında `call`-un ünvanı niyə sıfırdır? → Hədəf hələ bilinmir; relocation qeydi linker-ə "buranı doldur" deyir.

2. PIE nədir, niyə lazımdır? → İstənilən ünvanda işləyən executable; ASLR-in executable-a da tətbiq olunması üçün.
3. ld.so-nu kim yükləyir? → Kernel, `PT_INTERP`-dəki yola görə; icranı da birinci ona verir.
4. `x` icazəsi olmasa? → `execve` rədd edir, amma `ld-linux-x86-64.so.2 ./prog` işləyir: x biti təhlükəsizlik sərhədi deyil.
5. Static-də nə dəyişir? → `INTERP` yoxdur, kernel birbaşa `_start`-a verir; 18 system call.

> 🖼️ **Şəkil 285:** Follow-up nərdivanı · İ 3.1

Burada nəyə bax: nərdivan fayldan kernel-ə enir. Hər pillənin cavabı bu fəslin bir bölməsidir.

İ 3.2. "Static və dynamic linking fərqi nədir? PLT/GOT necə işləyir?"

- **Qısa cavab:** Static-də kitabxana kodu executable-a köçürülür. Dynamic-də yalnız `NEEDED` qeydi yazılır; kitabxana işə düşəndə map olunur, adlar GOT vasitəsilə bağlanır.
- **Mexanizm:** `call puts@plt` → `jmp *GOT[0x3fd0]` → libc-dəki `puts`. GOT xanasını ld.so `R_X86_64_JUMP_SLOT` relocation-ına görə doldurur: BIND_NOW-da başlanğıcda, lazy-də ilk çağırışda resolver vasitəsilə.
- **Rəqəm və sübut:** 785 336 B və 15 960 B; 18 və 36 system call; başlanğıc vaxtı hər ikisində ≈1 ms.
- **Trade-off:** dynamic — yaddaş paylaşımı, avtomatik təhlükəsizlik yaması, kiçik fayl; static — asılılıqsız paylanma, ld.so-suz başlanğıc.
- **Real nümunə:** NDK-da bir tətbiqdə bir neçə `.so` varsa, C++ standart kitabxanasını `c++_shared` kimi işlətmək tövsiyə olunur. `c++_static` hər `.so`-ya öz nüsxəsini köçürür: ölçü artar, qlobal vəziyyət ikiləşə bilər.
- **Fərqləndirici detal:** full RELRO — ld.so GOT-u doldurandan sonra `mprotect` ilə yalnız oxunan edir. Ona görə GOT-u dəyişmək üzərində qurulan hücumlar işləmir.

> 🖼️ **Şəkil 286:** İ 3.2 · çamadan və kitabxana kartı

Burada nəyə bax: hansı yaxşıdır sualının cavabı "hansı qiyməti ödəməyə hazırsan" sualıdır: ölçü və müstəqillik, ya da paylaşma və avtomatik yama.

> 🖼️ **Şəkil 287:** Follow-up nərdivanı · İ 3.2

Burada nəyə bax: hər pillə əvvəlkinin cavabından doğur. Bir pilləni bilməsən, növbəti sual gəlmir — interviewer sənin sərhədini məhz belə tapır.

Follow-up nərdivanı:

1. PLT nədir? → Hər xarici funksiya üçün kiçik keçid: GOT xanasına dolayı `jmp`.
2. GOT nə vaxt dolur? → BIND_NOW-da `main`-dən əvvəl (LD_DEBUG 87 < 104); lazy-də ilk çağırışda.
3. RELRO nədir? → Relocation-dan sonra GOT və digər cədvəllərin `r--` edilməsi; `/proc/self/maps`-dəki əlavə `r--` sətri.

4. LD_PRELOAD necə işləyir? → ld.so-nun axtarış sırasına birinci yazılır; static proqramda ld.so olmadığı üçün təsirsizdir.
5. Android-də fərq? → Lazy binding yoxdur, NDK default `-z now -z relro`; 16 KB səhifə üçün `LOAD` Align 0x4000 olmalıdır.

> 🖼️ **Şəkil 288:** İ 3.2 · Android-də dörd fərq

Burada nəyə bax: dörd fərqin dördü də başlanğıc vaxtına və ya Play tələblərinə toxunur. Android developer üçün İ 3.2-nin ən dəyərli pilləsi budur.

İ 3.3. "Bu dəyişənlər yaddaşda harada yaşayır?"

- **Qısa cavab:** ilkin dəyərli qlobal → `.data`; sıfır qlobal → `.bss`; `const` və string literalı → `.rodata`; kod → `.text`; lokal → stack və ya register; `malloc` → heap.
- **Mexanizm:** section-lar segment-lərə yığılır, loader onları icazə ilə map edir. `.bss` `NOBITS`-dir: faylda yox, yaddaşda yaranır.
- **Rəqəm və sübut:** `bss_zero` 15 992 B, `bss_one` 4 016 008 B — tək bir `1` fərqi.
- **Trade-off:** `.rodata` səhifələri proseslər arasında paylaşılır, `.data` isə yazılanda hər prosesin öz nüsxəsinə çevrilir. Çox qlobal vəziyyət həm yaddaş, həm thread təhlükəsizliyi problemləridir.
- **Real nümunə:** Kotlin-də top-level `var` — `XxxKt` sinfində `private static` sahə və getter/setter; `const val` istifadə yerinə yapışdırılır; böyük `intArrayOf` — `<clinit>`-də kod və 64 KB limiti.
- **Fərqləndirici detal:** `char *s = "hi";` — `"hi"` `.rodata`-dadır, `s` isə `.data.rel.local`-dadır: PIE-də göstəricinin dəyəri yüklənmə ünvanından asılıdır, ona görə runtime relocation tələb edir.

