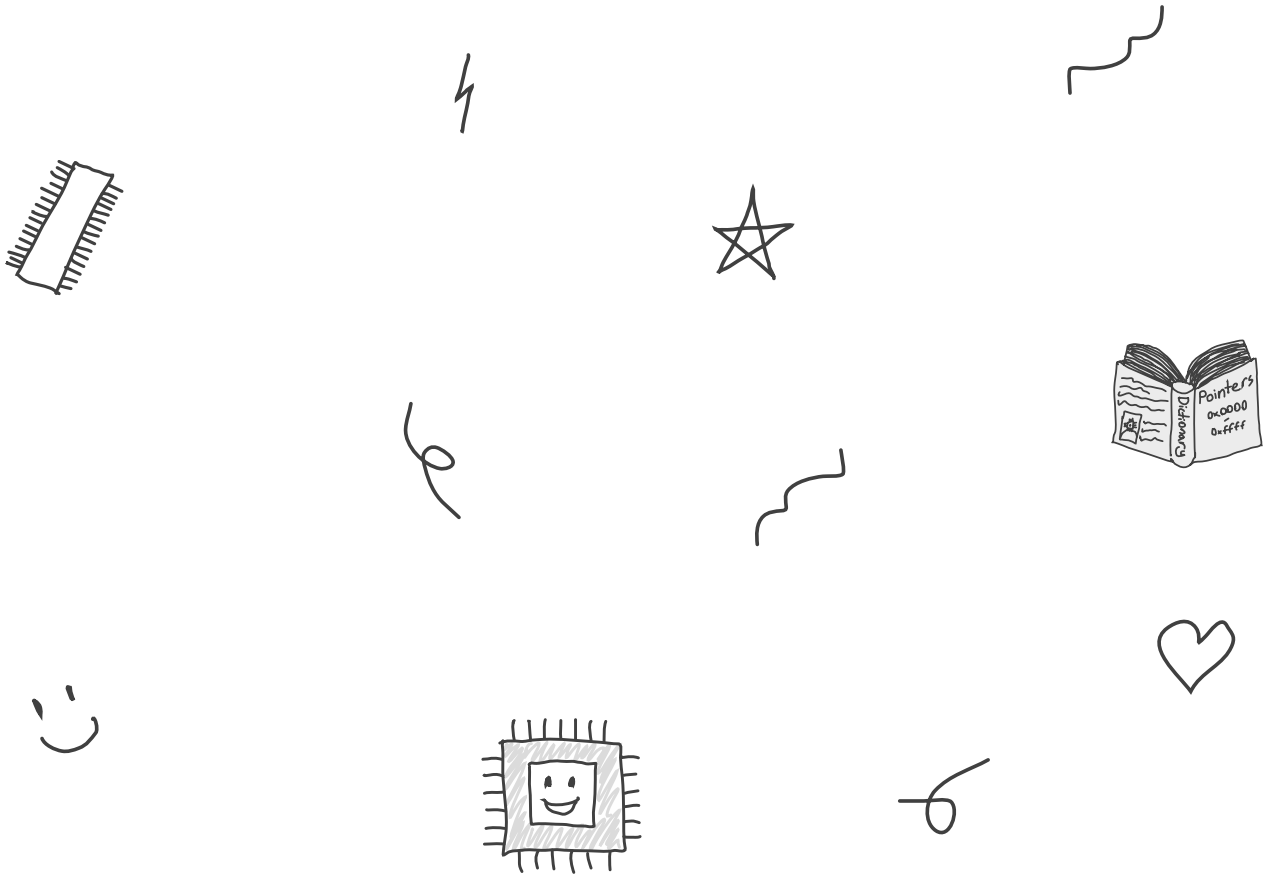
CPU'ya “Sen”i Katmak

Emircan Demir · Ağustos 2026



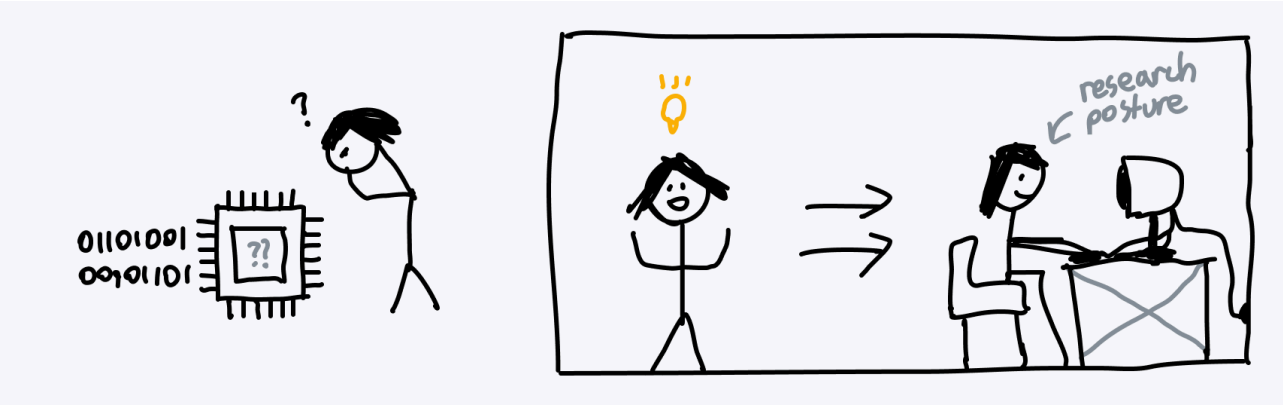
Bölüm 0: Giriş

Bilgisayarında bir program çalıştırdığında tam olarak ne oluyor?

Soru masum görünüyor. Ama cevabı, diskte duran ölü bir dosyadan CPU'nun içindeki milyarlarca transistöre kadar uzanan koca bir zincir — ve o zinciri baştan sona, hiçbir halkasını atlamadan kurabilen şaşırtıcı derecede az insan var. Çoğumuz halkaların bir kısmını gayet iyi biliriz: derleyicinin ne yaptığını, RAM'in ne işe yaradığını, syscall'ın hangi başlık dosyasında durduğunu. Parçaları tek tek bilmek yetmiyor; asıl iş onları doğru sırayla birbirine bağlamak.

Bu kitap tam olarak o bağlantıyı kuruyor. Üç soruyu omurga olarak alıyorum:

- Programlar gerçekten doğrudan CPU üzerinde mi çalışıyor, yoksa arada başka katmanlar mı var?
- Syscall dediğimiz şey tam olarak nedir ve *nasıl* çalışır?
- Tek bir çekirdek aynı anda tek bir talimat yürütebiliyorsa, birden fazla program nasıl aynı anda çalışıyor?



Bu konuyu tek parça hâlinde anlatan derli toplu bir kaynak, özellikle Türkçede neredeyse yok. Var olanlar ya tek bir katmana odaklanıyor — sadece CPU, sadece kernel, sadece derleyici — ya da okuru üç paragraf sonra kaynak kodun ortasında yalnız bırakıyor. Aradaki köprüyü kimse kurmuyor.

Anlatım sırasını buna göre kurdum: her bölüm bir öncekinin bıraktığı yerden başlıyor ve hiçbir kavramı, sana anlatmadan kullanmıyorum. Kitabı bitirdiğinde bilgisayarın açıldığı andan bir programın ilk talimatını yürüttüğü ana kadar olan zinciri kafanda baştan sona kurabiliyor olacaksın. Hedef bu, ölçü de bu.

Her Şeyin Tek Cümlelik Özeti

Bir uygulamaya çift tıkladığında ya da terminalde bir komut çalıştırdığında olay şuna benzer: programın dosyası diskten okunur, bellekte ona kendine ait bir çalışma alanı hazırlanır — o alan yalnızca ona görünür, başka hiçbir program oraya uzanamaz —, CPU o alandaki talimatları tek tek kendi içine çekip yürütür, kernel ise hem kaynakları paylaştıran hem de herkesin kendi sınırında kalmasını sağlayan hakem gibi davranır.

O tek cümlenin her kelimesini teker teker açacağız: CPU talimatları nasıl okur, RAM neden her programa sanki bilgisayardaki tek program oymuş gibi görünür, aynı anda birden fazla program nasıl çalışır, syscall neden gerekir ve Linux bir dosyayı gerçek bir programa nasıl dönüştürür?

Okurken Bilmen Gereken 6 Kavram

Kavram	Tek cümlelik karşılığı
CPU	Talimatları okuyup uygulayan işlemci; bilgisayarın hesap yapan motoru.
RAM	Çalışan programların o anda kullandığı hızlı çalışma masası; elektrik kesildiği anda üzerindeki her şey silinir.
Disk/SSD	Programların ve dosyaların kapalıyken de durduğu kalıcı depo.
Kernel	Donanımı, belleği ve programları yöneten işletim sistemi çekirdeği.
Process	Çalışan bir programın işletim sistemi tarafından takip edilen örneği.
Syscall	Programın kernel'dan dosya okuma, bellek ayırma gibi kontrollü yardım istemesi.

> **Program mı, process mi?**
>
> Kitap boyunca en çok karıştırılan ayrım bu, o yüzden baştan netleştirelim. **Program**, diskte duran ölü bir dosyadır; kimse çalıştırmadığı sürece hiçbir şey yapmaz. **Process** ise o dosyanın çalışır hâle gelmiş, kendi belleği ve kendi durumu olan canlı bir örneğidir. Aynı programdan aynı anda beş process çalıştırabilirsin: beşi de aynı dosyadan doğar, ama beşinin de belleği ayrıdır ve birbirlerinden habersizdirler.

Yol Haritası

Bölüm	Başlık	Ne öğreneceksin?
1	**[Başlamadan Önce]**	Bit, byte, ikilik sistem, hex, negatif sayılar, ASCII, bellek adresi ve assembly
2	**[Temeller]**	CPU, register, instruction pointer ve fetch-execute döngüsü
3	**[Kernel, User Mode ve Syscall]**	Ayrıcalık seviyeleri, interrupt, syscall giriş yolu ve libc sarmalayıcıları
4	**[Mimariler: x86, ARM ve Diğerleri]**	Talimat kümesi ne demek, 32/64 bit farkı, x86-64 ile ARM, CISC ve RISC
5	**[Bellek Hiyerarşisi]**	Register, L1/L2/L3 cache, RAM ve disk piramidi; SRAM ile DRAM farkı
6	**[Cache Nasıl Çalışır]**	Cache line, locality, tag/index/offset, associativity, MESI ve false sharing
7	**[Zamanı Dilimle]**	Timer interrupt, preemption, zaman dilimleri, context switch, EEVDF
8	**[İşlemciyi Hızlandıran Hileler]**	Dennard duvarı, pipeline, hazardlar, superscalar, çok çekirdek, SMT ve SIMD
9	**[Tahmin, Spekülasyon ve Spectre]**	Branch prediction, spekülatif yürütme, yan kanallar, Spectre ve Meltdown
10	**[Bir Program Nasıl Çalıştırılır?]**	execve syscall'ı, kernel'ın binary format dedektifliği, shebang, binfmt_misc

Bölüm	Başlık	Ne öğreneceksin?
11	**[Shell'den Kernel'e]**	Shell, PATH, fork+execve, pipeline, yönlendirme, iş kontrolü ve sinyaller
12	**[Bir ELF Ustasına Dönüşmek]**	ELF formatı, program header, section header, static ve dynamic linking, GOT/PLT
13	**[Bellek Aslında Sanal]**	Sanal adres, sayfa, çerçeve, MMU, izolasyon, higher-half kernel ve ASLR
14	**[Adres Çevirisi ve TLB]**	Çok seviyeli sayfa tabloları, sayfa yürüyüşü, TLB, page fault ve demand paging
15	**[Bellek Güvenliği ve Sertleştirme]**	Buffer overflow, NX, stack canary, ASLR, PIE ve RELRO ile savunma katmanları
16	**[Fork'lar ve COW'lar Hakkında Konuşalım]**	fork syscall'ı, Copy-on-Write, init process, boot zinciri
17	**[Dosya Sistemi ve I/O]**	Her şey dosyadır, file descriptor, VFS, inode, page cache, mmap ve epoll
18	**[Sanal Makineler ve Container'lar]**	Hypervisor, donanım desteği, namespace ve cgroup ile iki farklı izolasyon yolu
19	**[Son Söz]**	Büyük resim, buradan sonra ne okumalı, birkaç küçük not

Başlamadan önce sana üç şey söyleyeyim:

1. **İlk kez okuyorsan sırayla git ve ilk beş bölümü atlama.** Orada kurduğumuz temeli atlarsan sonraki bölümler gereğinden çok daha ağır gelir.
2. **Register, user/kernel mode, syscall ve process kavramları sana tanıdık geliyorsa** doğrudan **[10. bölüme]** geçebilirsin. Tanıdık gelmiyorsa hiç dert etme; hepsini ilk beş bölümde sıfırdan kuruyoruz.
3. **Katlanmış "Derinleşme" panellerini ilk okumada açman gerekmez.** Onlar ana anlatının dışında duruyor; her biri bir konuyu kaynağına kadar takip ediyor ve hiçbirini açmadan kitabın tamamını anlayabilirsin. İkinci okumada ya da merak ettiğin yerde aç.

Bir şeyi ancak başka birine anlatabildiğinde gerçekten anlarsın. Bu kitabın ölçüsü de bu: son sayfayı kapattığında, bilgisayarında bir programın nasıl çalıştığını baştan sona bir başkasına anlatabiliyor olacaksın.

Bölüm 1: Başlamadan Önce

Bilgisayarlar hakkında konuşurken sürekli karşımıza çıkan birkaç kelime var: *bit*, *byte*, *binary*, *hex*... Eğer bunlar sana yabancı geliyorsa endişelenme, bu bölüm tam olarak bunun için var. Sonraki bölümlere atlama; buradaki her kavram bir sonrakinin temeli.

Hepsi tek bir soruya bağlanıyor: bilgisayar neden sadece iki şey sayabiliyor ve bu iki şeyle nasıl her şeyi anlatıyor?

> **Bu bölümde neyi çözüyoruz?**
>
> - Bilgisayarların neden sadece 0 ve 1 anladığını göreceğiz.
> - Bit, byte ve ikilik sistemi pratikleştirip negatif sayıların nasıl tutulduğunu çözeceğiz.
> - Hexadecimal'in neden var olduğunu ve nerede işe yaradığını göreceğiz.
> - Metnin bellekte nasıl sayıya döndüğünü, ASCII ile Unicode farkını anlayacağız.
> - Bellek adresi fikrini ve byte sıralamasını (endianness) kafaya oturtacağız.
> - Assembly ile yüksek seviye diller arasındaki farkı ve derleme zincirini netleştireceğiz.

Neden Sadece 0 ve 1?

Bilgisayarların içinde milyarlarca minik anahtar var: *transistörler*. Bir transistörün yaptığı iş şaşırtıcı derecede basittir — bir devrede akımın geçmesine izin verir ya da vermez. Açık ve kapalı, o kadar.

Bir ucundan bakınca sıradan bir elektrik anahtarı. Ama bu anahtarları belirli kalıplarda birbirine bağladığında ortaya iki tür devre çıkar: **karar verebilenler** (lojik kapılar) ve **durumunu hatırlayabilenler** (bellek hücreleri). Bilgisayar dediğimiz şey, bu iki tür devrenin milyarlarcasının üst üste bindirilmesinden ibaret.

Biz de o iki kararlı duruma bir isim veririz: **0** ve **1**. Tüm hesaplamalar, tüm videolar, tüm oyunlar, hatta şu an okuduğun bu cümle bile — hepsi bu iki durumun kontrollü kombinasyonlarından oluşur.

> Buradaki en yaygın yanılgıyı baştan kapatalım: “bir transistör = bir bit” değildir. Bir transistör temel tuğladır; bit ise o tuğlalarla kurulan devrenin tuttuğu anlamdır. Hızlı ama pahalı olan SRAM’de tek bir bit, birbirini besleyen birkaç transistörden kurulu küçük bir kilitle (latch) tutulur. Ucuz ve yoğun olan DRAM’de ise tek transistör ve minik bir şarj deposu (kapasitör) yeter — bu yüzden DRAM’in içeriğinin sürekli tazelenmesi gerekir. İkisinin farkına **[5. bölümde]** cache’i konuşurken döneceğiz.

> “Milyarlarca” derken abartmıyorum: Apple M3 çipinde yaklaşık 25 milyar transistör var. Hepsi de yukarıda anlattığım tek işi yapıyor — geçir ya da geçirme.

Şimdi bu anahtarlardan mantığın nasıl çıktığına bakalım. Birkaç transistörü öyle bağlarsın ki çıkış ancak *her iki* giriş de 1 iken 1 olur — buna **AND** dersin. Öyle bağlarsın ki *herhangi biri* 1 iken çıkış 1 olur; bu **OR**. Girişi tersine çeviren tek düzenek **NOT**, iki giriş *birbirinden farklıysa* 1 veren düzenek de **XOR**’dur.

Aşağıdaki tabloda her kapının hangi girişe ne cevap verdiğini göreceksin. Aklında tutmanı istediğim şey şu: bilgisayarın yaptığı her şey — toplama, video çözme, bu sayfayı ekrana çizme — bu birkaç kapının milyarlarca kez üst üste bindirilmesinden ibaret.

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \cdot Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \cdot Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

Kaynak: William Stallings, Computer Organization and Architecture, Ch. 11 — Digital Logic, Slide 5.

İki kapının ayrıca not düşmeye değer bir özelliği var.

XOR kendi kendini geri alır. Bir değeri aynı anahtarla iki kez XOR’larsan orijinali geri elde edersin: $A \oplus K \oplus K = A$. Bu küçük özellik, en basit şifreleme yöntemlerinin, grafik programlarındaki “ters çevir” işlemlerinin ve hata tespit algoritmalarının temelidir.

NAND tek başına yeter. NAND (AND'in tersi) evrensel bir kapıdır: yalnızca NAND kapıları kullanarak AND'i de, OR'u da, NOT'u da, dolayısıyla herhangi bir devreyi kurabilirsin. Üretimde bunun karşılığı çok somut: tek tip bir kapıyı milyarlarca kez basmak, farklı tipleri karıştırmaktan hem ucuz hem güvenilirdir.

Bit ve Byte

- **Bit**: En küçük bilgi birimi. 0 veya 1.
- **Byte**: 8 bit'in bir araya gelmesi. 1 byte = 8 bit.

Neden 8? Aslında temiz bir sebebi yok; bu bir tasarım zorunluluğu değil, tarihsel bir uzlaşma.

Hikâye şöyle gelişti. ASCII 7 bit ile 128 karakteri temsil edebiliyordu, yani metin için yedi bit yetiyordu. Birçok sistem kalan sekizinci biti *parity* için ayırdı. Parity fikri çocukça basittir: diğer yedi bitteki 1'lerin sayısı tek mi çift mi, onu söyleyen bir bit eklersin. Hat üzerinde bir bit bozulursa sayı tutmaz ve hatayı yakalarsın. Tek bitlik hatayı fark eder, düzeltemez — ama 1960'ların gürültülü hatlarında bu bile büyük kazançtı.

Sonra IBM 1964'te System/360'ı çıkardı ve 8 bitlik byte'ı kendi mimarisinin merkezine koydu. Bu makine dönemin sektör standardı oldu; ardından gelen donanım da yazılım da bu ölçüye göre büyüyünce, 1 byte = 8 bit artık tartışılmayan bir alışkanlığa dönüştü.

Terim	Değer
1 byte	8 bit
1 kilobyte (kB)	1.000 byte
1 kibibyte (KiB)	1.024 byte
1 megabyte (MB)	1.000.000 byte
1 mebibyte (MiB)	1.024 × 1.024 byte

> Bu kitapta bellek boyutları bağlamında genelde kibibyte (KiB), mebibyte (MiB) gibi ikili tabanlı birimleri kullanacağız. Çünkü bilgisayarlar 2'nin kuvvetleriyle düşünür.

İkilik (Binary) Sistem

Sen on parmağınla saydığın için onluk sistemi doğal buluyorsun: 0'dan 9'a on rakam, sonra basamak atlarsın. Donanımın parmağı yok; elinde güvenilir biçimde ayırt edebildiği yalnızca iki durum var. O yüzden bilgisayar ikilik sistemle sayar: sadece 0 ve 1. Bu iki değere yazılım tarafında **false** ve **true**, devre tarafında düşük ve yüksek voltaj dersin — isim değişir, iş değişmez. Önemli olan voltajın kaç volt olduğu değil, iki durumun birbirine karışmamasıdır.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

Bu sistemi zaten biliyorsun, sadece farkında değilsin. Onluk sistemde `523` yazdığında aslında şunu kastediyorsun: $5\times10^2 + 2\times10^1 + 3\times10^0$. Her basamak, tabanın bir kuvveti. İkilik sistemde değişen tek şey taban: 10 yerine 2.

Yani binary'deki her basamak 2'nin bir kuvvetini temsil eder. Sağdan sola: 2^0, 2^1, 2^2, 2^3...

Örneğin `1011` binary:

- $1 \times 2^3 = 8$
- $0 \times 2^2 = 0$
- $1 \times 2^1 = 2$
- $1 \times 2^0 = 1$

- **Toplam: 11** (decimal)

Peki Eksi Sayılar?

Buraya kadar her şey pozitif. Ama `-5` nasıl saklanıyor? Elimizde eksi işareti diye bir şey yok; yalnızca 0'lar ve 1'ler var.

Akla gelen ilk çözüm en soldaki biti işaret olarak kullanmaktır: 0 ise artı, 1 ise eksi. Mantıklı görünür ve **çalışmaz**. İki sorunu vardır:

1. **İki tane sıfır olur.** `0000 0000` artı sıfır, `1000 0000` eksi sıfır. Birbirine eşit olması gereken iki farklı bit deseni. Her karşılaştırmada bunu ayrıca düşünmek gerekir.
2. **Toplama bozulur.** `5 + (-5)` yapmak istersen devrenin önce işaret bitlerine bakıp "aslında bu bir çıkarma" diye karar vermesi gerekir. Yani ayrı bir çıkarma devresi kurman gerekir.

Gerçekte kullanılan çözüm çok daha zarif: **ikiye tümleyen** (two's complement). Bir sayının negatifini bulmak için iki adım yeter — **tüm bitleri ters çevir, sonra 1 ekle.**

İşlem	8 bit
5	`0000 0101`
bitleri ters çevir	`1111 1010`
1 ekle → −5	`1111 1011`

Şimdi sihri gör. `5 + (-5)` toplamasını sıradan bir toplayıcıya verelim:

```
  0000 0101   (5)
+ 1111 1011   (-5)
-------------
1 0000 0000   → taşan bit atılır, geriye 0000 0000 kalır
```

Sonuç sıfır. Üstelik devre "bunlardan biri negatifmiş" diye bir şey bilmiyor; sadece topladı. **İkiye tümleyenin bütün amacı budur:** çıkarma diye ayrı bir işleme gerek kalmaz, negatifini alıp toplarsın. Donanımda tek bir toplayıcı hem toplamayı hem çıkarmayı yapar.

Bir de sıfır artık tek: `0000 0000`. Karşılığında küçük bir tuhaflık ortaya çıkar — aralık simetrik değildir. 8 bit ile temsil edilebilen sayılar **−128 ile +127** arasındadır. Negatif tarafta bir sayı fazladır, çünkü artık ikinci bir sıfıra yer harcanmıyor.

DERİNLEŞME **O eksik bir sayının gerçek hayattaki bedeli**

Aralığın simetrik olmaması akademik bir ayrıntı gibi durur. Değildir.

`-128`'in mutlak değeri `+128`'dir ve `+128`, 8 bitlik işaretli bir sayıya **sığmaz**. Yani `abs(-128)` sorusu 8 bit dünyasında cevapsızdır. Aynı şey her genişlikte tekrarlanır: 32 bitlik bir tam sayıda `abs(INT_MIN)` tanımsızdır.

Bu, teorik bir merak değil, gerçek bir açık sınıfıdır. Bir programın "uzunluk negatif olamaz, mutlak değerini alayım" diye yazılmış tek satırı, saldırganın `INT_MIN` göndermesiyle negatif kalır ve ardından gelen sınır kontrolünü sessizce geçer.

C ve C++ standartları işaretli tam sayı taşmasını **tanımsız davranış** ilan eder; taşmayı "sarma" olarak varsayan kod, derleyici optimizasyonu açıldığında bambaşka davranabilir. Derleyici "taşma olamaz" varsayımıyla kontrolü tamamen silebilir — ki bu, gerçekte yazılmış güvenlik kontrollerinin optimizasyonla yok olduğu meşhur hataların kaynağıdır.

Modern diller bu sınıfı kapatmayı seçer: Rust hata ayıklama derlemesinde taşmada panik eder, üretimde açıkça `wrapping_*` / `checked_*` demeni ister. Yani "hangi davranışı istediğini söyle" der; varsayım bırakmaz.

Hexadecimal (Onaltılık) Sistem

İkilik sistemin bir bedeli var: yazması ve okuması dayanılmaz derecede uzun. 255 sayısını `11111111` diye yazmak zorunda kalırsın; birkaç basamak sonra gözün kayar ve fazladan bir 1 saydığını fark bile etmezsin.

Bunu çözmek için **onaltılık** (hexadecimal) sisteme geçeriz: 0-9'un devamına A'dan F'ye altı harf ekleriz, böylece tek karakterle 0 ile 15 arasını anlatabiliriz.

Hex'i asıl kullanışlı yapan şey ise şu eşitlik: **16 = 2^4.** Yani her hex karakteri tam olarak 4 bit'e karşılık gelir. Bunun pratik sonucu çok büyük — binary ile hex arasında çevirmek bir hesap işi değil, sadece gruplama işidir:

`1101 0110` → sağdan dörderli böl → `1101` = D, `0110` = 6 → `D6`

Hiç toplama çıkarma yapmadın; sadece grupları okudun. 4 bitlik bu gruba *nibble* denir (byte'ın yarısı olduğu için "kemirik"). Sekizlik sistem (octal) bu yarışı neden kaybetti diye sorarsan: 8 = 2^3'tür ve 3, 8'i tam bölmez — bir byte octal'de iki buçuk basamak eder, ki bu kimsenin işine yaramaz.

Decimal	Binary	Hex
0	0000	0
10	1010	A
15	1111	F
16	0001 0000	10
255	1111 1111	FF

> Bir byte (8 bit) her zaman iki hex karakteriyle gösterilebilir. Bu yüzden bellek dökümleri ve makine kodları genelde hex formatında yazılır.

Sayıları hallettik. Ama bilgisayarın belleğinde sadece sayı yok: metin de var, adresler de var, programın kendisi de var. Şimdi sıra bunların hepsinin aslında yine sayı olduğunu görmekte — aradaki tek fark, o sayıyı nasıl yorumladığın.

ASCII ve Karakterlerin Sayı Hâli

Bilgisayarlar metni nasıl saklar? Her harf, rakam ve sembole bir sayı atanır. En temel standart **ASCII**'dir (American Standard Code for Information Interchange) ve 128 karakter tanımlar.

Bu 128 sayı rastgele dağıtılmamış; tablo üç bölgeye ayrılır:

- **0-31: kontrol karakterleri.** Ekranda görünmezler, iş yaparlar. Satır sonu `\n` = 10, sekme `\t` = 9, string sonunu işaretleyen `\0` = 0.
- **32-126: yazdırılabilir karakterler.** Boşluk 32'den başlar, `~` 126'da biter.
- **127: DEL.** Neden en sonda? Çünkü ASCII delikli bant çağında tasarlandı ve 127 = `1111111`, yani bütün delikler açık demek. Yanlış yazdığın bir karakteri iptal etmenin tek yolu üzerindeki tüm delikleri delmekti; geri alınamayan tek işlem, kodun da en sonuna kondu.

Tabloya biraz dikkatli bakarsan tesadüf olmayan iki şey daha görürsün.

'A' = 65 ve 'a' = 97. Aradaki fark tam olarak 32, yani 2^5. Bu bir tasarım tercihidir: büyük harfi küçük harfe çevirmek, altıncı biti 1 yapmaktan ibarettir. Onlarca yıl boyunca `toupper` ve `tolower` fonksiyonları tam olarak bunu yaptı — tek bir bit işlemi.

'0' = 48. Bu yüzden C'de `karakter - '0'` numarası çalışır: `'7'` karakteri 55'tir, 48'i çıkarınca 7 sayısını elde edersin. Karakterle sayı arasındaki köprü, bir çıkarma işleminden ibaret.

Karakter	Decimal	Hex
'A'	65	41
'a'	97	61
'0'	48	30
' ' (boşluk)	32	20

> ASCII yetersiz kalınca (Türkçe karakterler, Çince vb.) **Unicode** ve **UTF-8** devreye girdi. UTF-8, bir karakteri 1-4 byte arasında değişen uzunlukta temsil eder ve dünya üzerindeki neredeyse tüm yazı sistemlerini destekler.

DERİNLEŞME **UTF-8 neden bu kadar iyi tasarlanmış?**

Unicode, her karaktere evrensel bir numara (code point) atayan bir standarttır. Örneğin ‘A’ = U+0041, ‘ö’ = U+00F6, ’𐍈’ = U+10348. UTF-8 ise bu numaraları byte’lara çevirme yöntemidir.

UTF-8 neden değişken uzunluklu? Çünkü 1,1 milyondan fazla karakteri tek byte’la (0-255) ifade edemezsin. Öte yandan dünyadaki dijital metnin çok büyük kısmı — İngilizce metin, HTML etiketleri, JSON anahtarları, kaynak kodun tamamı — zaten ASCII aralığında kalır. Bunları 4 byte’lık sabit kutulara koymak, her karakter için üç byte’ı boşa harcamak demektir. İşte UTF-8’in zekâsı burada:

- U+0000 - U+007F (ASCII): 1 byte. `0xxxxxxx`
- U+0080 - U+07FF: 2 byte. `110xxxxx 10xxxxxx`
- U+0800 - U+FFFF: 3 byte. `1110xxxx 10xxxxxx 10xxxxxx`
- U+10000 - U+10FFFF: 4 byte. `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

ASCII karakterleriyle tam uyumlu olması, eski C programlarının ve protokollerin çalışmaya devam etmesini sağlar. UTF-8’de ayrıca bir karakterin ilk byte’ına bakarak o karakterin kaç byte süreceğini anlarsın. Devam byte’larının hepsi `10` ile başladığı için, metnin tam ortasından rastgele bir yere düşsen bile geriye doğru birkaç byte gidip karakter sınırını bulabilirsin — buna *self-synchronizing* denir.

UTF-16 ise aynı işi daha kaba yapar: her karakteri 16 bit’le başlatır, ama Unicode 16 bit’e sığmadığı için 65535’in üstündeki karakterlerde iki 16 bitlik birim kullanmak zorunda kalır. Bu ikiliye *surrogate pair* denir. Yani yaygın kanının aksine **UTF-16 sabit genişlikli değildir**; üstelik ASCII ile de uyumlu değildir, iki dünyanın da kötü tarafını alır. Gerçekten sabit genişlikli tek kodlama UTF-32’dir — o da her karaktere koşulsuz 4 byte harcadığı için pratikte neredeyse hiç kullanılmaz.

Endianness: Byte Sıralaması

Şimdiye kadar hep tek byte'lık değerlerle çalıştık. Peki sayı bir byte'a sığmıyorsa? `0x12345678` dört byte tutar ve bellekte peş peşe dört adrese yerleşir. İşte burada, ilk bakışta hiç sorulmaması gereken bir soru çıkıyor ortaya: bu dört byte hangi sırayla dizilecek? Cevabın herkes için aynı olmadığını göreceksin — ve bu tercihin bir adı var: **endianness** (byte sıralaması).

- **Little-endian**: En düşük değerli byte (LSB), en düşük bellek adresinde saklanır. Intel x86 ve x86-64 mimarileri little-endian kullanır.
- **Big-endian**: En yüksek değerli byte (MSB), en düşük bellek adresinde saklanır. Ağ protokolleri (TCP/IP) ve eski Motorola/PowerPC mimarileri big-endian kullanır.

"Endian" kelimesi, Jonathan Swift'in *Gulliver'in Gezileri* kitabından gelir: Yumurtanın hangi ucundan kırılacağı konusundaki savaş. 1981'de Danny Cohen bu terimleri bilgisayar bilimine taşımıştır.

Neden x86 little-endian? Cevap tarihte. Intel'in ilk mikroişlemcisi olan 4004 (1971) 4 bitlik bir çipti ve aritmetiği tıpkı senin kâğıt üzerinde toplama yapman gibi, en sağdaki basamaktan başlayarak yürütüyordu. En düşük anlamlı parçayı önce işlediği için, sayıyı bellekte de en düşük anlamlı byte önde tutmak işleri kolaylaştırdı. Bir yıl sonraki 8008 bu tercihi devraldı, 8080 ondan devraldı ve zincir bugünkü x86-64'e kadar hiç kırılmadı.

Bunun pratik bir faydası da var. Elle toplama yaparken en sağdaki basamaktan başlar, eldeyi sola doğru taşırsın; işlemci de aynısını yapar. Little-endian'da en düşük anlamlı byte zaten en düşük adreste durduğu için, işlemci sayıyı baştan okumaya başlayıp eldeyi ilerledikçe taşıyabilir — sayının kaç byte olduğunu önceden bilmesine bile gerek kalmaz.

Bir örnek: 32 bitlik `0x12345678` sayısı bellekte nasıl durur?

Bellekte byte sıralaması

```
Adres     Big-endian    Little-endian
0x1000    12            78
0x1001    34            56
0x1002    56            34
0x1003    78            12
```

Dikkat et: temsil edilen sayı ikisinde de aynı (`0x12345678`) ve kullanılan adresler de aynı. Değişen tek şey hangi byte'ın hangi adrese düştüğü. Yani bellekteki byte'lar gerçekten farklı yerlerde duruyor; bu bir okuma tercihi değil, fiziksel bir yerleşim farkı.

Bunu bilmezsen bir hex dump'a bakıp `78 56 34 12` görürsün ve sayıyı ters okursun — hata ayıklarken en çok vakit yakan şeylerden biri budur. Ağ tarafında ise herkesin anlaşabilmesi için tek bir sıra seçilmiştir: big-endian, yani *network byte order*. x86 makinen little-endian olduğu için bir sayıyı ağa koymadan önce byte'larını çevirmek zorundadır; `htons` ve `htonl` gibi fonksiyonların tek işi budur.

Bellek Adresi: Her Şeyin Bir Numarası Var

Bilgisayarın belleği (RAM), byte'ların yan yana dizildiği dev bir dizi gibidir. Her byte'ın bir **adresi** (numarası) vardır ve numaralar 0'dan başlar.

Adreslerin nereye kadar gidebileceğini belirleyen şey ise — burası çok sık yanlış bilinir — RAM'inin boyutu değil, işlemcinin adres genişliğidir. 64 bitlik bir makinede adres için 64 bit ayrılmıştır. Bu, RAM'inden kat kat büyük bir numara aralığı demek; o aralığın tamamının karşılığında fiziksel bir RAM hücresi olmak zorunda da değil. Bu tuhaflığın nasıl çözüldüğüne **[13. bölümde]** geleceğiz.

Şimdilik temel resmi şöyle tut: bellek, numaralanmış byte'lardan oluşan dev bir dizidir.

Benzetmeyle bakalım: bir apartman düşün ve her daire bir byte olsun. Üç ayrıntı önemli.

Daireler eşit büyüklükte. Bir byte'lık daireye 32 bitlik bir sayı sığmaz; o sayı dört ardışık daireyi kaplar. Biraz önce konuştuğumuz endianness meselesi tam olarak "hangi parça hangi daireye girecek" sorusudur.

CPU kapı kapı dolaşmaz. Doğrudan numarayı söyler ve o daireye anında ulaşır. RAM'deki "R" harfi buradan geliyor: *Random Access*, yani rastgele erişim. 5.000'inci daireye erişmek, 1'inci daireye erişmekle tam olarak aynı sürer. Kasete ya da plağa göre devrimsel olan şey buydu.

Belleğin umurunda değildir. Bir daireye ne koyduğun — harf mi, sayı mı, talimat mı — bellek açısından hiçbir anlam taşımaz; hepsi sekiz bittir. O sekiz bitin ne anlama geldiği tamamen CPU'nun onu nasıl yorumladığına bağlıdır. **[2. bölümde]** göreceğimiz gibi, bu kayıtsızlık bir kusur değil, bilgisayarın en temel tasarım kararlarından biridir.

Adres	Değer (Hex)	Değer (ASCII)
0x1000	48	'H'
0x1001	65	'e'
0x1002	6C	'l'
0x1003	6C	'l'
0x1004	6F	'o'

> Adresler genelde `0x` önekiyle hexadecimal olarak yazılır. `0x1000` = 4096 decimal.

Makine Kodunu Gözlemlemek

Şimdiye kadar öğrendiklerimizi terminalde doğrulayabiliriz. `xxd`, `hexdump` ve `od` gibi araçlar dosyaları ham byte düzeyinde gösterir.

`xxd`, hex dump için en yaygın kullanılan araçtır. Sol tarafta offset (adres), ortada hex byte'lar, sağda ASCII karşılığı görünür:

Shell oturumu

```
echo -n "AB" | xxd
```

Çıktısı şu şekilde olur:

xxd çıktısı

```
00000000: 4142                                     AB
```

`41` hex = 65 decimal = 'A', `42` hex = 66 decimal = 'B'. `echo` normalde çıktının sonuna bir satır sonu karakteri (`\n`, hex `0a`) ekler; `-n` bunu engeller. Bu yüzden dökümde yalnızca iki byte görüyorsun. `-n` olmadan denersen üçüncü byte olarak `0a` belirir — dene ve gör.

Bir binary dosyanın içindeki makine talimatlarını `objdump` ile assembly olarak okuyabiliriz. `/bin/ls` gibi yaygın bir programın başındaki talimatlar:

Shell oturumu

```
objdump -d /bin/ls | head -20
```

objdump çıktısı

```
0000000000004000 <.init>:
    4000:       f3 0f 1e fa             endbr64
    4004:       48 83 ec 08             sub    $0x8,%rsp
    400f:       48 85 c0                test   %rax,%rax
    4012:       74 02                   je     4016 <...>
```

Soldaki adresler (`4000`, `4004`...), ortadaki makine kodu (`f3 0f 1e fa`), sağdaki assembly (`endbr64`) aynı talimatın üç farklı temsilidir. Bellekte saklanan şey ortadaki byte'lardır; CPU bunları çözüp sağdaki gibi davranır.

Assembly ve Yüksek Seviye Diller

Buraya kadar makine kodunu hep byte olarak gördük. O byte'ları insanın elle yazması işkencedir; bu yüzden her talimata okunabilir bir isim verilmiştir. **Assembly** tam olarak budur: makine kodunun bire bir insan okunur karşılığı. Sen `mov` yazarsın, assembler onu `48 c7` gibi byte'lara çevirir — arada kaybolan ya da eklenen hiçbir şey yoktur.

Assembly'nin tek bir "dili" de yoktur. Talimat isimleri doğrudan CPU'nun anladığı komutlar olduğu için, x86'nın assembly'si ile ARM'ınki birbirinden bağımsız iki dildir; RISC-V'ninki bir üçüncüsü.

Aşağıdaki örnekte `rax` gibi isimler göreceksin. Bunlara **register** denir: CPU'nun içinde bulunan ve o an üzerinde çalıştığı birkaç değeri tuttuğu isimli kutulardır. RAM'den iki farkı var — RAM'de milyarlarca adres varken register sayısı on-yirmi civarındadır ve register CPU'nun *içinde* olduğu için erişimi kıyaslanamayacak kadar hızlıdır. **[2.]** ve **[5. bölümde]** ikisine de ayrıntılı döneceğiz.

hello.asm

```
; x86-64 assembly örneği
mov rax, 5      ; rax register'ına 5 yaz
add rax, 3      ; rax'a 3 ekle
```

Yüksek seviye diller (C, Python, Java, JavaScript) insanlar için çok daha kolaydır. Aradaki köprüyü ise bir program kurar ve burada sık karıştırılan bir ayrım var.

Derleyici, kodun tamamını önceden makine koduna çevirir ve ortaya çalıştırılabilir bir dosya çıkarır; C böyle çalışır. **Yorumlayıcı** ise hiçbir şeyi makine koduna çevirmez — programını satır satır okuyup gereken işi kendisi yapar. `python script.py` yazdığında ortada makine koduna dönüşmüş bir dosya yoktur; çalışan şey Python yorumlayıcısının kendisidir.