> 🖼️ **Şəkil 289:** İ 3.3 · bir proqram, altı ev

Burada nəyə bax: dörd ev ELF-dən gəlir və bir yerdə durur, iki ev isə proqram işləyərkən yaranır. Cavabı bu iki qrupla başla.

> 🖼️ **Şəkil 290:** İ 3.3 · paylaşma və copy-on-write

Burada nəyə bax: trade-off-un mənası bu şəkildədir. Sabitlər pulsuz paylaşılır, dəyişən qloballar isə yazılan an hər prosesdə yaddaş tutur.

Follow-up nərdivanı:

1. `.bss` niyə faylda yer tutmur? → Yalnız ölçüsü lazımdır; sıfırları loader verir.
2. `int b;` ilə `int b = 0;` fərqi? → İkisi də `.bss`-dədir. GCC 10-dan əvvəl birincisi "common" simvol ola bilirdi.
3. Funksiyanın içindəki `static int calls = 1;`? → `.data`, lokal simvol `calls.0` (`nm`-də kiçik `d`).
4. Kotlin `const val` ilə `val` fərqi? → `const val` compile vaxtı yapışdırılır; `val` statik sahə və getter-dir.
5. Hər thread-in öz qlobalı lazımdırsa? → `.tdata`/`.tbss` (thread-local storage) — Faza 3-də.

> 🖼️ **Şəkil 291:** İ 3.3 · Kotlin-də beş ev

Burada nəyə bax: Kotlin-də sual "hansı section" yox, "hansı sahə, hansı metod" olur. Cavabı `javap` ilə sübut et.

İ 3.4. "Android-də Kotlin kodun necə icra olunur? Başlanğıcı necə sürətləndirərdin?"

- **Qısa cavab:** kotlinc → `.class` → D8/R8 → DEX → APK. Cihazda ART icra edir: interpreter, JIT və profilə əsaslanan AOT (dex2oat → `.vdex` + `.odex`). Proses zygote-dan `fork` ilə yaranır.
- **Mexanizm:** framework sinifləri zygote-da əvvəlcədən yüklənib, boot image-də AOT kodu var. Tətbiq sinifləri lazy yüklənir, `<clinit>` ilk istifadədə işləyir. Native `.so`-lar linker64 ilə, BIND_NOW rejimində.
- **Rəqəm və sübut:** JVM-də ölçdük: 590 sinif, 589-u CDS-dən; CDS ilə 54 ms, onsuz 103 ms. Android rəqəmlərini cihazda ölç: Macrobenchmark və Perfetto (Faza 8).
- **Trade-off:** AOT — sürətli icra, amma disk və quraşdırma vaxtı; JIT — çevik, amma isinmə vaxtı. Profil ikisinin arasında körpüdür.
- **Real nümunə:** soyuq başlanğıcın regressiyası: başlanğıcda ağır `<clinit>`, böyük `.so`, çox sinif yüklənməsi. Düzəliş: Baseline Profile, ağır işi lazy etmək, böyük cədvəli `assets`-ə köçürmək, `.so`-nu lazım olanda yükləmək.
- **Fərqləndirici detal:** `.so`-lar APK-dan birbaşa map olunur (sıxılmamış, səhifəyə düzülmüş); 1 noyabr 2025-dən Play üçün `LOAD` Align 16 KB olmalıdır.

> 🖼️ **Şəkil 292:** İ 3.4 · soyuq başlanğıcın oğruları

Burada nəyə bax: dörd oğrunun hər biri bu fəslin bir bölməsidir. Düzəliş çox vaxt "etmə" yox, "sonra et"dir.

Follow-up nərdivanı:

1. DEX niyə var, `.class` niyə yox? → Bir fayl, ümumi constant pool, register əsaslı instruction-lar: mobil cihaz üçün kompakt.
2. JIT və AOT birlikdə necə işləyir? → JIT isti metodları tapır və profilə yazır, cihaz boş olanda dex2oat həmin profilə görə AOT edir.
3. Baseline Profile nə edir? → Profili APK ilə əvvəlcədən verir: ilk işə salmadan başlanğıc kodu AOT olur.
4. Zygote niyə sürətləndirir? → Framework sinifləri və resurslar bir dəfə yüklənir; `fork` səhifələri copy-on-write ilə paylaşır (Faza 1).
5. 16 KB nəyi dəyişdi? → `.so`-nun segmentləri 16 KB sərhədində başlamalıdır; NDK r28+ və `-z max-page-size=16384` bunu təmin edir.

> 🖼️ **Şəkil 293:** İ 3.4 · JIT, AOT və Baseline Profile

Burada nəyə bax: eyni kod dörd mərhələdə fərqli icra olunur. Baseline Profile birinci işə salmanı sonrakılar kimi edir.

> 🖼️ **Şəkil 294:** Rəqəm kartı · iddia + sübut

Burada nəyə bax: səkkiz rəqəm, səkkiz əmr. Bunları bilən namizəd fərziyyə ilə yox, ölçü ilə danışır.

Bilmədiyin sualda: “Bilmirəm” demə və dayanma. Əsasdan başla: “fayl ELF-dirsə, program header-ləri var; loader onları map edir; deməli cavab orada olmalıdır”. Sonra yoxlama yolunu de: hansı əmr (`readelf -l`, `strace`, `LD_DEBUG`), hansı mənbə faylı (`fs/binfmt_elf.c`, `elf/rtld.c`).

Bug-ı tap

B3.1. Ortaq ayar faylı

Komanda bir ayarı bütün fayllarda istifadə etmək üçün onu header-ə qoydu:

```
// config.h: bütün fayllar üçün ümumi ayarlar
#ifndef CONFIG_H
#define CONFIG_H
int retries = 3;
void load(void);
#endif
```

```
// a.c
#include <stdio.h>
#include "config.h"
int main(void) { load(); printf("%d\n", retries); return 0; }
```

```
// b.c
#include "config.h"
void load(void) { retries = 5; }
```

```
$ gcc a.c b.c -o app
/usr/bin/ld: /tmp/cc2ESniU.o:(.data+0x0): multiple definition of `retries'; /tmp/ccBdKVr0.o:
(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Suallar: Include guard (`#ifndef`) niyə kömək etmədi? Xəta niyə compiler-də yox, linker-də çıxır? Necə düzəldirsən? `int retries;` (dəyərsiz) yazsan nə olar?

> 🖼️ **Şəkil 295:** B3.1 · iki ev, eyni ünvan

Burada nəyə bax: hər iki ev header-dən eyni planı götürüb tikilib. Poçtalyon (linker) iki eyni ünvan görür.

> 🖼️ **Şəkil 296:** B3.1 · kim nəyi görür

Burada nəyə bax: iki compile ayrı-ayrı keçir və ikisi də xətasızdır. Problem yalnız faylları birlikdə görən addımda üzə çıxır.

Kotlin versiyası — eyni bug, başqa geyimdə:

```
// Network.kt
@file:JvmName("Config")
package app
```

```
        var retries = 3
```

```
        // Storage.kt
        @file:JvmName("Config")
        package app

        var cacheSize = 64
```

```
$ kotlinc Network.kt Storage.kt -d out
Network.kt:1:1: error: duplicate JVM class name 'app/Config' generated from: Config, Config
```

Sual: kotlinc nə üçün eyni adı iki dəfə yarada bilmir? İki düzəliş yolu tap.

> 🖼️ **Şəkil 297:** B3.1 · əvvəl və sonra

Burada nəyə bax: düzəlişdən sonra `retries` üçün bir `D` və iki `U` qalır. Bu, B3.1-in tam cavabı deyil — "niyə" hissəsi Cavab açarındadır.

B3.2. Paket adını dəyişdik

Refaktorinqdə `Native.kt` `demo` paketindən `demo.math` paketinə köçdü. C tərəfinə toxunulmadı:

> 🖼️ **Şəkil 298:** B3.2 · ev köçdü

Burada nəyə bax: Kotlin tərəfdə heç nə sınmayıb, kompilyasiya təmizdir. Uyğunsuzluq iki dil arasındakı addır.

```
        // Native.kt (refaktorinqdən sonra)
        package demo.math

        object Native {
            init { System.loadLibrary("square") }
            @JvmStatic external fun square(x: Int): Int
        }
```

```
        // native.c (dəyişməyib)
        JNIEXPORT jint JNICALL
        Java_demo_Native_square(JNIEnv *env, jclass cls, jint x) {
            return x * x;
        }
```

```
$ java -Djava.library.path=. -jar b2.jar
Exception in thread "main" java.lang.UnsatisfiedLinkError: 'int demo.math.Native.square(int)'
```

> 🖼️ **Şəkil 299:** B3.2 · JNI adının anatomiyası

Burada nəyə bax: JVM C adını Kotlin adından qurur. Paket bir söz uzandı, C adı da uzanmalı idi.

Suallar: `loadLibrary` uğurla işlədi, bəs xəta nə vaxt çıxdı? İki düzəliş yolu de. Android release build-da R8 bu sinfin adını dəyişsə nə olar?

> 🖼️ **Şəkil 300:** B3.2 · JNI açarı

Burada nəyə bax: JVM adı Kotlin tərəfindən qurur, `.so` isə köhnə adı saxlayır. Arada heç bir compiler yoxdur ki, bu fərqi görsün.

B3.3. C++-a keçdik

Komanda JNI kodunda C++ imkanları istədi: `native.c`-ni `native.cpp` adlandırdı və g++ ilə yığdı. Funksiyanın mətni dəyişmədi, Kotlin tərəfi (`Native.kt`, paket `demo`) də dəyişmədi:

```
        // native.cpp: eyni JNI funksiyası, amma C++ faylında
        #include <jni.h>

        JNIEXPORT jint JNICALL
        Java_demo_Native_square(JNIEnv *env, jclass cls, jint x) {
            return x * x;
        }
```

> 🖼️ **Şəkil 301:** B3.3 · iki pasport

Burada nəyə bax: mətn eynidir, fayldakı ad isə fərqlidir. JVM adı hərf-hərf axtarır.