(Modern yorumlayıcıların sık çalışan parçaları çalışma anında makine koduna çevirdiği bir üçüncü yol daha var: JIT. Ona şimdilik girmiyoruz.)

hello.c

```
// C ile aynı işlem
int x = 5;
x = x + 3;
```

> Bir C programı derlendiğinde, önce assembly'ye, oradan da makine koduna (binary) dönüştürülür. Bu makine kodu, CPU'nun talimat olarak okuduğu 0 ve 1 dizisidir.

C’den Makine Koduna: Derleme Pipeline’ı

`gcc hello.c -o hello` yazdığında aslında tek bir araç çalışmaz; bir **pipeline** (boru hattı) devreye girer. C kaynağı makine koduna dört aşamada dönüşür:

1. Önişlemci (Preprocessor)

`#include`, `#define` gibi direktifleri çözer; header dosyalarını içeri aktarır, makroları genişletir, yorumları siler. C dilinin syntax’ını bilmez; sadece metin değiştirme motorudur.

```
gcc -E hello.c -o hello.i
```

Sonucu `wc -l` ile saydırırsan şaşırırsın: beş satırlık bir `hello.c`, sekiz yüz satırı aşan bir `hello.i` üretebilir. Çünkü `<stdio.h>` tek başına yüzlerce satır tanım getirir ve o da başka header’ları içeri çeker.

2. Derleyici (Compiler)

İkinci aşamada asıl iş yapılır: derleyici, önişlemciden çıkan `hello.i` dosyasını okur ve hedef mimarinin assembly diline çevirir. Gördüğün derleme hatalarının neredeyse tamamı buradan çıkar — tip uyuşmazlıkları, tanımlanmamış değişkenler, eksik `return`’ler. `-O2` yazdığında optimizasyon da burada devreye girer. Kısacası C bilgisi gerektiren ne varsa bu aşamada olur; bundan sonraki iki aşama artık C’nin ne olduğunu bilmez.

Shell oturumu

```
gcc -S hello.i -o hello.s
cat hello.s
```

hello.s

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $5, -4(%rbp)
        addl    $3, -4(%rbp)
        ret
```

> `-S` bayrağı (büyük S) derleyiciyi assembly ürettikten sonra durdurur; daha sonraki aşamalara geçmez.

3. Assembler

Assembly metnini (`hello.s`) ham makine koduna çevirir. Çıktı bir **object file** (`hello.o`) olur; bu dosya binary talimatlar içerir ama henüz çalıştırılabilir değildir. Dışarıdan çağrılan fonksiyonların (örn. `printf`) adresleri henüz boştur.

```
gcc -c hello.s -o hello.o
```

`file hello.o` çalıştırırsan çıktıda `relocatable` kelimesini görürsün. Anahtar kelime odur: dosya makine kodu içerir ama henüz bir adrese yerleşmemiştir.

4. Bağlayıcı (Linker)

Son aşamada bağlayıcı, ortalıkta duran parçaları birleştirir. Senin `hello.o` dosyanı `libc` gibi hazır kütüphanelerle yan yana koyar; bir önceki aşamada boş bırakılan `printf` adresini bulup yerine yazar — buna **sembol çözümleme** denir — ve her şeyi son adreslerine oturtur. Ortaya artık eksiği olmayan, tek başına çalıştırılabilir bir dosya çıkar.

```
gcc hello.o -o hello
```

Şimdi `file hello` çıktısında `relocatable` yerine `executable` yazar. Aradaki tek fark, sembollerin çözülmüş ve adreslerin sabitlenmiş olmasıdır — **[12. bölümde]** bu iki ELF türünün farkına ayrıntılı bakacağız.

> Kendi makinende `executable` yerine `pie executable` ya da hatta `shared object` görürsen şaşırma. Sebebi, modern derleyicilerin varsayılan olarak konumdan bağımsız (PIE) çalıştırılabilir üretmesi; dosya yine çalıştırılabilirdir. `gcc -no-pie` ile denersen düz `executable` yazdığını görürsün. Bu ayrımın neden var olduğuna **[12. bölümde]** döneceğiz.

Günlük hayatta bu dört aşamayı tek tek çalıştırmazsın; `gcc hello.c -o hello` yazar geçersin. Ama aşamaları bilmenin çok somut bir getirisi var: bir hata mesajı geldiğinde onun hangi aşamadan çıktığını anlarsın. `No such file or directory` önişlemcidendir — bir header'ı bulamamıştır. `undefined reference to 'printf'` ise bağlayıcıdan gelir; kod gayet iyi derlenmiştir, sadece fonksiyonun gövdesi bulunamamıştır. İkisi bambaşka sorunlardır ve karıştırmak saatler yakar.

Dış Kaynaklar

- ASCII — Vikipedi
- Unicode — Vikipedi
- Endianness — Vikipedi
- Transistör — Vikipedi
- UTF-8 ve Unicode Standardı — unicode.org

Özet

> **Peki, ne öğrendik?**
>
> - ASCII 128 karakterle yetinir; dünya dillerinin tamamı için Unicode ve UTF-8 gerekir.
> - UTF-8 değişken uzunlukludur ve ASCII ile geriye dönük uyumlu olacak şekilde tasarlanmıştır.
> - Çok byte'lı sayılar bellekte **little-endian** (x86) veya **big-endian** (ağ) sırayla saklanabilir.
> - Her byte'ın bellekte bir adresi vardır; hex dump araçlarıyla bu adresleri ve içerikleri gözlemleyebilirsin.
> - Assembly CPU'ya yakındır, yüksek seviye diller insana. İkisi de sonunda aynı makine koduna iner.
> - C kodu derlenirken önişlemci → derleyici → assembler → bağlayıcı zincirinden geçer.
> - Hepsi aynı şeye çıkar: makine kodu, yani anlamı yorumlayana bağlı bir sayı dizisi.

Bu temelleri attıktan sonra bilgisayarın beyni olan CPU'yu derinlemesine inceleyebiliriz. **[Sonraki bölümde]** CPU'nun nasıl talimat çekip çözdüğünü göreceğiz. Daha ileride ise **[ELF formatı ve makine kodunun belleğe nasıl yerleştiğini]** adım adım çözüp, gerçek bir programın hayatını baştan sona takip edeceğiz.

Bölüm 2: Temeller

Bu kitabı yazarken beni tekrar tekrar şaşırtan şey, bilgisayarların ne kadar *basit* olduğuydu. İnsanın aklı bilgisayarların içinde, gerçekte olduğundan daha karmaşık ve daha soyut bir yapı arıyor; öyle bir yapı yok. Devam etmeden önce aklına kazımanı istediğim tek bir şey varsa o da şu: basit görünen birçok şey gerçekten de basittir. Bu sadelik hem çok güzel hem de zaman zaman epey lanetlidir.

Bilgisayarının özünde nasıl çalıştığına dair temel resimle başlayalım.

> **Bu bölümde neyi çözüyoruz?**
>
> - CPU'nun talimatları nasıl okuyup yürüttüğünü göreceğiz.
> - RAM, disk, register ve instruction pointer arasındaki farkı netleştireceğiz.
> - "Program" ile "process" arasındaki farkı yerine oturtacağız.
> - Bilgisayar açıldığında ilk talimatın nereden geldiğini çözeceğiz.

Bilgisayarlar Nasıl Tasarlanır?

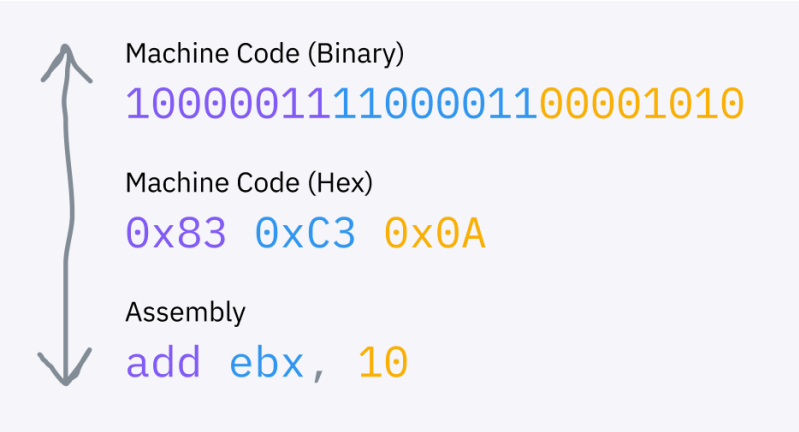
Bir bilgisayarın *merkezi işlem birimi* (CPU), bütün hesaplamalardan sorumludur. İşin patronu odur. Makine açıldığı anda çalışmaya başlar ve talimat üstüne talimat yürüterek durmadan devam eder.

Bir programı başlatmadan önce o programın dosyası kalıcı depoda durur: sabit diskte ya da SSD'de. Kernel onu çalıştırmaya karar verdiğinde dosyanın gerekli parçalarını RAM'e taşır. Şimdilik temel resmi şöyle tut: **disk kalıcı depo, RAM çalışma masası, CPU da masadaki talimatları uygulayan motor.**

> Bu resmin ileride iki yerinden inceleceğini şimdiden söyleyeyim. Birincisi, CPU talimatları çoğu zaman RAM'e hiç gitmeden, ona çok daha yakın duran küçük ara depolardan alır. İkincisi, bir programın gördüğü adresler ile RAM'deki gerçek adresler aynı şey değildir. İkisini de yeri geldiğinde ayrı ayrı açacağız; şimdilik basit resim yeterli.

Bu fikrin ne kadar eski olduğunu görmek için bir örnek vereyim: Intel'in Kasım 1971'de duyurduğu Intel 4004. Ne kadar küçük olduğunu bir rakamla anlatayım — 4004 aynı anda yalnızca 4 bit işleyebiliyordu, yani senin bugün kullandığın işlemcinin tek hamlede çevirdiği verinin on altıda birini. Ama içindeki fikir bugünküyle birebir aynıydı: programlanabilir bir çip, talimatları sırayla yürütür. Bu kitapta anlatacağım her şey o tek fikrin üstüne kuruldu.

CPU'nun yürüttüğü "talimatlar" sadece ikili veridir: önce hangi talimatın çalıştırıldığını belirten bir ya da birkaç baytlık opcode gelir, ardından o talimatın ihtiyaç duyduğu veri yer alır. *Makine kodu* dediğimiz şey, aslında bu ikili talimatların art arda dizilmesinden ibarettir. Assembly, insanların ham bitlere göre çok daha rahat okuyup yazabildiği bir gösterimdir; ama sonuçta her zaman CPU'nun anlayacağı ikili biçime derlenir.



> Bir not: Talimatlar makine kodunda her zaman yukarıdaki örnekteki gibi 1:1 görünmez. Örneğin `add eax, 512`, `05 00 02 00 00` şeklinde kodlanır.
>
> İlk bayt (`05`), özellikle *EAX register'ına 32-bit bir sayı ekleme* işlemini ifade eden opcode'dur. Kalan baytlar ise little-endian sıra ile yazılmış 512 (`0x200`) değeridir.
>
> Defuse Security, assembly ile makine kodu arasındaki çeviriyle oynamak için yararlı bir araç hazırlamış.

RAM, bilgisayarının ana belleğidir; çalışan programların kullandığı verilerin tutulduğu büyük ve genel amaçlı alan budur. Buna program kodunun kendisi de, işletim sisteminin kernel kodu da dahildir. CPU'nun yürüteceği talimatların RAM'de erişilebilir olması gerekir; yalnızca diskte duran bir dosya kendi kendine çalışmaz.

Burada, kitabın ilerleyen bölümlerinde birkaç kez karşımıza çıkacak bir ayrım var: **RAM uçucudur, disk kalıcıdır.**

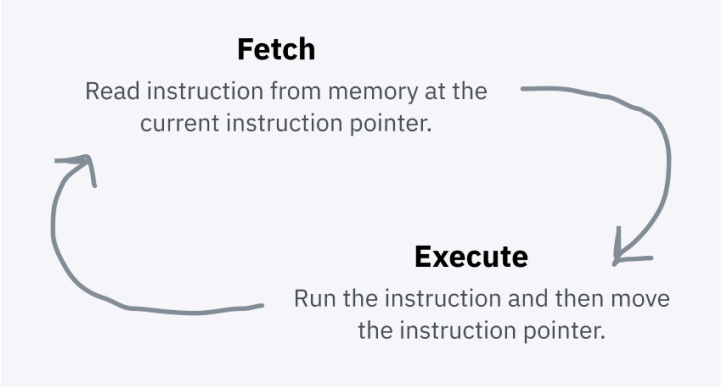
Sebebi fizikte. RAM'in her biti, minik bir kondansatörde tutulan elektrik yüküdür ve o yük kendiliğinden boşalır — o kadar hızlı boşalır ki, makine çalışırken bile her hücrenin saniyede binlerce kez yeniden doldurulması gerekir. Buna *refresh* denir ve RAM'in "D"si (Dynamic) tam olarak buradan gelir. Elektrik kesildiği anda refresh durur, yükler kaçar, veri gider. Disk ise veriyi manyetik yönelim ya da hapsedilmiş elektron olarak saklar; elektriğe ihtiyaç duymaz.

Gündelik hayatta bunun iki tanıdık sonucu var. Birincisi, kaydetmediğin dosyanın elektrik kesintisinde uçup gitmesi. İkincisi, her açılışta bilgisayarın "yeniden yükleniyor" olması — çünkü RAM'de gerçekten hiçbir şey kalmamıştır ve her şeyin diskten baştan taşınması gerekir. Bölümün ilerleyen kısmında, "peki RAM boşsa ilk talimat nereden geliyor?" sorusuna da döneceğiz.

CPU, sıradaki talimatın nerede olduğunu takip etmek için *instruction pointer* adında bir gösterge tutar. Burada karıştırması çok kolay bir ayrım var, o yüzden baştan netleştirelim:

Göstergenin kendisi CPU'nun içindedir; gösterdiği adres RAM'dedir. İkisi aynı yerde değil. Bir kitap okurken parmağını satırların üzerinde gezdirmen gibi düşün — parmak sende, satırlar kitapta. CPU bir talimatı bitirince parmağını bir sonrakine kaydırır ve aynı işi baştan yapar. Buna *fetch-execute cycle* denir.

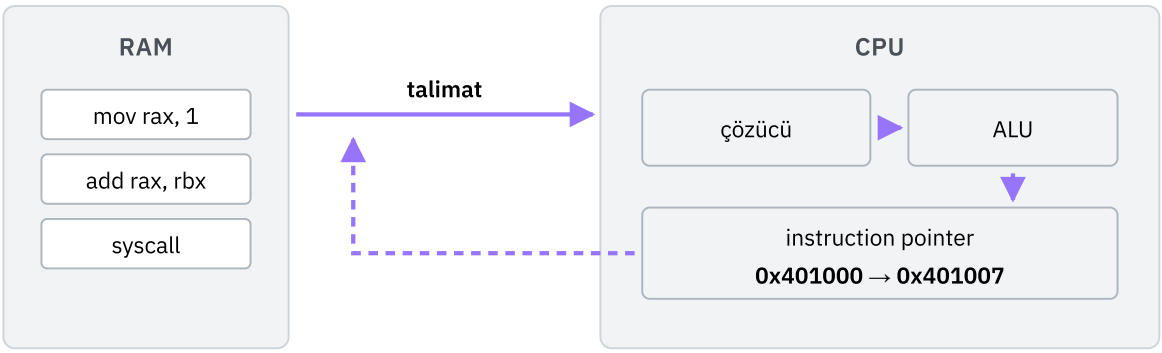
Bu ayrım küçük bir ayrıntı gibi görünüyor ama kitabın geri kalanının yarısı buradan çıkıyor. Göstergenin CPU'nun içinde olması, ona erişmenin bedava sayılacak kadar ucuz olması demek. Gösterdiği yerin RAM'de olması ise, oraya gitmenin pahalı olması demek. Aradaki bu uçurum, **[5. bölümde]** göreceğimiz bütün bellek hiyerarşisinin ve **[8. bölümdeki]** hız hilelerinin var olma sebebidir.



Bir talimat yürütüldükten sonra instruction pointer, RAM'de o talimatın hemen sonrasına ilerler; böylece sıradaki talimatı işaret eder. Kodun çalışması dediğimiz şey tam olarak budur. Talimatlar bellekte hangi sıradaysa CPU onları o sırayla yürütür. Bazı talimatlar ise instruction pointer'a başka bir adrese atlamasını söyler; böylece dallanma, koşullu mantık ve tekrar kullanılabilir kod mümkün olur.

Döngünün dört adımını tek tek izlemek istersen:

FETCH-EXECUTE DÖNGÜSÜ



1. **Getir (fetch).** CPU, instruction pointer'ın gösterdiği adresteki talimatı RAM'den okur.
2. **Çöz (decode).** Okunan byte'lar çözülür: hangi işlem, hangi register'lar, hangi sabit?
3. **Yürüt (execute).** İşlem gerçekten yapılır; sonuç bir register'a ya da belleğe yazılır.
4. **İlerlet.** Instruction pointer bir sonraki talimata kayar ve döngü baştan başlar. Bir atlama talimatı bunu başka bir adrese de çevirebilir.

> Pratikte CPU çekirdeği ile DRAM arasında L1 → L2 → L3 gibi cache katmanları bulunur. Programcıya görünen basit model "bellekten oku/yaz" olsa da donanım sık kullanılan veri ve talimatları CPU'ya yakın tutarak gecikmeyi azaltır; bu ayrıntı temel fetch-execute resmini değiştirmez.

Burada, fark etmesi kolay olmayan ama her şeyin üstünde durduğu bir tasarım kararı var: **talimatlar ve veriler aynı bellekte duruyor.** CPU'nun okuduğu makine kodu ile üzerinde çalıştığı sayılar aynı RAM'de, aynı adres uzayında, aynı byte'lardan yapılmış. Bir programın *dosya* olarak diskte durabilmesi, kopyalanabilmesi, indirilebilmesi tamamen bu karara bağlı.

DERİNLEŞME **Von Neumann kararı ve bugüne kadar uzanan faturası**

Talimat ile veriyi aynı belleğe koymak bir zorunluluk değil, bir tercihti. 1945'te John von Neumann'ın raporuyla anılan bu modele karşı bir alternatif de vardı ve hâlâ var: **Harvard mimarisi**, talimat belleği ile veri belleğini fiziksel olarak ayırır.

İkisinin sonuçları taban tabana zıt:

- **Von Neumann** esnektir. Program bir veridir; derleyici üretebilir, indirebilirsin, çalışırken üretebilirsin. JIT derleyiciler, dinamik yükleme ve bu kitabın anlattığı her şey bu esneklikten doğar.
- **Harvard** hızlı ve güvenlidir. Talimat ile veri ayrı yollardan gittiği için aynı anda ikisine birden erişilebilir; ve bir veri hatası kodun üzerine yazamaz. Mikrodenetleyiciler ve sinyal işlemcilerinde bugün de tercih edilir.

Modern bir x86 ya da ARM çekirdeği ikisinin arasında durur: mimari düzeyde von Neumann'dır (tek adres uzayı), ama çekirdeğin içinde ayrı bir talimat cache'i (L1i) ve ayrı bir veri cache'i (L1d) vardır. Buna *modified Harvard* denir. Bunun gözlemlenebilir bir sonucu şudur: kendi kodunu çalışırken üreten bir program (bir JIT gibi), yazdığı byte'ların L1d'de olup L1i'de olmadığını hesaba katmak ve talimat cache'ini açıkça geçersiz kılmak zorundadır. Aksi hâlde CPU eski talimatları çalıştırmaya devam eder.

İki büyük fatura daha var:

Von Neumann darboğazı. Talimat ve veri aynı yolu paylaştığı için, işlemcinin ne kadar hızlı çalışabileceğini o tek yolun kapasitesi sınırlar. Terimi 1977'de John Backus ortaya attı ve şu gözleme dayanıyordu: CPU ne kadar hızlanırsa hızlansın, veriyi belleğe götürüp getiren tek bir borudan geçmek zorundadır. Bunun üstüne DRAM'in işlemciler kadar hızlanamaması eklenince aradaki uçurum daha da açıldı. **[5. bölümdeki]** cache hiyerarşisi ve **[8. bölümdeki]** pipeline hilelerinin tamamı, bu iki basıncın etrafından dolaşma çabasıdır.

Kod enjeksiyonu diye bir şeyin var olabilmesi. Bir saldırgan veri alanına byte yazıp CPU'yu oraya atlatabiliyorsa, bunun mümkün olmasının sebebi verinin de talimat olabilmesidir. **[15. bölümde]** göreceğimiz NX biti ve W^X politikası, tam olarak

> Harvard'ın ayrımını von Neumann belleğine sonradan geri takma çabasıdır. Seksen yıl sonra hâlâ o kararın bedelini ödüyoruz.

Bu instruction pointer bir *register* içinde tutulur. Register'lar, CPU'nun çok hızlı okuyup yazabildiği küçük depolama alanlarıdır — ve "küçük" derken gerçekten küçük kastediyorum. Birkaç sayıyla bakalım:

- **Sayı ve boyut.** x86-64'te 16 genel amaçlı register vardır ve her biri 64 bit, yani 8 bayt tutar. Toplamı 128 bayt. Aynı makinede RAM gigabaytlarla ölçülüyor — arada milyarda bir mertebesinde bir oran var.
- **Neden bu kadar hızlı?** Çünkü register CPU'nun *içinde*, hesap yapan devrelerin hemen yanındadır; aynı silikon parçasının üzerinde. RAM ise anakart üzerinde ayrı bir yongadır ve oraya gidip gelmek, elektrik sinyalinin fiziksel olarak yol katetmesi demektir. Register erişimi bir saat çevrimi sürer, RAM erişimi yüzlerce.
- **İsimler neden bu kadar tuhaf?** `rax`, `eax` ve `ax` üç ayrı register değil; aynı register'ın farklı boylardaki görünümleridir. `rax` 64 bitin tamamı, `eax` alttaki 32 bit, `ax` alttaki 16 bit, `al` ise en alttaki 8 bit. Bu iç içe geçmişlik geriye uyumluluktan geliyor: 16-bit çağında yazılmış kod, 64-bit bir işlemcide hâlâ çalışsın diye.

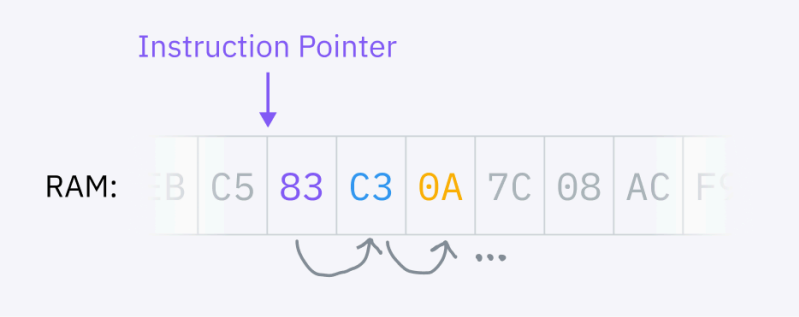
Yukarıdaki assembly örneğinde gördüğün `eax` ve `ebx` gibi genel amaçlı register'lara makine kodundan doğrudan erişilebilir.

Bazı register'lar ise CPU tarafından daha içsel amaçlarla kullanılır; yine de çoğu zaman özel talimatlarla okunabilir veya güncellenebilirler. Instruction pointer buna iyi bir örnektir: doğrudan okunmaz ama örneğin bir jump talimatı ile değiştirilebilir.

İşlemciler Naiftir

Asıl soruya dönelim: Bilgisayarında çalıştırılabilir bir programı başlattığında ne oluyor? Önce onu çalıştırmaya hazırlamak için bir sürü kurulum yapılır, bunların hepsini birazdan göreceğiz, ama sonunda dosyanın içindeki makine kodu RAM'e yerleştirilir. Ardından işletim sistemi CPU'ya instruction pointer'ı o konuma ayarlamasını söyler. CPU da fetch-execute cycle'ı normal şekilde sürdürür ve program çalışmaya başlar.

Burada bir saniye durmanı istiyorum. Bu satırları okumak için kullandığın program da tam olarak böyle çalışıyor. Tarayıcının makine kodu şu anda RAM'de duruyor; CPU'n instruction pointer'ı o kodun üzerinde bir adım bir adım ilerletiyor ve okuduğun cümle, saniyede milyarlarca kez tekrarlanan bu tek döngünün çıktısı. Sihir yok; sadece çok hızlı tekrarlanan çok basit bir iş var.



Burada bir terimi de yerine oturtalım, çünkü kitabın geri kalanında sürekli kullanacağız. **Program**, diskte duran dosyadır; kimse çalıştırmadıkça hiçbir şey yapmaz. **Process** ise o dosyanın çalışan hâlidir: kendi belleği, kendi açık dosyaları ve kendi instruction pointer değeri olan canlı bir örnek. Aynı programı iki kez açtığında ortada tek bir dosya ama iki process olur.

Şimdi hükmü verebiliriz: CPU'ların dünya görüşü inanılmaz derecede dardır. Yalnızca o andaki instruction pointer'ı ve biraz da iç durumu görürler. *Process* tamamen bir işletim sistemi soyutlamasıdır; CPU'nun doğal olarak bildiği ya da takip ettiği bir kavram değildir.

process diye bir şey yok; ~~işletim sistemi geliştiricileri~~ daha çok bilgisayar satılsın diye uydurulmuş bir hikâye bu

Bu kadar dar bir dünya görüşü, hemen üç soru doğuruyor. Kitabın geri kalanı büyük ölçüde bunların cevabı:

1. CPU çoklu işlemeyi bilmiyorsa ve sadece talimatları sırayla yürütüyorsa, neden tek bir programın içinde sıkışıp kalmıyor? Birden fazla program aynı anda nasıl çalışabiliyor?
2. Programlar doğrudan CPU üzerinde çalışıyorsa ve CPU doğrudan RAM'e erişebiliyorsa, neden başka process'lerin belleğine, hele hele kernel belleğine erişemiyorlar?
3. Madem konu açıldı: Her process'in istediği talimatı çalıştırıp bilgisayarına istediğini yapmasını engelleyen şey ne? Ve syscall denen şey tam olarak nedir?

Bellek meselesi kendi bölümünü hak ediyor; onu **[Bölüm 13]**'da ele alacağız. Kısa versiyon şu: user-space programı kendi sanal adres alanına erişir; fiziksel RAM'e, kernel belleğine ve ayrıcalıklı donanım giriş/çıkışına doğrudan el uzatamaz. Şimdilik bellek koruması yokmuş ve bilgisayar aynı anda sadece tek bir process çalıştırıyormuş gibi sadeleştirelim. Bu varsayımların ikisini de birazdan bozacağız.

Şimdi ilk derin dalışımızın zamanı: syscall'lar ve ring'ler.

> **Bu arada kernel nedir?**
>
> Bilgisayarındaki macOS, Windows ya da Linux gibi işletim sistemi, temel işlerin yürümesini sağlayan bütün yazılım katmanıdır. "Temel işler" çok muğlak bir ifade ve "işletim sistemi" terimi de kime sorduğuna göre değişir; kimi insanlar buna varsayılan uygulamaları, font'ları ve ikonları da katar.
>
> Ama kernel, işletim sisteminin çekirdeğidir. Bilgisayarı açtığında çalışan ilk şey kernel değildir: önce anakart üzerindeki firmware (BIOS ya da UEFI) donanımı ayağa kaldırır, sonra bir bootloader (GRUB gibi) kernel'ı diskten bulup RAM'e yükler ve kontrolü ona devreder. O andan sonra makinenin sahibi kernel'dır. Belleğe, çevre birimlerine ve sistem kaynaklarına neredeyse tam erişimi vardır ve user-space programlarını çalıştırmak onun işidir. Kitap boyunca kernel'ın bu erişime nasıl sahip olduğunu, programların neden olmadığını göreceğiz.
>
> Linux tek başına bir kernel'dır; kullanılabilir bir sistem olması için shell'ler, görüntü sunucuları ve başka birçok user-space yazılımına ihtiyaç duyar. macOS'un kernel'ının adı XNU'dur ve Unix benzeridir; modern Windows kernel'ı ise NT Kernel olarak bilinir.

Bu boot zincirini okurken çoğu kişinin aklına çok yerinde bir itiraz gelir, o yüzden hemen cevaplayalım.

DERİNLEŞME **RAM kapanınca siliniyorsa, ilk talimat nereden geliyor?**

İtiraz şu: bilgisayarı kapattığında RAM'deki her şey silinir. Bu doğru — hem de tahmin ettiğinden daha sert biçimde doğru. DRAM veriyi minik kapasitörlerde tutar, kapasitörler de saniyenin küçük bir kesrinde kendiliğinden boşalır; makine **açıkken bile** bu hücrelerin saniyede binlerce kez tazelenmesi gerekir. Güç gidince tazeleme durur, yük kaçar, veri gider. CPU'nun içindeki register'lar için de aynısı geçerli.

O hâlde ortada gerçek bir bilmece var: açılış anında RAM boş, register'lar boş. CPU ilk talimatı nereden okuyor?

Cevap, "bellek adresi" ile "RAM" kelimelerini eş anlamlı kullanmayı bırakmakla başlıyor. **CPU'nun ürettiği her adres RAM'e gitmez.** CPU'nun tek bir adres uzayı vardır, ama bu uzayın hangi diliminin nereye bağlanacağına chipset karar verir: bir aralık DRAM'e gider, başka bir aralık ekran kartının belleğine, bir başkası ağ kartının kontrol register'larına, bir dilim de anakart üzerindeki küçük bir flash yongasına. Adres uzayını bir binanın kat planı gibi düşün — RAM, o binadaki dairelerden yalnızca biri.

Firmware işte o flash yongasında durur. Anakartın üzerinde, tipik olarak 8-32 MiB kapasiteli, SPI flash denen kalıcı bir yonga vardır ve içeriği güç kesilince silinmez; telefonundaki depolama gibi düşünebilirsin. "BIOS güncellemek" dediğin şey tam olarak bu yongayı yeniden yazmaktır.

Geri kalanı artık düz mantık. CPU'nun reset devresi, güç geldiği anda instruction pointer'ı donanımsal olarak sabit bir değere kurar; x86-64'te bu adres `0xFFFFFFF0`, yani 4 GiB sınırının on altı bayt altıdır. CPU o adresi ister, chipset adresi flash yongasına yönlendirir, ilk talimat oradan gelir. CPU açısından olağanüstü hiçbir şey olmadı: her zamanki gibi bir adresten talimat okudu. Olağanüstü olan, o adresin arkasında RAM değil flash olması.

Resmin son parçası da en şaşırtıcı olanı. O ilk anda DRAM henüz *kullanılabilir durumda bile değildir*: bellek denetleyicisinin takılı çipleri tanıması, zamanlamalarını okuması ve hatları kalibre etmesi gerekir. Bu işleme *memory training* denir ve onu yapan kod firmware'in kendisidir. Yani firmware, RAM'i hazırlayan koddur — dolayısıyla çalışırken RAM'e güvenemez. Peki değişkenlerini ve çağrı yığınını nerede tutar? Cevap zarif: L1/L2

cache'i geçici olarak sıradan bir bellek gibi davranmaya zorlar. Bu numaranın adı **Cache-As-RAM** (CAR). Bilgisayarın hayatının ilk milisaniyelerinde bütün dünyası, birkaç yüz kilobaytlık cache'ten ibarettir.

Zinciri tek satırda toplarsak: reset vector → flash'taki firmware → memory training → RAM artık kullanılabilir → bootloader diskten RAM'e → kernel RAM'e → kontrol kernel'a.

Bölümü kapatmadan elimizdekini toparlayalım.

> **Peki, ne öğrendik?**
>
> - CPU sonsuz bir **fetch-execute cycle** içinde çalışır: talimatı oku, yürüt, göstergeyi ilerlet, baştan başla.
> - **Instruction pointer** CPU'nun içindedir, gösterdiği adres RAM'dedir. Aradaki bu uzaklık kitabın geri kalanının yarısını açıklar.
> - **Register'lar** CPU'nun içindeki çok küçük ve çok hızlı depolardır; RAM ile aralarında yüzlerce kat gecikme farkı vardır.
> - **RAM uçucudur, disk kalıcıdır.** Bilgisayarın her açılışta her şeyi baştan yüklemesinin sebebi budur.
> - **Program** diskteki dosyadır, **process** onun çalışan hâlidir. CPU process diye bir şey bilmez; process tamamen bir işletim sistemi soyutlamasıdır.

Bölümü üç soruyla kapatmıştık. Birincisinin cevabı **[7. bölümde]** bizi bekliyor. Diğer ikisi ise doğrudan bir sonraki bölümün konusu: aynı çip, aynı talimatı neden bazen çalıştırıp bazen reddediyor?

Bölüm 3: Kernel, User Mode ve Syscall

Bir önceki bölümü üç soruyla kapattık. Şimdi ikincisini ve üçüncüsünü cevaplayacağız: programlar doğrudan CPU üzerinde çalışıyorsa ve CPU doğrudan RAM'e erişebiliyorsa, neden birbirlerinin belleğini okuyamıyorlar? Ve syscall denen şey tam olarak nedir?

Cevabın tamamı tek bir donanım özelliğine dayanıyor — ve o özellik, taşıdığı yükün yanında şaşırtıcı derecede basit.

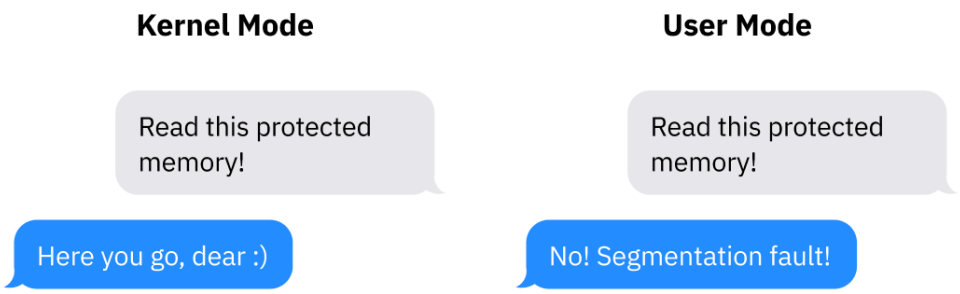
> **Bu bölümde neyi çözüyoruz?**
>
> - CPU'nun ayrıcalık seviyelerini, yani kernel mode ile user mode ayrımını göreceğiz.
> - Interrupt'ın ne olduğunu ve neden syscall için biçilmiş kaftan olduğunu anlayacağız.
> - Bir syscall'ın register düzeyinde nasıl göründüğünü somut olarak izleyeceğiz.
> - `exit(1)` gibi tek satırlık bir çağrının altında kaç katman olduğunu çözeceğiz.

Hepsine Hükmedecek İki Halka

Şimdi CPU'nun en az konuşulan ama en çok işe yarayan özelliğine geldik. CPU her an bir *mode* içindedir (ayrıcalık seviyesi ya da ring olarak da geçer) ve o an neleri yapmasına izin verildiğini bu belirler. Aynı çip, aynı talimatı, hangi mode'da olduğuna göre ya çalıştırır ya da reddeder. Modern mimarilerde iki temel mode vardır: kernel/supervisor mode ve user mode. İkiden fazlası desteklenebilir ama pratikte bugün çoğunlukla bu ikisi kullanılır.

Kernel mode'da neredeyse her şey serbesttir: CPU desteklediği her talimatı çalıştırabilir ve her belleğe erişebilir. User mode'da ise yalnızca belirli talimatlara izin verilir, I/O ve bellek erişimi kısıtlanır ve birçok CPU ayarı kilitlenir. Genel olarak kernel ve sürücüler kernel mode'da, uygulamalar ise user mode'da çalışır.

İşlemciler açılışta kernel mode'da başlar. Bir programı çalıştırmadan önce kernel, user mode'a geçiş yapar.



Gerçek bir mimaride bunun nasıl göründüğüne örnek: x86-64'te mevcut ayrıcalık seviyesi (CPL), `cs` adlı code segment register'ından okunabilir. Daha net söyleyeyim: CPL, `cs` register'ının en düşük değerlikli iki bitinde tutulur. Bu iki bit, x86-64'ün dört olası ring'ini temsil eder: ring 0 kernel mode'dur, ring 3 ise user mode'dur. Ring 1 ve 2 sürücüler için düşünülmüştür ama bugün yalnızca birkaç eski ve niş işletim sistemi tarafından kullanılır. Örneğin CPL bitleri `11` ise CPU ring 3, yani user mode'dadır.

Syscall Tam Olarak Nedir?

Programlar user mode'da çalışır çünkü bilgisayara tam erişim konusunda onlara güvenilmez. User mode, sistemin büyük kısmına erişimi engeller; ama programların yine de I/O yapması, bellek ayırması ve bir şekilde işletim sistemiyle konuşması gerekir. Bunun için user mode'da çalışan yazılımın kernel'dan yardım istemesi gerekir. Kernel da bu sırada kendi güvenlik kontrollerini uygulayabilir.

İşletim sistemiyle etkileşen kod yazdıysan muhtemelen `open`, `read`, `fork` ve `exit` gibi fonksiyonları görmüşsündür. Bu fonksiyonların altında, işletim sisteminden yardım istemek için *syscall*'lar kullanılır. Syscall, bir programın user space'ten kernel space'e geçiş başlatmasına; yani program kodundan işletim sistemi koduna atlamasına izin veren özel prosedürdür.

Bu geçişin nasıl yapıldığını anlatabilmek için önce bir kavramı yerine oturtmam gerekiyor: **interrupt**.

Interrupt, CPU'ya "yürüttüğün işi bırak, şu adrese git" diyen mekanizmadır. Fetch-execute cycle normalde bir sonraki talimata ilerler; interrupt bu akışı keser, CPU'nun nerede kaldığını

kaydeder ve onu önceden belirlenmiş başka bir adrese götürür. Oradaki iş bitince CPU kaldığı yere döner ve hiçbir şey olmamış gibi devam eder.

İki türü var ve farkları kimin tetiklediğidir:

- **Hardware interrupt** donanımdan gelir. Klavyeye bastığında, ağ kartına paket düştüğünde ya da bir timer'ın süresi dolduğunda üretilir. Program bunu istemez, başına gelir. **[7. bölümde]** çoklu görevin tamamen bunun üstüne kurulduğunu göreceğiz.
- **Software interrupt**'ı ise programın kendisi bilerek tetikler. "Şu anda kesilmek istiyorum" der.

Syscall için biçilmiş kaftan olmasının sebebi şu: bu, **atlanacak yeri programın değil kernel'ın önceden belirlediği tek atlama biçimidir.** Sıradan bir `jmp` talimatıyla istediğin adrese gidebilirsin; interrupt ile ancak kernel'ın senin için hazırladığı yere gidebilirsin.

İşte user space'ten kernel space'e kontrol devrinin temel fikri budur: CPU, sadece kernel'ın önceden hazırladığı güvenli giriş noktalarına geçişe izin verir. Tarihsel olarak bu iş software interrupt ile anlatılır; modern mimarilerde ise aynı rolü çoğu zaman özel syscall giriş talimatları üstlenir.

Bu cümlenin içindeki "önceden hazırladığı" ifadesi, bütün güvenlik modelinin kilit taşı. Neden bu kadar önemli olduğunu görmek için bir an tersini hayal et: eğer bir program kernel'ın içinde istediği adrese atlayabilseydi, dosya açma kodunun **izin kontrolünden sonraki** satırına atlar ve kontrolü tamamen atlardı. Kapıyı kilitlemenin bir anlamı kalmazdı.

Bu yüzden CPU şu kuralı donanım seviyesinde uygular: user mode'daki bir program kernel'a *nereden* gireceğini seçemez. Yalnızca "girmek istiyorum" diyebilir; nereye düşeceğine kernel'ın önceden yazdığı liste karar verir.

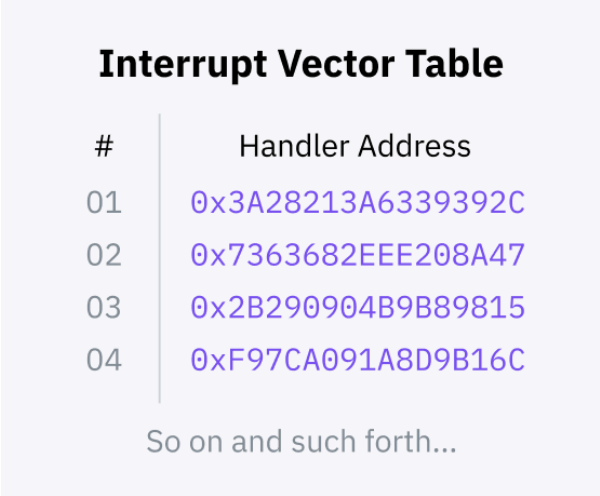
Acil durum numaralarını düşün. Yangın çıktığında itfaiyecinin cep telefonunu aramazsın, 112'yi ararsın. O numaranın karşılığında kimin oturduğunu sen değil, sistemi kuran kişi belirlemiştir. Sana düşen tek şey bir numara söylemektir.

Kernel de açılışta tam olarak böyle numaralı bir liste hazırlar: "0 numaralı olay olursa şu adrese git, 14 numaralı olursa bu adrese git." Sonra CPU'ya bu listenin nerede durduğunu bir kez bildirir. Listenin x86-64'teki adı **IDT**'dir (Interrupt Descriptor Table). Adı şimdilik önemli

değil; önemli olan, listeyi yalnızca kernel mode'daki kodun değiştirebilmesi. User mode'daki bir program ne listeye yeni bir numara ekleyebilir ne de mevcut bir girdiyi baypas edebilir.

Şimdi mekanizmanın kendisine bakalım.

1. Boot sırasında işletim sistemi bu tabloyu RAM'de kurar ve adresini CPU'ya kaydeder. Tablo, olay numaralarını handler adresleriyle eşler. (Eski 16-bit dünyada aynı yapının adı *interrupt vector table* — IVT idi; x86-64'te interrupt descriptor table olarak geçer.)



2. Eski 32-bit x86 Linux modelinde user-space programları, INT gibi bir talimat kullanarak CPU'ya şu işi yaptırabilirdi: IVT/IDT'de ilgili interrupt numarasını bul, kernel mode'a geç ve instruction pointer'ı oradaki handler adresine atla.

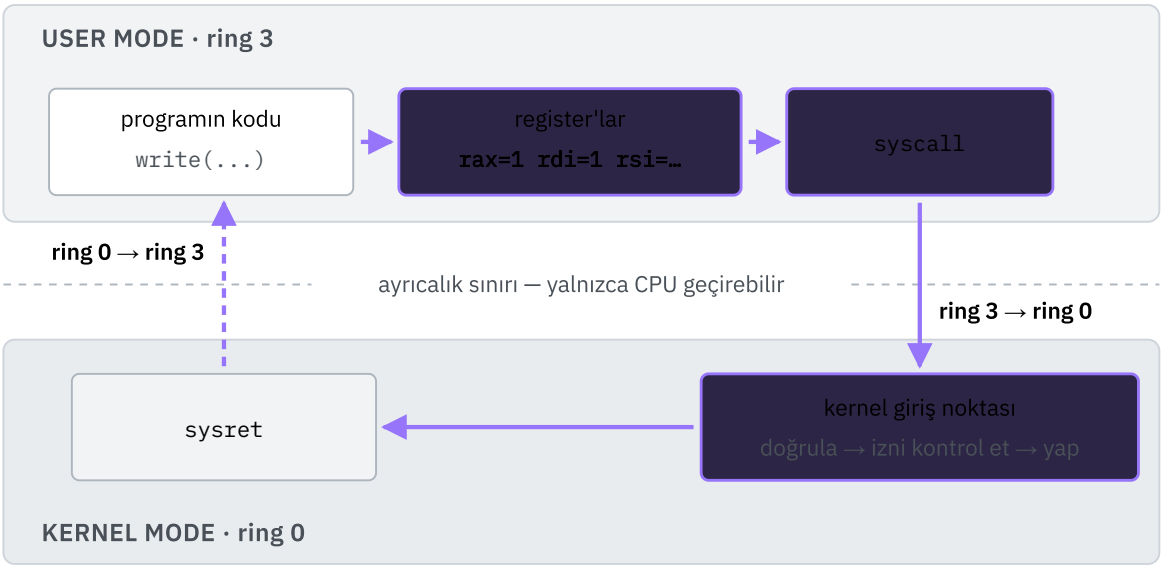
Modern 64-bit Linux'ta x86-64 için yerel syscall yolu `syscall` talimatıdır. `int 0x80` ve `sysenter` hâlâ eski uyumluluk bağlamlarında karşına çıkabilir, ama bugünün 64-bit programları için ana yol değildir.

Dönüş de gidiş kadar kontrollüdür. Kernel işini bitirince CPU'ya iki şeyi birden yaptırır: ayrıcalık seviyesini user mode'a düşürür ve instruction pointer'ı programın kaldığı yere geri koyar. Bu iki işi tek hamlede yapan özel bir dönüş talimatı vardır; adı mimariye göre değişir (x86-64'te `iret` ya da `sysret`, AArch64'te exception return). İsimleri ezberlemene gerek yok; önemli olan dönüşün de rastgele bir atlama olmadığı.

(Merak ediyorsan, 32-bit x86 Linux'ta klasik syscall interrupt numarası `0x80`'dir. Linux syscall listesini Michael Kerrisk'in çevrimiçi manpage dizininde görebilirsin.)

Bütün yolculuğu tek bir şemada adım adım izleyelim:

BİR SYSCALL'IN GİDİŞ VE DÖNÜŞÜ



1. **Program çalışıyor.** Kod user mode'da (ring 3) yürüyor. Bu modda I/O yok, kernel belleğine erişim yok.
2. **Argümanlar register'lara yazılır.** Syscall numarası `rax`'e, argümanlar `rdi`, `rsi`, `rdx`'e konur. Bu düzenin adı calling convention.
3. **`syscall` talimatı tetiklenir.** CPU tek hamlede iki iş yapar: ayrıcalık seviyesini kernel mode'a çıkarır ve kernel'ın önceden belirlediği giriş noktasına atlar. Program nereye gideceğini seçemez.
4. **Kernel işi yapar.** Önce argümanları doğrular ve izinleri kontrol eder, sonra asıl işi yürütür. Sonuç `rax`'e yazılır.
5. **Geri dönülür.** `sysret` yine tek hamlede iki iş yapar: ayrıcalık user mode'a düşer ve instruction pointer programın kaldığı yere geri konur.

Wrapper API'ler: Interrupt Ayrıntısını Gizlemek

Şu ana kadar bildiklerimizi toparlayalım:

- User-mode programları ayrıcalıklı I/O'ya, fiziksel belleğe ya da kernel belleğine doğrudan erişemez. Dış dünyayla etkileşime geçmek için işletim sisteminden yardım istemeleri gerekir.

- Programlar, `syscall`/`sysret`, `svc`/exception return veya legacy `int`/`iret` gibi mimariye özgü yollarla kontrolü işletim sistemine devredebilir.
- Programlar ayrıcalık seviyesini doğrudan değiştiremez. Bu kontrollü geçişler güvenlidir çünkü CPU, işletim sistemi kodunda nereye atlanacağını önceden kernel tarafından yapılandırılmış giriş noktalarından alır. Bu tablolar ve register'lar yalnızca kernel mode'da değiştirilebilir.

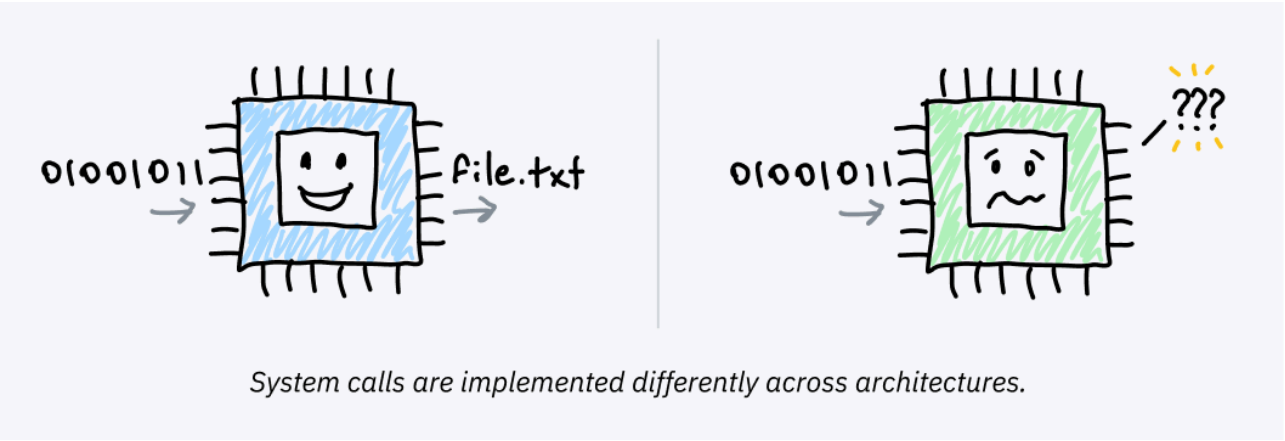
Bir program syscall tetiklerken kernel'a veri de aktarmalıdır. Kernel'ın hangi syscall'ın çağrıldığını ve örneğin hangi dosya adının açılacağını bilmesi gerekir.

Bunun nasıl yapıldığını somut görmek her şeyi yerine oturtuyor. 64-bit Linux'ta kural şu: syscall numarası `rax` register'ına, ilk üç argüman da sırasıyla `rdi`, `rsi` ve `rdx` register'larına yazılır. Sonra `syscall` talimatı tetiklenir. Programı sonlandıran `exit(1)` çağrısının aslı, iki register'a değer koyup bir talimat çalıştırmaktan ibarettir:

- `rax = 60` — 60, Linux'ta `exit` syscall'ının numarası.
- `rdi = 1` — çıkış kodu.
- `syscall`

Kernel devraldığında yaptığı ilk iş bu register'ları okumaktır. Yani register'lar burada iki dünya arasındaki ortak dil işlevi görüyor: program yazıyor, kernel okuyor. Bu düzenin adı *calling convention*'dır ve mimariden mimariye, işletim sisteminden işletim sistemine değişir.

Mimariler arasında syscall çağırma biçimlerinin değişmesi, programcıların her program için bunu elle yazmasını pratik olmaktan çıkarır. Dahası, işletim sistemini kendi ayrıntılarına mahkûm eder: her program bu ayrıntıları doğrudan kullanıyorsa, en küçük değişiklik çalışan her programı bozar. Üstelik artık ham assembly ile program yazmıyoruz; bir dosya okumak isteyen herkesin assembly yazması beklenemez.



System calls are implemented differently across architectures.

Bu yüzden işletim sistemleri, bu mimariye özgü kernel giriş yollarının üstüne bir soyutlama katmanı koyar. Gerekli assembly talimatlarını saran yeniden kullanılabilir üst seviye kütüphane fonksiyonları Unix benzeri sistemlerde libc, Windows'ta ise ntdll.dll tarafından sağlanır. Bu kütüphane fonksiyonlarına yaptığın çağrı doğrudan kernel mode'a geçmez; bunlar normal fonksiyon çağrılarıdır. Kütüphanenin içinde bir yerde assembly devreye girer ve kontrol gerçekten kernel'a aktarılır.

Unix benzeri bir sistemde C içinden `exit(1)` çağırdığında, bu fonksiyon önce syscall numarasını ve argümanlarını doğru register'lara yerleştirir, sonra da mimariye özgü syscall talimatını tetikler. Yani senin tek satırlık `exit(1)` çağrın, altında register düzenleyen bir assembly bloğuna, oradan da CPU'nun ayrıcalık seviyesini değiştiren tek bir talimata iniyor. Bu kadar katmanın üst üste dizilip tek satır gibi görünmesi, bilgisayarların en iyi yaptığı iş.

Bir Syscall Ne Kadar Pahalı?

Bütün bu mekanizmayı gördükten sonra doğal bir soru geliyor: madem syscall sıradan bir fonksiyon çağrısı değil, ne kadar pahalı?

Karşılaştırma şöyle. Sıradan bir fonksiyon çağrısı birkaç saat çevrimi sürer. Bir syscall ise mertebe olarak **yüzlerce** çevrim. Aradaki farkı üç şey doğuruyor:

- **Mod geçişinin kendisi.** CPU ayrıcalık seviyesini değiştirir, register'ları kaydeder, kernel yığınına geçer. Dönüşte hepsini geri alır.
- **Kernel'ın doğrulama işi.** Kullanıcıdan gelen her işaretçi ve her sayı şüphelidir; kernel bunları kontrol etmek ve kendi tarafına kopyalamak zorundadır.

- **Spekülatif yürütme yamalarının bedeli.** Meltdown ve Spectre sonrası eklenen korumalar, her geçişte sayfa tablosu değiştirmek ya da tahmin tamponlarını temizlemek gibi ek işler getirdi. Bu bedel, 2018 sonrası sistemlerde syscall maliyetini gözle görülür biçimde artırdı.

Bu sayı, kitabın ilerisindeki birçok tasarım kararının arkasındaki asıl sebep. Bir programın neden veriyi tek tek değil de tampon dolusu yazdığı, `epoll` ve `io_uring` gibi arayüzlerin neden var olduğu — hepsi aynı hedefe hizmet eder: **aynı işi daha az syscall ile yapmak.**

İşin ilginç tarafı, bazı syscall'ların bu bedeli hiç ödememesi. Saati sormak gibi çok sık yapılan ve hiçbir gizli bilgi gerektirmeyen birkaç iş için kernel, gerekli kodu doğrudan programın adres alanına eşler; çağrı kernel'a hiç sıçramadan user space'te biter. Bu numaranın adı **vDSO** ve son bölümde ayrıntısına döneceğiz.

Toparlayalım:

- Hangi talimatların çalıştırılabileceğini, işlemcinin o anki **mode**'u belirler. İşletim sistemi kodu kernel mode'da çalışır; bir programı çalıştırmadan önce user mode'a geçer.
- Programların ayrıcalıkları kısıtlı olduğu için dış dünyayla etkileşim kurmak istediklerinde kernel'dan yardım istemek zorundadırlar. **Syscall**, bu geçişin standart yoludur.
- Geçişin güvenli olmasının sebebi, atlanacak yeri programın değil kernel'ın belirlemesidir. User mode'daki kod kernel'a *nereden* gireceğini seçemez; yalnızca girmek istediğini söyleyebilir.
- Programlar syscall'ları genelde libc gibi kütüphane fonksiyonları üzerinden kullanır. Mimariye özgü ayrıntıyı saran katman odur.

Son maddedeki "mimariye özgü" ifadesi, şimdiye kadar birkaç kez kullandığım ama hiç açmadığım bir konuya işaret ediyor. x86-64, AArch64, RISC — bu isimler tam olarak neyi anlatıyor ve neden bir program bir makinede çalışıp diğerinde çalışmıyor? Sıradaki bölümün konusu bu.

Bölüm 4: Mimariler: x86, ARM ve Diğerleri

Buraya kadar birkaç kez “x86-64’te şöyle olur”, “ARM tarafında böyle” diye cümleler kurdum ve bu isimlerin ne anlama geldiğini açıklamadan geçtim. Şimdi o borcu ödeyelim.

Bu isimler marka adı değil. Bilgisayarın hangi dili konuştuğunu söylüyorlar — ve o dili bilmeden, bir programın neden bir makinede çalışıp diğerinde çalışmadığını açıklamak mümkün değil.

> **Bu bölümde neyi çözüyoruz?**
>
> - “Talimat kümesi” ile “işlemci modeli” arasındaki farkı netleştireceğiz.
> - 32 bit ile 64 bit arasındaki farkın tam olarak neyi ölçtüğünü göreceğiz.
> - x86-64 ile ARM’ın nereden geldiğini ve neden ikisinin de var olduğunu anlayacağız.
> - Aynı programın neden her makinede çalışmadığını cevaplayacağız.

Talimat Kümesi: Donanımla Yazılım Arasındaki Sözleşme

[2. bölümde] CPU’nun makine kodu yürüttüğünü, makine kodunun da opcode’lardan oluştuğunu gördük. Şimdi bir soru soralım: `05 00 02 00 00` baytlarının “EAX register’ına 512 ekle” demek olduğuna kim karar verdi?

Cevap bir belge. Her işlemci ailesinin, hangi bayt dizisinin hangi işi yapacağını satır satır tanımlayan bir sözleşmesi vardır. Bu sözleşmenin adı **talimat kümesi mimarisi** — kısaca **ISA** (Instruction Set Architecture).

Bir ISA şunları söyler:

- Hangi talimatlar var ve her biri hangi bayt dizisiyle yazılıyor.
- Kaç register var, adları ne, kaç bit genişliğinde.
- Bellek nasıl adreslenir, byte’lar hangi sırayla dizilir.
- Ayrıcalık seviyeleri nasıl çalışır, kernel’a nasıl girilir.

Yani ISA, donanım ile yazılım arasındaki sınırdır. Üstünde kalan her şey — derleyicin, işletim sistemin, tarayıcın — bu sözleşmeye göre yazılır. Altında kalan her şey de bu sözleşmeyi yerine getirmek zorundadır.

Burada karıştırılması çok kolay bir ayrım var, o yüzden altını çizeyim:

ISA, işlemcinin *ne* yaptığını söyler; mikro-mimari ise *nasıl* yaptığını.

Bir Intel Core i9 ile bir AMD Ryzen aynı ISA'yı konuşur: ikisine de aynı `.exe` dosyasını verirsin, ikisi de çalıştırır. Ama içeride bambaşka makinelerdir — farklı pipeline derinliği, farklı cache düzeni, farklı dal tahmincisi. **[8. bölümde]** anlattığım hilelerin hepsi mikro-mimari tarafındadır ve üreticiden üreticiye, hatta nesilden nesile değişir.

> Bu ayrımın günlük hayattaki karşılığı şu: yeni bir işlemci aldığında programlarını yeniden derlemen gerekmez. Çünkü değişen mikro-mimaridir, sözleşme aynı kalır. Sözleşme değiştiğindeyse — ARM'a geçen bir Mac'te olduğu gibi — her şeyin yeniden derlenmesi gerekir.

"32 bit" ve "64 bit" Tam Olarak Neyi Ölçer?

Bu ikili neredeyse her yerde karşına çıkıyor ama neyi ölçtüğü çoğu zaman havada kalıyor. Aslında tek bir şeyi değil, birbirine bağlı iki şeyi birden ölçüyor.

Birincisi: register genişliği. 64-bit bir işlemcide genel amaçlı register'lar 64 bit, yani 8 bayt tutar. 32-bit bir işlemcide 4 bayt. Bu, CPU'nun tek hamlede işleyebildiği sayının büyüklüğü demek. 32 bitle en fazla 4.294.967.295'e kadar sayabilirsin; daha büyük sayılarla çalışmak için işi parçalara bölmek gerekir.

İkincisi, ve asıl önemlisi: adres genişliği. Bir bellek adresi de bir sayıdır ve o sayı register'a sığmak zorundadır. 32 bitlik bir adresle en fazla 2^{32} farklı bayt gösterebilirsin — yani **4 GiB**. Kaç çubuk RAM taktığının önemi yok; adresi yazacak yerin yoksa oraya erişemezsin.

2000'lerin başında 32-bit Windows'un 4 GB'ın üstünü göremiyor olmasının sebebi tam olarak buydu. Sorun RAM'de değil, adresi tutan sayının genişliğindeydi. 64 bite geçince tavan 2^{64}'e, yani 16 exbibyte'a çıktı — öngörülebilir gelecek için fazlasıyla yeterli bir sayı.

> Pratikte x86-64 işlemciler bu 64 bitin tamamını kullanmaz; sanal adreslerin yalnızca alttaki 48 biti anlamlıdır (yeni nesillerde 57). Sebebini **[14. bölümde]** göreceğiz: adresin her biti bir sayfa tablosu seviyesi demek ve her seviye çeviriye ek maliyet bindiriyor. 48 bit bile 256 tebibyte eder; kimse bu tavana yaklaşmıyor.

Bunun bir de ters yönlü bedeli var: 64-bit bir programda her işaretçi 4 bayt yerine 8 bayt yer kaplar. İşaretçi bakımından yoğun yapılarda (bağlı listeler, ağaçlar) bu, aynı verinin cache'te iki kat yer tutması demektir. Aynı program 64 bitte bazen *daha yavaş* çalışabilir; nedeni **[6. bölümde]** konuştuğumuz cache basıncıdır.

x86'nın Hikâyesi: Bir Geriye Uyumluluk Anıtı

Bugün masaüstü ve sunucu dünyasına hâkim olan ISA'nın adı **x86-64**. Bu tuhaf ismin nereden geldiğini anlamak için elli yıl geriye gitmek gerekiyor.

[2. bölümde] andığım Intel 4004 ile başlayan zincir 1978'de **8086**'ya ulaştı: 16-bit bir işlemci. Ondan sonra gelen modeller 80186, 80286, 80386, 80486 diye devam etti. Sektör bu aileyi kısaca "86 ile biten işlemciler", yani **x86** diye anmaya başladı. 80386 ile birlikte aile 32 bite geçti ve bu 32-bit sürüm **IA-32** adını aldı.

Sonra beşinci nesil geldi ve Intel bir sorunla karşılaştı: sayılar ticari marka olarak tescil edilemiyordu, yani rakipleri de "586" diye çip satabiliyordu. Çözüm, sayı yerine bir isim koymak oldu: **Pentium**. Yani Pentium yeni bir mimari değil, x86 ailesinin beşinci neslinin pazarlama adıdır. Sonraki Core, Core i7, Ryzen gibi isimler de aynı şey — hepsi aynı sözleşmeyi konuşan farklı mikro-mimarilerdir.

64 bite geçiş ise Intel'in beklemediği bir yerden geldi. Intel, geleceği tamamen yeni ve x86 ile uyumsuz bir mimaride (Itanium) gördü. AMD ise farklı düşündü: mevcut x86'yı 64 bite genişletip eski programların çalışmaya devam etmesini sağladı. Piyasa, elindeki yazılımı çöpe atmak istemedi. AMD'nin tasarımı fiilen standart oldu; Intel de onu benimsemek zorunda kaldı.

İşte bu yüzden bugün kullandığın mimarinin adı **x86-64** (ya da AMD64, ya da Intel 64 — üçü de aynı şey).

DERİNLEŞME **Modern bir işlemci hâlâ 1978 modunda açılıyor**

Geriye uyumluluğun bedeli soyut bir şey değil; her açılışta ödüyorsun.

Bir x86-64 işlemci güç geldiğinde 64-bit modda başlamaz. **Real mode** denen, 8086'nın 1978'deki modunda başlar: 16-bit, koruma yok, 1 MiB adres sınırı. Firmware önce onu 32-bit korumalı moda, sonra 64-bit "long mode"a taşımak zorundadır. Yani modern bir sunucu, kırk yıl önceki bir işlemcinin taklidini yaparak uyanır ve ancak birkaç adım sonra kendisi olur.

Aynı yükün başka izleri de var: değişken uzunluklu talimat kodlaması (bir talimat 1 ile 15 bayt arasında olabilir), hâlâ duran ama kimsenin kullanmadığı segment register'ları, x87 gibi terk edilmiş ondalık sayı birimleri.

Buna karşılık aynı yük, x86'yı bu kadar başarılı yapan şeyin ta kendisi. 1990'larda yazılmış bir programı bugünkü bir işlemcide çalıştırabiliyor olmak, sektörün kırk yıllık yatırımını korumak demekti. Temizlik uğruna bu bağı kesmeyi deneyen mimariler — Itanium bunlardan biriydi — genellikle kaybetti.

ARM: Aynı Soruya Verilen Başka Bir Cevap

x86 masaüstünde kazanırken bambaşka bir yerde, tamamen farklı bir baskı altında ikinci bir aile büyüyordu.

ARM, 1985'te İngiliz Acorn Computers'ta doğdu. Açılımı *Acorn RISC Machine*'di. Hedefi en hızlı işlemciyi yapmak değil, işi *en az güçle* yapmaktı — çünkü ucuz ev bilgisayarlarında ne büyük bir soğutucu ne de büyük bir güç bütçesi vardı.

Bu kısıt, cep telefonu çağı geldiğinde altın değerinde çıktı. Pilden çalışan bir cihazda önemli olan tepe hız değil, watt başına iş. Bugün elindeki telefonun içinde neredeyse kesinlikle bir ARM çekirdeği var.

ARM'ın ikinci sıra dışı yanı iş modeli: **ARM şirketi çip üretmez.** Tasarımı lisanslar. Apple, Qualcomm, Samsung, Amazon ve daha onlarcası bu lisansı alıp kendi çipini üretir. Bu yüzden

“ARM işlemci” dediğimizde tek bir üründen değil, aynı sözleşmeyi konuşan çok geniş bir aileden söz ediyoruz.

64-bit sürümün adı **AArch64** (ya da ARM64). 2011’de duyuruldu ve bugün telefonlardan Apple Silicon Mac’lere, oradan bulut sunucularına kadar her yerde.

> Apple’ın 2020’de Mac’leri Intel’den kendi ARM çiplerine taşıması, bu bölümün en somut örneği. Sözleşme değişti, dolayısıyla bütün programların yeniden derlenmesi gerekti. Apple bu geçişi **Rosetta 2** ile yumuşattı: x86-64 programlarını kurulum sırasında AArch64 makine koduna çeviriyor. Çeviri işe yarıyor ama bedava değil; yerel derlenmiş bir uygulama her zaman daha hızlı.

CISC ve RISC: Kavga Neydi, Bugün Ne Kaldı?

Bu iki kısaltma seksenlerin büyük tartışmasıdır ve isimlerinden tahmin edebileceğinden daha az şey ifade ederler.

CISC (Complex Instruction Set Computer) yaklaşımı şuydu: talimatlar zengin ve güçlü olsun, tek bir talimat çok iş yapsın. Bellek hem pahalı hem küçükken, programı kısa tutmak gerçek bir kazançtı. x86 bu okuldan.

RISC (Reduced Instruction Set Computer) tersini savundu: talimatlar az sayıda, sabit uzunlukta ve basit olsun; karmaşık işi derleyici birkaç basit talimatla kursun. Böylece donanım sadeleşir, hızlanır ve daha az güç harcar. ARM, RISC-V ve MIPS bu okuldan.

Aradaki en görünür fark talimat uzunluğu. AArch64’te her talimat tam olarak 4 bayttır; işlemci sıradaki talimatın nerede bittiğini bilmek için onu çözmek zorunda değildir. x86-64’te ise 1 ile 15 bayt arasında değişir; nerede bittiğini anlamak için başından okumaya başlaman gerekir. **[8. bölümde]** gördüğümüz “aynı anda birden fazla talimatı çözme” işi, sabit uzunlukta çok daha kolaydır.

Peki kim kazandı? Dürüst cevap: ikisi de, çünkü tartışma ortada kalktı.

Modern bir x86-64 işlemci dışarıdan CISC görünür ama içeride RISC gibi çalışır. Aldığı karmaşık talimatı, **mikro-op** denen küçük ve düzenli parçalara böler; pipeline’ın geri kalanı bu

parçaları yürütür. Yani x86, RISC fikirlerini kendi kabuğunun içine taşıdı. Geriye kalan asıl fark, o çeviriyi yapan decoder'ın maliyeti: sabit uzunluklu bir ISA'da bu iş neredeyse bedavayken, x86'da gerçek bir transistör ve güç bütçesi tüketir.

Bu bölümdeki bir davranış farkını da buraya bağlayabiliriz. **[3. bölümde]** syscall giriş yollarından bahsetmiştik. x86 tarafında bu iş için ayrı ayrı talimatlar eklendi — Intel 32-bit dönem için `SYSENTER`'ı, AMD 64-bit için `SYSCALL`'ı tasarladı ve 64-bit dünyada AMD'ninki standart oldu. AArch64 ise böyle özel bir talimat eklemedi; hem syscall'lar hem de software interrupt'lar için tek bir `SVC` talimatı kullanılır. Bu, iki okulun tavrını tek bir örnekte özetliyor: CISC ihtiyaç gördüğü yere yeni talimat ekler, RISC var olanı yeniden kullanır.

RISC-V: Sözleşmenin Kendisi Açık Olsa?

Son yılların yeni ismi **RISC-V**. Teknik olarak sıra dışı bir şey vaat etmiyor; sıra dışı olan lisansı. x86 ve ARM'ın aksine RISC-V'yi kullanmak için kimseye para ödemen ya da izin almana gerek yok — ISA açık bir standart.

Bunun pratik sonucu, bir üniversitenin, bir startup'ın ya da bir devletin kendi işlemcisini kimseye sormadan tasarlayabilmesi. Şu an ağırlıklı olarak gömülü sistemlerde ve araştırma tarafında; masaüstünde ciddi bir varlığı yok. Ama bu bölümün asıl dersini iyi gösteriyor: ISA bir belgedir, ve belgenin kime ait olduğu teknik bir mesele değil, stratejik bir meseledir.

Peki Aynı Program Neden Her Yerde Çalışmıyor?

Artık bu sorunun cevabı elimizde. Derleyici kaynak kodunu makine koduna çevirirken **belirli bir ISA'yı hedefler**. Ürettiği baytlar başka bir sözleşmede ya bambaşka bir anlama gelir ya da hiçbir anlama gelmez.

İşletim sistemi de bu dosyayı çalıştırmadan önce hangi mimariye ait olduğuna bakar. **[12. bölümde]** göreceğimiz ELF başlığında bunun için ayrılmış bir alan vardır; kernel oraya bakıp uyumsuzsa dosyayı hiç açmaz.

Bu tek gerçek, gündelik hayatta karşına çıkan bir sürü ayrıntıyı açıklıyor:

- Bir program indirirken neden “Intel” ve “Apple Silicon” diye iki ayrı sürüm sunuluyor.
- Docker imajlarının neden `amd64` ve `arm64` etiketleri var ve yanlış olanı çektiğinde neden çalışmıyor.
- Telefonun uygulamasının bilgisayarında doğrudan çalışmamasının sebebinin işletim sisteminden önce mimari olduğu.

Bu duvarı aşmanın yolu her zaman aynı: **çeviri**. Apple’ın Rosetta’sı, QEMU gibi öykünücüler ve Windows’un ARM üzerinde x86 programlarını çalıştıran katmanı, hepsi bir ISA’nın talimatlarını diğerininkine çevirir. Çalışır, ama arada bir çevirmen olduğu için hep bir bedel ödersin.

> Kendi makinenin hangi sözleşmeyi konuştuğunu merak ediyorsan tek komut yeter: `uname -m`. `x86_64` görürsen bu bölümün ilk yarısındasın, `aarch64` ya da `arm64` görürsen ikinci yarısında.

Özet

- **ISA**, donanım ile yazılım arasındaki sözleşmedir: hangi bayt hangi işi yapar, kaç register vardır, kernel’a nasıl girilir.
- **Mikro-mimari** o sözleşmenin nasıl yerine getirildiğidir ve üreticiden üreticiye değişir. Aynı ISA, bambaşka içeriye sahip olabilir.
- **32/64 bit** hem register genişliğini hem de adres genişliğini ölçer. 32 bitin 4 GiB tavanı, RAM’den değil adresin kendisinden gelir.
- **x86-64**, 1978’deki 8086’ya kadar uzanan kesintisiz bir geriye uyumluluk zincirinin bugünkü hâlidir; her açılışta hâlâ real mode’dan geçer.
- **AArch64**, güç verimliliği baskısı altında büyümüş ayrı bir ailedir ve lisanslama modeli sayesinde çok sayıda üreticide karşımıza çıkar.
- **CISC/RISC** ayrımı bugün büyük ölçüde tarihseldir: x86 içeride talimatları mikro-op’lara bölerek RISC gibi çalışır.
- Bir binary tek bir ISA’ya aittir. Başka bir mimaride çalıştırmanın tek yolu çeviridir ve çevirinin bedeli vardır.

Sözleşmeyi tanıdığımıza göre artık bir kademe aşağı inebiliriz. Sıradaki soru şu: CPU bu talimatları yürütürken ihtiyaç duyduğu veriyi nereden alıyor ve neden bunu almak

sandığından çok daha pahalı?

Bölüm 5: Bellek Hiyerarşisi

Bir önceki bölümde CPU'nun talimatları nasıl yürüttüğünü, RAM'in çalışma masası olduğunu ve diskin kalıcı depo olduğunu gördük. Ama bu üçlü arasında aslında çok daha fazla katman var. Bilgisayarın belleği tek düzeyli değil; bir piramit gibi katman katman düzenlenmiş. Neden? Çünkü **hızlı bellek pahalıdır, ucuz bellek yavaştır**. Piramidin tepesi hızlı ve küçük, tabanı büyük ve yavaş.

> **Bu bölümde neyi çözüyoruz?**
>
> - CPU'dan diske kadar olan bellek piramidini adım adım çıkacağız.
> - Register, cache, RAM ve disk arasındaki hız ve maliyet farkını anlayacağız.
> - SRAM ile DRAM arasındaki farkın nereden geldiğini göreceğiz.
> - "Hızlı bellek neden büyük olamıyor?" sorusunu cevaplayacağız.

Üç Soru: Ne Kadar, Ne Kadar Hızlı, Ne Kadar Pahalı?

Bir bilgisayarın belleğini tasarlarken cevaplaman gereken üç soru vardır ve üçünü aynı anda memnun etmek mümkün değildir:

- **Ne kadar?** Kapasite ne olsun?
- **Ne kadar hızlı?** Erişim ne kadar sürsün?
- **Ne kadar pahalı?** Bit başına maliyet ne olsun?

Aralarındaki ilişki inatçı ve neredeyse yasa gibidir:

- Erişim hızlandıkça bit başına maliyet **artar**.
- Kapasite büyüdükçe bit başına maliyet **düşer**.
- Kapasite büyüdükçe erişim süresi **uzar**.

Son maddeyi atlamak kolay ama en önemlisi o. Büyük bellek yalnızca pahalı olduğu için değil, **fiziksel olarak da** yavaştır: kapasite arttıkça adres çözme devresi büyür, sinyalin kat etmesi gereken mesafe uzar. Bir cache'i iki katına çıkarmak, içinde arama yapmayı da yavaşlatır.

Yani “hem çok büyük hem çok hızlı bir bellek yapalım” diye bir seçenek yok. Tasarımcının önünde iki kötü seçenek var: az ve hızlı, ya da çok ve yavaş.

Çözüm ikisinden birini seçmek değil, **ikisini birden kullanmak** oldu. Küçük ve hızlı belleği CPU’nun yanına, büyük ve yavaş belleği uzağa koy; sık kullanılanı yukarıda tut. Ortaya çıkan katmanlı yapıya **bellek hiyerarşisi** denir ve bu bölümün konusu o.

Peki Bu Neden İşe Yarıyor?

Buradaki numarayı bir kere görünce her şey yerine oturuyor, o yüzden sayılarla gösterelim.

Diyelim ki hızlı katman 1 nanosaniyede, yavaş katman 100 nanosaniyede cevap veriyor. İstenen verinin hızlı katmanda bulunma oranına **hit oranı** diyelim. Ortalama erişim süresi şu basit hesapla çıkar:

ortalama = hit oranı × hızlı süre + (1 − hit oranı) × yavaş süre

Hit oranı %50 ise ortalama 50,5 ns. Hâlâ kötü. Ama gerçek programlarda bu oran %50 değil; **%95’in üzerinde**. %95 ile hesap 5,95 ns’ye iner — yani yalnızca küçük ve pahalı katmanı ekleyerek, tamamı yavaş bellekten oluşan bir sistemi on yedi kat hızlandırmış olursun.

Hiyerarşinin bütün varlık sebebi o hit oranıdır. Peki neden bu kadar yüksek çıkıyor? Cevabın adı **locality**: programlar belleğe rastgele dağılmış biçimde erişmez, kümeler hâlinde erişir. Bir sonraki bölümde bunu ayrıntısıyla açacağız; şimdilik hiyerarşinin ayakta durmasını sağlayan tek varsayımın bu olduğunu bilmek yeterli.

> Bu hesabın ters yönü de öğretici. Hit oranı %95’ten %90’a düştüğünde ortalama süre 5,95 ns’den 10,9 ns’ye, yani neredeyse iki katına çıkar. Kaçırılan her yüzde puanı orantısız biçimde pahalıya patlar; performans yazılarında cache miss oranının bu kadar takıntılı biçimde konuşulmasının sebebi budur.

Piramidin Tepesi: Register'lar

CPU'nun içindeki register'lar, daha önce gördüğümüz gibi, işlemcinin doğrudan erişebildiği küçük depolama alanlarıdır. Register erişimi çoğu modern işlemcide bir veya birkaç saat çevrimi mertebesindedir; yani aşağıdaki sayıları kesin ölçüm değil, katmanlar arasındaki büyüklük farkını göstermek için tipik mertebeler olarak oku.

- Hız: bir veya birkaç saat çevrimi; saat hızına göre alt nanosaniye mertebesi
- Boyut: 64-bit (modern x86-64'te) veya 32-bit (eski sistemlerde)
- Miktar: Birkaç on adet (x86-64'te ~16 genel amaçlı register)
- Maliyet: Çok yüksek (transistör başına en pahalı alan)

Birinci Kademe: L1 Cache

Register'lar tek başına yetmez. Bir düşün: elinin altında toplam birkaç yüz baytlık bir çalışma alanı var, işlediğin veri ise gigabaytlarca. Eksik olan her şeyi RAM'den istemen gerekiyor — ve RAM'den bir değer gelene kadar CPU yüzlerce saat çevrimi boyunca boş oturuyor.

İşte bu bekleyişi kısaltmak için CPU ile RAM'in arasına küçük ama çok hızlı bir ara depo koyuyoruz: **cache** (önbellek). L1 cache bu ara depoların CPU'ya en yakın olanı; çekirdeğin tam içinde durur ve register'lardan sonraki en hızlı bellektir.

- Hız: ~1 - 5 nanosaniye (4-10 saat çevrimi)
- Boyut: Genelde 32 KiB - 128 KiB (veri cache'i + talimat cache'i olarak ayrılmış)
- Teknoloji: **SRAM** (Static RAM)

> **SRAM vs DRAM**
>
> SRAM, tipik hücrelerde her bit için birden fazla transistör kullanır; bu yüzden hızlı ama pahalıdır. "Static" demek elektriksiz kalınca veriyi tutar demek değildir; sadece DRAM gibi sürekli refresh gerektirmediği anlamına gelir. SRAM de **uçucudur** (volatile): güç kesilirse veri gider. DRAM ise her biti transistör + kapasitör hücresiyle temsil eder; daha ucuz ve yoğundur ama kapasitörler düzenli tazelenmelidir. Bu yüzden ana RAM dediğimiz şey çoğunlukla DRAM'dir.

İkinci Kademe: L2 Cache

Akla ilk gelen çözüm L1'i büyütmek. Ama bu işe yaramaz: bir cache ne kadar büyürse içinde arama yapmak da o kadar uzun sürer, yani L1'i büyütmek onu yavaşlatır. L1'in küçük kalması bir eksiklik değil, bilinçli bir tercihtir.

Bu yüzden büyütmek yerine altına ikinci bir katman koyuyoruz. L2, L1'de bulunamayan veriye bakılan ilk yerdir: L1'den birkaç kat büyük, karşılığında birkaç kat yavaş. Eskiden ayrı bir çip olarak anakarta oturuyordu; bugün çekirdeğin hemen yanında, aynı silikonun üzerinde duruyor.

- Hız: ~5 - 15 nanosaniye (10-30 saat çevrimi)
- Boyut: 256 KiB - 1 MiB (çekirdek başına)
- Teknoloji: SRAM

Üçüncü Kademe: L3 Cache

L3'ün diğer ikisinden yapısal bir farkı var. L1 ve L2 her çekirdeğin kendi malıdır; komşu çekirdek onlara bakamaz. L3 ise ortak alandır — sekiz çekirdek de aynı L3'e bakar.

Bu ortaklık iki yönlü çalışır. Faydası şu: bir çekirdeğin RAM'den çekip getirdiği veriyi diğeri hazır bulur, aynı uzun yolu ikinci kez yürümez. Bedeli ise bu bölümün sonunda ayrıntısıyla konuşacağımız mesele: aynı verinin birden fazla kopyası ortalıkta dolaşmaya başlayınca, hangisinin doğru olduğuna birinin karar vermesi gerekir.

- Hız: ~15 - 40 nanosaniye (30-80 saat çevrimi)
- Boyut: 8 MiB - 128 MiB (işlemciye göre değişir; AMD Epyc'lerde yüzlerce MiB olabilir)
- Teknoloji: SRAM

Ana Bellek: RAM (DRAM)

Cache'ler bitince sıra RAM'e gelir. Üç katmanın hiçbirinde bulunamayan her veri buradan istenir; yani RAM hiyerarşinin asıl belleği, cache'ler ise onun önüne çekilmiş perdelerdir.

Buraya kadar anlattığım her şeyi tek cümlede toplayabilirim: **o katmanların tamamı, RAM'e gitmek zorunda kalmamak için verilmiş bir mücadeledir.** Çünkü RAM'e gitmek CPU ölçeğinde uzun bir yolculuktur — L1'den yüz kat, register'dan iki yüz kat uzak.

- Hız: ~80 - 120 nanosaniye (yüzlerce saat çevrimi)
- Boyut: 8 GiB - 128 GiB (günümüz masaüstü/sunucu sistemlerinde)
- Teknoloji: DRAM (DDR4, DDR5 vb.)

> **DDR nedir?** Double Data Rate; her saat çevriminde iki kez veri aktarımı yapar. DDR5, DDR4'e göre daha yüksek bant genişliği sunar.

Fiziksel tarafına bakalım. Bellek modülleri (DIMM'ler), üzerinde DRAM çipleri bulunan ve anakarttaki yuvalara takılan kartlardır. CPU ile bu çipler arasındaki trafiği **bellek denetleyicisi** (memory controller) yönetir: hangi satırın açılacağına, tazelemenin ne zaman yapılacağına ve biriken isteklerin hangi sırayla karşılanacağına o karar verir. Eskiden bu denetleyici anakart üzerinde ayrı bir çipte otururdu — kuzey köprüsü (northbridge) denen parçada. Modern işlemcilerde ise CPU'nun kendi silikonuna taşındı ve bu taşıma, yolun bir durağını tamamen ortadan kaldırdığı için RAM gecikmesini gözle görülür biçimde düşürdü.

DERİNLEŞME **Aynı makinede bellek de uzak olabilir: NUMA**

Şimdiye kadar RAM'i tek bir havuz gibi anlattım: CPU bir adres ister, bellek denetleyicisi getirir. Tek soketli bir masaüstünde bu doğru.

Sunucularda ise resim değişiyor. Bir anakartta iki, dört ya da daha fazla fiziksel işlemci olabilir ve **her birinin kendi bellek denetleyicisi, kendi RAM yuvaları vardır.** İşlemciler birbirine hızlı bir bağlantıyla (Intel'de UPI, AMD'de Infinity Fabric) bağlanır.

Sonuç şu: bir çekirdek, kendi soketine takılı RAM'e doğrudan erişir; başka bir sokete takılı RAM'e ise komşu işlemcinin üzerinden gider. İkisi de "RAM" ama gecikmeleri farklıdır — uzak erişim tipik olarak %50 ile iki kat arası daha yavaştır. Bu düzene **NUMA** denir (Non-Uniform Memory Access): erişim süresi tek tip değildir.

Pratikte bunun iki görünen sonucu var:

- **İşletim sistemi bunu bilmek zorundadır.** Linux bellek ayırırken, isteyen process'in çalıştığı sokete yakın olan bölgeden vermeye çalışır. Scheduler da bir görevi başka bir sokete taşımakta isteksizdir; taşırsa görevin bütün belleği bir anda "uzak" hâle gelir.
- **Ölçüm yaparken tuzak kurar.** Aynı programın aynı makinede iki kez farklı hızda koşması, çoğu zaman ikinci koşuda başka bir sokete düşmüş olmasındandır. `numactl` gibi araçlar bu yüzden var: hangi soketin belleğini ve hangi çekirdekleri kullanacağını açıkça söylemene yarar.

Buradaki genel ders bu bölümün özetiyle aynı: **bellekte mesafe, gecikmeye dönüşür.** Piramidin katmanları arasında da böyleydi, aynı katmanın içinde de böyle.

En Alt Kademe: Disk / SSD

Piramidin tabanında disk var ve buradaki fark artık yalnızca hız değil.

Yukarıdaki bütün katmanlar — register, cache, RAM — **uçucudur**: elektrik kesildiği anda içindekiler kaybolur. Disk ve SSD ise **kalıcıdır**; bilgisayarı kapatıp bir yıl sonra açsan da

yazdığın şey yerinde durur. Bir programın "bilgisayarında yüklü olması" tam olarak bu demek: dosyaları diskte, program çalışmıyorken de orada duruyor. Çalıştırdığında bu dosyalar diskten silinmez, ihtiyaç duyulan parçaları RAM'e kopyalanır.

- Hız: SSD ~10-100 mikrosaniye; HDD ~1-10 milisaniye
- Boyut: 256 GiB - 8 TiB ve daha fazlası
- Teknoloji: NAND flash (SSD) veya manyetik plaka (HDD)

> SSD, HDD'ye göre 10-100 kat daha hızlıdır ama yine de RAM'e göre 1000 kat yavaştır. Bu yüzden çalışan programlar diskten değil, RAM'den okunur.

SSD'nin neden hâlâ mikrosaniye mertebesinde kaldığını merak ediyorsan, sebebi hareketli parça değil: NAND flash'ın kendisi. Bir SSD veriyi **sayfa** birimiyle okur ve yazar (tipik olarak birkaç kilobayt), ama silmeyi sayfa sayfa yapamaz — silme yalnızca çok daha büyük **blok**lar hâlinde mümkündür.

Bunun tuhaf bir sonucu var: var olan bir veriyi değiştirmek istediğinde denetleyici onu yerinde güncelleyemez. Yeni hâlini boş bir sayfaya yazar, eskisini "geçersiz" diye işaretler ve o bloğu ileride, uygun bir zamanda topluca siler. Bu arka plan temizliğine **çöp toplama** denir ve diskin dolu olduğu durumlarda yazma hızının neden düştüğünü açıklar: temizlenecek boş blok kalmamıştır.

Bir de her flash hücresinin sınırlı sayıda silme çevrimine dayandığını ekle. Denetleyici bu yüzden yazmaları bütün hücrelere eşit dağıtmaya çalışır; buna **wear leveling** denir. Yani bir SSD'nin içinde, işletim sisteminden tamamen bağımsız çalışan küçük bir yönetim yazılımı var ve senin gördüğün adreslerle çipteki gerçek konumlar arasındaki eşlemeyi o tutuyor. Tanıdık geldi mi? **[14. bölümde]** göreceğimiz sayfa tablolarının aynı fikri, bir kat aşağıda.

Neden Aradaki Fark Bu Kadar Açıldı?

Bir de tarihsel tarafı var, çünkü bu piramit hep bu kadar dik değildi.

1980'lerde bir işlemci RAM'den veri istediğinde birkaç saat çevrimi beklerdi; aradaki fark önemsizdi ve cache diye bir katmana çoğu makinede gerek bile duyulmuyordu. Sonraki yirmi

yılda işlemci hızları yılda yaklaşık %50 arttı, DRAM gecikmesi ise yılda ancak %7 iyileşti. İki eğri arasındaki makas her yıl biraz daha açıldı.

Bugün geldiğimiz nokta şu: RAM'den bir değer beklerken işlemci **yüzlerce talimat** çalıştırabilecek zamanı boşa harcıyor. Bu soruna literatürde **bellek duvarı** (memory wall) deniyor.

Dikkat edilecek ayrıntı şu: DRAM'in *bant genişliği* bu süre boyunca çok arttı — bir DDR5 modülü saniyede on gigabaytlarca veri taşıyabiliyor. Artmayan şey **gecikme** oldu. Yani boruyu genişletmeyi başardık, ama borunun bir ucundan diğerine gitmenin süresini kısaltamadık. Bu ayrım pratikte şu anlama geliyor: veriyi sırayla ve büyük parçalar hâlinde okuyan bir program hızlıdır, rastgele adreslere tek tek dokunan bir program ise duvara toslar.

Bu bölümdeki bütün katmanlar ve bir sonraki bölümdeki bütün hileler, o duvarı aşmak için değil, **etrafından dolaşmak** için var.

Hız ve Boyut Karşılaştırması

Katman	Tipik Gecikme / Mertebe	Tipik Boyut	Teknoloji
Register	~0.5 ns	~1 KiB toplam	CPU transistörü
L1 Cache	~1-5 ns	32-128 KiB	SRAM
L2 Cache	~5-15 ns	256 KiB - 1 MiB	SRAM
L3 Cache	~15-40 ns	8-128 MiB	SRAM
RAM	~80-120 ns	8-128 GiB	DRAM
SSD	~10-100 µs	256 GiB - 8 TiB	NAND Flash
HDD	~1-10 ms	1-20 TiB	Manyetik

> Fark dikkat çekici: Register ile HDD arasında **milyonlarca kat** hız farkı var.

Özet

> **Peki, ne öğrendik?**
>
> - Bellek tek bir katman değil; hız ile maliyet arasında denge kuran bir piramittir.
> - Tepede register'lar var: bir saat çevrimi mertebesinde erişim, toplamda birkaç yüz bayt.
> - L1, L2 ve L3 cache'ler **SRAM** ile yapılır. Hızlıdır ama bir bit için altı transistör harcadıkları için pahalı ve küçüktürler.
> - RAM **DRAM** ile yapılır. Bir bit için bir transistör ve bir kondansatör yeter; bu yüzden ucuz ve büyüktür, ama sürekli tazelenmesi gerekir.
> - Disk ve SSD kalıcıdır, buna karşılık RAM'e göre yüz binlerce kat yavaştır.
> - Her iniş, gecikmeyi kabaca bir mertebe artırır. Bir programın hızı çoğu zaman bu piramitte nerede çalıştığıyla belirlenir.

Piramidi tanıdık. Şimdi asıl soruya geçelim: CPU istediği veriyi cache'te *nasıl* buluyor, o veri oraya en baştan neden gelmişti ve yerini kime bırakıyor?

Bölüm 6: Cache Nasıl Çalışır

Bir önceki bölümde bellek piramidini yukarıdan aşağı gezdik ve her katmanda gecikmenin bir mertebe arttığını gördük. Ama asıl soruyu erteledim.

CPU bir adres istediğinde cache, o adresin elinde olup olmadığını nasıl anlıyor? Veri oraya ne zaman geldi, yerini kime bırakıyor, ve iki çekirdek aynı veriye aynı anda dokunduğunda kim haklı çıkıyor?

Bu bölüm o mekanizmanın içi. Aynı algoritmayı çalıştıran iki programın neden farklı hızlarda koştuğunu da burada göreceğiz — cevap neredeyse her zaman burada.

> **Bu bölümde neyi çözüyoruz?**
>
> - Cache'in neden tek tek bayt değil, 64 baytlık bloklar hâlinde çalıştığını göreceğiz.
> - Bir adresin cache içinde nasıl arandığını (tag, index, offset) çözeceğiz.
> - Locality'nin ne olduğunu ve cache-friendly kodun neden hızlı olduğunu göstereceğiz.
> - Çok çekirdekli bir işlemcide aynı verinin kopyalarının nasıl tutarlı tutulduğunu bağlayacağız.

Cache Line: Cache'in Çalışma Birimi

Cache'ler tek tek byte'ları değil, **cache line** denen bloklar hâlinde veri taşır. x86-64'te tipik cache line boyutu **64 byte**'tır.

Neden? Çünkü programlar genelde komşu verilere sırayla erişir. CPU tek bir bayt istese bile, cache o baytın içinde bulunduğu 64 baytlık bloğun tamamını getirir; böylece bir sonraki erişim büyük ihtimalle cache'te bulunur.

Buradaki blok gelişigüzel değil, **hizalıdır**. Bellek, başlangıç adresi 64'ün katı olan bloklara bölünmüş kabul edilir ve cache bu blokları ancak bütün hâlinde taşır. Yani istediğin bayt bloğun ortasına denk geliyorsa, ondan öncekiler de sonrakiler de aynı anda gelir. Bu ayrıntı

önemsiz görünüyor ama bölümün sonunda göreceğimiz *false sharing* sorununun tam olarak sebebi budur.

Cache Veriyi Nasıl Buluyor?

Şu ana kadar cache'i "hızlı bir kutu" gibi anlattım. Peki CPU bir adres istediğinde cache, o adresin içinde olup olmadığını nasıl anlıyor? Bütün satırları tek tek taramak mümkün değil; o kadar zamanı olsa zaten RAM'e gitmek yeterdi.

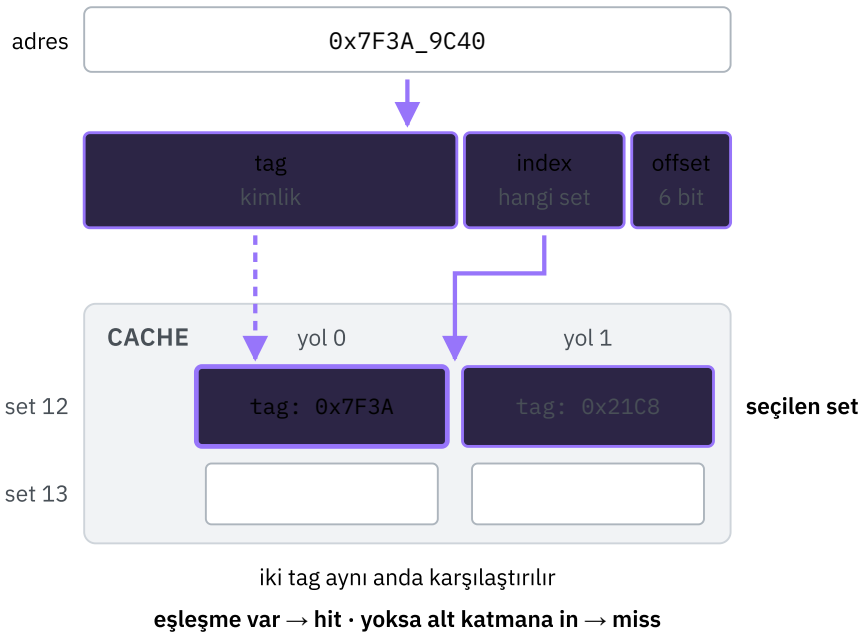
Numara, adresi üç parçaya bölmekte. 64 baytlık satırlar kullanan bir cache için:

- **Offset (son 6 bit).** Bulunan satırın içinde kaçıncı bayttasın? 64 bayt için tam 6 bit gerekir, çünkü $2^6 = 64$. **[1. bölümde]** ikinin kuvvetleriyle çalışırken kullandığımız hesabın aynısı.
- **Index (ortadaki bitler).** Bu adres cache'in hangi *kümesine* düşer? Cache, satırları kümelere böler ve bir adres yalnızca tek bir kümeye düşebilir. Böylece arama, koca cache yerine tek bir kümeyle sınırlanır.
- **Tag (kalan üst bitler).** Aynı kümeye düşen çok sayıda farklı adres var. Kümedeki satırın gerçekten *senin istediğin* adres olup olmadığını tag söyler.

Yani cache bir arama yapmıyor; index ile doğrudan doğru kümeye gidiyor ve orada yalnızca birkaç tag karşılaştırıyor. Hepsi tek saat çevriminde bitiyor.

Adımları tek tek izlemek istersen:

CACHE BİR ADRESİ NASIL BULUYOR?



1. **CPU bir adres istiyor.** Cache'in eline düz bir sayı geçiyor. Bu sayıyı tek başına aramanın yolu yok; önce parçalara ayrılması gerekiyor.
2. **Adres üçe bölünür.** En alttaki bitler satır içindeki konumu (offset), ortadakiler hangi sete bakılacağını (index), üstte kalan her şey ise kimlik bilgisini (tag) taşır. Hiçbir hesap yok — sadece bitleri okumak.
3. **Index seti seçer.** Index doğrudan bir satır numarasıdır. Cache o numaradaki sete gider; başka hiçbir yere bakmaz. Aramanın bu kadar hızlı olmasının sebebi budur.
4. **Tag'ler karşılaştırılır.** Sette kaç yol (way) varsa hepsinin tag'i aynı anda, donanımda karşılaştırılır. İki yollu bir cache'te iki karşılaştırıcı paralel çalışır.
5. **Hit ya da miss.** Tag tutuyor ve satır geçerliyse istenen bayt offset ile satırın içinden seçilir: hit. Tutmuyorsa bir alt katmana inilir ve dönen 64 baytlık satır bu setin bir yoluna yerleşir: miss.

Şimdi asıl tasarım sorusu: bir kümede kaç satır olmalı?

- **Direct-mapped** (küme başına 1 satır). En basit ve en hızlı: her adresin gidebileceği tek bir yer var, tek bir tag karşılaştırması yeter. Kusuru ağır: aynı kümeye düşen iki veriyi dönüşümlü kullanırsan, ikisi sürekli birbirini kovar. Cache'in geri kalanı bomboş dururken bu iki adres birbirini yiyip bitirir. Buna **conflict miss** denir.
- **Fully associative** (tek küme, bütün satırlar). Her adres her yere gidebilir, conflict miss diye bir şey kalmaz. Ama her erişimde bütün tag'lerin karşılaştırılması gerekir; bu da pahalı ve yavaştır. Yalnızca çok küçük yapılarda (örneğin bazı TLB'lerde) kullanılır.
- **N-way set associative** (küme başına N satır). İkisinin arası ve gerçek işlemcilerin tercihi. Tipik bir L1 8-way, L3 ise 16-way olabilir. Bir adres N farklı yerden birine düşebilir;

conflict miss ihtimali dramatik biçimde azalır, karşılaştırma maliyeti ise N ile sınırlı kalır.

Son soru: küme doluyken yeni bir satır gelirse hangisi atılır? İdeal cevap **LRU**'dur (Least Recently Used) — en uzun süredir dokunulmayanı at. Ama gerçek donanımda tam LRU tutmak, her erişimde kullanım sırasını güncellemek demektir ve bu, 16 yollu bir kümede pahalıya patlar. Bu yüzden işlemciler LRU'nun ucuz yaklaşıklarını kullanır: her satıra bir "yakında kullanıldı" biti koyup sırayla temizleyen `pseudo-LRU` gibi düzenekler. Sonuç LRU kadar iyi değildir, ama farkı ölçülemeyecek kadar küçük ve maliyeti kat kat düşüktür.

Yazma Nereye Gider?

Okumayı anlattık, peki CPU bir değere *yazdığında* ne oluyor? İki seçenek var ve aralarındaki tercih bölümün ilerisi için kritik.

Write-through — her yazma hem cache'e hem de bir alt katmana aynı anda gider. Basit ve güvenli: alt katman hiçbir zaman bayat kalmaz. Bedeli, her yazmanın yavaş katmanın hızına mahkûm olması.

Write-back — yazma yalnızca cache'te yapılır ve satır "kirli" (dirty) olarak işaretlenir. Alt katmana ancak o satır cache'ten atılırken yazılır. Aynı değişkene arka arkaya bin kez yazarsan, RAM'e yalnızca bir kez gidilir.

Modern L1/L2/L3'ün tamamı write-back çalışır; fark, RAM trafiğinde kat kat azalma demek. Ama bunun bir bedeli var ve o bedeli birazdan ödeyeceğiz: **cache'teki değer ile RAM'deki değer bir süre boyunca farklı olabilir.** Tek çekirdekli bir dünyada bu kimseyi ilgilendirmez. Sekiz çekirdek aynı veriye baktığında ise tam bir soruna dönüşür.

Prefetch: İstenmeden Getirmek

Cache'in hikâyesinde bir parça daha var ve satır bazlı döngünün neden bu kadar hızlı olduğunun yarısı burada saklı.

Modern işlemcilerde **prefetcher** denen bir devre, senin erişim düzenini izler. Ardışık adreslere eriştiğini ya da sabit bir adım aralığıyla ilerlediğini fark ederse, sıradaki cache line'ları sen istemeden çekmeye başlar. Tahmini tutarsa veri sen isteyene kadar çoktan cache'e gelmiş olur; yani gecikme ortadan kalkmaz, sadece görünmez olur — CPU beklerken değil, çalışırken taşınmıştır.

Tahmin tutmazsa bedelini de ödersin: boşuna taşınan satırlar hem bellek bant genişliğini yer hem de cache'te işine yarayacak başka satırların yerini kapar. Prefetcher'ın rastgele erişim düzenlerinde işe yaramamasının, hatta bazen zarar vermesinin sebebi budur.

Locality: Hızın Sırrı

Cache'lerin işe yaramasının nedeni iki locality prensibidir:

1. **Temporal Locality (Zamansal Yakınlık):** Az önce kullandığın bir veriyi yakında tekrar kullanma ihtimalin yüksektir. Örneğin bir döngüdeki sayaç değişkeni.
2. **Spatial Locality (Mekânsal Yakınlık):** Kullandığın verinin komşusuna da yakında erişme ihtimalin yüksektir. Örneğin bir dizinin elemanlarını sırayla okumak.

> **Hit ve miss.** İstediğin veri cache'te bulunursa buna **cache hit**, bulunamayıp bir alt katmana inmek gerekiyorsa **cache miss** denir. Toplam erişimlerin ne kadarının hit olduğu da **cache hit rate**'tir. Bir cache miss bedava değildir: bulunamayan veri, o katmanın kendi alt katmanından istenir ve zincir gerekirse RAM'e kadar iner. Aynı algoritmayı çalıştıran iki programın hız farkı çoğu zaman tam olarak burada saklıdır — aynı işlemci, aynı sayıda işlem, farklı erişim düzeni.

Cache-Friendly ve Cache-Unfriendly Kod

Locality prensibi pratikte nasıl işler? Aşağıdaki iki C kodu aynı matrisi topluyor ve neredeyse aynı görünüyor; aralarındaki tek fark iki döngünün sırası. Buna rağmen büyük matrislerde biri diğerinden kat kat hızlı çalışır ve sebebi tamamen cache'tir.

Cache-friendly örnek (satır bazlı erişim):

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    sum += matrix[i][j];  // Sequential access = cache hit
```