```
$ g++ -O2 -shared -fPIC -I$J/include -I$J/include/linux native.cpp -o libsquare.so
$ java -Djava.library.path=. -jar native.jar
Exception in thread "main" java.lang.UnsatisfiedLinkError: 'int demo.Native.square(int)'
    at demo.Native.square(Native Method)
    at demo.NativeKt.main(Native.kt:12)
$ nm -D libsquare.so | grep -i square
0000000000001100 T _Z23Java_demo_Native_squareP7JNIEnv_P7_jclassi
$ nm -D -C libsquare.so | grep -i square
0000000000001100 T Java_demo_Native_square(JNIEnv_*, _jclass*, int)
```

> 🖼️ **Şəkil 302:** B3.3 · mangle olunmuş adın anatomiyası

Burada nəyə bax: “23” sənin yazdığın adın uzunluğudur, sonrası isə arqumentlərin tipləridir. C-də bu əlavələr yoxdur.

Suallar: ad niyə dəyişdi? `nm -D -C` nə göstərir və niyə aldadıcıdır? Bir söz əlavə etməklə necə düzəldərsən? Android Studio-nun C++ şablonunda hər JNI funksiyasının qarşısında həmin söz niyə var?

> 🖼️ **Şəkil 303:** B3.3 · nm -D və nm -D -C

Burada nəyə bax: `-C` adı insan üçün açır, amma fayldakı ad dəyişmir. JVM `-C` işlətmir.

Öz sözlərinlə izah et

Bu fəsli 5 cümlə ilə bir dostuna izah et: `gcc hello.c && ./hello` yazanda nə baş verir? Yaxşı izahda bunlar var:

- [ ] Dörd build mərhələsi və hər birinin çıxardığı fayl.
- [ ] Object faylında niyə boşluqlar qalır və onları kim doldurur.
- [ ] Section və segment fərqi; `.bss`-in faylda yer tutmaması.
- [ ] `main`-dən əvvəl kim işləyir: kernel, ld.so, `_start`.
- [ ] Kotlin-də eyni işlərin harada və nə vaxt baş verdiyi (JVM, ART).

Boş xəritə: aşağıdakı xəritəni kitaba baxmadan doldur. Hər qutuya bir mərhələnin adını və onun əsas sözünü yaz. Sonra fəslin sonundakı Xülasə şəkli ilə müqayisə et.

> 🖼️ **Şəkil 304:** Fəsil 3 · boş xəritə

Keçid qapısı

Qeydə baxmadan cavab ver. 8 sualdan ən azı 7-sinə rubrikanın 3-cü səviyyəsində (mexanizm) cavab verə bilirsənsə, Fəsil 4-ə keç.

1. **K3.1 (yadına sal):** Build-in dörd mərhələsini və hər birinin çıxış faylını say.
2. **K3.2 (niyə):** `hello.o`-da `call` instruction-ının ünvan hissəsi niyə `00 00 00 00`-dır?
3. **K3.3 (nəticəni təxmin et):** `int big[1000000];` əvəzinə `int big[1000000] = {0};` yazsan, faylın ölçüsü dəyişərmi?
4. **K3.4 (bug-ı tap):** `gcc -L. -lutil main.c` niyə `undefined reference` verir, `gcc main.c -L. -lutil` isə yox?
5. **K3.5 (dizayn):** Android tətbiqində 30 000 elementli lookup cədvəli lazımdır. Kotlin `intArrayOf`, C `.so`, yoxsa `assets`? Niyə?
6. **K3.6 (nəticəni təxmin et):** `LD_PRELOAD=./libmyputs.so ./hello_static` nə çap edər? Niyə?
7. **K3.7 (Fəsil 2):** `"salam"` mətni proqram işləyəndə ilk dəfə oxunanda haradan gəlir və ikinci oxunuş niyə daha sürətlidir?
8. **K3.8 (Fəsil 1):** `lea 0x0(%rip),%rax`-da `rip` nədir və ünvan niyə məhz ona nisbətən hesablanır?

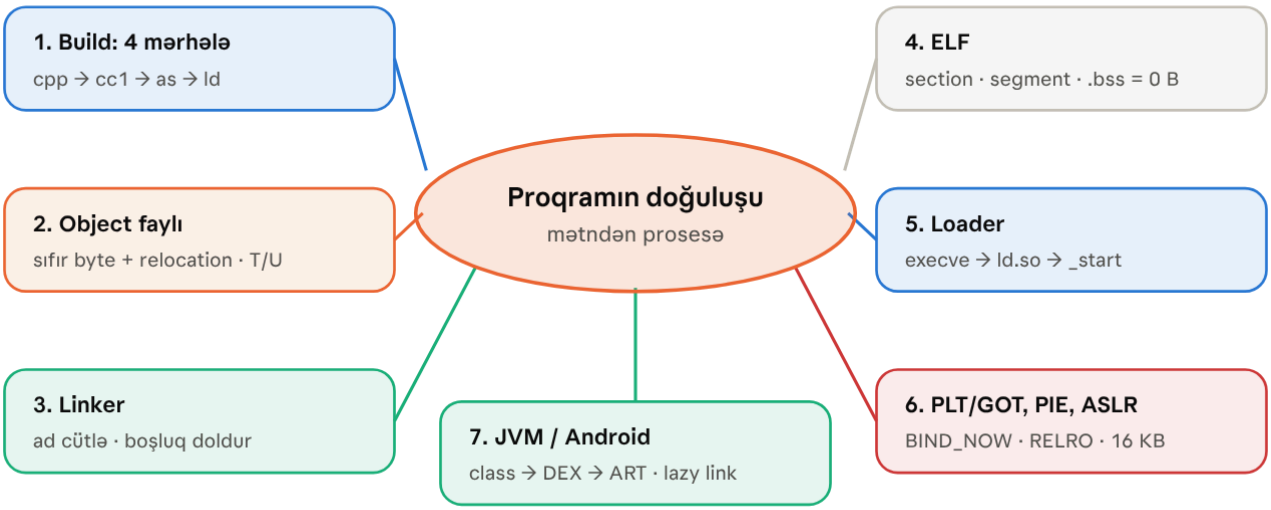
Yadda saxla

- Build dörd addımdır: preprocessor mətni genişləndirir, compiler assembly (Kotlin-də bytecode) yazır, assembler boşluqlu machine code yaradır, linker boşluqları doldurur.
- Object faylında adlar var, ünvanlar yox: relocation qeydləri və `T`/`U` simvollar. Linker hər `U` üçün dəqiq bir `T` axtarır.
- ELF-in iki planı var: section-lar linker üçün, segment-lər loader üçün. `.bss` faylda 0 byte tutur.
- Proqram `main`-dən başlamır: kernel → ld.so (kitabxanaların constructor-ları) → `_start` → `__libc_start_main` → `main`. Adlar BIND_NOW ilə əvvəlcədən bağlanır, GOT sonra kilidlənir.
- JVM və ART-da eyni işlər runtime-dadır: lazy class loading, lazy resolution; CDS, zygote və AOT işi əvvəlcədən görür. Kotlin massiv literalı data yox, koddur.

Bir sətirlik model: proqram — adların ünvana çevrilmə hekayəsidir; compiler adları yazır, linker və loader onları ünvana çevirir, dillər yalnız bunun nə vaxt olmasında fərqlənir.

Xülasə şəkli

Bütün fəsil bir şəkildə: mərkəzdə doğuluş, ətrafında yeddi addım — build-dən JVM və Android-ə qədər.



Bir cümlə: mətn dörd build mərhələsində ELF faylına, bir yükləmədə isə prosese çevrilir; hər addımda adlar bir az da ünvana yaxınlaşır.

Lüğət