Cache-unfriendly örnek (sütun bazlı erişim):

```
for (int j = 0; j < N; j++)
  for (int i = 0; i < N; i++)
    sum += matrix[i][j];  // Strided access = cache miss
```

Neden satır bazlı (row-major) daha hızlı? C dilinde çok boyutlu diziler satır bazlı saklanır. `matrix[i][j]` erişimi komşu bellek adreslerine sırayla erişir; bu **spatial locality** sağlar. Bir cache line (64 byte) içinde ardışık veriler gelir ve CPU bir sonraki erişimde cache'de bulur. Sütun bazlı erişimde ise iki ardışık okuma arasındaki adres mesafesi bir matris satırının tamamı kadardır. Bu mesafeye **stride** (adım aralığı) denir. Stride bir cache line'dan büyük olduğu anda spatial locality tamamen kaybolur: getirilen 64 baytın yalnızca birini kullanır, kalan 63'ünü çöpe atarsın. Sonuç, neredeyse her erişimde bir cache miss — üstelik prefetcher da bu düzeni çoğu zaman takip edemez.

Bellek Kanalları ve Bant Genişliği

CPU ile RAM arasında veri, **bellek kanalları** (memory channels) üzerinden taşınır. Çift kanallı (dual-channel) bellek, iki kanalı paralel kullanarak teorik bant genişliğini iki katına çıkarır.

Burada çok sık yapılan bir çıkarım hatası var, onu baştan kapatalım: **çift kanal, tek bir erişimi hızlandırmaz.** İki farklı şeyden söz ediyoruz.

- **Gecikme** (latency), tek bir isteğin cevabının gelmesi için geçen süredir. Kanal sayısı bunu değiştirmez; istediğin bayt yine ~80 nanosaniye sonra gelir.
- **Bant genişliği** (bandwidth), birim zamanda taşınabilen toplam veri miktarıdır. Kanal sayısı yalnızca bunu artırır.

Benzetmesi otoyol: şerit eklemek arabaların hızını artırmaz, sadece aynı anda daha çok araba geçmesini sağlar. Bu yüzden tek bir çekirdek genellikle bant genişliğini tek başına doyuramaz; kanal sayısına asıl duyarlı olanlar çok çekirdekli iş yükleri ve sistem belleğini paylaşan entegre GPU'lardır.

> Bant genişliği, birim zamanda taşınabilen veri miktarıdır. DDR5-5600 bellek, kanal başına ~45 GB/s bant genişliği sunabilir.

Çok Çekirdek: Aynı Veriyi Kim Doğru Biliyor?

Şimdiye kadar tek bir çekirdeğin bellek görüşünü anlattık. Ama modern bir işlemcide sekiz çekirdek var ve **her birinin kendi L1 cache'i** var. Aynı değişkeni iki çekirdek birden okuduysa iki ayrı kopyası oluşur. Biri o kopyayı değiştirirse ne olur?

Cevap, çekirdekler arasında sessizce dönen bir konuşmadır: **cache coherence** (cache tutarlılığı). Donanım, bir cache line'ın her kopyasına bir durum etiketi tutturur ve çekirdekler birbirini haberdar eder. En yaygın anlatım MESI protokolüdür ve dört durumu vardır:

Durum	Anlamı
Modified	Yalnızca bende var ve değiştirdim; RAM'deki hâli bayat
Exclusive	Yalnızca bende var ama değiştirmedim
Shared	Başka çekirdeklerde de aynı kopya var, hepsi temiz
Invalid	Bendeki kopya geçersiz, yeniden almam gerek

Bir çekirdek yazmaya kalktığında önce diğerlerine "bu satır artık benim" der; onlar da kendi kopyalarını `Invalid` işaretler. Bu konuşma bedava değildir ve bedelini ödeyen çoğu zaman farkında bile olmaz.

DERİNLEŞME **False sharing: paralel kodun en sinsi yavaşlatıcısı**

İki thread'in tamamen **farklı** değişkenlere yazdığını düşün. Hiçbir paylaşım yok, kilit yok, yarış yok. Kod kâğıt üzerinde kusursuz paralel. Ama tek thread'li hâlinden yavaş çalışıyor.

Sebep şu: cache tutarlılığı değişken düzeyinde değil, **cache line düzeyinde** çalışır. İki değişken bellekte yan yanaysa aynı 64 baytlık satırı paylaşıyor olabilir. A thread'i kendi değişkenine yazdığında satırın tamamı sahiplenilir ve B'nin kopyası geçersizleşir. B kendi değişkenine yazmak için satırı geri ister. A yine ister. Satır iki çekirdek arasında ping-pong oynar.

Buna **false sharing** denir; "sahte" çünkü paylaşılan hiçbir veri yoktur, yalnızca adres komşuluğu vardır. Sonuç, gerçek bir kilit kadar pahalıya mal olabilir — üstelik kodda kilit görünmediği için teşhis etmesi çok daha zordur.

Çözüm, komşuluğu bozmaktır: her thread'in yazdığı değeri kendi cache line'ına yaslamak. C++'ta `alignas(64)`, Rust'ta `#[repr(align(64))]`, Java'da `@Contended` hep aynı işi yapar. Bellek israfı gibi görünen bu hizalama, ölçülebilir hız kazancı sağlar.

TLB: Adres Çevirisinin Hızlı Sözlüğü

Bellekle ilgili bir cache daha var ve hiyerarşinin parçası olduğu için burada anmam gerekiyor — tam hikâyesi ise sanal belleği anlatmadan tamamlanmıyor.

Kısaca durum şu: CPU'nun kullandığı adresler doğrudan RAM'deki konumlar değildir. Programın gördüğü adres ile RAM'deki gerçek adres arasında bir çeviri yapılır ve bu çevirinin tablosu da bellekte durur. Yani her bellek erişimi aslında iki erişim demektir: önce çeviriyi bul, sonra veriyi al.

TLB (Translation Lookaside Buffer) bu çevirilerin sonucunu saklayan küçük ve çok hızlı bir tablodur. Tipik olarak birkaç yüz giriş tutar — evet, sadece birkaç yüz. Bir TLB isabeti neredeyse bedavadır, çünkü çeviri araması cache erişimiyle paralel yürütülür; CPU veriyi ararken adresi de çeviriyordur.

TLB'yi asıl önemli kılan ise ıskalamanın maliyeti. Sıradan bir cache miss'te veri bir alt katmandan gelir. TLB miss'te ise CPU'nun **çeviriyi bulmak için belleğe birden fazla ek erişim yapması** gerekir; yani asıl veriye ulaşmadan önce birkaç tur atarsın. Bu yüzden birkaç yüz girişlik minik bir tablo, gigabaytlarca RAM'i olan bir makinede performansı belirleyebilir.

Bir ipucu daha vereyim: TLB kapasitesini artırmanın en etkili yolu tabloyu büyütmek değil, **sayfaları büyütmektir**. Tek bir TLB girişi 4 KiB yerine 2 MiB'lık bir alanı kapsarsa, aynı tabloyla beş yüz kat geniş bir bellek alanını çevirebilirsin. Bu mekanizmanın nasıl çalıştığını ve neden her zaman iyi bir fikir olmadığını **[13. bölümde]** ayrıntısıyla göreceğiz.

> **Bir yana: Neden Cache Line 64 Byte?**
>
> Cache line boyutu tarihsel olarak evrimleşmiştir. Pentium 4 döneminde 32 byte kullanılırken, modern x86-64 işlemcilerde standart 64 byte'tır. Bazı ARM tasarımlarında ise 128 byte denenmektedir.
>
> Daha büyük cache line = daha fazla spatial locality avantajı (komşu veriler tek seferde gelir). Ancak erişilen verinin komşuları gerçekten kullanılmayacaksa daha büyük line, daha fazla gereksiz byte taşımak ve bant genişliği harcamak demektir. 64 byte, modern x86 sistemlerde bu denge için yaygın bir sweet spot'tur; başka mimarilerde farklı line boyutları görülebilir.

Kendi Sisteminde Cache'leri Keşfet

Buraya kadar anlattığım katmanlar soyut sayılar olarak kalmasın. Okuduğun her şeyin karşılığı kendi makinende duruyor ve birkaç komutla görebilirsin. Linux kullanıyorsan doğrudan aşağıdakileri dene; çıktıdaki rakamları bölümün başındaki tabloyla yan yana koy.

Cache line boyutu:

```
getconf LEVEL1_DCACHE_LINESIZE
```

Tüm cache hiyerarşisi:

```
lscpu | grep -i cache
```

Beklenen çıktıya örnek:

```
L1d cache:                       384 KiB (8 instances)
L1i cache:                       256 KiB (8 instances)
L2 cache:                        4 MiB (8 instances)
L3 cache:                        32 MiB (2 instances)
```

Bu çıktıyı okurken iki şey kafa karıştırır, ikisini de açalım.

“(8 instances)” ne demek? Sekiz ayrı çekirdekte sekiz ayrı kopya var demek — ve gösterilen boyut bu kopyaların **toplamı**. Yani çekirdek başına L1d aslında 384 / 8 = 48 KiB. Bölümün başında “L1 genelde 32-128 KiB” derken kastettiğim de çekirdek başına düşen bu değerdi. L3’ün “(2 instances)” çıkması ise cache’in iki ayrı dilime bölündüğü anlamına gelir; AMD işlemcilerde her çekirdek kümesinin (CCX) kendi L3 dilimi olur ve bir çekirdeğin komşu dilime erişmesi kendi dilimine erişmesinden yavaştır.

Neden L1d ve L1i ayrı? Çünkü CPU aynı saat çevriminde hem sıradaki talimatı okumak hem de veri üzerinde çalışmak ister. İkisi tek bir cache’i paylaşsaydı her çevrimde birbirlerini beklerlerdi. Ayırmak, ikisine aynı anda erişebilmeyi sağlar. **[2. bölümdeki]** von Neumann tartışmasını hatırlıyorsan: mimari düzeyde tek bir bellek var, ama çekirdeğin içinde iş ikiye bölünüyor. *Modified Harvard* denen şey tam olarak budur ve `lscpu` çıktısında gözünle gördüğün kanıtı bu iki satır.

Tek satırda cache bilgisi:

```
cat /proc/cpuinfo | grep -m1 cache
```

Özet

> **Peki, ne öğrendik?**
>
> - Cache tek tek bayt değil, hizalı **cache line**'lar hâlinde çalışır; x86-64'te bir satır 64 bayttır.
> - Bir adres cache içinde aranırken **offset, index ve tag** olmak üzere üçe bölünür.
> - **Associativity**, bir adresin cache'te kaç farklı yere düşebileceğini söyler. Arttıkça çakışma azalır, karşılaştırma maliyeti artar.
> - **Locality**, cache'in çalışmasının tek sebebidir: programlar yakın zamanda ve yakın adreste kullandıklarına tekrar dokunur.
> - Yazmalar genelde **write-back** yapılır; bu yüzden cache'teki değer ile RAM'deki değer bir süre farklı olabilir.
> - Çok çekirdekte tutarlılığı **MESI** gibi protokoller sağlar ve bu tutarlılık cache line düzeyinde çalışır. **False sharing** tam olarak buradan doğar.
> - Cache miss CPU pipeline'ını duraksatır; bunu **[8. bölümde]** işleyeceğiz. TLB'nin arkasındaki sayfa tablolarını ise **[14. bölümde]** açacağız.

Bellek tarafını burada kapatıyoruz. Sırada kitabın en başında sorduğumuz o ilk soru var:

> CPU birden fazla process'i takip etmiyorsa ve sadece talimat üstüne talimat yürütüyorsa, neden tek bir programın içinde sıkışıp kalmıyor? Birden fazla program aynı anda nasıl çalışabiliyor?

Cevap şu: saatler. Coldplay'in *Clocks*'uyla aynı cevap — ama burada kastettiğim timer'lar. Bu şakayı yapmayacağıma dair kimseye söz vermemiştim.

Bölüm 7: Zamanı Dilimle

Diyelim ki bir işletim sistemi yazıyorsun ve kullanıcıların aynı anda birden fazla program çalıştırabilmesini istiyorsun. Ama elinde çok çekirdekli bir işlemcinin lüksü yok; CPU aynı anda yalnızca tek bir talimat yürütebiliyor.

Neyse ki akıllı bir işletim sistemi geliştiricisisin. CPU'yu process'ler arasında sırayla paylaştırarak sahte bir paralellik yaratabileceğini fark ediyorsun: aralarında yeterince hızlı geçiş yapar ve her birine küçük bir çalışma payı verirsen, hiçbiri CPU'yu tek başına işgal etmeden sistem duyarlı kalır.

Bu bölümde çalışan her programa **process** diyeceğiz. Terimi baştan sabitliyorum, çünkü aynı şey için "görev", "süreç" ve "iş" kelimelerinin dönüşümlü kullanılması, scheduler anlatısını gereksiz yere bulandırıyor.

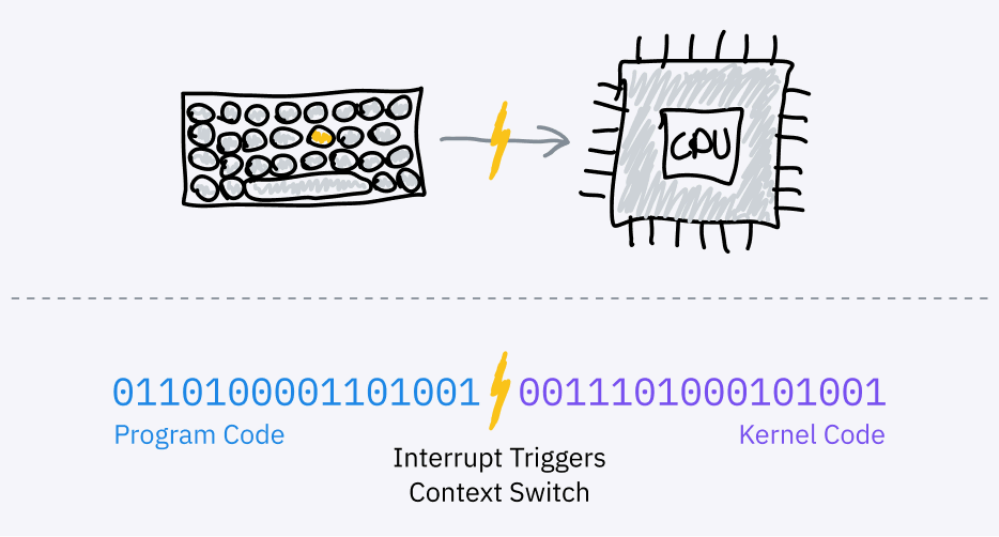
> **Bu bölümde neyi çözüyoruz?**
>
> - Bir CPU çekirdeğinin aynı anda tek yürütme akışı çalıştırmasına rağmen çoklu görev hissini nasıl verdiğini göreceğiz.
> - Timer interrupt, preemption ve context switch kavramlarını ayıracağız.
> - Linux scheduler anlatısını güncel EEVDF modeliyle uyumlu ama basit bir zihinsel modele indireceğiz.

Peki program kodu çalışırken kontrolü geri nasıl alacaksın? Biraz araştırınca, çoğu bilgisayarda timer chip'ler bulunduğunu keşfediyorsun. Belirli bir süre geçince işletim sisteminin interrupt handler'ına geçişi tetikleyecek şekilde bu timer chip'leri programlayabiliyorsun.

Donanım Interrupt'ları

Bir önceki bölümde kontrolün user-space programından işletim sistemine nasıl geçtiğini görmüştük: program özel bir talimat çalıştırıyor, CPU mode değiştirip kernel'ın önceden belirlediği bir adrese atlıyordu.

Buradaki kilit ayrıntıyı fark et: geçişi **program** başlatıyordu. Yani kernel ancak program ona söz verdiği anda söz sahibi oluyordu. Çoklu görev için bu yeterli değil — sonsuz döngüye giren bir program hiçbir zaman söz vermez ve makine kilitlenir. Kernel'ın, kimseden izin almadan sözü geri alabilmesi gerekiyor.



Neyse ki donanımın kendisi de CPU'nun sözünü kesebilir. Bir interrupt'ı program değil de donanım tetikliyorsa buna *hardware interrupt* denir; farkı kimin başlattığıdır.

En sezgisel örneği yukarıdaki çizimde: klavyede bir tuşa bastığında klavye denetleyicisi CPU'ya bir sinyal gönderir. CPU o an ne yapıyorsa yapsın, sıradaki talimatı almadan önce işini askıya alır, nerede kaldığını kaydeder, kernel'ın ilgili handler'ına atlar; handler tuşu okuyup işini bitirince CPU kaldığı yerden devam eder. Kesilen programın bundan haberi bile olmaz.

Scheduler'ın bütün numarası, bu mekanizmayı kendi lehine kullanmaktır. Klavyeden gelen sinyali beklemek yerine, kendi sinyalini kendisi sipariş eder — bunu bir *timer chip*'e söyleyerek yapar. Klasik örnek PIT'tir; modern x86 makinelerde işi her çekirdeğin kendi local APIC timer'ı, ARM tarafında ise generic timer üstlenir. Fikir hepsinde aynı: donanıma "şu kadar sonra beni dürt" demek.

1. Program koduna geçmeden önce işletim sistemi, timer chip'i belirli bir süre sonra interrupt tetikleyecek şekilde ayarlar.
2. İşletim sistemi user mode'a geçer ve programın bir sonraki talimatına atlar.
3. Süre dolunca timer chip, kernel mode'a geçişi ve işletim sistemi kodunun devreye girmesini sağlayan bir hardware interrupt üretir.

4. İşletim sistemi artık mevcut programın durumunu kaydedebilir, başka bir programı yükleyebilir ve aynı döngüyü sürdürebilir.

Buna *preemptive multitasking* denir; bir process'in zorla kesilmesine de *preemption* denir. Bu satırları tarayıcıda okurken aynı anda müzik de dinliyorsan, bilgisayarın şu anda bu döngüyü saniyede binlerce kez çeviriyor.

Timeslice Hesabı

Peki bu süreyi ne kadar tutacaksın? Scheduler'ın bir process'e, kesilmeden önce tanıdığı bu süreye *timeslice* deniyor ve seçimi tamamen sana ait. En basit yol herkese aynı süreyi verip sırayla dönmektir; buna *fixed-timeslice round-robin* scheduling denir.

> **Küçük bir jargon notu**
>
> Timeslice için bazen "quantum" dendiğini biliyor muydun? Artık biliyorsun ve bunu uygun bir ortamda kullanarak teknik arkadaşlarını etkileyebilirsin. Kitap boyunca her cümlede *quantum* kelimesini kullanmadığım için takdir bekliyorum.
>
> Linux kernel geliştiricileri ayrıca timer tick'lerini saymak için jiffy adlı zaman birimini kullanır. Frekans kernel config'e göre değişebilir; masaüstü/server kurulumlarında 100, 250 veya 1000 Hz gibi değerler görebilirsin.
>
> Günümüz Linux kurulumları **tickless** çalışır ama bunun iki ayrı seviyesi var, karıştırmak kolay. CPU boştayken tick'i durdurmak (`CONFIG_NO_HZ_IDLE`) bugün varsayılan davranıştır; laptop pilini ve CPU'nun uyku modlarına (C-states) inebilmesini doğrudan etkiler. Bir çekirdekte tek bir görev varken tick'i tamamen kesmek (`CONFIG_NO_HZ_FULL`) ise ayrı bir yapılandırmadır, varsayılan değildir ve ancak çekirdek izole edilerek elle açılır — yüksek başarımlı hesaplama ve düşük gecikmeli iş yüklerinin dünyasıdır.

Sabit timeslice yaklaşımının küçük ama önemli bir geliştirmesi, "herkes makul sürede tekrar CPU görebilsin" hedefini koymaktır. Eski CFS anlatılarında bu fikir çoğu zaman *target latency* üzerinden açıklanırdı: makul sayıda process varken birinin preempt edildikten sonra tekrar CPU görmesi için geçmesini istediğin ideal en uzun süre. Tanım kulağa soyut geliyor; birazdan göreceğin diyagramda yerine oturacak.

Timeslice süresi, target latency'nin toplam process sayısına bölünmesiyle hesaplanır. Bu, sabit süre vermekten daha iyidir; çünkü çalışan process sayısı az olduğunda gereksiz context switch'leri azaltır. Örneğin target latency 15 ms ve toplam 10 process varsa, her biri yaklaşık 1,5 ms çalışır. Yalnızca 3 process varsa her biri 5 ms'lik daha uzun bir dilim alır ve yine aynı target latency içinde sıraya girmiş olur.

Context Switch Tam Olarak Neyi Değiştirir?

"Context switch pahalıdır" cümlesini sık duyarsın ama neyin pahalı olduğu genelde söylenmez. Açalım, çünkü kitabın geri kalanında birkaç kez daha karşımıza çıkacak.

Bir process'i durdurup başka birine geçmek için kernel'ın kaydetmesi gereken "durum" üç parçadan oluşur:

1. **CPU register'ları.** Genel amaçlı register'lar, instruction pointer, stack pointer, flag'ler. Bunlar RAM'e, kernel'ın o process için tuttuğu veri yapısına yazılır (Linux'ta `task_struct` içindeki alanlara). Birkaç yüz bayt; kopyalaması ucuz.
2. **Bellek haritası.** CPU'ya, sanal adresleri çevirirken artık başka bir tablo kullanması söylenir. **[13. bölümde]** ayrıntısına ineceğiz.
3. **Kernel tarafındaki muhasebe.** Hangi process ne kadar çalıştı, sırada kim var.

Şimdi sürprize gel: yukarıdakilerin hiçbiri asıl maliyet değil. Asıl maliyet, geçişin **kendisinden sonra** ortaya çıkar. Yeni process çalışmaya başladığında CPU'nun cache'i hâlâ bir öncekinin verisiyle doludur ve adres çevirilerini hızlandıran TLB de öyle. Yeni process ilk birkaç bin talimatı boyunca sürekli ıskalar; cache ve TLB yeniden ısınana kadar yavaş çalışır. Kopyalanan birkaç yüz bayt nanosaniyeler sürer, bu ısınma ise mikrosaniyeler.

Bu yüzden çok küçük timeslice'lar performansı düşürür: sistem, işi yapmaktan çok cache ısıtmakla uğraşır. Scheduler'larda bir *minimum granularity* sınırının bulunmasının sebebi budur. Sınır devreye girdiğinde target latency aşılabilir, ama sistem kendi kuyruğunu kovalamaktan kurtulur.

Bu sayıları "Linux her zaman tam şunu yapar" diye okuma. CFS döneminde bu adalet hesabını `sched_latency_ns` ve `sched_min_granularity_ns` gibi ayarlar ile virtual runtime fikri belirlerdi. Linux 6.6'dan itibaren ise fair scheduler'ın karar mantığı CFS'den **EEVDF**'ye

(Earliest Eligible Virtual Deadline First) taşındı; bu bir belge güncellemesi değil, kodun kendisinde olan bir değişiklik. Bugün masaüstünde çalışan kernel'da işi yapan hesap budur ve eski ayarların yerini tek bir `sched_base_slice_ns` aldı. EEVDF de CPU süresini adil bölüştürmeye çalışır, ama bunu görevlerin geride kalma miktarını (*lag*) ve sanal deadline'larını hesaba katarak yapar.

DERİNLEŞME **CFS neyi çözemedi, EEVDF ne getirdi?**

CFS'nin fikri tek cümleyle anlatılır: her göreve *virtual runtime* adlı bir sayaç tut, kim en az CPU görmüşse onu çalıştır.

Sayacın adındaki "virtual" boşuna değil: biriktirdiği şey gerçek geçen süre değil, görevin ağırlığına bölünmüş süredir. Yüksek öncelikli (düşük `nice` değerli) bir görevin sayacı daha yavaş ilerler, dolayısıyla sıraya daha sık girer. Öncelik kavramı CFS'ye tam olarak buradan girer.

Adalet açısından kusursuz bir tasarım. Ama bir şeyi hiç sormaz: **bu görev ne kadar acil?**

Sorunu somutlaştıralım. Aynı makinede iki iş var:

- Bir ses uygulaması. Saniyede yüzlerce kez uyanıp birkaç yüz mikrosaniyelik iş yapması gerekiyor. Toplamda çok az CPU tüketiyor, ama **geç kalırsa ses cızırdıyor**.
- Bir derleme işi. CPU'yu doyasıya kullanmak istiyor, ne kadar geç başladığı hiç önemli değil.

CFS'nin gözünde ses uygulaması "az CPU almış" bir görevdir, o kadar. Uyandığında bir preemption kontrolü yapılırdı elbette — ama karar yalnızca vruntime farkına ve `sched_wakeup_granularity_ns` eşiğine bakardı. Ses uygulamasının "acilim" diyebileceği bir kanal yoktu; oysa az CPU almış olmak ile acil olmak aynı şey değildir. Adalet sağlanır, ses cızırdar. Bu yüzden yıllarca `sched_latency_ns` gibi ayarlar elle kurcalandı ve "düşük gecikmeli kernel" yamaları dolaştı.

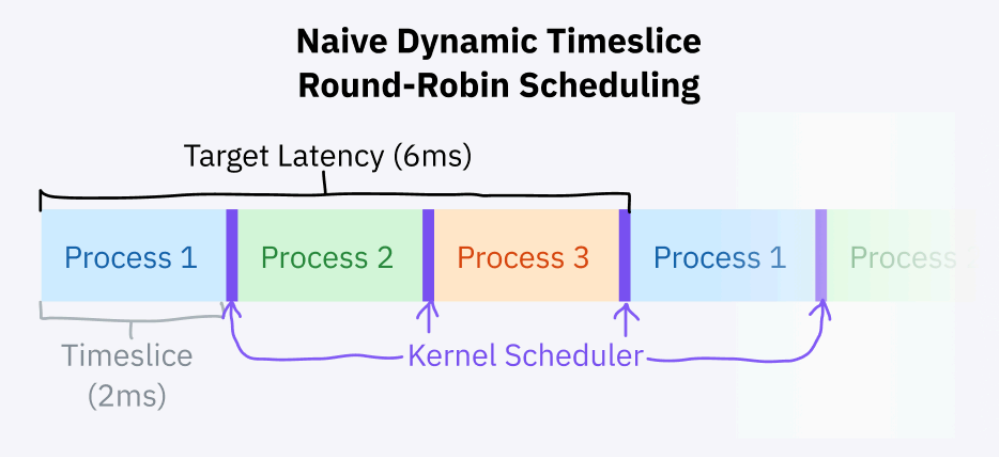
EEVDF (Earliest Eligible Virtual Deadline First) iki kavram ekleyerek bunu düzeltir:

- **Lag (geride kalma).** Her görev için "adil payına göre ne kadar geridesin?" hesaplanır. Lag pozitifse görev alacaklıdır ve *eligible* sayılır; negatifse hakkından fazlasını almıştır ve sırasını beklemek zorundadır.
- **Sanal deadline.** Eligible görevler arasında yarış, deadline'a göre çözülür. Bir görevin deadline'ı, **istediği dilim uzunluğuna** göre hesaplanır: kısa dilim isteyen erken bir deadline alır.

İşte kritik nokta burada. Ses uygulaması "bana çok CPU ver" demez; **"bana kısa dilimler ver ama çabuk ver"** der. EEVDF'de bu bir dilek değil, hesabın parçasıdır: kısa dilim istemek erken deadline demektir, erken deadline önce çalışmak demektir. Karşılığında görev daha sık preempt edilir — yani gecikme ile verim arasındaki takası artık scheduler değil, görevin kendisi seçer.

Linux'ta sıradan bir görev bile `sched_setattr` çağrısının `sched_runtime` alanıyla "bana şu uzunlukta dilimler ver" diyebilir; bu, EEVDF ile gelen ve görev başına dilim uzunluğu tanımlayan yeni bir yetenektir. `nice` değerleri de kaybolmaz; ağırlık olarak lag hesabına girer.

Tarihsel bir not: EEVDF yeni bir icat değil. Ion Stoica ve Hussein Abdel-Wahab'ın 1995 tarihli oransal paylaşım (proportional share) çalışmasına dayanır. Yani otuz yıl bekleyip nihayet üretim scheduler'ı olan bir teori. Sistem araştırmalarında bu hiç de sıra dışı değildir.



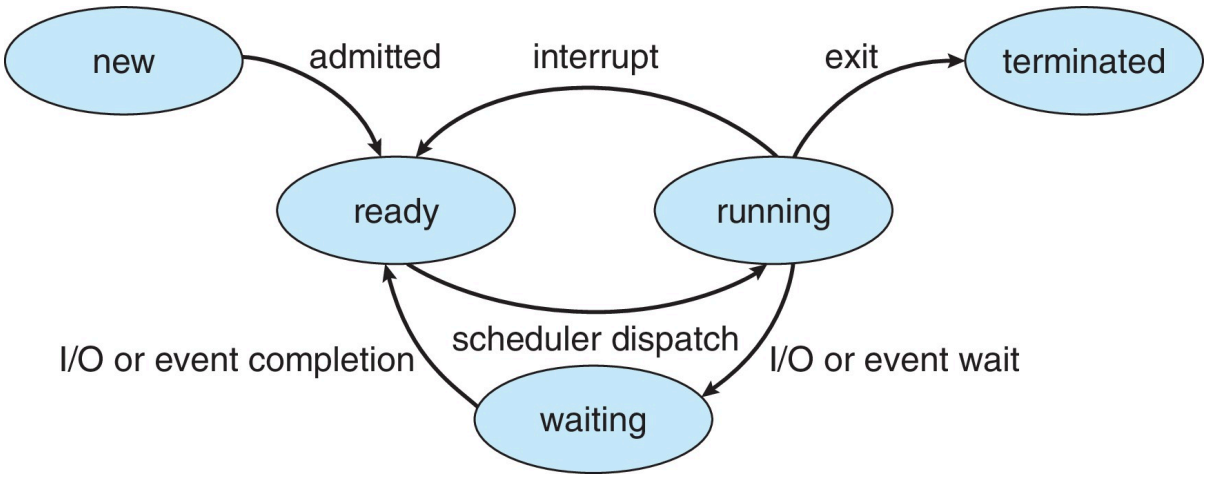
Bu temel timeslice hesabı, scheduler'ı anlamak için iyi bir ilk basamak; ama günümüz makinelerinin yaptığı şeyin tamamı değil. Gerçek scheduler'lar öncelikleri, deadline'ları, uyku/uyanma davranışını ve **CPU affinity**'yi de hesaba katar.

Bunlardan sonuncusu bir sonraki bölüme doğrudan köprü kuruyor, o yüzden açalım: CPU affinity, bir görevin hangi çekirdeklerde çalışabileceğini belirler. Scheduler bir görevi mümkün olduğunca **son çalıştığı çekirdekte** tutmaya çalışır, çünkü o çekirdeğin cache'i zaten o görevin

verisiyle doludur. Görevi başka bir çekirdeğe taşımak, biraz önce konuştuğumuz ısınma maliyetini sıfırdan ödetir. Yani "boştaki çekirdeğe ver" her zaman doğru cevap değildir.

Linux uzun süre Completely Fair Scheduler ile anlatıldı; güncel kernel belgeleri artık EEVDF tarafını anlatıyor. Temel ders değişmiyor: CPU kısa aralıklarla paylaştırılıyor, sadece "sıradaki kim?" sorusunun hesabı akıllanıyor.

Scheduler'ın ne yaptığını tam oturtmak için bir process'in hayatına yukarıdan bakalım. Bir process doğduğu andan (`new`) sonlandığı ana (`terminated`) kadar üç durum arasında gidip gelir: çalışmaya hazır bekliyor (`ready`), CPU'da çalışıyor (`running`), bir şey bekliyor (`waiting`). Scheduler'ın işi bu haritanın yalnızca iki okundan ibarettir: `ready → running` ve `running → ready`. `waiting` ise scheduler'ın kararı değildir — process diskten veri, ağdan paket ya da klavyeden tuş beklerken kendi isteğiyle oraya düşer.



Kaynak: Silberschatz, Galvin, Gagne — Operating System Concepts, Ch. 3 — Processes, Slide 9.

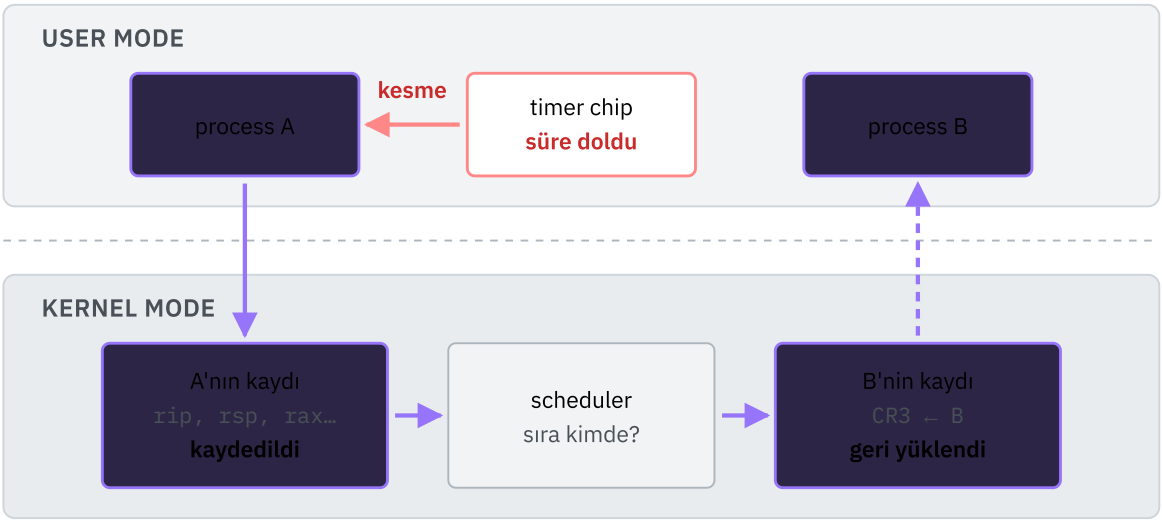
> Yukarıdaki target latency diyagramını bir tarih dersi gibi oku: fikri doğru anlatır, ama bugünkü kernel'ın yaptığı şeyin birebir kendisi değildir. Gerçek scheduler'ın davranışı kernel sürümüne ve derleme yapılandırmasına göre değişir. Değişmeyen tek cümle şu: CPU kısa dilimlere bölünüyor.

İşletim sistemi bir process'i her preempt ettiğinde, sıradakinin kayıtlı *context*'ini (yürütme bağlamını) yüklemesi gerekir; buna bellek context'i de dâhildir. Bu, CPU'ya sanal adresleri çevirirken farklı bir *page table* kullanmasını söyleyerek yapılır. Dikkat: page table'ın kendisi RAM'de durur, CPU'ya verilen şey yalnızca o tablonun bellekteki adresidir (x86-64'te bu adres

`CR3` adlı özel bir register'a yazılır). Yani kernel tek bir register'ı değiştirerek CPU'nun gördüğü bütün bellek haritasını değiştirmiş olur. Programların birbirinin belleğine erişememesinin sebebi tam olarak budur; burası tek başına iki bölüm eden bir konu ve **[Bölüm 13]** ile **[Bölüm 16]** tam olarak oraya iniyor.

Bütün bu geçişi baştan sona izleyelim:

BİR CONTEXT SWİTCH ADIM ADIM



1. **A çalışıyor.** Process A user mode'da yürüyor. Register'lar onun değerlerini taşıyor, sayfa tablosu onun belleğini gösteriyor.
2. **Timer interrupt.** Anakart üzerindeki timer'ın süresi doluyor ve CPU'ya bir hardware interrupt gönderiyor. A bunu istemedi, haberi bile yok.
3. **Kernel devralır.** CPU ayrıcalık seviyesini yükseltir ve kernel'ın önceden kurduğu handler'a atlar. A'nın bütün görünür durumu — register'lar, instruction pointer, bayraklar — kernel'daki kaydına yazılır.
4. **Scheduler seçim yapar.** Sırada kimin olduğuna scheduler karar verir. Linux'ta bugünkü varsayılan EEVDF; her process'in ne kadar hak ettiği ile ne kadar aldığı arasındaki farka bakar.
5. **B geri yüklenir.** B'nin kaydı register'lara yazılır ve `CR3` değişir — tek bir register, process'in gördüğü bütün bellek haritasını değiştirir. User mode'a dönüldüğünde B, hiç durmamış gibi devam eder.

Öncelik Tersine Dönmesi: Mars'ta Yaşanmış Bir Hata

Scheduler'ın işini zorlaştıran şey yalnızca "kim ne kadar aldı" değil. Görevler birbirine bağlanabilir ve o zaman öncelik sıralaması tersine dönebilir.

Klasik senaryo üç görevle kurulur:

1. **Düşük öncelikli** bir görev, paylaşılan bir kaynağın kilidini alır.
2. **Yüksek öncelikli** bir görev uyanır, aynı kilidi ister ve bekler. Kilit düşük öncelikli görevde olduğu için bekleyecektir.
3. Tam o sırada **orta öncelikli** bir görev uyanır. Düşük öncelikliyi preempt eder — çünkü ondan önceliklidir.

Sonuç şu: yüksek öncelikli görev, orta öncelikli görev yüzünden beklemektedir. Aralarında hiçbir doğrudan ilişki yoktur. Öncelik sıralaması fiilen tersine dönmüştür ve buna **priority inversion** denir.

Bu senaryonun en meşhur örneği bir laboratuvarda değil, Mars'ta yaşandı. 1997'de Mars Pathfinder'ın **iniş aracı** yüzeye kondaktan sonra düzenli aralıklarla kendini yeniden başlatmaya başladı. (Karıştırılması çok yaygın: sorun aracın taşıdığı Sojourner gezgininde değildi; gezginin bilgisayarı çok daha basit ve tamamen ayrı bir sistemdi.) Sebep tam olarak yukarıdaki üçgendi: yüksek öncelikli bir görev, paylaşılan veri alanını — VxWorks üzerindeki *information bus*'ı — koruyan yazılım kilidini bekliyor, bu arada orta öncelikli iletişim görevi çalışıyor ve watchdog zamanlayıcısı "yüksek öncelikli görev takıldı" deyip sistemi resetliyordu. Hata Dünya'daki ikiz sistemde yeniden üretildi ve milyonlarca kilometre öteye gönderilen bir yapılandırma değişikliğiyle düzeltildi.

Çözümün adı **priority inheritance** (öncelik devralma): kilidi tutan düşük öncelikli görev, kendisini bekleyen en yüksek önceliği geçici olarak devralır. Böylece orta öncelikli görev onu preempt edemez, kilit hızlıca bırakılır ve öncelikler normale döner.

Linux'ta programların birbirini beklemek için kullandığı temel kilit mekanizmasının adı *futex*'tir (fast userspace mutex). İsmindeki "fast" şuradan geliyor: kilit için çekişme yoksa iş tamamen user-space'te, kernel'a hiç girilmeden biter; yalnızca gerçekten beklemek gerektiğinde syscall yapılır. Öncelik devralmayı destekleyen sürümüne **PI-futex** denir ve gerçek zamanlı iş yükleri için kullanılan `PREEMPT_RT` yapılandırmasının önemli bir parçasıdır.

Buradaki asıl ders şu: **scheduler tek başına adaleti garanti edemez.** Görevler kaynak paylaşıyorsa, kilitlerin de öncelikten haberi olması gerekir.

Kenar Notu: Kernel'in Kendisi Kesilebilir mi?

Şu ana kadar yalnızca user-space process'lerinin preempt edilmesinden söz ettik. Oysa bir syscall'ın işlenmesi ya da driver kodunun yürütülmesi çok uzun sürerse, kernel kodu da programları yavaşlatabilir.

Linux kernel'ı derleme seçeneğine göre dört farklı preemption modelinden biriyle çalışır ve her biri farklı bir kullanıcıyı memnun etmek için var:

- `PREEMPT_NONE` — kernel kodu hiç kesilmez. Toplam verimi en yükseğe çıkarır; toplu iş çalıştıran sunucuların tercihi.
- `PREEMPT_VOLUNTARY` — kernel yalnızca kendi önceden işaretlediği güvenli noktalarda sırayı bırakır. Genel amaçlı sunucuların dengesi.
- `PREEMPT` — kernel kodu da kesilebilir. Masaüstünde tercih edilen model; arayüzün takılmamasını buna borçlusun.
- `PREEMPT_RT` — gerçek zamanlı. Kernel o kadar agresif kesilebilir hâle gelir ki spinlock'ların çoğu bile uyuyabilir olur. Ses üretimi, robotik ve endüstriyel kontrol bu modeli kullanır.

Genel kural şu: kernel kritik bir kilit tutmadığı sürece kendi kodu da kesilebilir ve scheduler başka bir işe geçebilir. Modern kernel'larda `CONFIG_PREEMPT_DYNAMIC` sayesinde bu seçim derleme anına çakılı değildir; boot sırasında belirlenebilir.

Kernel ya da driver yazmıyorsan bu ayrıntıyı ezberlemene gerek yok. Yine de aklının bir köşesinde dursun: bir programın neden "takıldığını" araştırırken karşılaştığın gecikmelerin bir kısmı user-space'te değil, tam olarak burada — kernel'ın kendi kodunda — doğar.

Kenar Notu: Programların Gönüllü Sıra Verdiği Günler

Klasik Mac OS ve NT öncesi Windows sürümleri dâhil, eski işletim sistemleri preemptive multitasking'i değil onun atası olan bir modeli kullanıyordu. İşletim sistemi "şimdi seni

keseceğim” demiyordu; programlar kontrolü kendi istekleriyle bırakıyordu. Program bir software interrupt tetikleyip “istersen şimdi başkasını çalıştırabilirsin” diyordu ve işletim sisteminin sözü geri alabilmesinin tek yolu buydu.

Buna *cooperative multitasking* denir. Büyük kusurları vardır: kötü niyetli ya da kötü yazılmış programlar tüm sistemi kolayca dondurabilir ve zaman hassasiyetinin önemli olduğu işlerde kararlı davranış elde etmek neredeyse imkânsızdır. Bu yüzden teknoloji dünyası preemptive multitasking’e uzun zaman önce geçti ve bir daha geri dönmedi.

Özet

> **Peki, ne öğrendik?**
>
> - Çoklu görev bir yazılım hilesi değil, bir **donanım** özelliğinin üstüne kurulur: timer chip’in düzenli aralıklarla ürettiği interrupt.
> - Interrupt geldiğinde kontrol kernel’a geçer; kernel çalışan process’in bütün durumunu kaydedip başka birininkini yükler. Buna **context switch** denir.
> - Hangi process’in ne kadar çalışacağına **scheduler** karar verir. Linux’ta bugünkü varsayılan EEVDF’tir.
> - Zaman dilimi sabit bir sayı değildir; sistemdeki yüke ve önceliklere göre hesaplanır.
> - **Öncelik tersine dönmesi**, düşük öncelikli bir işin yüksek öncelikli bir işi dolaylı yoldan bekletmesidir; Mars’a kadar gitmiş gerçek bir hata sınıfıdır.
> - Eski sistemlerdeki **cooperative multitasking** sırayı programın gönüllü bırakmasına dayanıyordu ve tek bir kötü program tüm sistemi dondurabiliyordu.

Peki CPU aynı anda tek bir talimat yürütüyorsa, bilgisayarın nasıl bu kadar hızlı olabiliyor? Cevap işlemcinin kendi içinde sakladığı hilelerde. Sıradaki bölümde onları tek tek açacağız.

Bölüm 8: İşlemciyi Hızlandıran Hileler

Buraya kadar CPU'yu basit bir "talimat oku, yürüt, tekrar et" makinesi olarak anlattım. O resim yanlış değil, sadece eksik.

Sen bu cümleyi okurken önündeki işlemci muhtemelen yüzden fazla talimatı aynı anda havada tutuyor, bir sonraki `if`'in hangi tarafa sapacağını daha koşulu hesaplamadan tahmin ediyor ve o tahmine güvenip çoktan işe koyulmuş durumda. Bu bölümde o hilelerin her birini tek tek açacağız.

Bu bölümde işlemcinin "aynı anda daha fazla iş yapma" yollarını göreceğiz. Tahmin etmeye dayananları ise bir sonraki bölüme bırakıyorum, çünkü onların hikâyesi güvenlikle iç içe.

> **Bu bölümde neyi çözüyoruz?**
>
> - CPU'nun neden sadece saat hızı artırılarak hızlanamadığını göreceğiz.
> - Pipeline ve superscalar yürütmenin nasıl çalıştığını anlayacağız.
> - Talimatların neden sırasız yürütülüp sıralı emekli edildiğini çözeceğiz.
> - Çok çekirdek, SMT ve SIMD'in birbirinden farkını netleştireceğiz.

Önce Şunu Soralım: Bu Hilelere Neden İhtiyaç Var?

2004'te bir masaüstü işlemcisi 3 GHz'de çalışıyordu. Bugün, yirmi yıl sonra, hâlâ 3-5 GHz civarındayız. Oysa aynı sürede transistör sayısı yüzlerce kat arttı. Neden saat hızı artmadı?

Cevap, adını Robert Dennard'dan alan bir kuralın çökmesi. **Dennard ölçekleme**, transistörler küçüldükçe güç yoğunluğunun sabit kaldığını söylüyordu: daha küçük transistör daha az voltaj ister, daha az voltaj daha az ısı demektir, yani hızı artırmak bedavaya yakındır. Yaklaşık 2005'e kadar sektör bu bedavaya yakın hızlanmanın üzerinde yaşadı — yazılımcılar hiçbir şey yapmadan, sadece bekleyerek programlarının hızlandığını gördü.

Sonra fizik araya girdi. Transistörler o kadar küçüldü ki **sızıntı akımı** baskın hâle geldi; kapalı transistör bile akım harcamaya başladı. Voltajı daha fazla düşürmek mümkün olmayınca, saat hızını artırmanın bedeli doğrudan ısıya dönüştü. Duvara çarpıldı.

Sektörün cevabı yön değiştirmek oldu ve iki koldan ilerledi:

1. **Daha çok çekirdek.** Tek çekirdeği hızlandıramıyorsan yan yana koy.
2. **Tek çekirdekten daha çok iş çıkar.** Aynı saat hızında saat çevrimi başına daha fazla talimat bitir.

Bu bölümün ağırlığı ikinci kolda: pipeline, superscalar, out-of-order ve branch prediction, hepsi tek bir sorunun cevabı — *saat hızı artmıyorsa aynı saatte nasıl daha fazla iş yaparım?* Birinci kola, yani çekirdek çoğaltmaya, bölümün sonunda döneceğiz.

DERİNLEŞME **Amdahl mı Gustafson mu: daha çok çekirdek ne kadar işe yarar?**

Birinci kol, yani “daha çok çekirdek”, ilk bakışta sınırsız görünür. Değildir ve sınırı 1967’de Gene Amdahl koydu.

Amdahl yasası şunu söyler: bir programın ancak paralelleşebilen kısmı hızlanır; seri kalan kısım hızlanmaz ve toplam hızlanmayı o belirler. Formülü basit — programın %5’i seri ise, sonsuz çekirdekle bile en fazla **20 kat** hızlanabilirsin. %1 seri ise tavan 100 kattır. Çekirdek eklemek bu tavanı yükseltmez, yalnızca ona yaklaştırır.

Sayılarla: %95 paralel bir programda 16 çekirdek yaklaşık 9,1 kat hızlanma verir. 64 çekirdek 15,4 kat. Yani çekirdeği dört katına çıkarmak hızlanmayı %70 artırır. 1024 çekirdek ise 19,6 katta durur; artık çekirdek eklemek neredeyse hiçbir şey getirmez.

Bu, 1980’lerde paralel hesaplamaya karamsar bakılmasının sebebiydi. Sonra 1988’de John Gustafson karşı bir argüman getirdi: **Amdahl yanlış soruyu soruyor.**

Gustafson’ın gözlemi şu: pratikte insanlar aynı problemi daha hızlı çözmek için makine büyütmez; **daha büyük problem çözmek için** büyütür. Hava tahmini yapan biri, aynı çözünürlüklü tahmini daha hızlı almak istemez; aynı sürede daha yüksek çözünürlüklü tahmin ister. Problem büyüdükçe seri kısım (kurulum, toplama, çıktı) sabit kalırken paralel kısım büyür, yani seri oran düşer.

İkisi çelişmez; farklı soruları cevaplar. **Amdahl:** problem sabitken çekirdek eklemenin getirisi nedir? **Gustafson:** çekirdek sayısı arttıkça ne kadar büyük problem çözebilirim?

Pratik sonuç ikisinin ortasındadır ve bugün hâlâ geçerlidir: sabit boyutlu bir işi paralelleştirerek çok az kazanırsın, o yüzden ölçeklenme hikâyesinin gerçekten ölçeklenen kısmı neredeyse her zaman *veri* tarafındadır. Sunucu dünyasında çekirdeklerin işe yaramasının sebebi tek bir isteği hızlandırmaları değil, aynı anda binlerce bağımsız istek işlemeleridir.

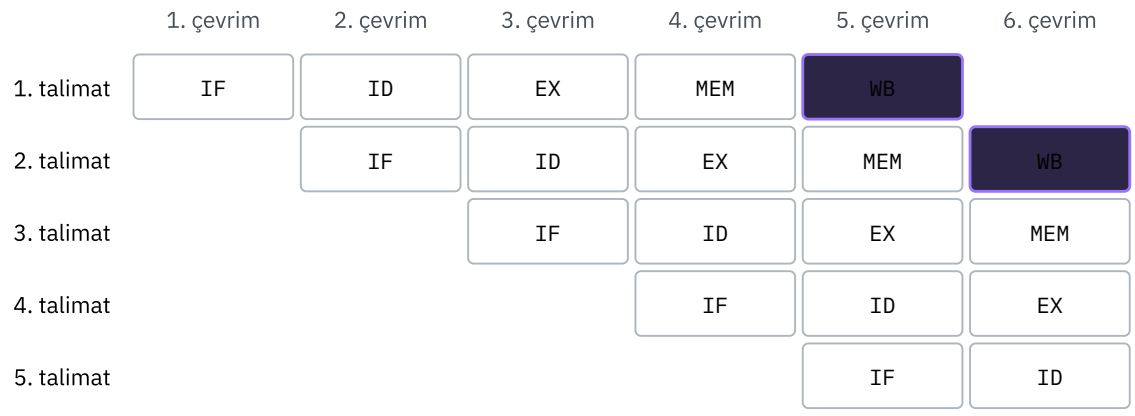
Pipeline: Fabrika Bandı Gibi Çalışmak

İlk CPU'lar bir talimatı tamamen bitirmeden diğerine geçmezdi. Fetch, decode, execute, write-back adımları tek tek yapılırdı. Modern CPU'lar ise **pipeline** (boru hattı) kullanır. Fabrika bandı gibi, her aşamada farklı bir talimat bulunur.

Aşama	Ne Yapar?
Fetch	Bellekten bir sonraki talimatı getir
Decode	Talimatı çözümle (ne yapacağını anla)
Execute	İşlemi gerçekleştir (toplama, çarpma vb.)
Memory	Gerekirse belleğe oku/yaz
Write-back	Sonucu register'a geri yaz

Bunu bir tablo olarak değil, zaman içinde görmek çok daha ikna edici. Aşağıdaki şemayı adım adım ilerlet: her adım bir saat çevrimi, her satır bir talimat.

BEŞ AŞAMALI PİPELİNE ÇEVRİM ÇEVRİM DOLUYOR



IF getir · ID çöz · EX yürüt · MEM belleğe eriş · WB sonucu yaz

1. **1. çevrim.** Yalnızca ilk talimat işlenmeye başladı: bellekten getiriliyor (IF). Diğer dört aşama boş duruyor.

2. **2. çevrim.** Birinci talimat çözülürken (ID) ikinci talimat getiriliyor. Getirme birimi boş kalmadı — pipeline'ın bütün fikri bu.
3. **3. çevrim.** Üç aşama aynı anda çalışıyor. Hiçbir talimat daha bitmedi ama üçü birden yolda.
4. **4. çevrim.** Dört aşama dolu. Buraya kadar tek bir talimat bile tamamlanmadı; pipeline hâlâ doluyor.
5. **5. çevrim: ilk sonuç.** Birinci talimat WB aşamasında bitiyor ve beş aşama da dolu. Pipeline tam kapasiteye ulaştı.
6. **6. çevrim: kararlı hâl.** Artık her çevrimde bir talimat bitiyor. Tek bir talimat hâlâ beş çevrim sürüyor — değişen tek şey, aynı anda beş talimatın havada olması.

Beş çevrim boyunca hiçbir talimat bitmiyor, sonra birden her çevrimde bir tane bitmeye başlıyor. Pipeline’ın vaadi tam olarak bu: tek bir talimatı hızlandırmıyor, aynı anda kaç talimatın havada olduğunu artırıyor.

Aynı deseni daha uzun bir örnekte, dokuz talimat ve on dört çevrim üzerinde de görebilirsin:

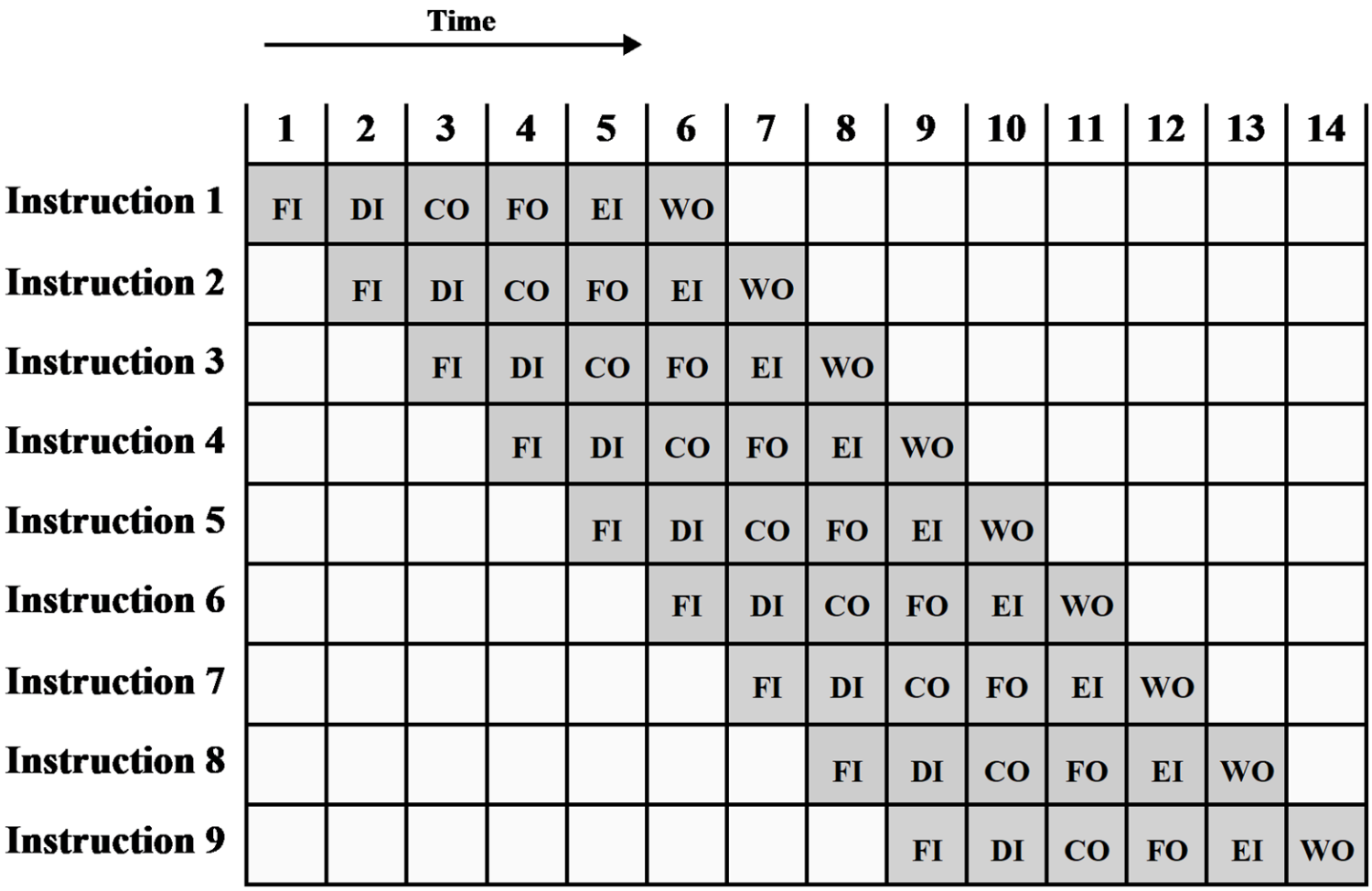


Figure 14.10 Timing Diagram for Instruction Pipeline Operation

Kaynak: William Stallings, Computer Organization and Architecture, 10. baskı, Şekil 14.10.

Şemanın iki şeyi birden anlattığına dikkat et. **Merdiven deseni** pipeline'ın dolmasıdır: ilk talimat altı çevrim sürer, ama altıncı çevrimde artık altı talimat aynı anda ilerlemektedir. Bu noktadan sonra **her çevrimde bir talimat tamamlanır** — tek tek altı çevrim süren işler, ardışık geldiklerinde çevrim başına bire iner.

Bir de sol üstteki boşluğa bak: pipeline dolana kadar geçen o birkaç çevrim bedavaya gelmez. Kısa süren bir kod parçasında pipeline dolmadan iş biter ve vaat edilen hızlanma hiç gerçekleşmez.

> Buradaki aşama isimleri (FI, DI, CO, FO, EI, WO) Stallings'in altı aşamalı modelinden geliyor ve yukarıdaki beş aşamalı tablodan biraz farklı bölünmüş. Aşama sayısı mimariye göre değişen bir tasarım tercihidir: klasik RISC beşte kalır, Pentium 4 yirmiyi aşmıştı. Derin pipeline saat hızını yükseltmeyi kolaylaştırır ama birazdan göreceğimiz yanlış tahmin cezasını da o oranda büyütür.

Pipeline Hazardları

Pipeline'ın verimli çalışması için talimatlar sürekli akmalıdır. Ama bant her zaman akmaz. Bazen bir talimat sırası geldiği hâlde kendi aşamasına giremez ve arkasındaki herkes onu bekler. Bu duruma **hazard** deniyor. Üç ayrı sebepten çıkar ve üçünü de tanıman gerekiyor, çünkü bu bölümün geri kalanındaki bütün hilelerin varlık sebebi bunlar.

- **Structural hazard (yapısal):** İki talimat aynı anda aynı donanım birimini kullanmaya çalışır. Örneğin tek bir bellek portu hem fetch hem memory aşamasında kullanılmak istenirse biri beklemek zorundadır.
- **Veri Hazardı (Data):** Bir talimatın çalışması için önceki talimatın sonucuna ihtiyaç duyulur ama o sonuç henüz yazılmamıştır.
- **Kontrol Hazardı (Control):** Koşullu dallanma (`if`, `jmp`) sonrası hangi talimatın geleceği bilinmez; pipeline'ın doğru yoldan devam etmesi için beklenir veya tahmin edilir.

Veri hazardlarından en yaygın olanı **Read-After-Write (RAW)** bağımlılığıdır:

RAW bağımlılığı örneği

```
1  add eax, ebx
2  mov ecx, eax    ; önceki talimatın sonucunu bekliyor — stall'e sebep olur
```

`mov ecx, eax` talimatı `eax`'in güncellenmiş değerine ihtiyaç duyar. Eğer `add` talimatının write-back aşaması henüz tamamlanmadıysa, CPU bir veya daha fazla saat çevrimi **stall** (duraklama) ekler veya **bypassing/forwarding** ile sonucu doğrudan execute birimine yönlendirir.