Hər sətir həm də flashcard-dır: termini ört, izahı yaddaşdan de. Siyahı fəsillər əlavə olunduqca böyüyür.

Termin	Sadə izah	Dəqiq tərif	Fəsil
Bit	"Hə" və ya "yox"	İki vəziyyətli ən kiçik informasiya vahidi: 0 və ya 1	0
Byte	8 bitlik qutu	8 bit; 256 fərqli dəyər; yaddaşın ünvanlanan vahidi	0
Binary	İkilik say sistemi	Əsası 2 olan pozisional say sistemi	0
Hex	On altlıq say sistemi	Əsası 16 olan say sistemi; 1 rəqəm = 4 bit	0
KiB / KB	Kilobaytın iki mənası	KiB = 1024 byte, KB = 1000 byte	0
ASCII	İngilis hərfləri cədvəli	128 simvolu 0–127 ədədlərinə uyğunlaşdıran 1963-cü il standartı	0
Unicode	Dünyanın bütün hərfləri cədvəli	Hər simvola code point verən standart	0
Code point	Simvolun nömrəsi	Unicode-da simvolun ədədi, məsələn U+0259 (ə)	0
Encoding	Nömrəni byte-a çevirmə qaydası	Code point-ləri byte ardıcıllığına çevirən qayda (UTF-8, UTF-16)	0
UTF-8	1–4 byte-lıq mətn yazılışı	ASCII ilə uyğun, self-synchronizing Unicode encoding-i	0
UTF-16	2 və ya 4 byte-lıq yazılış	16 bitlik code unit-lərlə encoding; JVM `String`-inin daxili modeli	0
Surrogate pair	Emoji üçün iki `Char`	BMP-dən kənar code point-i ifadə edən iki UTF-16 unit (D800–DFFF)	0
Grapheme cluster	Ekranda görünən "bir simvol"	İstifadəçinin bir simvol saydığı bir və ya bir neçə code point	0
BOM	Faylın başındakı sıra işarəsi	UTF-16 üçün FE FF və ya FF FE; UTF-8-də EF BB BF (lazım deyil)	0
Overlong encoding	Lazım olandan uzun yazılış	Bir code point-in qadağan olunmuş uzun UTF-8 forması (C0 AF = "/")	0
Signed / unsigned	İşarəli / işarəsiz	Bitlərin mənfi ədəd daşıyıb-daşımadığını deyir; bitlər eyni, məna fərqli	0
Two's complement	Mənfi ədədi yazma üsulu	−x = NOT x + 1; ən yüksək bitin çəkisi −2^(n−1)	0
Overflow	Ədəd qutuya sığmayır	Nəticə tipin aralığından çıxır; JVM-də wrap, C-də UB	0
Wraparound	Sayğac sıfırlanır	Daşan bitlər atılır, aşağı n bit qalır	0
Undefined behavior	"Nə olacağına söz yoxdur"	Standartın nəticə vəd etmədiyi hal; kompilyator onu yox sayır	0
Sign extension	İşarə bitinin yayılması	Kiçik signed tipi böyüyə çevirəndə işarə bitinin yuxarı bitlərə kopyalanması	0

Termin	Sadə izah	Dəqiq tərif	Fəsil
Endianness	Byte-ların sırası	Çox-byte-lı ədədin yaddaşda byte sırası: little (kiçik əvvəl) və ya big	0
Network byte order	Şəbəkənin byte sırası	İnternet protokollarında big-endian	0
Fayl imzası (magic number)	Faylın "pasportu"	Formatı tanıdan ilk byte-lar: PNG 89 50 4E 47, ELF 7F 45 4C 46	0
IEEE 754	Kəsr ədəd standartı	`Double`: 1 işarə, 11 eksponent, 52 kəsr biti	0
Locale	Dil və regional qaydalar	Mətn çevirməsi, format və müqayisə qaydaları dəsti; az-da I → ı	0
CPU	Proqramı icra edən çip	Instruction-ları fetch, decode, execute edən prosessor	1
Instruction	CPU-nun bir əmri	ISA-nın bir əməliyyatı; byte-larla kodlanır	1
Machine code	CPU-nun dili	Instruction-ların byte forması	1
Assembly	Machine code-un oxunan yazılışı	Bir sətir = bir instruction olan mətn dili	1
ISA	CPU ilə proqram arasında müqavilə	Instruction set architecture: instruction-lar, register-lər, qaydalar (x86-64, ARM64)	1
Register	CPU-nun içindəki kiçik qab	Ən sürətli, sayı az yaddaş xanası; ARM64-də x0–x30	1
ALU	Hesablayan dövrə	Arithmetic Logic Unit: toplama, çıxma, məntiqi əməliyyatlar	1
PC	"Haradayam" barmağı	Program counter: icra olunacaq instruction-ın ünvanı (x86-64: RIP)	1
SP	Stack-in ucu	Stack pointer: stack-in cari ucunun ünvanı (x86-64: RSP)	1
LR	Qayıdış ünvanı	ARM64-də x30: `bl`-in yazdığı qayıdış ünvanı	1
Flag	Son nəticə haqqında qeyd	Bir bitlik vəziyyət: ARM64-də N, Z, C, V; x86-64-də SF, ZF, CF, OF	1
XZR	Həmişə sıfır olan register	ARM64-də register 31-in bir mənası: 0 oxunur, yazılan atılır	1
Fetch / decode / execute	Götür / aç / icra et	Instruction icrasının əsas mərhələləri	1
Clock, cycle	Metronom və onun bir tiqqiltısı	CPU-nu sinxronlaşdıran siqnal; iki tiq arası vaxt	1
GHz	Saniyədə milyard tiq	10^9 cycle/s; 2,1 GHz-də cycle ≈ 0,48 ns	1
IPC	Bir tiqdə neçə iş	Instructions per cycle: ortalama göstərici	1
Core	Bir aşpaz	Müstəqil icra maşını: öz PC-si, register-ləri, ALU-ları	1
SMT / hardware thread	Bir aşpaz, iki resept	Bir core-da bir neçə icra konteksti (Hyper-Threading)	1