Superscalar: Birden Fazla Fabrika Bandı

Pipeline tek bir fabrika bandıdır. **Superscalar** işlemciler ise aynı anda birden fazla talimatı farklı birimlerde yürütebilir. Örneğin bir toplama birimi ve bir çarpma birimi aynı anda çalışabilir.

- **IPC (Instructions Per Cycle):** Bir saat çevriminde kaç talimat yürütülür? Basit bir pipeline'da bu sayı en iyi ihtimalle 1'e yaklaşır; superscalar bir çekirdekte 1'in üstüne çıkabilir.
- **Out-of-Order Execution:** İşlemci, talimatların bağımsız olanlarını önce yürütüp bağımlı olanları bekletebilir. Bu, pipeline'ın boş kalmamasını sağlar.

Sıra Neden Bozulur?

Out-of-order execution modern işlemcinin en büyük hilesi ve — göreceğimiz gibi — Spectre'ın da altyapısı. O yüzden üç basamakta açalım.

Birinci basamak: beklemenin bulaşıcılığı. Bellekten veri bekleyen bir talimat düşün; cache miss olmuş, veri yüzlerce çevrim sonra gelecek. Sıralı bir işlemcide bu talimatın *arkasındaki* talimatlar da bekler. Oysa onların çoğunun bekleyen veriyle hiçbir alakası yoktur. Boş oturmalarının tek sebebi, kodda daha sonra yazılmış olmaları.

İkinci basamak: sahte bağımlılıklar. Sıra bozacaksan gerçek bağımlılıkları bilmen gerekir, ama register isimleri seni yanıltır:

```
mov eax, [mem1]   ; birinci is
add eax, 5
mov eax, [mem2]   ; ikinci is -- birinciyle hicbir ilgisi yok
add eax, 9
```

Üçüncü satır ikinci satırı beklemek zorunda mı? Mantıken hayır; bunlar bambaşka iki hesap. Ama ikisi de `eax` adını kullanıyor. Bu bir veri bağımlılığı değil, **isim** bağımlılığıdır. Donanım bunu **register renaming** ile çözer: `eax` aslında bir isimdir ve çekirdeğin içinde onlarca fiziksel register vardır. İşlemci ikinci `eax`'e farklı bir fiziksel register verir, iki hesap birbirinden tamamen kopar ve aynı anda yürüyebilir.

Üçüncü basamak: sıra geri nasıl kuruluyor? Talimatlar sırasız *yürütülür* ama sıralı *emekli edilir* (retirement). Sonuçlar önce **reorder buffer** denen bir kuyrukta bekler; bir talimatın sonucu ancak kendinden önceki bütün talimatlar tamamlandığında dünyaya görünür olur. Programın gördüğü sonuç bu yüzden her zaman sıralıdır.

Bu üçüncü basamağı aklında tut: birazdan "mimari sonuçları geçici tutmak" dediğimde kastettiğim mekanizma tam olarak budur.

Çok Çekirdekli İşlemciler

Pipeline ve superscalar tek bir çekirdek (core) içinde paralellik sağlar. Ama fiziksel sınırlar — ısı ve güç tüketimi — tek bir çekirdeği sınırsız hızlandırmana izin vermez. Bu noktada sektörün bulduğu çözüm yön değiştirmek oldu: tek çekirdeği daha da hızlandırmak yerine, aynı çipin üzerine birden fazla bağımsız çekirdek koymak.

Modern bir masaüstü işlemcisinde sekiz, sunucu tarafında yüzü aşkın çekirdek görmen sıra dışı değil. Bu çekirdeklerin her biri kendi register setine, kendi L1 ve L2 cache'ine sahiptir; yani birbirlerinin defterine bakmadan, kendi programlarını kendi hızlarında yürütürler. Ortak kaldıkları yer L3'tür: **[5. bölümde]** tanıştığın o büyük ve görece yavaş katman, artık aynı zamanda çekirdeklerin buluşma noktası.

Buradan doğal olarak bir soru çıkıyor ve cevabı bu kitabın en sinsi performans tuzaklarından birine bağlanıyor: **iki çekirdek aynı veriye dokunursa ne olur?** İkisi de o cache line'ı kendi L1'ine kopyalar; biri değeri değiştirdiğinde diğerinin elinde bayat bir kopya kalır. Donanımın bunu engellemek için yürüttüğü sessiz pazarlığa **cache coherence** denir ve **[5. bölümde]** MESI protokolüyle birlikte ayrıntısını konuşmuştuk. Oradaki *false sharing* bölümünü atladıysan, paralel kod yazmadan önce dönüp okumanı öneririm: paralelleştirdiğin bir programın tek çekirdekliden yavaş çıkmasının en yaygın sebebi orada anlatılıyor.

> Programlama dillerinde `std::thread` (C++), `Thread` (Java), `threading` (Python) gibi yapılar bu çok çekirdekli yapıyı kullanmak içindir. Ama birden fazla çekirdek kullanmak, programın paralel çalışacak şekilde yazılmasını gerektirir.

Simultaneous Multithreading (SMT / Hyper-Threading)

Intel'in **Hyper-Threading** teknolojisi (genel adıyla **SMT**), tek bir fiziksel çekirdeğin içinde iki (veya daha fazla) mantıksal yürütme akışı çalıştırmasına olanak tanır.

Fikri anlamak için çekirdeği ikiye ayırmak gerekiyor. Hesabı yapan birimler — ALU, FPU, cache — tek takımdır ve paylaşılır. Buna karşılık *durum* bilgisi, yani register seti ve instruction pointer, çekirdeğin içinde ikişer kopya hâlinde tutulur; iki akış birbirinin defterine bakmaz.

Kazanç şuradan geliyor: bir akış bellekten veri beklerken (cache miss) o pahalı hesap birimleri boş oturacaktı. SMT, o boşluğu ikinci akışın işiyle doldurur. Yani SMT sana ikinci bir çekirdek vermez, birinci çekirdeğin boş anlarını satar.

> SMT, tek thread'in performansını ikiye katlamaz ama genel verimliliği %10-30 artırabilir. Güvenlik açısından bazı bulut sağlayıcıları SMT'yi devre dışı bırakır (yan yana thread'ler arası bilgi sızıntısı riski).

SIMD: Tek Talimat, Çok Veri

SIMD (Single Instruction, Multiple Data), tek bir talimatın aynı anda birden fazla veri parçası üzerinde çalışmasını sağlar. Örneğin iki dizinin karşılıklı elemanlarını dörder dörder tek talimatta toplarsın: `a[0]+b[0]`, `a[1]+b[1]`, `a[2]+b[2]`, `a[3]+b[3]` — dört ayrı toplama değil, bir tane toplama talimatı.

Bunu mümkün kılan şey, normal register'lardan çok daha geniş özel register'lar. Genişliği anlamlı kılan da içine kaç sayı sığdığı:

Kuşak	Register genişliği	32-bit float	64-bit double	8-bit tam sayı
SSE	128 bit	4	2	16
AVX / AVX2	256 bit	8	4	32
AVX-512	512 bit	16	8	64

Register'ın bu dilimlerinin her birine **lane** (şerit) denir. Sekiz lane'li bir toplama talimatı, sekiz ayrı toplamanın işini tek çevrimde bitirir.

SIMD'in sınırını da hemen söyleyeyim, çünkü çoğu hayal kırıklığı buradan çıkar: **bütün lane'ler aynı talimatı yürütmek zorundadır.** Lane'ler arasında `if` yazamazsın. "Şu elemanlar için topla, şunlar için çıkar" demek istiyorsan, ikisini de hesaplayıp sonucu bir maske ile seçmen gerekir — yani işin bir kısmını baştan çöpe atarsın. Verinin düzenli olmadığı yerde SIMD'in kazancı hızla erir.

> Video işleme, yapay zeka, bilimsel hesaplama gibi alanlarda SIMD kritik öneme sahiptir. Derleyiciler kodu kendiliğinden vektörize edebilir (auto-vectorization), ama işi sonuna kadar sıkmak istiyorsan intrinsic fonksiyonlarla vektör talimatlarını elle yazarsın.

DERİNLEŞME **Daha geniş SIMD neden her zaman daha hızlı değil?**

AVX-512 daha geniş register'lar ve daha fazla paralellik vaat eder, ama bu verimlilik bedelsiz değildir. 512-bit işlemler çekirdek içindeki vektör birimlerini çok daha yoğun kullanır ve bu doğrudan **ısı artışına** yol açar.

Bunun iki ayrı sonucu var. Birincisi klasik **termal throttling**: çip ısınır, saat hızı düşer. İkincisi daha sinsi — bazı Intel nesilleri AVX-512 talimatını görür görmez, daha ısınmamışken bile, o çekirdeği (bazen tüm çekirdekleri) önceden tanımlı daha düşük bir frekans kademesine indirir. Yani cezayı ısındığın için değil, ısınma ihtimalin olduğu için ödersin. Bu yüzden bulut tarafında AVX-512'nin varlığı instance tipine göre değişir; aynı sağlayıcının iki farklı nesil makinesinde aynı kod farklı hızlarda koşabilir. "Daha fazla SIMD" her zaman "daha hızlı" anlamına gelmez; iş yüküne, soğutmaya ve güç bütçesine bağlıdır.

Özet

> **Peki, ne öğrendik?**
>
> - Saat hızı tek başına artık artmıyor; kazanç **aynı anda daha fazla iş yapmaktan** geliyor.
> - **Pipeline**, talimatları aşamalara böler ve her aşamada farklı bir talimat ilerletir.
> - **Superscalar** yürütme aynı çevrimde birden fazla talimat başlatır; **out-of-order** yürütme ise bekleyen bir talimatın arkasındaki bağımsız işi öne alır.
> - Talimatlar sırasız yürütülür ama **sıralı emekli edilir**. Programın gördüğü sonuç bu yüzden her zaman sıralıdır.
> - **Çok çekirdek** ayrı işlemciler, **SMT** tek çekirdeğin boş anlarının satılması, **SIMD** ise tek talimatla çok veri demektir. Üçü birbirinden bağımsız üç paralellik türüdür.

Bu hilelerin hepsi sessiz bir varsayıma dayanıyor: CPU, sıradaki talimatın hangisi olduğunu biliyor. Peki kodda bir `if` varsa? Sıradaki bölümün konusu, işlemcinin bu soruya verdiği cevap — ve o cevabın kırk yıl sonra açtığı devasa güvenlik deliği.

Bölüm 9: Tahmin, Spekülasyon ve Spectre

Bir önceki bölümde işlemcinin aynı anda onlarca talimatı havada tuttuğunu gördük. Ama o resmin ortasında bilerek bir boşluk bıraktım.

Pipeline'ın dolu kalması için CPU'nun sıradaki talimatı önceden getirmesi gerekir. Peki kodda bir `if` varsa? Koşulun sonucu daha hesaplanmamışken hangi tarafı getireceksin?

İşlemcinin bu soruya verdiği cevap "beklerim" değil. Cevap şu: **tahmin et ve işe koyul.** Bu bölümde önce o tahmin makinesini, sonra da onun modern bilgisayarların en ünlü güvenlik açığına nasıl dönüştüğünü göreceğiz.

> **Bu bölümde neyi çözüyoruz?**
>
> - CPU'nun bir dallanmanın sonucunu nasıl tahmin ettiğini göreceğiz.
> - Yanlış tahminin bedelini ve spekülatif yürütmenin ne olduğunu anlayacağız.
> - Spectre ile Meltdown'ın neden bir yazılım hatası değil, bir tasarım sonucu olduğunu çözeceğiz.
> - Bir soyutlamanın neden en çok hızlandığı yerden sızdığını göreceğiz.

Branch Prediction: Geleceği Tahmin Etmek

Programlar sık sık koşullu dallanmalar kullanır (`if`, `while`, `for`). Pipeline'ın verimli çalışması için CPU bir sonraki talimatı önceden getirmelidir. Ama dalın nereye gideceğini bilmeden ne getireceksin?

Branch prediction (dal tahmini) devreye girer. CPU, geçmişteki dallanma davranışlarına bakarak "bu dal muhtemelen atlayacak" veya "atlamayacak" diye tahminde bulunur.

- **Static prediction:** Basit kurallar (örneğin "geriye dönen dallar atlar, ileriye dönenler atlamaz").
- **Dynamic prediction:** Donanım iki ayrı soruya iki ayrı tabloyla cevap verir. *Atlayacak mı?* sorusunu, geçmiş davranışı tutan bir yön tablosu cevaplar — içinde birazdan göreceğimiz

2-bit sayaçlar durur. *Atlayacaksa nereye?* sorusunu ise **Branch Target Buffer** (BTB) cevaplar; orada dalın son gittiği hedef adres saklanır.

> Modern CPU'ların branch predictor'ları %95+ isabet oranına ulaşabilir. Yanlış tahmin edilirse pipeline boşaltılır (flush) ve doğru yoldan devam edilir; bu ceza onlarca saat çevrimi sürebilir.

Bu cezanın neye benzediğini de aynı şemayla görebiliriz. Bu kez 3. talimat koşullu bir dallanma:

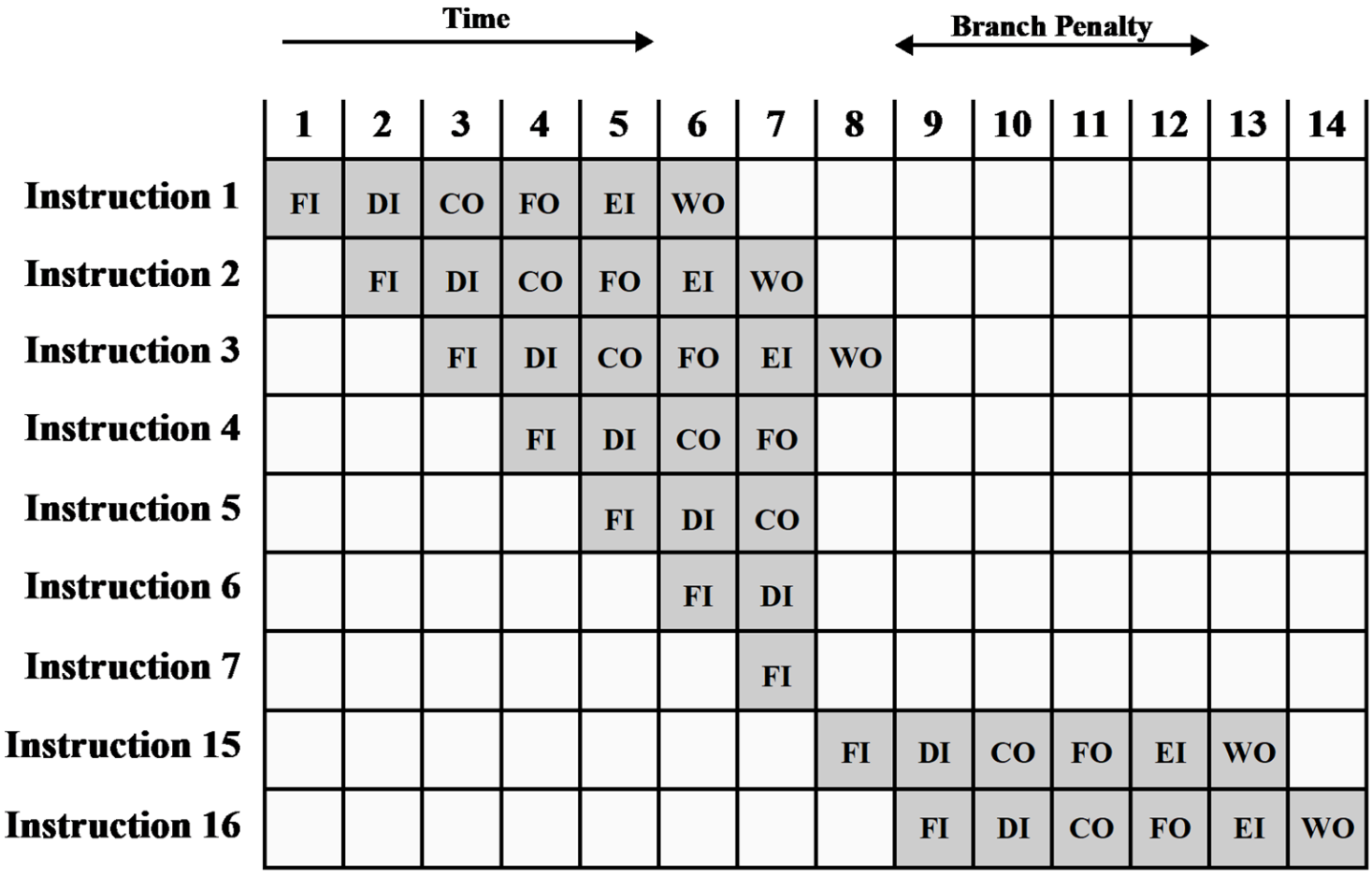


Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

Kaynak: William Stallings, Computer Organization and Architecture, 10. baskı, Şekil 14.11.

4’ten 7’ye kadar olan talimatların yarıda kesildiğine dikkat et — hiçbiri tamamlanmıyor. İşlemci onları çoktan pipeline’a almıştı, çünkü dallanmanın nereye gideceğini 7. çevrime kadar bilemedi. Öğrendiği anda o dört talimatın tamamı çöpe gidiyor ve pipeline 15. talimattan sıfırdan doluyor.

Şemadaki **Branch Penalty** aralığı işte bu israfın adı. Pipeline ne kadar derinse, yanlış tahmin anında çöpe giden iş de o kadar artar. Branch prediction’ın neden bu kadar önemsendiği tek cümleyle buradadır: doğru tahmin bedavadır, yanlış tahmin bu boşluğu ödetir.

DERİNLEŞME **Tahmin nasıl hatırlıyor? 2-bit saturating counter**

Dinamik branch prediction'da en yaygın mekanizmalardan biri **2-bit saturating counter**'dır. Bu sayaç, bir dalın geçmişteki davranışına göre dört durumdan birinde bulunur:

- **Strongly Not Taken** — Kesin atlamaz
- **Weakly Not Taken** — Büyük ihtimalle atlamaz
- **Weakly Taken** — Büyük ihtimalle atlar
- **Strongly Taken** — Kesin atlar

Dal atlarsa sayaç bir artar, atlamazsa bir azalır. Sınır değerlerde "sıkışır" (saturates); bu yüzden tek bir yanlış tahmin hemen yön değiştirmez. Bu, kısa döngülerdeki dal davranışındaki gürültüye karşı dayanıklılık sağlar.

Dört durumun birbirine nasıl bağlandığını tabloda görmek işi tamamen netleştiriyor:

Durum	Tahmini	Dal atlarsa	Dal atlamazsa
Strongly Not Taken	atlamaz	→ Weakly Not Taken	Strongly Not Taken
Weakly Not Taken	atlamaz	→ Weakly Taken	→ Strongly Not Taken
Weakly Taken	atlar	→ Strongly Taken	→ Weakly Not Taken
Strongly Taken	atlar	Strongly Taken	→ Weakly Taken

Uçlardaki iki durumun kendine dönmesi kasıtlıdır: sayaç orada **doyuma ulaşır**. Bu sayede yüz kez atlamış bir dal, tek bir istisnada tahminini değiştirmez — yalnızca bir kademe zayıflar. Döngülerin son turu her seferinde yanlış tahmin edilir ama döngünün geri kalanı bundan etkilenmez.

Spekülatif Yürütme ve Güvenlik

Devam etmeden önce iki farklı "durum" arasındaki ayrımı yapmam gerekiyor, çünkü bölümün geri kalanı bunun üzerine kuruluyor.

Mimari durum, programın görmesine izin verilen şeydir: register'ların değerleri, belleğin içeriği. **Mikro-mimari durum** ise programın göremediği, işlemcinin kendi iç defteridir: hangi satır cache'e girdi, tahmin edici ne öğrendi, hangi tampon doldu.

Şimdi bölümün en önemli cümlesi: **spekülatif yürütme mimari durumu geri alabilir, mikro-mimari durumu geri alamaz.** Spectre'ın tamamı burada saklı.

Tahmin edilen dalın talimatlarını **spekülatif olarak yürütme** (speculative execution), tahmin doğru çıkana kadar mimari sonuçları geçici tutmak demektir. Tahmin doğruysa sonuçlar görünür hâle gelir, yanlışsa mimari durum geri alınır. Sorun şu: geri alınan spekülatif yürütme bile cache gibi mikro-mimari yapılarda ölçülebilir iz bırakabilir.

2018'de duyurulan **Spectre** ve **Meltdown** ailesi açıklar bu farkı görünür yaptı:

- **Spectre:** Branch prediction ve spekülatif yürütme, normalde okunmaması gereken veriye bağlı cache izleri oluşturabilir. Saldırgan bu izleri zamanlama ölçümleriyle okuyarak gizli veriyi dolaylı çıkarabilir.
- **Meltdown:** Bazı Intel işlemcilerinde sıra bozuktu. CPU, kernel'a ait bir bellek adresini önce *okuyor*, izin kontrolünü *sonra* yapıyordu. Kontrol yapılınca sonuç iptal ediliyordu — ama okunan değer o arada cache'i çoktan değiştirmişti. Sıradan bir programın, hiçbir zaman göremeyeceği kernel verisini cache'in davranışından geri çıkarabilmesi buradan doğdu.

> Bu açıkların ardından işlemci üreticileri ve işletim sistemleri birkaç ayrı savunma geliştirdi: **mikrokod güncellemeleri** işlemcinin davranışını sahada değiştirdi, **KPTI** kernel'ın bellek haritasını kullanıcı programlarının görüş alanından çıkardı, **site isolation** tarayıcıda her siteyi ayrı bir process'e taşıdı, **retpoline** ise dolaylı dalları tahmin edilemez hâle getiren bir derleyici hilesi olarak devreye girdi. Bu savunmalardan hangilerinin senin makinende açık olduğunu tahmin etmene gerek yok; bölümün sonunda tek komutla nasıl kontrol edeceğini göstereceğim.

DERİNLEŞME **Meltdown yamalandı, Spectre neden yamalanamıyor?**

İki açık aynı anda duyuruldu ve aynı mekanizmadan yararlanıyor gibi göründü. Ama aralarında temel bir fark var ve bu fark, birinin çözülüp diğerinin çözülememesini açıklıyor.

Meltdown bir hataydı. Bazı Intel tasarımlarında izin kontrolü, spekülatif erişimden *sonra* yapılıyordu. Yani CPU önce kernel belleğini okuyor, sonra "aslında buna hakkın yokmuş" diyerek sonucu iptal ediyordu — ama okunan değer o esnada cache'i etkilemiş oluyordu. Bu, mimarinin gerektirdiği bir davranış değildi; bir sıralama hatasıydı. Nitekim aynı dönemin AMD ve birçok ARM tasarımı bu açıktan etkilenmedi; sonraki nesil Intel çipleri de izin kontrolünü öne alarak sorunu doğrudan donanımda kapattı. Yazılım tarafındaki yama ise **KPTI** oldu (Kernel Page Table Isolation): kernel'ın bellek haritası, kullanıcı programı çalışırken haritadan tamamen çıkarılır. Pahalı olmasının sebebi de bu — artık her syscall'da sayfa tablosunun baştan kurulması gerekiyordu ve syscall yoğun iş yüklerinde bunun bedeli %5 ile %30 arasında değişebiliyordu.

Spectre bir hata değil. İşlemci tam olarak tasarlandığı gibi davranıyor: dalı tahmin ediyor, tahmin ettiği yolu spekülatif olarak yürütüyor, yanılırsa mimari durumu geri alıyor. Sorun şu ki mimari durumu geri almak, **mikro-mimari izleri** geri almaz. Cache'e giren satır cache'te kalır ve zamanlama ölçümüyle görülebilir.

Bunu "düzeltmek" için spekülasyonun cache üzerinde hiçbir gözlenebilir iz bırakmaması gerekir. Ama spekülasyonun bütün amacı veriyi önceden getirmektir; izi silmek, faydayı da silmek demektir. Yani Spectre'ın kökü bir kusur değil, **performans ile izolasyon arasındaki temel gerilimdir**.

Bu yüzden savunmalar açığı kapatmaz, sömürülmesini zorlaştırır: retpoline, hedefi ancak çalışma anında belli olan **dolaylı** dallara (fonksiyon pointer'ları, sanal metot çağrıları) uygulanır: tahmin ediciyi kasten yanıltıp o dalı tahmin edilemez hâle getirir. `lfence` talimatı ise konduğu noktada işlemciye "buradan öteye spekülatif geçme" der, tarayıcılar zamanlayıcıların çözünürlüğünü kasten düşürür, bulut sağlayıcıları güvenilmeyen iş yüklerini ayrı çekirdeklere alır. Hepsinin bir performans bedeli vardır ve hiçbiri kesin değildir; nitekim aradan geçen yıllarda aynı aileden yeni varyantlar (Spectre-BHB, Retbleed, Downfall, Inception) düzenli aralıklarla yayımlandı.

Buradan çıkan ders bu kitabın en önemli fikirlerinden biri: **bir soyutlamanın sızdırdığı yer, çoğu zaman onu hızlandıran yerdir.** Cache, program için görünmez olmalıydı; zamanlama onu görünür kılıyor. Sanal bellek izolasyon vaat ediyordu; paylaşılan donanım o vaadi delik bırakıyor. Performans için kurulan her paylaşım, aynı zamanda bir yan kanaldır.

Bu bölümün en önemli beş adımını tek bir şemada topladım. Özellikle son ikisine dikkat et: geri alınan şey ile geri alınmayan şey.

SPEKÜLATİF YÜRÜTME VE GERİ ALINMAYAN İZ

1. **Koşula gelinir.** Kodda bir `if` var ve koşulun sonucu henüz hesaplanmadı. Değer bellekten geliyorsa bu bekleyiş yüzlerce çevrim sürebilir.
2. **Tahminci karar verir.** CPU beklemez. Aynı dalın geçmişte ne yaptığına bakar ve bir yön seçer. Modern işlemcilerde bu tahmin %95'in üzerinde tutar.
3. **Tahmin edilen yol yürütülür.** İşlemci tahmin ettiği daldaki talimatları gerçekten çalıştırır. Yol boyunca bellekten okunan her veri, yan etki olarak cache'e girer.
4. **Koşul çıkar: tahmin yanlış.** Mimari durum geri alınır. Reorder buffer'daki sonuçlar atılır, pipeline boşaltılır, doğru yoldan yeniden başlanır. Programın gördüğü sonuç kusursuzdur — sanki hiç olmamış gibi.
5. **Ama cache'teki iz kalır.** Geri alınmayan tek şey cache'in içeriği. Hangi satırın oraya girdiğini, erişim sürelerini ölçerek anlamak mümkündür. Spectre ve Meltdown'ın tamamı bu tek cümleden doğar.

Bunların İzi Kendi Makinende Duruyor

Bu bölümde anlattıklarım soyut kalmasın: hepsinin izi şu anda önünde duran makinede var.

Linux'ta `lscpu` komutu işlemcinin modelini, çekirdek sayısını ve çekirdek başına kaç thread çalıştığını (yani SMT açık mı) tek bakışta gösterir. Aynı komutun uzun çıktısında işlemcinin desteklediği özelliklerin listesi de yer alır ve o liste güzel bir tarih dersidir: `avx2` SIMD kuşağını, `ht` hyper-threading'i, `nx` **[15. bölümde]** göreceğimiz çalıştırılamaz sayfa bitini gösterir. `pti` gördüğünde ise doğrudan bu bölümün konusuna bakıyorsun demektir — Meltdown'a karşı alınan sayfa tablosu izolasyonu açık.

Yani o listede yalnızca yetenekler değil, geçmişte yaşanmış açıkların izleri de duruyor.

Bir adım daha ileri gitmek istersen ölçülecek tek bir sayı var: **IPC** (Instructions Per Cycle), yani saat çevrimi başına biten talimat sayısı. IPC birin üzerindeyse işlemci gerçekten superscalar çalışıyor, aynı çevrimde birden fazla talimat bitiriyor demektir. Birin altındaysa pipeline boş bekliyordur: ya cache'ten veri gelmiyordur, ya da bir dal tahmini tutmamıştır. Bu bölümü okuduktan sonra o tek sayıyı gördüğünde arkasında ne olduğunu biliyor olacaksın.

Özet

> **Peki, ne öğrendik?**
>
> - **Branch prediction**, dallanmanın yönünü geçmiş davranışa bakarak tahmin eder ve modern işlemcilerde %95'in üzerinde isabet tutturur.
> - Yanlış tahminin bedeli pipeline'ın boşaltılmasıdır; onlarca çevrim çöpe gider.
> - **Spekülatif yürütme**, tahmin edilen yolu koşul daha hesaplanmadan yürütmektir. Tahmin yanlışsa mimari sonuçlar geri alınır — ama cache'te bıraktığı iz geri alınmaz.
> - **Spectre ve Meltdown** tam olarak o izin okunabilmesinden doğar. Bir yazılım hatası değil, performans için kurulmuş paylaşımın doğal sonucudurlar.
> - Bu bölümün tek cümlelik dersi: **bir soyutlamanın sızdığı yer, çoğu zaman onu hızlandıran yerdir.**

Artık CPU'ya bir daha "talimatları sırayla yürüten motor" diye bakamazsın. Baktığın şey, her çevrimde onlarca talimatı havada tutan, geleceği tahmin eden ve bu tahminlerin izini cache'te bırakan bir makine.

Donanım tarafını burada kapatıyoruz. Sıradaki bölümden itibaren yazılım tarafına geçiyor ve tek bir soruyu sonuna kadar takip ediyoruz: bir program çalıştırdığında tam olarak ne oluyor?

Bölüm 10: Bir Program Nasıl Çalıştırılır?

Şimdiye kadar CPU'ların executable dosyalardan yüklenen makine kodunu nasıl yürüttüğünü, ring tabanlı güvenliğin ne olduğunu ve syscall'ların nasıl çalıştığını gördük. Bu bölümde ise Linux kernel'ının içine biraz daha dalıp programların gerçekten nasıl yüklendiğini ve çalıştırıldığını anlamaya çalışacağız.

Başlamadan önce bu bölümün en ters gelen fikrini masaya koyalım, çünkü geri kalan her şey ona dayanıyor.

Bir program çalıştırmak, sezgisel olarak bir **doğum** gibi görünür: yeni bir şey ortaya çıkar. Linux'ta öyle olmaz. `execve` yeni bir process yaratmaz; **var olan process'in içini boşaltıp yerine yeni programı koyar.** Aynı PID, aynı açık dosyalar, aynı kullanıcı kimliği — ama bambaşka bir kod ve bambaşka bir bellek. Bir deri değiştirme, bir doğum değil.

Bu yüzden `execve` başarılı olduğunda **geri dönmez**. Dönecek bir yer kalmamıştır; onu çağıran kod artık bellekte yoktur. Bir fonksiyonun başarısını "geri dönmemesinden" anlamak tuhaf gelir, ama mantığı tam olarak budur.

Peki o zaman yeni process nereden geliyor? Ayrı bir syscall'dan: `fork`. İkisi birlikte çalışır ve **[16. bölümde]** o tarafı ele alacağız. Bu bölüm tek bir soruya odaklanıyor: kernel, elindeki dosyayı **çalışan bir programa** nasıl çeviriyor?

> **Kernel kaynak koduyla ilk kez karşılaşıyorsan:** Ana anlatı boyunca ihtiyacın olan her şey düz metinle anlatılıyor. Kernel kaynağına inen kısımlar katlanmış "Derinleşme" panellerine alındı; hiçbirini açmadan bölümün tamamını anlayabilirsin. Merak edersen oradalar.

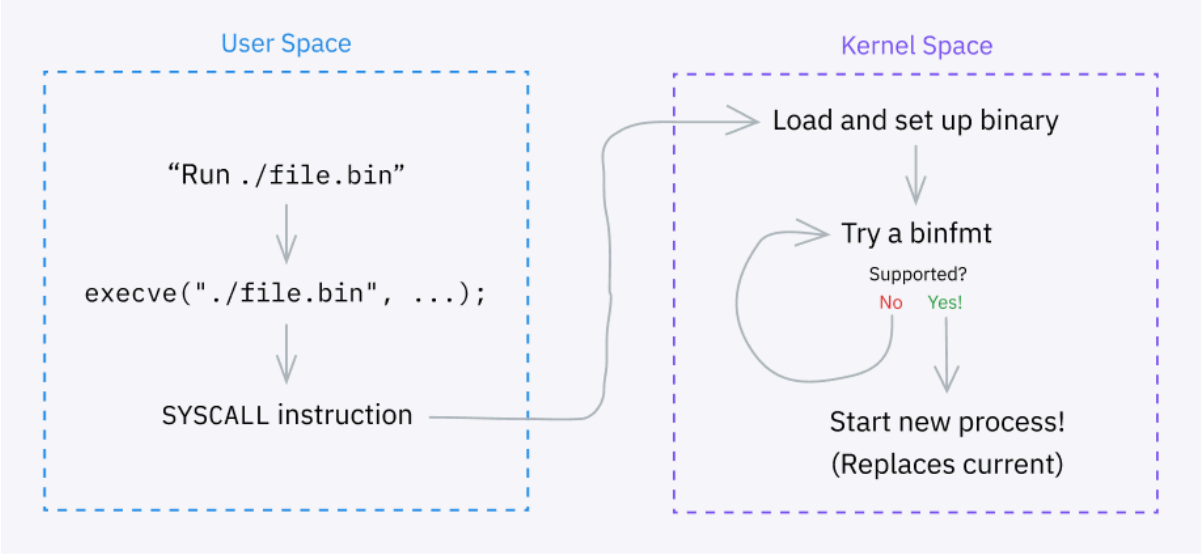
> **Bu bölümde neyi çözüyoruz?**
>
> - `execve` çağrısının mevcut process'i yeni bir programla nasıl değiştirdiğini göreceğiz.
> - Kernel'ın dosya formatını nasıl tanıdığını ve shebang satırını nasıl ele aldığını anlayacağız.
> - "Terminalden çalıştırdım" ile "CPU gerçekten yeni koda atladı" arasındaki köprüyü kuracağız.

Özellikle x86-64 üzerindeki Linux'a bakacağız. Neden?

- Linux, masaüstünden mobile ve sunucuya kadar geniş kullanım alanına sahip tam teşekküllü bir üretim işletim sistemidir. Üstelik açık kaynak olduğu için doğrudan kaynak koda bakarak ilerlemek çok kolaydır. Bu kitapta kernel koduna bol bol referans vereceğim.
- x86-64, modern masaüstü bilgisayarların büyük kısmında kullanılan mimaridir ve pek çok kodun hedefidir. Burada göreceğimiz x86-64'e özgü ayrıntıların önemli kısmı yine de daha genel fikirlere açılır.

Öğreneceğimiz şeylerin çoğu, ayrıntılar değişse de başka işletim sistemlerine ve mimarilere de kolayca genellenebilir.

Exec Syscall'larının Temel Davranışı



Çok önemli bir syscall ile başlayalım: `execve`. Bir programı yükler ve başarılı olursa mevcut process'i o programla değiştirir.

Bunların yanında `execl`, `execlp`, `execvp`, `execvpe` gibi isimler de görürsün. Bunlar syscall değil, libc'nin sağladığı sarmalayıcı fonksiyonlardır; argümanları farklı biçimlerde alırlar — `l` tek tek liste, `v` vektör, `p` `PATH` üzerinden arama, `e` ortam değişkenlerini elle verme — ama hepsi sonunda aynı yere, `execve` syscall'ına iner. Kernel'ın gerçekten bildiği iki tanedir: `execve` ve `execveat`.

> **Bir not: `execveat`**
>
> Linux'ta `execve` ve `execveat` kullanıcıya görünen ayrı syscall'lardır, ama kernel içinde büyük ölçüde ortak bir yardımcı yola bağlanırlar: `do_execveat_common`. `execveat` bazı ek seçeneklerle program yürütmeye izin verir. Biz sadelik adına çoğunlukla `execve` üzerinden konuşacağız; pratikte fark, kernel'ın ortak helper'a hangi dosya descriptor'ı ve flag'lerle girdiğidir.
>
> `ve` ne demek diye merak ediyorsan: `v`, argüman vektörü olan `argv`'yi; `e` ise environment vektörü olan `envp`'yi temsil eder. Diğer `exec` varyantları da farklı çağrı imzalarını son eklerle ayırt eder. `execveat` içindeki `at` ise, programın hangi konuma göre çalıştırılacağını belirtebildiği için oradadır.

`execve` çağrı imzası:

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

- `filename`, çalıştırılacak programın yolunu belirtir.
- `argv`, null-terminated bir argüman listesidir; yani son öğe null pointer'dır. C'deki `main` fonksiyonunda gördüğün `argc` değeri aslında daha sonra bu listeden hesaplanır.
- `envp` ise programın ortam değişkenlerini taşıyan, yine null-terminated başka bir listedir. Bunlar... geleneksel olarak `KEY=VALUE` çiftleridir. *Geleneksel olarak.* Bilgisayarları seviyorum.

Küçük ama önemli bir ayrıntı: Bir programın ilk argümanının program adı olması diye bildiğimiz şey sadece bir *konvansiyon*dur. `execve` bunu kendiliğinden ayarlamaz. İlk argüman, çağıran kod `argv` içine ne koyduysa odur; ister program adı olsun ister bambaşka bir şey.

Yine de ilginç biçimde, bazı kod yollarında `execve` ve çevresindeki logic, `argv[0]`'ın program adını temsil ettiğini varsayar. Birazdan interpreted dillerden söz ederken bunun bir örneğini göreceğiz.

Adım 0: Çağrı Kernel'a Nasıl Ulaşıyor?

`execve` yazdığında çalışan şey aslında libc'nin ince bir sarmalayıcısıdır. O sarmalayıcı, **[2. bölümde]** gördüğümüz gibi argümanları register'lara yerleştirir ve mimariye özgü giriş talimatını tetikler. CPU kernel mode'a geçer, kernel da bu çağrıyı tek bir ortak yardımcıya yönlendirir.

Yol boyunca yapılan ilk iş küçük ama önemli: dosya adı **user space'ten kernel space'e kopyalanır**. Kernel, kullanıcının verdiği bir işaretçiye asla doğrudan güvenmez; kullanıcı o belleği çağrının ortasında değiştirebilir ya da geçersiz bir adres verebilir. Kopyalamak, kontrol edilen değerin sonradan değişmemesini garanti eder. Bu "önce kopyala, sonra doğrula" refleksi kernel kodunun her yerinde karşına çıkar ve bir güvenlik açığı sınıfının tamamını kapatır.

Adım 1: Kurulum

Kernel artık ortak yürütme yolundadır ve ilk işi bir çalışma dosyası hazırlamaktır: `linux_binprm` adlı bir yapı. Adı `binary program`'ın kısaltmasıdır ve bu syscall boyunca "yeni program hakkında bildiğimiz her şey" onun içinde taşınır.

İçindeki alanlar bölümün geri kalanının haritası gibidir:

- **Yeni bir bellek düzeni.** Henüz kurulmamış, ama yeri ayrılmıştır. Eski programın belleği hâlâ ayakta; ikisi bir süre yan yana durur.
- **`argc` ve `envc`.** Argümanlar ve ortam değişkenleri sayılıp kernel tarafına kopyalanır, çünkü birazdan eski adres alanı yok olacaktır ve onları orada bırakmak olmaz.
- **`filename` ve `interp`.** Başlangıçta ikisi de aynı dosyayı gösterir. Bir Python betiği çalıştırdığında `filename` betiğin kendisi olarak kalır ama `interp` yorumlayıcının yoluna kayar. Bu iki alanın ayrılabiliyor olması, birazdan göreceğimiz shebang mekanizmasının tamamını mümkün kılan şeydir.
- **256 baytlık bir tampon.** Dosyanın **ilk 256 baytı** buraya okunur.

Son madde bu bölümün en verimli ayrıntısı, o yüzden üstünde duralım. Kernel, bir dosyanın ne olduğunu anlamak için dosyanın tamamını okumaz. Baştan sabit sayıda bayt alır ve

kararını yalnızca ona bakarak verir. Bu, hem hızlıdır hem de kötü niyetli bir dosyanın kernel'ı keyfi miktarda okumaya zorlamasını engeller — ama birazdan göreceğin gibi kendi tuhaf sonuçları vardır.

Bir dürüstlük notu: `execve`'nin "başarısız olursa eski program devam eder" sözü tam değildir. Kernel uygun bir format bulamazsa, yetki yetmezse veya hazırlık erken bozulursa gerçekten hata döner ve hiçbir şey olmamış gibi devam edilir. Ama bir noktadan sonra eski adres alanı sökülür; buna **point of no return** denir ve ondan sonra ortaya çıkan bir hata artık geri dönülebilir değildir. Process sonlandırılır. Günlük mental modelde "başarırsa dönmez, başarısız olursa hata verir" yeterlidir; gerçekte arada küçük bir gri şerit vardır.