Pipeline	Paltaryuyan xətti	Instruction mərhələlərinin üst-üstə düşməsi	1

Termin	Sadə izah	Dəqiq tərif	Fəsil
Latency / throughput	Nə qədər gözləyirəm / nə qədər bitir	Bir əməliyyatın vaxtı / vahid vaxtda bitən əməliyyat sayı	1
Branch prediction	`if`-in nəticəsini təxmin	CPU-nun keçidin hansı tərəfə gedəcəyini əvvəlcədən seçməsi	1
Misprediction	Səhv təxmin	Yanlış yolda görülən işin atılması; təxminən 15–20 cycle	1
Out-of-order execution	Hazır olanı əvvəl et	Instruction-ları hazırlıq sırası ilə icra edib proqram sırası ilə rəsmiləşdirmək	1
Speculative execution	Ehtimalla qabaqcadan işləmək	Təxmin edilmiş yolu nəticə məlum olmadan icra etmək; Spectre-in mənbəyi	1
Branchless kod	Jump-sız seçim	`csel`, `cmov` kimi şərtli seçim instruction-ları ilə yazılmış kod	1
Calling convention	Funksiyalar arası razılaşma	Arqument və nəticənin hansı register-lərdə ötürüldüyü qaydası (Fəsil 4)	1
Bytecode	JVM-in instruction dili	JVM və ART-ın icra etdiyi ara kod (`getfield`, `iadd`)	1
Lost update	İtən artırma	İki oxu-dəyiş-yaz əməliyyatının qarışması ilə bir yazının itməsi	1
Race condition	Nəticə vaxt sırasından asılıdır	Nəticənin thread-lərin sıralanmasından asılı olması (Faza 4)	1
Atomic əməliyyat	Bölünməz addım	Başqa core-ların ortasını görmədiyi əməliyyat: `lock add`, `ldaddal`	1
LL/SC	"Oxu, sonra yalnız toxunulmayıbsa yaz"	Load-linked / store-conditional: ARM64-də `ldaxr` / `stlxr`	1
ABI (Android)	Native kodun platforma növü	`arm64-v8a`, `armeabi-v7a`, `x86_64`, `x86`	1
TSO	Güclü yazı sırası	x86-64-ün yaddaş modeli: yazılar başqalarına sıra ilə görünür	1
µop	Daxili kiçik addım	x86 instruction-larının CPU daxilində parçalandığı sadə əməliyyatlar	1
big.LITTLE	Böyük və kiçik core-lar	Fərqli güc və enerji sərfiyyatlı core-ların bir çipdə olması (DynamIQ)	1
DVFS, thermal throttling	Tezlik dəyişir	Yükə və temperatura görə tezlik və gərginliyin dəyişməsi	1
TSC	Sabit sürətli sayğac	Time Stamp Counter: `rdtsc` ilə oxunur, core tezliyindən asılı deyil	1

Fəsil 2 terminləri:

Termin	Sadə izah	Dəqiq tərif	Fəsil
RAM	İşləyən proqramın yaddaşı	İstənilən ünvana birbaşa müraciət edilən, elektrik kəsiləndə silinən əsas yaddaş (DRAM)	2

Termin	Sadə izah	Dəqiq tərif	Fəsil
Ünvan	Byte-ın nömrəsi	Yaddaşda byte-ın yerini göstərən ədəd; 64 bitlik sistemdə 64 bitlik dəyər	2
Pointer, reference	"Orada bax" qeydi	Dəyəri başqa obyektin ünvanı olan dəyişən; JVM-də reference adlanır, ünvan gizlidir	2
Latency	Gözləmə vaxtı	Bir sorğunun başlanğıcından cavabına qədər keçən vaxt	2
Bandwidth	Yolun eni	Vahid vaxtda ötürülə bilən data miqdarı (GB/s)	2
Memory wall	Yaddaş divarı	CPU sürətinin RAM latency-sindən çox sürətlə artması problemi (Wulf və McKee, 1994)	2
Cache	Yaxın rəf	RAM-dakı data-nın CPU yanındakı kiçik, sürətli SRAM surəti	2
Cache hit, cache miss	Tapdı, tapmadı	İstənilən line cache-dədir; və ya yoxdur və aşağı səviyyədən gətirilir	2
Cache line	Bir kisə	Cache ilə yaddaş arasında daşınan blok; x86-64 və ARM64-də 64 byte	2
Temporal locality	Yenə lazım olacaq	Yaxın vaxtda eyni ünvana təkrar müraciət	2
Spatial locality	Qonşusu da lazım olacaq	Yaxın vaxtda qonşu ünvanlara müraciət	2
L1, L2, L3	Cache pillələri	Core-a yaxınlıq sırası ilə cache səviyyələri; L1 və L2 adətən core-un özünündür, L3 ortaqdır	2
Working set	İşlək data	Proqramın müəyyən müddətdə toxunduğu data-nın həcmi	2
Row-major	Sətir-sətir düzülüş	2D massivin sətirlərinin yaddaşda ardıcıl saxlanması: `[i][j]` = `m[i * n + j]`	2
Pointer chasing	Zəncir boyunca qaçmaq	Hər oxunuşun ünvanı əvvəlki oxunuşun nəticəsindən asılı olan giriş; prefetcher kömək etmir	2
Boxing	Ədədi qutuya qoymaq	Primitiv dəyəri obyektə (`Integer`) çevirmək; `List<Int>`-də hər element belədir	2
SRAM, DRAM	Cache yaddaşı, RAM yaddaşı	SRAM: bit başına ~6 tranzistor, refresh-siz; DRAM: 1 tranzistor və 1 kondensator, refresh lazımdır	2
Set, way, tag	Cache-in rəfləri və yerləri	Index set-i seçir, line set-in N way-indən birinə düşür, tag line-ı tanıyır	2
Conflict miss	Yer var, amma rəf doludur	Line-ların eyni set-ə düşdüyü üçün qovulması	2