Adım 2: Binfmt'ler

Kurulum bitti; şimdi kernel'ın asıl sorusu geliyor: elimdeki bu dosya *ne*?

Kernel bunu tek bir yerden bilmez. Bunun yerine, her biri yalnızca tek bir dosya biçiminden anlayan bir handler listesi tutar ve dosyayı sırayla hepsine gösterir. Bu handler'lara **binfmt** denir (binary format). Her biri kernel kaynağında kendi dosyasında yaşar: ELF için `fs/binfmt_elf.c`, gömülü sistemlerde kullanılan düz formatlar için `fs/binfmt_flat.c`, birazdan uzun uzun konuşacağımız shebang için `fs/binfmt_script.c`. Liste sabit de değildir; yüklenen bir kernel modülü havuza kendi binfmt'ini ekleyebilir.

Her binfmt, `load_binary()` adında tek bir fonksiyon dışa açar. Bu fonksiyon `linux_binprm` yapısını alır ve tek bir soruya cevap verir: bu dosya benim tanıdığım format mı?

Kontrolün en yaygın biçimi çok basittir. Handler, buffer'ın ilk birkaç baytına bakar ve kendi imzasını arar. Bu imzaya **magic number** denir — bir dosyanın en başına vurulmuş kimlik damgası gibi düşün. ELF dosyaları `0x7F` baytıyla ve ardından gelen `E`, `L`, `F` harfleriyle başlar; `binfmt_script` ise yalnızca `#` ve `!` baytlarına bakar. Bazı handler'lar bunun yerine dosyanın başındaki başlığı çözmeye çalışır ya da dosya uzantısına bakar.

Tanıdıysa programı çalıştırmaya hazırlar ve başarı döner. Tanımadıysa hiçbir şeye dokunmadan geri çekilir ve `ENOEXEC` döner — yani "bu benim işim değil, sıradakine sor".

Kernel bu listeyi biri başarılı olana kadar dolaşır. Zincir bazen kendi üzerine de katlanır: bir script'in yorumlayıcısı varsa ve o yorumlayıcı da bir script'se sıra `binfmt_script` → `binfmt_script` → `binfmt_elf` şeklinde ilerleyebilir. Dikkat et, zincirin sonunda her zaman gerçek bir binary format duruyor; script bir yerde durup asıl makine koduna teslim olmak zorunda.

Format Vurgulama: Script'ler

Linux'un desteklediği pek çok formattan `binfmt_script` özellikle bahsetmek istediğim ilk format.

Hiç shebang satırı gördün mü? Hani bazı script'lerin en başında interpreter yolunu söyleyen şu satır:

```
1  #!/bin/bash
```

Buradaki en yaygın yanılgı, shebang satırını shell'in okuduğunu sanmaktır. Okumaz. Shebang bir **kernel** özelliğidir; bir script, terminalden mi yoksa bambaşka bir programın içinden mi çağrıldığından bağımsız olarak, tam olarak derlenmiş bir binary ile aynı syscall üzerinden yürütülür. Shell'i tamamen devre dışı bıraksan bile `#!` satırı çalışmaya devam eder.

`fs/binfmt_script.c` dosyasının, bir dosyanın `#!` ile başlayıp başlamadığını nasıl kontrol ettiğine bak:

load_script @ fs/binfmt_script.c

```
40          /* Not ours to exec if we don't start with "#!". */
41          if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!'))
42                  return -ENOEXEC;
```

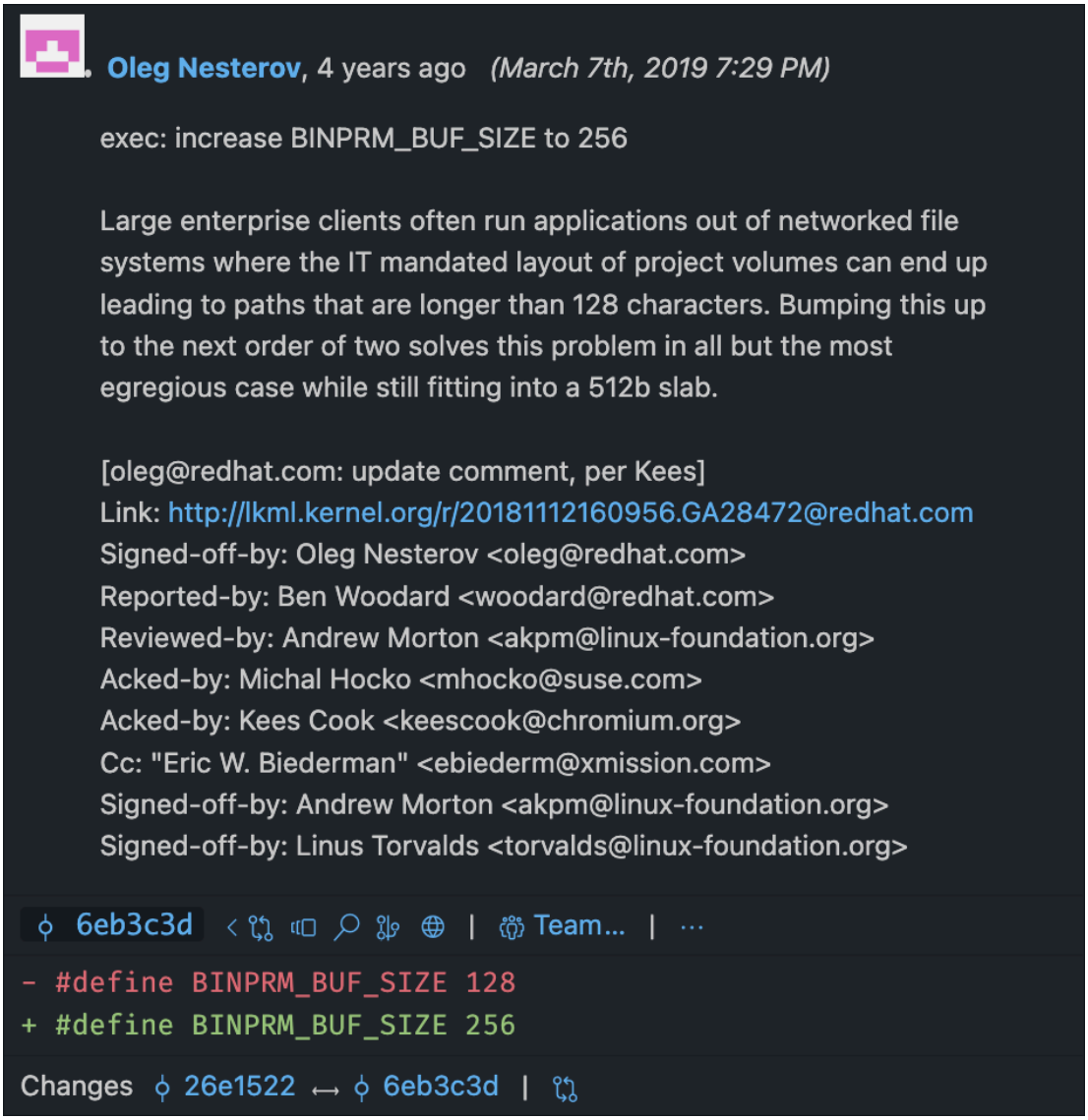
Dosya bir shebang ile başlıyorsa `binfmt_script`, ilk boşluğa kadar olan kısmı yorumlayıcının yolu olarak alır. Kalan kısmın tamamını ise — içinde kaç tane boşluk olursa olsun — **tek bir argüman** olarak yorumlayıcıya geçer. Okuma, yeni satıra ya da buffer'ın sonuna gelindiğinde durur.

Bu ayrım hayatında en az bir kez başını ağrıtacak. `#!/usr/bin/env python3 -u` yazdığında `env` programı `python3` ve `-u` diye iki argüman almaz; `"python3 -u"` diye tek parça bir metin alır, bu isimde bir program bulamaz ve script çalışmaz. (GNU coreutils 8.30 ve sonrasında gelen `env -S` bayrağı, o satırı senin adına parçalamak için vardır.)

Burada iki ilginç, riskli şey oluyor.

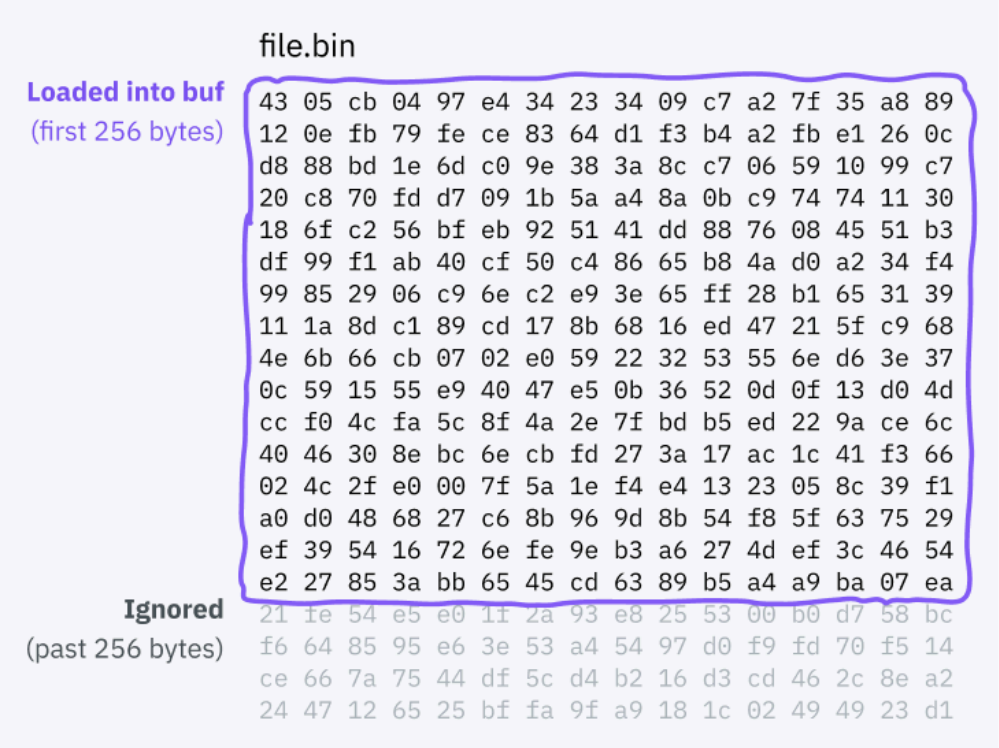
Birincisi, `linux_binprm` içindeki ve dosyanın ilk 256 baytıyla doldurulan buffer'ı hatırlıyor musun? Yürütülebilir formatı tespit etmek için kullanılan bu aynı buffer, `binfmt_script` içinde shebang satırlarını okumak için de kullanılıyor.

Bu buffer her zaman 256 bayt değildi; uzun süre 128 bayttı. `BINPRM_BUF_SIZE` satırının Git blame kaydına bakarsan sınırın neden ikiye katlandığını tek bir commit'te bulursun:



Yani bugün shebang satırına sığdırabildiğin yol uzunluğu, birinin gerçek bir hata raporunun peşine düşüp bu sabiti ikiye katlamasının sonucu. Kernel'daki pek çok sayı böyledir: teoriden değil, birinin canını yakan somut bir olaydan gelir.

Shebang kernel tarafından işlendiği ve tüm dosya yerine yalnızca `buf` içinden okunduğu için, interpreter satırı sabit bir üst sınıra takılır. Linux 5.1'den beri `#!` sonrasında okunan metin sınırı 255 karakterdir; daha eski kernel'larda bu sınır 127 karakterdi. `BINPRM_BUF_SIZE = 256` teknik detayı buradan gelir. Satır bu sınıra sığmazsa fazla kısım yok sayılır; kernel yarım kalan yolu/argümanı denediği için sonuç çoğu zaman `ENOENT` veya benzeri bir `execve` hatasıdır — veri sadece kaybolup program sessizce doğru çalışmaya devam etmez.



Böyle bir bug yaşadığını düşün. Kodunu bozan şeyin kök nedenini arıyorsun. Sonra problemin, Linux kernel'ının derinliklerinde duran bir buffer uzunluğu sınırı olduğunu öğreniyorsun. Büyük kurumsal path'lerde bir kısmın gizemli biçimde silindiğini fark eden bir sonraki BT çalışanına şimdiden sabır diliyorum.

İkinci riskli şey: Az önce `argv[0]`'ın program adı olmasının sadece bir *konvansiyon* olduğunu ve çağıranın istediği `argv`'yi verebileceğini konuşmuştuk ya? İşte `binfmt_script`, `argv[0]`'ın program adı olduğunu varsayan yerlerden biri.

Bu handler, önce `argv[0]`'ı siler ve ardından `argv`'nin başına şunları ekler:

- Interpreter'ın yolu
- Interpreter'a verilecek tek argüman (varsa)
- Script dosyasının adı

> **Örnek: Argümanların Yeniden Yazılması**
>
> Örnek bir `execve` çağrısına bakalım:
>
> ```
> // Argümanlar: filename, argv, envp
> execve("./script", [ "A", "B", "C" ], []);
> ```
>
> Bu varsayımsal `script` dosyasının ilk satırı şöyle olsun:
>
> script
>
> ```
> 1  #!/usr/bin/node --experimental-module
> ```
>
> Sonuçta Node interpreter'ına giden değiştirilmiş `argv` şu olur:
>
> ```
> [ "/usr/bin/node", "--experimental-module", "./script", "B", "C" ]
> ```

`argv` güncellendikten sonra handler, `linux_binprm.interp` değerini interpreter yoluna ayarlayarak yürütme hazırlığını tamamlar. Son olarak programın başarıyla hazırlandığını göstermek için `0` döndürür.

Format Vurgulama: ELF

Zincirin sonunda neredeyse her zaman aynı handler duruyor: `binfmt_elf`. Linux'ta çalıştırdığın hemen her derlenmiş program — `ls`'ten tarayıcına kadar — ELF formatındadır.

Tanıma adımı, biraz önce anlattığım şablonun ders kitabı örneği. `binfmt_elf`, buffer'ın ilk dört baytına bakar ve şunu arar: `0x7F`, ardından `E`, `L`, `F` harfleri. Kendi gözünle görmek istersen:

Shell oturumu

```
$ head -c 4 /bin/ls | xxd
00000000: 7f45 4c46                                .ELF
```

O `7f` baytının neden orada olduğu bile düşünülmüş: yazdırılabilir bir karakter değildir, dolayısıyla bir ELF dosyasını yanlışlıkla metin olarak açan bir program onu hemen ikili dosya olarak tanır.

İmza tutunca `binfmt_elf` işi devralır ve asıl ağır iş başlar: dosyanın hangi parçasının belleğin neresine yerleşeceğini okumak, o parçaları yeni adres alanına haritalamak, gerekiyorsa dinamik linker'ı devreye sokmak ve nihayet CPU'yu programın giriş noktasına atlatmak.

O ağır işin tamamı **[12. bölümün]** konusu; ELF'in içini oraya bırakıyorum. Bu bölüm açısından önemli olan tek şey şu: kernel dosyayı **tanıdı** ve sorumluluğu doğru handler'a devretti.

Format Vurgulama: Çeşitli Yorumlayıcılar

Son olarak bir tanesinden daha söz edeceğim, çünkü diğerlerinden farklı bir şey yapıyor: `binfmt_misc`.

Buraya kadar gördüğümüz her binfmt kernel'ın içinde C ile yazılmıştı; yeni bir format desteklemek istiyorsan kernel'a kod yazman gerekiyordu. `binfmt_misc` bu kapıyı user space'e açar: hangi dosyanın hangi yorumlayıcıya gideceğini, kernel'a tek satır kod eklemeden sen tarif edersin. Kernel `/proc/sys/fs/binfmt_misc/` altına küçük bir dosya sistemi mount eder ve oraya belirli bir biçimde yazdığın her satır yeni bir format kaydı olur.

Her kayıt üç şeyi söyler:

- **Format nasıl tanınacak.** Belirli bir konumdaki bir magic number ya da aranacak bir dosya uzantısı.
- **Hangi yorumlayıcı çalıştırılacak.** Yorumlayıcıya argüman geçirmenin bir yolu yoktur; gerekiyorsa araya bir sarmalayıcı script koyman gerekir.
- **Birkaç yapılandırma bayrağı.** Bunlardan biri `binfmt_misc`'in `argv`'yi nasıl kuracağını belirler.

Bu `binfmt_misc` sistemi geçmişte Java class/JAR dosyaları gibi formatlar için kullanılabildiği gibi, bugün pratikte QEMU user emulation ve container/multi-arch geliştirme akışlarında da sık karşına çıkar: örneğin ARM için derlenmiş bir binary'yi x86-64 makinede uygun QEMU interpreter'ına otomatik yönlendirmek mümkün olur. Bazı sistemlerde özel bytecode, `.pyc`

benzeri dosyalar veya kurum içi executable formatları da bu yolla bir kullanıcı alanı yorumlayıcısına aktarılabilir.

Buradaki tasarım fikri hoşuma gidiyor: kernel, kendisine yeni format öğretme yetkisini dışarı veriyor ama bunu ayrıcalıklı kod çalıştırmadan yapıyor. Yeni bir format desteklemek için artık kernel geliştiricisi olman gerekmiyor.

Zincirin Sonu

Bir exec syscall'ı pratikte iki yoldan biriyle biter:

- Belki birkaç kat script yorumlayıcısından geçtikten sonra, kernel tanıdığı bir binary formata ulaşır ve o kodu çalıştırır. İşte tam bu anda eski program artık yoktur; yerini yenisi almıştır.
- Ya da kernel elindeki bütün handler'ları dener, hiçbiri bu dosyayı tanımaz ve çağıran programa eli boş döner: bir hata kodu.

Nadir kernel-içi hata yollarında point of no return sonrası eski programa dönülemeyeceğini az önce not etmiştik; bu ayrıntı günlük mental modelde değil, doğruluk payı olarak aklının köşesinde dursun.

Unix benzeri bir sistem kullandıysan, terminalden çalıştırılan shell script'lerin bazen ne shebang ne de `.sh` uzantısı olmadan yine de yürütüldüğünü fark etmiş olabilirsin. Elinin altında Unix benzeri bir terminal varsa hemen deneyebilirsin:

Shell oturumu

```
$ echo "echo hello" > ./file
$ chmod +x ./file
$ ./file
hello
```

(`chmod +x`, işletim sistemine dosyanın çalıştırılabilir olduğunu söyler. Bunu yapmazsan dosyayı yürütemezsin.)

Peki shell script neden shell script olarak çalışıyor? Kernel'ın format handler'larının, üzerinde açık bir etiket olmayan shell script'i güvenilir biçimde tanıması mümkün görünmüyor.

Çünkü bu davranış aslında kernel'ın işi değil. Bu, *shell* tarafında başarısız bir exec çağrısını ele almanın yaygın yolu.

Bir dosyayı shell üzerinden çalıştırdığında exec syscall'ı başarısız olursa, çoğu shell dosyayı yeniden denemek için bu kez bir shell process'i başlatır ve dosya adını ona ilk argüman olarak verir. Bash genelde kendi kendisini interpreter olarak kullanır; ZSH ise çoğu zaman `sh`'in işaret ettiği şeyi, yani genellikle Bourne shell'i çağırır.

Bu davranış o kadar yaygındır ki, Unix sistemleri arasında taşınabilirliği hedefleyen eski standartlardan biri olan POSIX'te bile yer alır. POSIX bugün her araç ve sistem tarafından birebir takip edilmese de, pek çok davranış hâlâ onun izini taşır. Yine de hangi shell'in hangi interpreter'ı seçtiği ve binary olmayan dosyaya ne kadar tolerans gösterdiği implementation'a göre değişebilir.

> Bir exec syscall'ı `[ENOEXEC]` ile eşdeğer bir hatayla başarısız olursa, shell komut adını ilk argüman olarak verdiği yeni bir shell process'i başlatır ve kalan argümanları da bu yeni shell'e aktarır. Çalıştırılmak istenen dosya bir metin dosyası değilse shell bu denemeyi atlayabilir; bu durumda hata yazdırır ve `126` çıkış kodu döndürür.
>
> *Kaynak: Shell Command Language, POSIX.1-2017*

Yani terminalde gördüğün o basit davranışın arkasında iki ayrı katman var: kernel "bu formatı tanımıyorum" diyor, shell de bu cevabı kesin bir hayır olarak değil, bir öneri gibi okuyup dosyayı kendisi çalıştırmayı deniyor.

Özet

> **Peki, ne öğrendik?**
>
> - Bir programı çalıştırmanın tek yolu `execve` ailesidir. Yeni bir process kurmaz; mevcut process'in kimliğini değiştirir.
> - Başarılı bir `execve` **dönmez**, çünkü dönecek program artık yoktur.
> - Kernel dosyanın ne olduğunu anlamak için yalnızca ilk 256 baytına bakar ve bunu sırayla her **binfmt** handler'ına sorar.
> - Tanıma çoğu zaman bir **magic number** kontrolüdür: ELF için `0x7F ELF`, betikler için `#!`.
> - **Shebang** satırı bir kernel özelliğidir, shell özelliği değil. Kernel yorumlayıcıyı bulur ve asıl dosyayı ona argüman olarak verir.
> - `binfmt_misc` ile kernel'a yeni format tanıtılabilir; `.exe` dosyalarının Linux'ta çift tıklamayla çalışması bu sayededir.

Peki terminalde `./program` yazdığında bu `execve` çağrısını kim yapıyor? `ls`, `cat`, `echo` gibi komutlar neden doğrudan çalışıyor da kendi programının başına `./` koymak gerekiyor? Sıradaki bölümde, kullanıcı ile kernel arasındaki ilk elçiyi tanıyacağız: **shell**.

Bölüm 11: Shell'den Kernel'e

Bir önceki bölümde `execve` syscall'ının mekaniklerini gördük. Peki terminale `./program` yazıp Enter'a bastığında bu çağrıyı senin adına kim yapıyor? `ls`, `cat`, `echo` gibi komutlar neden doğrudan çalışıyor da `./benim-programim` yazarken başına `./` koymamız gerekiyor? Bu bölümde, kullanıcı ile kernel arasındaki ilk elçi olan **shell** dünyasına dalacağız.

> **Bu bölümde neyi çözüyoruz?**
>
> - Shell'in bir komutu nasıl çözümlediğini göreceğiz.
> - PATH, built-in komutlar ve harici programlar arasındaki farkı ayıracağız.
> - Pipeline (`|`), yönlendirme (`>`, `<`) ve fork+execve+wait üçlüsünü bağlayacağız.
> - Environment variable'ların programlara nasıl aktarıldığını netleştireceğiz.
> - `Ctrl-C`'nin aslında kime gittiğini ve iş kontrolünün nasıl çalıştığını çözeceğiz.

Shell Nedir?

Terminali açtığında karşına çıkan o yanıp sönen imleç bir programa ait: **shell**. İşi tek cümleyle şu — yazdığın satırı okur, ne demek istediğini çözer ve kernel'dan doğru şeyi ister.

Bugün en çok karşılaşacakların **Bash** (Bourne Again Shell), **Zsh** ve **Fish**; hepsinin atası olan sade sürümün adı ise **sh**. Aralarındaki fark ne yaptıkları değil, aynı işi ne kadar konforlu yaptıkları.

Şu anda hangisinin içinde olduğunu merak ediyorsan iki komut yeter:

Shell oturumu

```
$ echo $SHELL      # giris shell'in olarak kayitli olan
$ ps -p $$         # su anda gercekten calisan
```

İkisi her zaman aynı çıkmaz ve bu şaşırtıcı değil: `$SHELL` sistemin senin için kayıtlı tuttuğu tercihtir, `ps -p $$` ise o anda çalışan process'i gösterir. Bir Bash içinden `zsh` yazdığında ikincisi değişir, birincisi değişmez.

> Script yazarken en sık yapılan hatalardan biri buraya bağlanıyor: dosyanın başına `#!/bin/sh` yazıp içeride Bash'e özgü bir özellik kullanmak. Pek çok dağıtımda `/bin/sh` Bash değildir (Debian ve Ubuntu'da `dash`'tir) ve script, senin makinende çalışıp sunucuda patlar. Bash'e özgü bir şey kullanacaksan shebang'i de `#!/bin/bash` yap.

Shell'in kendisi de sıradan bir kullanıcı programıdır. Kernel'ın gözünde `bash` ile hesap makinesi uygulaması arasında hiçbir fark yoktur; ayrıcalığı yoktur, sihri yoktur.

Komut Çözümleme: PATH ve Built-in'ler

Bir komut girdiğinde shell, onu çalıştırmadan önce "bu isim kime ait?" sorusunu sabit bir sırayla cevaplar. Sıra önemli, çünkü öndeki her basamak arkasındakini gölgeler:

1. **Alias** — `alias ls='ls --color'` yazdıysan burada yakalanır.
2. **Shell fonksiyonu** — kendi tanımladığın fonksiyonlar.
3. **Built-in** — `cd`, `export` gibi shell'in içinde olanlar.
4. **Hash tablosu** — daha önce bulunmuş bir programın hatırlanan yolu.
5. **PATH araması** — hiçbiri tutmazsa diskte aranır.

Bunu tahmin etmene gerek yok; `type` sana doğrudan söyler:

Shell oturumu

```
$ type cd
cd is a shell builtin
$ type ls
ls is aliased to `ls --color=auto'
$ type -a python3
python3 is /usr/local/bin/python3
python3 is /usr/bin/python3
```

> Bir komutun neden beklediğinden farklı davrandığını anlamanın en hızlı yolu `type -a` çalıştırmaktır. Son örnekteki gibi iki aday çıkıyorsa kazanan her zaman listedeki ilkidir.

Built-in Komutlar

`cd`, `exit`, `export`, `alias` gibi komutlar shell'in kendisinin içindedir. Bunlar için ayrı bir program çalıştırılmaz; shell doğrudan kendi kodunu çalıştırır.

> Neden `cd` built-in'dir? Çünkü `cd` mevcut process'in çalışma dizinini değiştirir. Eğer `cd` ayrı bir program olsaydı, o program kendi process'inde çalışır ve senin shell'inin dizini hiç değişmezdi. İşte bu yüzden `cd` shell'in içindedir.

Harici Programlar

`ls`, `grep`, `node`, `python` gibi komutlar diskteki programlardır. Shell bunları çalıştırmak için önce nerede olduklarını bulmalıdır. Bunun için **PATH** ortam değişkenini kullanır.

Shell oturumu

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

PATH, dizinlerin iki nokta üst üste (`:`) ile ayrıldığı bir listedir. Shell, komutunu bu dizinlerde **sırayla** arar ve ilk bulduğunda durur. Yani aynı adı taşıyan iki program varsa kazanan, daha iyi olan değil, PATH'te önce gelen.

Bulamazsa ne olur? Shell `command not found` mesajını kendisi basar ve `127` çıkış kodu döner. Biraz sonra göreceğimiz kod örneğindeki `_exit(127)` tam olarak bu kuralı uyguluyor.

Bir de az bilinen bir davranış var: Bash bulduğu yolları bir hash tablosunda saklar, her seferinde baştan aramaz. Bu yüzden bir programı başka bir dizine taşıdığında Bash ısrarla eski yolu denemeye devam edebilir; `hash -r` komutu bu hafızayı temizler.

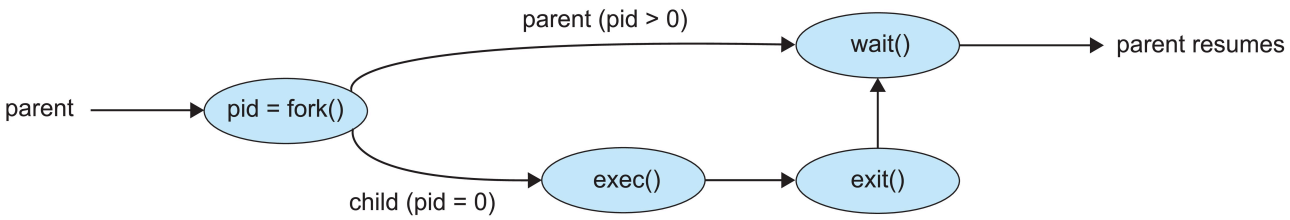
> `./` **neden gerekli?** Yaygın açıklama "şu anki dizinde ara" demektir ama bu tam olarak doğru değil ve yanlış bir model kuruyor. Gerçek kural şu: **içinde `/` geçen her şeyi shell PATH'te aramaz, doğrudan bir yol olarak okur.** `./program` da bunu yapar — "arama yapma, şu anki dizindeki şu dosyayı çalıştır" der. PATH'e mevcut dizin eklenmiş falan değildir; nitekim bölümün sonunda göreceğin gibi, eklemek de ciddi bir güvenlik hatasıdır.

Fork + Execve + Wait: Shell'in Motoru

Shell bir harici program çalıştıracağı zaman şu üç adımı uygular:

1. `fork()`: Shell kendini klonlar. Artık iki shell process'i var.
2. **Child'ta** `execve()`: Child process, shell kodunu yeni programın koduyla değiştirir.
3. **Parent'ta** `wait()`: Parent (orijinal shell), child'ın bitmesini bekler.

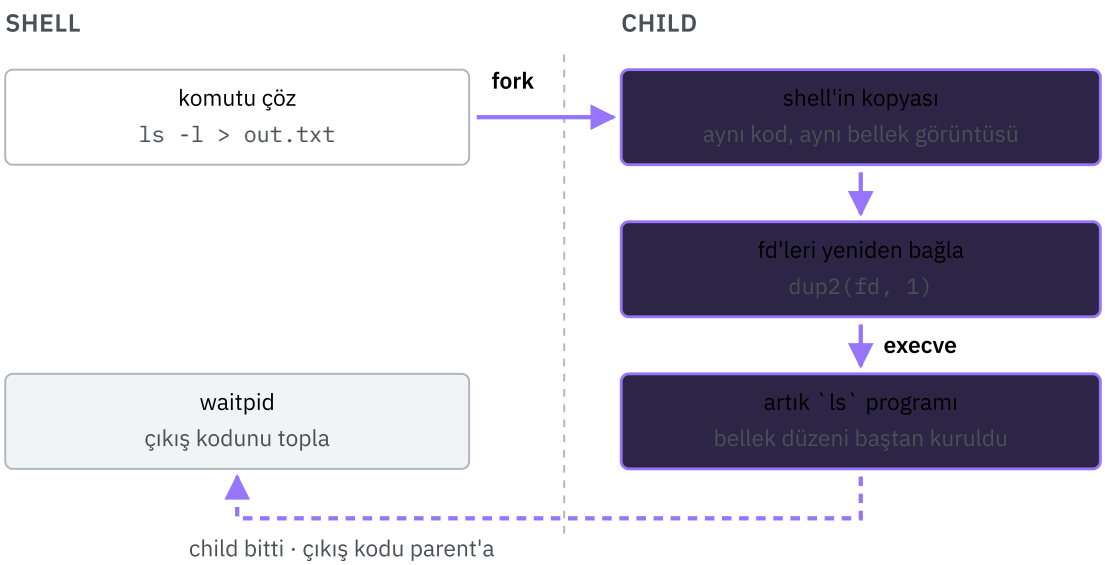
Bu üç adımı tek bir bakışta görmek için aşağıdaki diyagrama bakabilirsin. Shell parent process olarak `fork()` ile bir child üretir; child, `exec()` ile kendi bellek alanını yeni programın koduyla değiştirir. Parent ise `wait()` ile child'ın sonlanmasını bekler. Bu model, Unix dünyasında yeni bir program başlatmanın temel kalıbıdır.



Kaynak: Silberschatz, Galvin, Gagne — Operating System Concepts, Ch. 3 — Processes, Slide 21.

Aynı akışı adım adım ilerleterek de izleyebilirsin; üçüncü adım özellikle önemli, çünkü yönlendirme ve pipe kurulumunun tamamı orada yaşanır:

BİR KOMUT ÇALIŞTIRILDIĞINDA SHELL NE YAPAR?



1. **Shell komutu çözer.** Satır kelimelere ayrılır, `*.txt` gibi kalıplar dosya adlarına açılır ve çalıştırılacak programın yolu `PATH` üzerinden bulunur.
2. **`fork`: kopya çıkarılır.** Kernel shell'in neredeyse birebir kopyasını üretir. İki process de aynı satırdan devam eder; tek ayırt edici işaret `fork`'un döndürdüğü değerdir.
3. **Child kendi dünyasını kurar.** Asıl iş bu küçük pencerede yapılır: `dup2` ile file descriptor'lar yeniden bağlanır, yönlendirme ve pipe burada kurulur. Shell bunu child'ın içinde yapar, çünkü kendi fd'lerini bozmak istemez.
4. **`execve`: kimlik değişir.** Child'ın bellek düzeni tamamen sökülür ve yerine yeni programınki kurulur. Process numarası ve açık fd'ler kalır; geri kalan her şey gider.
5. **`waitpid`: parent bekler.** Shell child bitene kadar bekler ve çıkış kodunu toplar. Toplamazsa child ölü ama kayıtlı kalır — buna zombie denir.

Bir shell'in kalbi

```
pid_t pid = fork();                 // buradan sonrası iki process'te birden akar

if (pid == 0) {                     // child: kendini yeni programa dönüştür
    execvp(argv[0], argv);          // başarılıysa aşağısı hiç çalışmaz
    perror("execvp");               // buraya düştüysek program bulunamadı
    _exit(127);
}

int status;
waitpid(pid, &status, 0);           // parent: child bitene kadar prompt'u verme
```

Child Ne Diyerek Öldü?

Yukarıdaki kodda `waitpid`'in doldurduğu `status` değişkenini fark ettin mi? İçinde küçük ama günlük hayatta sürekli karşına çıkan bir bilgi var: **çıkış kodu**.

Sözleşme basit ve evrensel: **0 başarı, sıfırdan farklı her şey hata.** Terminalde son komutun ne döndürdüğünü `echo $?` ile görebilirsin. Birkaç değerin özel anlamı da vardır:

Kod	Anlamı
`0`	Başarı
`1`, `2`, …	Programın kendi tanımladığı hatalar
`126`	Dosya bulundu ama çalıştırma izni yok
`127`	Komut bulunamadı (yukarıdaki `_exit(127)`)
`130`	`Ctrl-C` ile sonlandırıldı

Son satır ilk bakışta tuhaf görünüyor. Sebebi şu: bir process sinyalle öldürüldüğünde çıkış kodu `128 + sinyal numarası` olarak raporlanır. `Ctrl-C`, `SIGINT` gönderir, `SIGINT`'in numarası 2'dir, 128 + 2 = 130. Bölümün ilerleyen kısmında sinyalleri konuşurken bu sayıya geri döneceğiz.

Geri Kalan Her Şey Konfor

Bu on satır, `bash` yazdığında olan biten her şeyin özü. Etrafındaki her şey — komut satırını kelimelere ayırmak, `*.txt` gibi kalıpları genişletmek, alias çözmek, geçmişi tutmak — kolaylık katmanıdır. Çıkarırsan shell kullanışsız olur ama çalışmaya devam eder. `fork`/`exec`/`wait` üçlüsünü çıkarırsan geriye shell diye bir şey kalmaz.

Yine de o "kolaylık katmanı"nın bir parçası var ki, bilmezsen günün birinde başını gerçekten ağrıtır.

`*` Karakterini Kim Çözüyor?

`ls *.txt` yazdığında `ls` programının `*.txt` diye bir şey gördüğünü sanmak çok yaygın bir yanılgı. Görmez. O kalıbı **shell** çözer ve `ls` çalışmaya başlamadan *önce* iş biter. Kendi gözünle gör:

Shell oturumu

```
$ echo *.txt
notlar.txt rapor.txt taslak.txt
```

`echo` sadece kendisine verilen argümanları yazdırır. Ekranda üç dosya adı görüyorsan, `echo`'nun `argv`'sine giren şey `*.txt` değil, o üç isimdir. Buna **glob genişletme** denir.

Üç sonucu var ve üçü de pratikte karşına çıkar:

- **Tırnak genişletmeyi durdurur.** `rm *.txt` üç dosyayı siler; `rm "*.txt"` ise gerçekten `*.txt` adında bir dosya arar ve muhtemelen bulamaz.
- **Eşleşme yoksa kalıp olduğu gibi geçer.** Hiç `.txt` dosyası yoksa Bash varsayılan ayarında `*.txt`'yi harfi harfine argüman olarak verir; programın aldığı şey anlamsız bir dosya adı olur.
- **Boşluklu dosya adları tırnaksız değişkenlerde parçalanır.** `rm $dosya` yazarsan ve `$dosya` içinde `tatil fotoğrafı.jpg` varsa shell bunu iki ayrı argümana böler. Bu yüzden script'lerde değişkenler her zaman `"$dosya"` diye tırnak içinde yazılır.

> Program `*` diye bir şey görmez. Gördüğü an, shell işini yapmamış demektir.

Dikkat çeken bir ayrıntı: `execvp` başarılı olursa **geri dönmez**. Dönüş değeri kontrol edilmez, çünkü dönmüş olması zaten başarısızlık demektir. Bu, C'de görmeye alışık olmadığın bir çağrı sözleşmesidir ve doğrudan **[10. bölümde]** gördüğümüz "exec bir doğum değil, başkalaşımdır" fikrinden gelir.

Pipeline: `|` Karakteri

Pipeline, bir komutun çıktısını diğerinin girdisi yapar:

Shell oturumu

```
$ cat dosya.txt | grep "arama" | wc -l
```

Arka planda olanlar:

1. Shell bir **pipe** (boru) oluşturur: iki ucu olan tek yönlü bir veri kanalı. Terimleri baştan ayıralım — *pipe* tek bir kanal, *pipeline* ise o kanallarla birbirine dizilmiş komut zinciri.
2. Shell `cat` için bir child üretir ve child'ın stdout'unu pipe'ın yazma ucuna bağlar. `cat` bunu hiç fark etmez; her zamanki gibi stdout'a yazar, ama yazdığı yer artık ekran değil, pipe'tır.
3. Shell `grep` için ikinci bir child üretir; onun stdin'ini pipe'ın okuma ucundan besler, stdout'unu ise yeni bir pipe'a bağlar.
4. `wc -l` için üçüncü bir child üretir ve girdisini ikinci pipe'dan alır.
5. Shell bu üç child'ın da bitmesini bekler.

> **Pipe'ın içinde ne var?** Pipe, kernel tarafında tutulan bir bellek tamponudur; veri diske hiç uğramaz. Ama bu tampon sonsuz değil — Linux'ta varsayılan olarak 64 KiB.
>
> Bunun iki gözlemlenebilir sonucu var. Birincisi: tampon dolduğunda yazan taraf uyutulur, okuyan boşaltana kadar bekler. Bir pipeline'ın hızını en yavaş halkası belirler, çünkü öndekiler ona doğru geri basınç uygular. İkincisi: okuyan taraf kanalı kapatırsa yazan tarafa `SIGPIPE` sinyali gider ve o program varsayılan olarak ölür. `yes | head -1` komutunun sonsuz döngüye girmeyip anında bitmesinin sebebi tam olarak budur — `head` bir satır alıp çıkar, `yes` de bir sonraki yazmasında `SIGPIPE` yiyip ölür.

Yönlendirme: `>`, `<`, `>>`

Yönlendirme, bir programın nereden okuyup nereye yazdığını — program bundan hiç haberdar olmadan — değiştirmenin yoludur. `ls`'in "dosyaya yazan" bir sürümü yoktur; `ls`'i buna ikna eden shell'dir.

Önce üç sayıyı yerine oturtalım, çünkü geri kalan her şey bunların üstüne kuruluyor. **Her process üç file descriptor ile doğar:**

fd	Adı	Varsayılan olarak nereye bağlı
`0`	stdin	klavye
`1`	stdout	ekran
`2`	stderr	ekran

Şimdi işaretler anlam kazanıyor:

- `>` : stdout'u bir dosyaya yazar (üzerine yazarak). Aslında `1>`'ın kısaltmasıdır.
- `>>` : stdout'u bir dosyanın sonuna ekler.
- `<` : stdin'i bir dosyadan okur.
- `2>` : stderr'i bir dosyaya yazar.
- `2>&1` : "2 numaralı fd'yi, 1'in **şu anda** gösterdiği yere bağla."