Prefetcher	Qabaqcadan gətirən	Giriş naxışını görüb line-ları istənməmişdən əvvəl gətirən hardware	2
Cache coherence	Surətlərin uyğunluğu	Core-ların cache-lərindəki eyni line-ın surətlərini uyğun saxlayan hardware protokolu (MESI ailəsi)	2
False sharing	Yalançı paylaşma	Fərqli thread-lərin eyni line-dakı fərqli dəyişənlərə yazması və line-ın core-lar arasında qaçması	2
Padding	Boş yer	Dəyişənləri fərqli line-lara ayırmaq üçün qoyulan istifadəsiz byte-lar; JDK `@Contended` default 128 byte	2

Termin	Sadə izah	Dəqiq tərif	Fəsil
LongAdder	Pay-pay sayğac	Toqquşmada thread-ləri ayrı `@Contended` hüceyrələrə paylayan JDK sayğacı; `sum()` hamısını toplayır	2
TLB	Ünvan tərcüməsi cache-i	Virtual ünvanın fiziki ünvana son tərcümələrini saxlayan kiçik cache	2
Huge page	Böyük səhifə	Adi 4 KB əvəzinə 2 MB-lıq yaddaş səhifəsi; TLB miss-ləri azaldır	2

Fəsil 3 terminləri:

Termin	Sadə izah	Dəqiq tərif	Fəsil
Source code	İnsanın yazdığı mətn	Proqramlaşdırma dilində yazılmış, compiler-in oxuduğu mətn fayl	3
Build	Yığma	Mənbə kodundan icra olunan fayl yaradan addımlar zənciri	3
Preprocessor	Mətn redaktoru	`#include`, `#define` kimi direktivləri açan ilk mərhələ	3
Macro	Avtomatik əvəzetmə	Preprocessor-un mətni mətnlə əvəz etdiyi qayda; tip bilmir	3
Compiler	Tərcüməçi-redaktor	Dili assembly-yə (C) və ya bytecode-a (Kotlin) çevirən proqram	3
Assembler	Byte yazan	Assembly mətnini machine code byte-larına və object faylına çevirən proqram (`as`)	3
Object faylı	Boşluqlu blank	Relocation-ları və simvol cədvəli olan, hələ işə salınmayan ELF faylı (`REL`)	3
Relocation	“Bura sonra yaz” qeydi	Linker və ya loader-in hansı byte-lara hansı ünvanı yazacağını göstərən qeyd	3
Symbol table	Adlar siyahısı	Faylın təyin etdiyi (`T`, `D`, `B`) və istifadə etdiyi (`U`) adların cədvəli	3
Declaration, definition	Vəd, tərif	Elan adın tipini bildirir; tərif yaddaş və ya kod ayırır	3
Linker	Yığıcı	Object fayllarını birləşdirən, adları cütləyən və boşluqları dolduran proqram (`ld`)	3
Static library	Hazır parçalar qutusu	Object fayllarının arxivi (`.a`); lazım olan parçalar executable-a köçürülür	3
Shared library	Ortaq kitabxana	İşə düşəndə yüklənən və proseslər arasında paylaşılan ELF (`.so`)	3
ELF	Linux-un icra faylı formatı	Executable, object və shared library üçün format: header, section-lar, segment-lər	3
Section	Fəsil	Linker üçün bölmə: `.text`, `.rodata`, `.data`, `.bss`	3
Segment	Yükləmə planı	Loader üçün bölmə (program header): faylın hansı hissəsi hansı icazə ilə map olunur	3
`.bss`	Boş yer vədi	Sıfırla başlayan qlobal dəyişənlər; faylda yer tutmur (`NOBITS`)	3

Termin	Sadə izah	Dəqiq tərif	Fəsil
Entry point	Giriş qapısı	Proqramın ilk icra olunan instruction-ının ünvanı (`_start`)	3
Loader	Yükləyici	Faylı prosese çevirən kernel və ld.so cütü	3
`execve`	"Bu proqrama çevril"	Prosesin yaddaşını yeni proqramla əvəz edən system call	3
Dynamic linker (ld.so)	Köməkçiləri çağıran	Shared library-ləri yükləyən və adları bağlayan proqram; Android-də `linker64`	3
auxv	Kernel-in qeydi	Kernel-in stack-ə yazdığı açar-dəyər siyahısı: `AT_ENTRY`, `AT_BASE`, `AT_RANDOM`	3
PIE	İstənilən yerdə işləyən	Kodu PC-relative olan, istənilən ünvana yüklənə bilən executable	3
ASLR	Ünvan lotereyası	Hər işə salmada yükləmə ünvanlarını təsadüfi dəyişən qoruma	3
PLT, GOT	Operator və kitabça	PLT — xarici funksiyaya keçid kodu; GOT — ünvanların cədvəli	3
BIND_NOW, lazy binding	İndi və ya sonra	Adların start-da və ya ilk çağırışda həll olunması	3
RELRO	Kilidlənmiş cədvəl	Relocation-dan sonra GOT-u yalnız oxunan edən qoruma	3
Interposition	Yerinə keçmə	Eyni adlı simvolun axtarış sırasında daha əvvəl gələn kitabxanadan götürülməsi (`LD_PRELOAD`)	3
JVM bytecode	Virtual maşının dili	JVM-in icra etdiyi stack əsaslı instruction-lar (`.class`)	3
Constant pool	Adlar anbarı	Class faylında sinif, metod adlarının və sabitlərin cədvəli	3
Lazy class loading	Lazım olanda gətir	JVM sinfi yalnız ilk istifadədə yükləyir və link edir	3
CDS	Hazır sinif arxivi	Class Data Sharing: JDK sinifləri əvvəlcədən hazırlanmış, map olunan arxivdə	3
DEX	Android-in bytecode-u	Bütün sinifləri bir faylda saxlayan register əsaslı format (`classes.dex`)	3
ART	Android-in icraçısı	Android Runtime: DEX-i interpreter, JIT və AOT ilə icra edir	3
JIT, AOT	İndi və əvvəlcədən tərcümə	JIT — işləyərkən, AOT — əvvəlcədən (dex2oat) machine code-a çevirmə	3
Zygote	Ana proses	Framework-u əvvəlcədən yükləyən və tətbiqləri `fork` ilə doğuran Android prosesi	3
JNI	Kotlin-C körpüsü	JVM/ART ilə native kod arasında çağırış interfeysi; ad `Java_paket_Sinif_metod`	3
Class faylı	Kotlin/Java compiler-inin çıxışı	JVM üçün bytecode, constant pool və metodları saxlayan fayl; magic ca fe ba be	3
Executable	işə salına bilən fayl	program header-ləri və entry point-i olan ELF faylı (EXEC və ya PIE üçün DYN)	3

Termin	Sadə izah	Dəqiq tərif	Fəsil
Static və dynamic linking	kodu fayla köçürmək və ya işləyərkən qoşmaq	static: kitabxana kodu executable-a köçürülür; dynamic: yalnız NEEDED yazılır, ld.so başlanğıcda map edir	3
OAT, .vdex, .odex	ART-ın əvvəlcədən hazırladığı fayllar	dex2oat çıxışı: .vdex — yoxlanılmış DEX, .odex — ELF içində OAT formatında machine code	3
D8, R8	Android-in DEX yazan alətləri	D8 .class-ı DEX-ə çevirir; R8 əlavə olaraq kodu kiçildir, adları qısaldır, mapping.txt yazır	3
Baseline Profile	başlanğıc kodunun siyahısı	APK ilə gələn profil; dex2oat quraşdırmada həmin kodu AOT edir	3
Boot image	framework-ün hazır machine code-u	zygote-un yüklədiyi, əvvəlcədən AOT olunmuş framework sinifləri	3
Heap	işləyərkən istənən yaddaş	malloc (C) və ya new (JVM) ilə ayrılan, ömrü funksiyadan asılı olmayan yaddaş (Faza 2)	3
Include guard	header-i ikinci dəfə saymamaq	#ifndef X, #define X, #endif; yalnız bir .c faylının daxilində təkrarı önləyir	3
Top-level	sinfin içində olmayan Kotlin elanı	kotlinc onu FaylKt sinfinin static üzvünə çevirir	3
Symbol resolution	adı sahibi ilə cütləmək	linker-in hər U üçün bir T tapması; JVM-də simvolik istinadın ilk istifadədə həlli	3
Interpreter	kodu addım-addım icra edən	bytecode-u machine code-a çevirmədən instruction-ları bir-bir icra edən runtime hissəsi	3
Runtime	proqramın işlədiyi vaxt və ya mühit	build vaxtının əksi; həm də JVM və ART kimi icra mühiti	3
System call	proqramın kernel-dən xidmət istəməsi	xüsusi instruction ilə kernel-ə keçid: execve, mmap, write (Fəsil 6)	3
AppCDS	öz siniflərin üçün CDS	-XX:ArchiveClassesAtExit ilə yaradılan dinamik arxiv (JDK 13, JEP 350)	3
Verification, initialization	sinfi yoxlamaq və hazırlamaq	verification — bytecode-un tip yoxlaması; initialization — ilk aktiv istifadədə <clinit>-in işləməsi	3
Symbol visibility	adın kitabxanadan kənara görünməsi	-fvisibility=hidden və visibility("default"); dynamic symbol table-a nəyin düşdüyünü idarə edir	3
Constructor (.init_array)	main-dən əvvəl işləyən funksiya	__attribute__((constructor)) və C++ qlobal obyektləri; ünvanları .init_array-dədir, ld.so və libc çağırır	3
Debug məlumatı (DWARF)	ünvanı sətrə çevirən xəritə	-g ilə yazılan .debug_* section-ları; strip onları silir	3
strip	debug və simvolları silən alət	.symtab və .debug_* section-larını silir; kod dəyişmir	3
Build ID	faylın barmaq izi	linker-in .note.gnu.build-id-ə yazdığı hash; strip olunmuş faylı debug faylı ilə cütləyir	3
SONAME	kitabxananın ABI adı	DT_SONAME, məsələn libc.so.6; linker NEEDED-ə faylın yolunu yox, onu yazır	3

Termin	Sadə izah	Dəqiq tərif	Fəsil
Simvol versiyası	funksiya adındakı versiya etiketi	puts@GLIBC_2.2.5; bir kitabxanada eyni funksiyanın bir neçə versiyası ola bilər	3
Name mangling	C++-ın adı tiplərlə bəzəməsi	Itanium ABI: _Z + uzunluq + ad + tip kodları; extern "C" bunu söndürür	3
extern "C"	C++-a: adı C kimi yaz	mangling-i söndürür; JNI funksiyaları üçün lazımdır	3
TLS (thread-local storage)	hər thread-in öz qlobalı	__thread, .tbss və .tdata, TLS program header; Kotlin-də ThreadLocal	3
Class initialization lock	sinfin init kilidi	JLS 12.4.2: <clinit>-i bir thread işlədir, qalanları gözləyir; dövrə deadlock yaradır	3
Loader lock	dlopen kilidi	dlopen və constructor-lar qlobal kilid altında işləyir; orada başqa thread-i gözləmək deadlock-dur	3