Son maddedeki "şu anda" vurgusu boşuna değil ve klasik bir tuzağın kaynağı:

Shell oturumu

```
$ komut > log.txt 2>&1    # ikisi de log.txt'ye gider
$ komut 2>&1 > log.txt    # stderr EKRANA gider, sadece stdout dosyaya
```

İkinci satırda `2>&1` çalıştığı anda `1` hâlâ ekranı gösteriyordu; stderr ekrana bağlandı. Sonraki `> log.txt` yalnızca `1`'i değiştirdi ve `2`'yi olduğu yerde bıraktı. Yönlendirmeler **soldan sağa** işlenir; sıra gerçekten önemli. (İkisini birden istiyorsan `&> log.txt` kısayolu da var.)

Shell oturumu

```
$ ls -la > dosyalar.txt    # Çıktıyı dosyaya yaz
$ cat < dosyalar.txt       # Dosyayı girdi olarak oku
$ ./program 2> hatalar.log # Hataları ayrı dosyaya yaz
```

Dördü de aynı tek numaraya dayanıyor: programı doğurduktan hemen sonra, çalıştırmadan hemen önce onun file descriptor'larını değiştirmek.

> Sıraya dikkat: shell **önce** `fork` eder, yönlendirmeyi child'ın içinde `dup2` ile yapar ve ancak ondan sonra `execve` çağırır. Eğer bu iş fork'tan önce yapılsaydı parent shell'in kendi çıktısı da dosyaya giderdi ve terminalinde bir daha hiçbir şey göremezdin. Dosyaya yazan, o an doğmuş olan child'dır.

Environment Variables (Ortam Değişkenleri)

Her process doğarken yanında küçük bir not defteri taşır: anahtar-değer çiftlerinden oluşan **environment**. Shell yeni bir program başlatırken kendi defterinin bir kopyasını child'a verir. `PATH`'in çalıştırdığın her komuta kadar ulaşmasının sebebi tam olarak bu — kopyalana kopyalana iniyor. Ve dikkat et, verilen şey bir *kopya*: child kendi defterini değiştirse parent'ınki etkilenmez.

Shell oturumu

```
$ export MY_VAR="merhaba"
$ echo $MY_VAR
merhaba
$ ./program             # MY_VAR, programın envp'sine gider
```

> `export` ile tanımlanan değişkenler, shell'in fork ettiği tüm child process'lere aktarılır. `export` kullanmadan tanımlanan değişkenler sadece shell'in kendisi için geçerlidir.

Önemli environment variable'ları:

Değişken	Anlamı
`PATH`	Komut arama dizinleri
`HOME`	Kullanıcının ana dizini
`USER`	Kullanıcı adı

Değişken	Anlamı
`SHELL`	Varsayılan shell
`LANG`	Dil ve yerel ayarlar
`PWD`	Şu anki çalışma dizini

Subshell ve Parantezler

Parantez, shell'e "bunu sen değil, senin bir kopyan yapsın" demenin yolu.

Parantezi gördüğü an shell yine `fork` eder — ama bu kez child'ta `execve` çağırmaz. Child, shell'in kendisi olarak kalır ve içerideki komutları çalıştırır; işi bitince ölür, üzerinde ne değiştirdiyse onunla birlikte gider. Buna **subshell** denir. Yani bölümün başındaki üçlünün eksik hâli: fork var, exec yok.

Shell oturumu

```
$ (cd /tmp && ls)   # Subshell'de calisir; ana shell'in dizini degismez
$ cd /tmp && ls     # Ana shell'in dizini degisir
```

(Aradaki `&&` şunu söylüyor: soldaki komut `0` döndürürse sağdakini de çalıştır. Az önce konuştuğumuz çıkış kodu sözleşmesinin doğrudan kullanımı.)

Subshell'i tanıdıktan sonra, ilk bakışta anlamsız görünen birkaç davranış yerine oturuyor.

Komut ikamesi de bir subshell'dir. `$(...)` yazdığında shell içerideki komutu bir subshell'de çalıştırır ve çıktısını yakalayıp yerine koyar.

Pipeline'ın her halkası subshell'de çalışır. Bu, Bash'in en meşhur tuzağının kaynağı:

Shell oturumu

```
$ echo merhaba | read x
$ echo $x
                    # bos!
```

`read` gerçekten çalıştı ve `x`'e "merhaba" yazdı — ama bunu kendi subshell'inin içinde yaptı. O subshell öldüğünde değişken de onunla gitti. Ana shell'in `x`'i hiç dokunulmamış olarak kaldı.

`(...)` fork eder, `{ ...; }` etmez. Süslü parantez komutları gruplar ama aynı process'te çalıştırır; bu yüzden içeride yapılan `cd` dışarıyı da etkiler.

Peki Kernel Tarafında Ne Oluyor?

Önce küçük bir sürpriz: kernel'ın içinde `fork` diye ayrı bir kapı yoktur. `fork`, `clone` adlı çok daha genel bir kapının belirli bir ayarla çağrılmış hâlidir. Aynı kapı, farklı bayraklarla çağrıldığında yeni bir process yerine yeni bir thread üretir — ikisi arasındaki fark kernel açısından yalnızca "neyi paylaşıyorlar" sorusunun cevabıdır.

Bu kapıdan girildiğinde varılan yerin işi tek cümleyle özetlenebilir: **çalışan process'i tarif eden her yapıyı kopyala, ama içindeki veriyi kopyalama.**

> Kaynağa bakmak isteyenler için: bu iş `kernel/fork.c` içindeki `copy_process` fonksiyonunda yapılıyor.

Kopyalanan şeyler process'in kimliğidir: kernel'ın o process için tuttuğu kayıt yapısı (`task_struct`), kimlik bilgileri, açık dosya tablosu, sinyal ayarları. Kopyalanmayan şey ise belleğin kendisi.

Burada bir kavramı önden vermem gerekiyor: kernel belleği tek parça hâlinde değil, sabit boyutlu bloklar hâlinde yönetir ve bu bloklara **sayfa** denir. Nasıl çalıştığını **[13. bölümde]** göreceğiz; şimdilik şunu tut. Fork sonrası iki process aynı sayfalara bakar ve kernel o sayfalara "yazmaya kalkan olursa bana haber ver" işareti koyar. Yani `fork` pahalı görünen ama pratikte ucuz olan bir çağrıdır — **[16. bölümde]** Copy-on-Write'ı ele alırken bu ucuzluğun tam olarak nereden geldiğini göreceğiz.

`execve` tarafında ise kontrol `fs/exec.c` içindeki ortak yardımcıya geçer: dosya açılır, argümanlar ve ortam değişkenleri kernel tarafına kopyalanır, sonra uygun format yükleyicisi aranır. **[10. bölümde]** bu yolun tamamını adım adım izlemiştik.

Buradan çıkarılacak asıl ders şu: shell'in yaptığı şey özel değil. Kernel'da "shell" diye bir kavram yoktur. `bash` de, bir masaüstü uygulaması da, `systemd` de yeni program başlatırken tam olarak aynı iki syscall'ı aynı sırayla çağırır.

Pratik Deney: `strace` ile Fork, Exec ve Pipe

Shell'in arka planda `clone`, `execve` ve `dup2` syscall'larını nasıl kullandığını görmek için `strace` ile bir pipeline izleyebiliriz. Aşağıdaki komut, bash'in `ls | wc` komutunu çalıştırırken yaptığı syscall'ları gösterir:

Terminal

```
$ strace -f -e trace=clone,execve,dup2 bash -c "ls | wc"
clone(...) = 2847
[pid  2847] dup2(3, 1)  = 1          # ls: stdout artik pipe'in yazma ucu
[pid  2847] execve("/usr/bin/ls", ["ls"], ...) = 0
clone(...) = 2848
[pid  2848] dup2(3, 0)  = 0          # wc: stdin artik pipe'in okuma ucu
[pid  2848] execve("/usr/bin/wc", ["wc"], ...) = 0
```

Gözünü üç şeye dikmen yeterli. **İki** `clone` var, yani iki child doğdu — biri `ls`, biri `wc` için. Her child kendi `dup2`'siyle bir fd'yi başka bir fd'nin üstüne kopyalıyor; yönlendirmenin gerçekleştiği an tam olarak burası. Sonra gelen `execve` ise child'ın kimliğini değiştirdiği an.

İki satırda da 3 numarasını görmen tesadüf değil ama aynı şeyi de göstermiyor. Pipe'ın iki ucu parent'ta iki ayrı fd olarak açılır; her child kendi kopyasında yalnızca ihtiyacı olan ucu tutup diğerini kapatır. Yani 3, `ls`'in dünyasında yazma ucu, `wc`'nin dünyasında okuma ucudur.

`pipe()` ve `dup2()` ile Pipeline

Shell, `|` karakteri gördüğünde arka planda `pipe()` syscall'ını çağırır. `pipe()` iki uçlu bir kanal oluşturur: `pipefd[0]` okuma, `pipefd[1]` yazma ucudur. Sonra sol komut için child fork edilir, `dup2` ile stdout pipe'a yönlendirilir; sağ komut için başka bir child fork edilir, `dup2` ile stdin pipe'dan okur.

ls | wc -l elle kurulursa

```
pipe(pipefd);                        // [0] okuma ucu, [1] yazma ucu
if (fork() == 0) {                   // sol komut
    dup2(pipefd[1], STDOUT_FILENO); // stdout artik pipe'a akiyor
    execlp("ls", "ls", NULL);
}
if (fork() == 0) {                   // sag komut
    dup2(pipefd[0], STDIN_FILENO);  // stdin artik pipe'dan geliyor
    execlp("wc", "wc", "-l", NULL);
}
close(pipefd[0]); close(pipefd[1]); // parent kendi kopyalarini birakir
```

Yukarıda kısalık için atladım ama gerçek bir programda her child, kullanmadığı ucu da kapatmak zorundadır — ve parent'ın sondaki `close` çağrıları süs değil. Bir pipe'ın okuma ucu, **yazma ucunu tutan son file descriptor kapandığında** EOF verir. Parent kendi kopyasını açık bırakırsa kernel'ın gözünde hâlâ yazabilecek biri vardır; `wc` sonsuza kadar bekler ve komut asılı kalır. Unix'te en sık rastlanan "sebepsiz donan pipeline" hatasının tamamı bu tek cümleden çıkar.

Bunun neden böyle olduğunu **[17. bölümde]** file descriptor'ların üç katmanlı yapısına baktığımızda daha net göreceğiz: `fork` fd'yi kopyalamaz, aynı açık dosya tanımına ikinci bir referans ekler. Referans sayısı sıfırlanmadan kanal kapanmaz.

> **Bir yana: PATH Hijacking ve Güvenlik**
>
> Eğer `PATH`'inin içinde `.` (mevcut dizin) varsa, oraya dosya bırakabilen herkes senin komutunu gölgeleyebilir. `/tmp`'ye girdiğini ve birinin oraya kendi `ls`'ini koyduğunu düşün: sen `ls` yazdığında `/usr/bin/ls` değil, onunki çalışır — üstelik hiçbir uyarı almazsın. Bu yüzden modern Linux dağıtımları `.`'yi varsayılan `PATH`'ten çıkarmıştır. Güvenlik için `./program` kullanımı, `PATH`'te olmayan dizinlerdeki programları açıkça belirtmenin en temiz yoludur.

İş Kontrolü: Ctrl-C Aslında Kime Gidiyor?

Şimdiye kadar shell'i tek yönlü bir başlatıcı gibi anlattık: komutu al, çalıştır, bitmesini bekle. Ama terminalde her gün yaptığın bir şey bu resme sığmıyor. `Ctrl-C`'ye bastığında ne oluyor?

Akla gelen ilk cevap "shell programı durduruyor" olur. Değil. Shell o sırada `waitpid` içinde uyuyor; hiçbir şey yapmıyor. `Ctrl-C`'yi gören ve karşılığında bir şey yapan taraf **terminal sürücüsüdür**.

Mekanizma şöyle işler. Kernel, birbirine bağlı process'leri **process group** denen kümelerde tutar. Terminalin her an tam olarak bir tane *foreground* process group'u vardır; klavyeden gelen `Ctrl-C`, `Ctrl-Z`, `Ctrl-\` gibi tuşlar birer sinyale çevrilip **o gruptaki bütün process'lere birden** gönderilir.

Bu tek cümle, günlük hayatta tuhaf görünen birkaç şeyi aynı anda açıklar:

- **`ls | wc -l` çalışırken `Ctrl-C` neden ikisini birden öldürür?** Çünkü shell, bir pipeline'ın tüm üyelerini aynı process group'a koyar. `SIGINT` gruba gider, ikisi de alır.
- **Shell neden ölmüyor?** Çünkü shell o grubun içinde değil. Kendi grubunda, arka planda oturuyor.
- **`sleep 100 &` ile arka plana attığın işe `Ctrl-C` neden işlemiyor?** Çünkü artık foreground grupta değil. Terminal sinyali yalnızca ön plandaki gruba yollar.

`Ctrl-Z` aynı yolu izler ama farklı bir sinyal gönderir: `SIGTSTP`. Bu sinyal process'i öldürmez, **durdurur** — bellekte olduğu gibi kalır, sadece scheduler ona bir daha CPU vermez. `fg` ile ön plana, `bg` ile arka plana devam ettirmek, kernel'a `SIGCONT` gönderip terminalin foreground grubunu yeniden ayarlamaktan ibarettir.

Tuş	Sinyal	Varsayılan davranış
`Ctrl-C`	`SIGINT`	Nazikçe sonlandır; program yakalayıp temizlik yapabilir
`Ctrl-Z`	`SIGTSTP`	Durdur; program yakalayabilir ama çoğu yakalamaz
`Ctrl-\`	`SIGQUIT`	Sonlandır ve core dump al
—	`SIGKILL`	Öldür; **yakalanamaz, engellenemez, ertelenemez**

Son satır önemli: `SIGKILL` ve `SIGSTOP` process'e hiç ulaşmaz, kernel işi doğrudan halleder. Bu yüzden `kill -9` "en güvenilir" seçenektir ve tam da bu yüzden en kötüsüdür — programa kapanırken dosyasını kapatma, kilidini bırakma, geçici dizinini silme şansı vermez.

Bir bağlantı daha: terminal penceresini kapattığında ön plandaki gruba `SIGHUP` (hang up — telefonu kapatmak) gider. Adı 1970'lerden kalmadır; gerçekten telefon hattı üzerinden bağlanan terminaller için icat edilmişti. Uzun süren bir işi `nohup` ile başlatmanın ya da `disown` ile shell'in iş listesinden çıkarmanın tek yaptığı şey, bu sinyalin yolunu kesmektir.

> **Özet ve Sonraki Adımlar**
>
> - Shell, kullanıcı ile kernel arasındaki köprüdür.
> - Built-in komutlar shell'in içindedir; harici programlar diskten `PATH` üzerinden bulunur.
> - Her harici komut için shell `fork` + `execve` + `waitpid` üçlüsünü uygular.
> - Pipeline (`|`), `pipe()` ve `dup2()` syscall'larıyla kurulur.
> - Yönlendirme (`>`, `<`), shell tarafından file descriptor'ların yeniden atanmasıyla yapılır.
> - Environment variable'lar `export` ile child process'lere aktarılır.
> - `O_CLOEXEC`, `execve` sonrası istenmeyen fd mirasını önler.
> - Terminal sinyalleri tek bir process'e değil, foreground **process group**'una gider; `Ctrl-C`'nin pipeline'ın tamamını durdurması bundandır.
> - `SIGKILL` ve `SIGSTOP` yakalanamaz; bu yüzden `kill -9` en güvenilir ve en kaba seçenektir.
>
> Artık shell'in komutları nasıl çözümlediğini, kernel'a nasıl ilettiğini ve terminalin sinyalleri kime yönlendirdiğini biliyoruz. Bir sonraki bölümde, çalıştırılan programların formatına dalıyoruz: ELF dünyası.

Bölüm 12: Bir ELF Ustasına Dönüşmek

Artık `execve`'ı epey iyi anlıyoruz. Çoğu yolun sonunda kernel, çalıştırılacak makine kodunu içeren son programa ulaşır. Ama koda atlamadan önce genelde bir kurulum süreci gerekir; örneğin programın farklı bölümlerinin bellekte doğru yerlere yüklenmesi gerekir. Her programın farklı türde ve miktarda belleğe ihtiyacı olduğundan, bir programın nasıl hazırlanacağını tarif eden standart dosya formatlarına ihtiyaç duyarız. Linux pek çok formatı destekler ama açık ara en yaygın olanı *ELF*'dir (Executable and Linkable Format).

> **Bu bölümde neyi çözüyoruz?**
>
> - ELF dosyasının kernel'a "beni belleğe şöyle yerleştir" bilgisini nasıl verdiğini göreceğiz.
> - Program header ile section header arasındaki farkı ayıracağız.
> - Static/dynamic linking, GOT, PLT ve dynamic loader rollerini basit bir akışa bağlayacağız.

(Bu sevimli çizim için Nicky Case'e teşekkürler.)

> **Bir not: elfler her yerde mi?**
>
> Linux'ta bir uygulama ya da komut satırı programı çalıştırdığında, bunun bir ELF binary olması çok olasıdır. macOS tarafında fiili format Mach-O'dur. Mach-O da temelde ELF ile aynı işi yapar, sadece farklı biçimde düzenlenmiştir. Windows'ta ise `.exe` dosyaları, yine benzer kavramları farklı bir paketle sunan Portable Executable formatını kullanır.

Bu dosyaları kernel tarafında karşılayan koda `binfmt_elf` denir ve bu kod, binfmt handler'ları arasında açık ara en kalabalık olanıdır: iki binden fazla satır. Bu kalabalığı aklının bir köşesine not et; bir ELF dosyasını çalıştırmak, onu okuyup belleğe atmaktan çok daha fazlası ve bölümün sonunda neden bu kadar iş çıktığını göreceksin. Bu kod, ELF dosyasındaki ayrıntıları ayrıştırmaktan ve bunları süreci belleğe yerleştirip çalıştırmak için kullanmaktan sorumludur.

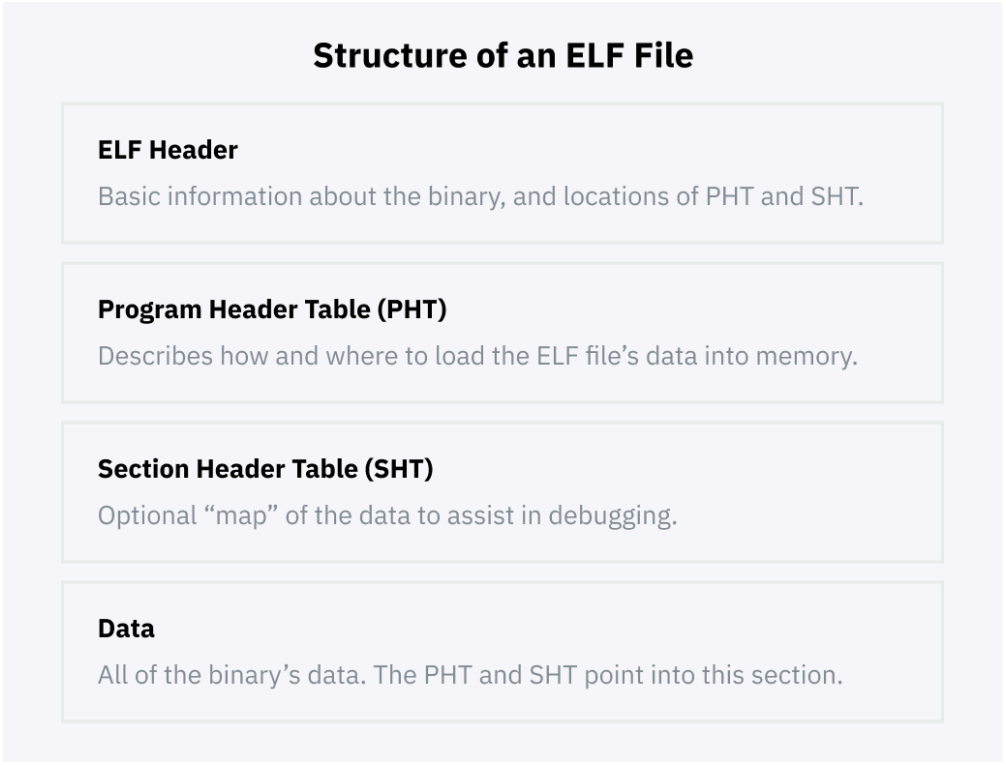
Binfmt handler'larını satır sayısına göre sıralamak için biraz command-line kung fu yaptım:

Shell oturumu

```
$ wc -l binfmt_* | sort -nr | sed 1d
    2181 binfmt_elf.c
    1658 binfmt_elf_fdpic.c
     944 binfmt_flat.c
     836 binfmt_misc.c
     158 binfmt_script.c
      64 binfmt_elf_test.c
```

Dosya Yapısı

`binfmt_elf`'in ELF dosyalarını nasıl çalıştırdığına daha derin girmeden önce, dosya formatının kendisine bakalım. ELF dosyaları genelde dört ana parçadan oluşur:



ELF Header

Her ELF dosyasının bir ELF header'ı vardır ve her zaman dosyanın en başında durur. Görevi, dosyanın geri kalanının nasıl okunacağını söylemek.

İlk dört bayt: kimlik damgası. `7f 45 4c 46` — yani `0x7F` ve ardından `E`, `L`, `F` harfleri. **[10. bölümde]** kernel'ın dosyayı bu dört bayta bakarak tanıdığını görmüştük; `file` komutunun kararı da buradan geliyor. Kendi gözünle bakabilirsin:

Shell oturumu

```
$ xxd /bin/ls | head -1
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
```

Sonraki iki bayt: sınıf ve bayt sırası. Yukarıdaki çıktıda `02` ve `01` olarak görünüyorlar. `02` dosyanın 64-bit olduğunu, `01` ise little-endian olduğunu söylüyor. Bunların bu kadar başta durması zorunluluk: header'ın geri kalanındaki her sayının kaç bayt olduğunu ve hangi sırayla okunacağını bu iki bayt belirler. Onları okumadan devamını okuyamazsın.

Dosyanın türü (`e_type`). Üç değeri sık göreceksin:

- `ET_REL` — henüz link edilmemiş ara ürün, yani bir `.o` dosyası.
- `ET_EXEC` — sabit bir adrese yüklenen klasik executable.
- `ET_DYN` — bellekte herhangi bir adrese yüklenebilen dosya.

Burada küçük bir sürpriz var: bugün dağıtımların çoğu güvenlik gerekçesiyle programları da `ET_DYN` olarak derliyor — buna **PIE** (Position Independent Executable) deniyor ve sebebini **[15. bölümde]** ASLR'yi konuşurken göreceğiz. Yani `/bin/ls` ile `libc.so.6`'nın tür alanı aynı görünür; ayrımı entry point'in ve `PT_INTERP`'in varlığı yapar.

Hedef mimari (`e_machine`). ELF dosyaları ARM, x86-64, RISC-V ve başka mimariler için makine kodu taşıyabilir; bu alan hangisi olduğunu söyler. Yanlış mimarideki bir binary'yi çalıştırmaya kalktığında aldığın `Exec format error` hatası tam olarak buradan çıkar.

Üç adres (`e_entry`, `e_phoff`, `e_shoff`). Sırasıyla: ilk çalışacak talimatın adresi, program header table'ın dosya içindeki konumu ve section header table'ın konumu. Yani header, "ilk komut nerede" ve "iki tablo dosyanın neresinde" sorularını cevaplayıp kenara çekiliyor.

Bu alanların hepsini okunabilir hâlde tek komutla görebilirsin:

Shell oturumu

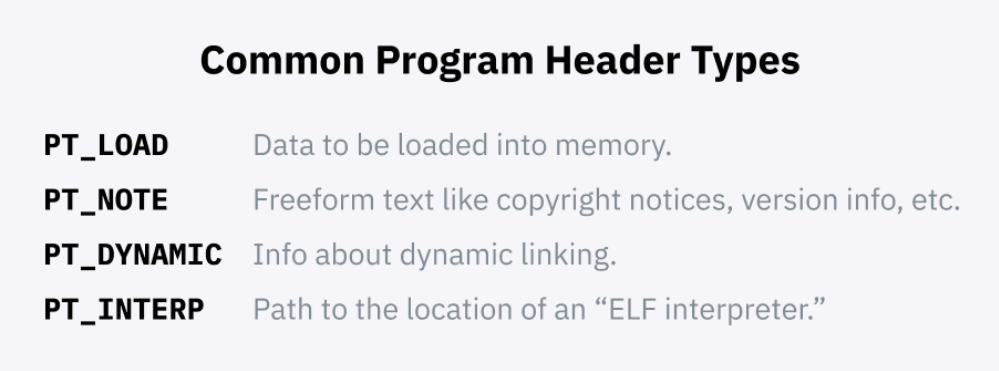
```
$ readelf -h /bin/ls
```

Program header table ile section header table'ın dosya içinde nerede olduğunu bildirir. Bu tablolar da dosyanın başka yerlerinde duran veri bloklarına işaret eder.

Program Header Table

Şimdi kernel'ın gerçekten umursadığı tabloya geldik. Program header table'ı, programın kernel'a bıraktığı bir yerleştirme talimatı gibi düşün: "şu parçamı şu adrese koy, şuna yazma izni ver, şunu hiç yükleme." Tabloyu oluşturan her girdi bu talimatlardan biridir ve her girdinin başında, hangi tür talimat olduğunu söyleyen bir tür alanı bulunur. Örneğin `PT_LOAD`, belleğe

yüklenmesi gereken veri segmentini ifade eder; `PT_NOTE` ise herhangi bir yere yüklenmesi gerekmeyen serbest biçimli bilgi notlarını temsil eder.

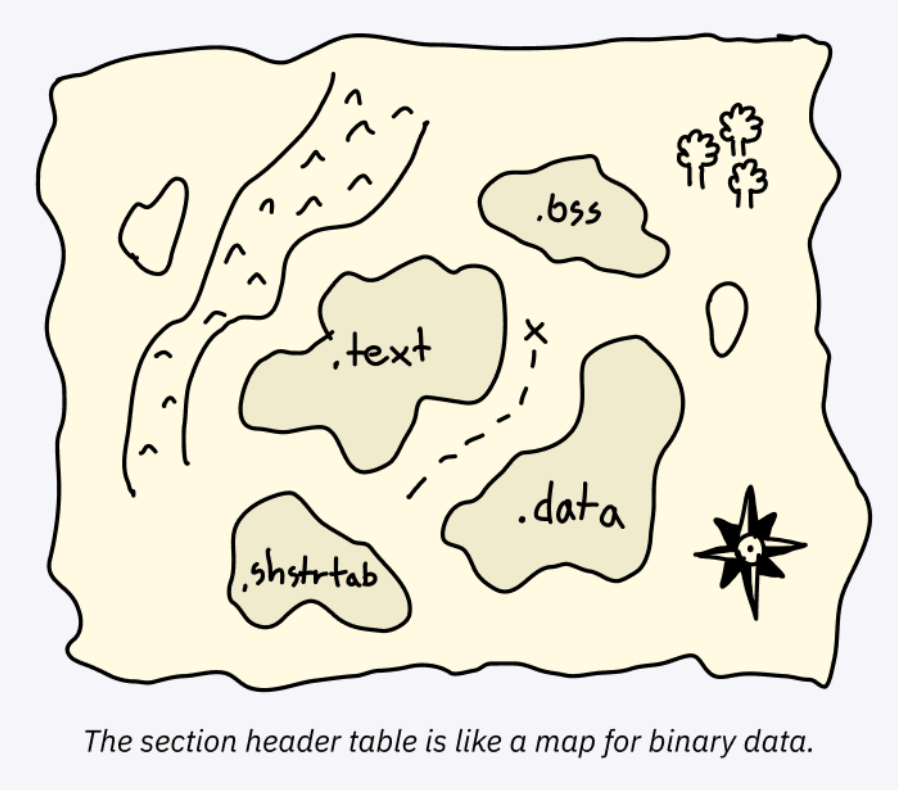


Her giriş, verisinin dosya içinde nerede olduğunu ve bazen belleğe nasıl taşınacağını belirtir:

- Verinin ELF dosyası içindeki konumunu söyler.
- Verinin belleğe hangi sanal adresten yüklenmesi gerektiğini belirtebilir. Segment belleğe yüklenmeyecekse bu alan genelde boş kalır.
- İki ayrı alan verinin boyutunu tutar: biri dosyadaki boyut, diğeri ise bellekte ayrılacak bölgenin boyutudur. Bellek boyutu dosya boyutundan büyükse, eksik kısım sıfırlarla doldurulur. Bu, çalışma zamanında sıfırlanmış bir bellek bölgesi isteyen programlar için kullanışlıdır; bu tür alanlara genelde BSS denir.
- Son olarak, bir bayrak alanı belleğe yüklendiğinde hangi işlemlere izin verileceğini belirtir: `PF_R` okunabilir, `PF_W` yazılabilir, `PF_X` ise çalıştırılabilir demektir.

Section Header Table

Section header table, *section*'lar hakkında bilgi taşıyan bir dizi girdidir. Bunu, ELF dosyasındaki verileri gösteren bir harita gibi düşünebilirsin. Böylece debugger gibi araçlar, farklı veri bölgelerinin ne işe yaradığını anlayabilir.



The section header table is like a map for binary data.

Örneğin program header table, belleğe birlikte yüklenecek geniş bir veri aralığı tanımlayabilir. Tek bir `PT_LOAD` girdisi hem kodu hem de global değişkenleri içerebilir. Programın *çalışması* için bunların ayrıca tek tek işaretlenmesi gerekmez; CPU entry point'ten başlar ve program ne zaman nereye erişmek istiyorsa oraya gider. Ama *analiz* yapmak isteyen bir debugger'ın her alanın tam olarak nerede başladığını ve bittiğini bilmesi gerekir; yoksa `hello` yazan bir metni kod sanıp çözmeye çalışabilir ve doğal olarak ortalık dağılır. İşte bu bilgi section header table'da tutulur.

Genelde ELF dosyalarında bulunsa da, section header table aslında isteğe bağlıdır. Tamamen kaldırılmış olsa bile ELF binary'leri düzgün çalışabilir. Kodlarının ne yaptığını zorlaştırmak isteyen geliştiriciler bazen section header table'ı bilerek siler ya da bozar; kötü biçimlendirilmiş ELF başlıklarıyla analizden kaçma diye bir dünya bile var.

Her section'ın bir adı, türü ve nasıl çözümlenip kullanılacağını anlatan bazı bayrakları vardır. Geleneksel isimler çoğu zaman nokta ile başlar. Sık görülen örnekler şunlardır:

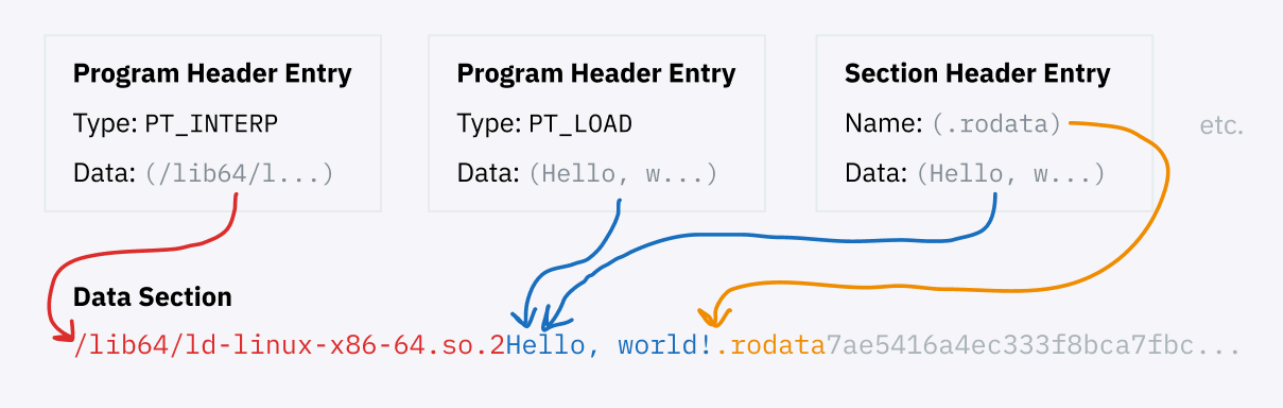
- `.text`: belleğe yüklenecek ve CPU'da yürütülecek makine kodu. Türü `SHT_PROGBITS` olur; çalıştırılabilir olduğunu belirtmek için `SHF_EXECINSTR`, belleğe yükleneceğini belirtmek için de `SHF_ALLOC` bayrakları kullanılır. (İsmi seni şaşırtmasın; bu bölüm hâlâ dümdüz

binary makine kodudur. Ad, Unix'ten kalma bir alışkanlık: bir programın makine kodunu taşıyan bölgeye tarihsel olarak *text segment* denirdi. Format değişti, isim yerinde kaldı.)

- `.data`: executable içine gömülmüş, başlatılmış veriler. Örneğin bazı metinleri taşıyan global bir değişken burada bulunabilir. Low-level kod yazıyorsan `static` verilerin gittiği yer kabaca burasıdır. Bunun da türü `SHT_PROGBITS`'tir; bayrakları genelde `SHF_ALLOC` ve `SHF_WRITE` olur.
- `.bss`: Daha önce sıfırla başlatılmış ama dosyada fiziksel olarak yer kaplamayan bellek alanlarından söz etmiştik. ELF dosyasına yığınla boş bayt koymak israf olacağı için bunun için özel bir temsil kullanılır. Debugging açısından bu alanların da görünür olması faydalıdır; bu yüzden section header table'da ayrılacak bellek boyutunu belirten bir giriş bulunur. Türü `SHT_NOBITS`, bayrakları ise `SHF_ALLOC` ve `SHF_WRITE` olur.
- `.rodata`: Yazılabilir olmaması dışında `.data` gibidir. Çok basit bir C programındaki `"Hello, world!"` dizesi burada durabilir; onu ekrana yazdıran makine kodu ise `.text` içinde olur.
- `.shstrtab`: Bu hoş bir implementasyon detayıdır. Section isimleri (`.text` ya da `.shstrtab` gibi) doğrudan section header table içine yazılmaz. Bunun yerine her girdi, dosyada isminin tutulduğu yere giden bir offset saklar. Böylece tablodaki tüm girdiler sabit boyutta kalır ve ayrıştırılmaları kolaylaşır. Bu isim dizelerinin tamamı, `SHT_STRTAB` türündeki ayrı bir `.shstrtab` section'ında tutulur.

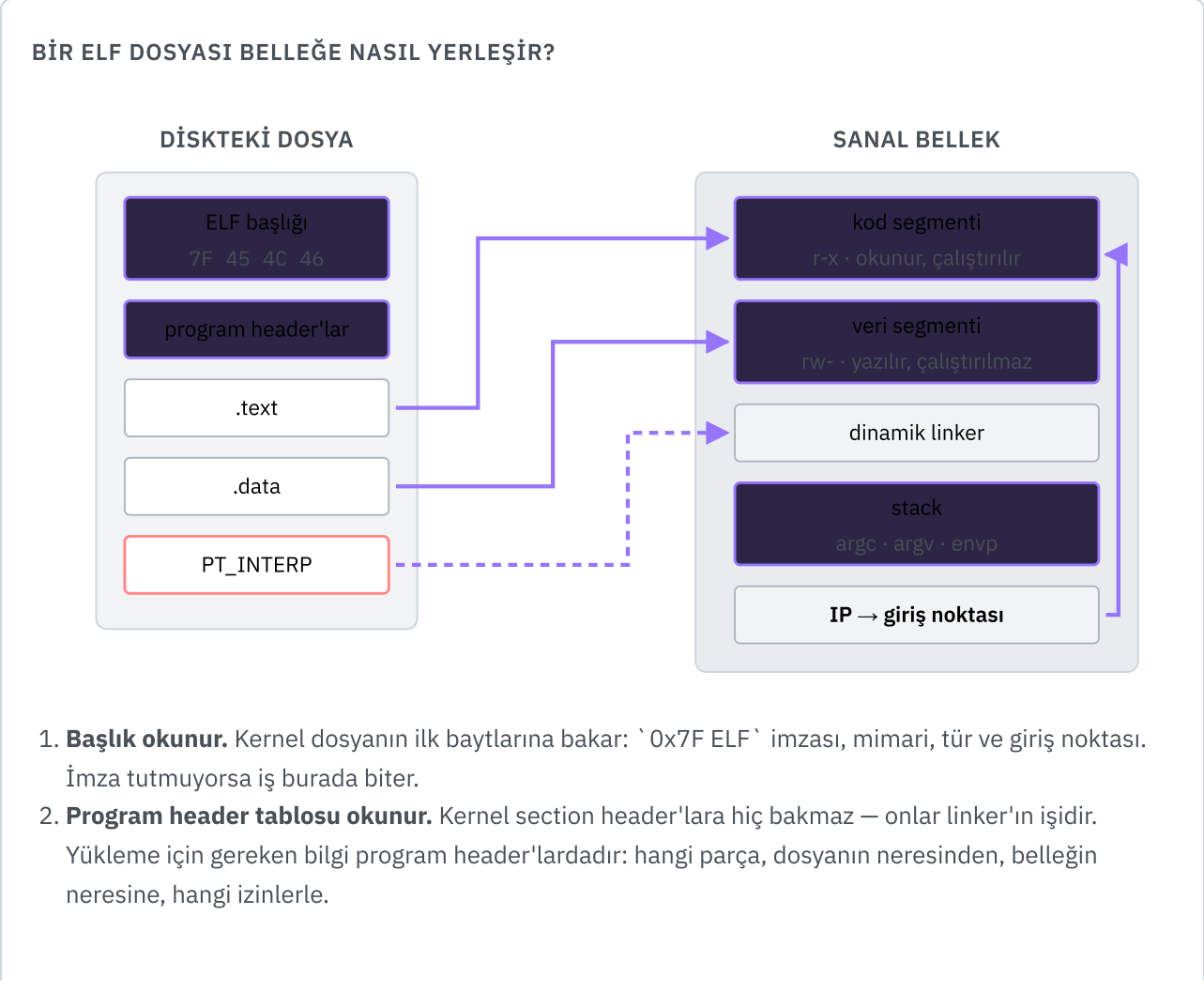
Veri

Program ve section header girdilerinin tamamı, ister belleğe yüklenecek veri olsun ister program kodunun nerede durduğunu gösteren referans olsun, ELF dosyası içindeki veri bloklarına işaret eder. Bu parçaların tamamı ELF dosyasının veri alanında bulunur.



> Kendi sisteminde dene: `readelf -h /bin/ls`, `readelf -l /bin/ls`, `readelf -d /bin/ls | grep NEEDED`, `ldd /bin/ls`

Dosyanın belleğe nasıl döküldüğünü adım adım izlemek istersen:



1. **Başlık okunur.** Kernel dosyanın ilk baytlarına bakar: `0x7F ELF` imzası, mimari, tür ve giriş noktası. İmza tutmuyorsa iş burada biter.
2. **Program header tablosu okunur.** Kernel section header'lara hiç bakmaz — onlar linker'ın işidir. Yükleme için gereken bilgi program header'lardadır: hangi parça, dosyanın neresinden, belleğin neresine, hangi izinlerle.

3. **LOAD segmentleri eşlenir.** Her `PT_LOAD` girdisi sanal belleğe eşlenir. Kod salt okunur ve çalıştırılabilir, veri yazılabilir ama çalıştırılamaz olur. Bu ayrım rastgele değil; bellek güvenliğinin temeli.
4. **Gerekiyorsa linker de yüklenir.** Dosyada `PT_INTERP` varsa program dinamik bağlıdır. Kernel o yoldaki linker'ı da belleğe yükler ve kontrolü ona verir; kütüphaneleri o bulur.
5. **Kontrol devredilir.** Stack kurulur, `argc`, `argv` ve ortam değişkenleri yazılır, register'lar temizlenir. Instruction pointer giriş noktasına ayarlanır ve user mode'a dönülür. Program artık çalışıyor.

Linking'e Kısa Bir Bakış

Şimdi yeniden `binfmt_elf` koduna dönelim: kernel, program header table'daki iki tür girdiye özellikle dikkat eder.

`PT_LOAD` girdileri, `.text` ve `.data` gibi program verilerinin bellekte nereye yerleştirileceğini belirtir. Kernel bu girdileri okur ama verinin tamamını hemen RAM'e taşımaz; yaptığı şey dosyanın ilgili parçalarını programın adres alanına iliştirmektir. Fiziksel RAM'e gelen kısım, program o adrese gerçekten dokunduğu anda gelir. Yani devasa bir binary'yi çalıştırmak, onu baştan sona belleğe kopyalamak demek değildir — nasıl olduğunu **[13. bölümde]** göreceğiz.

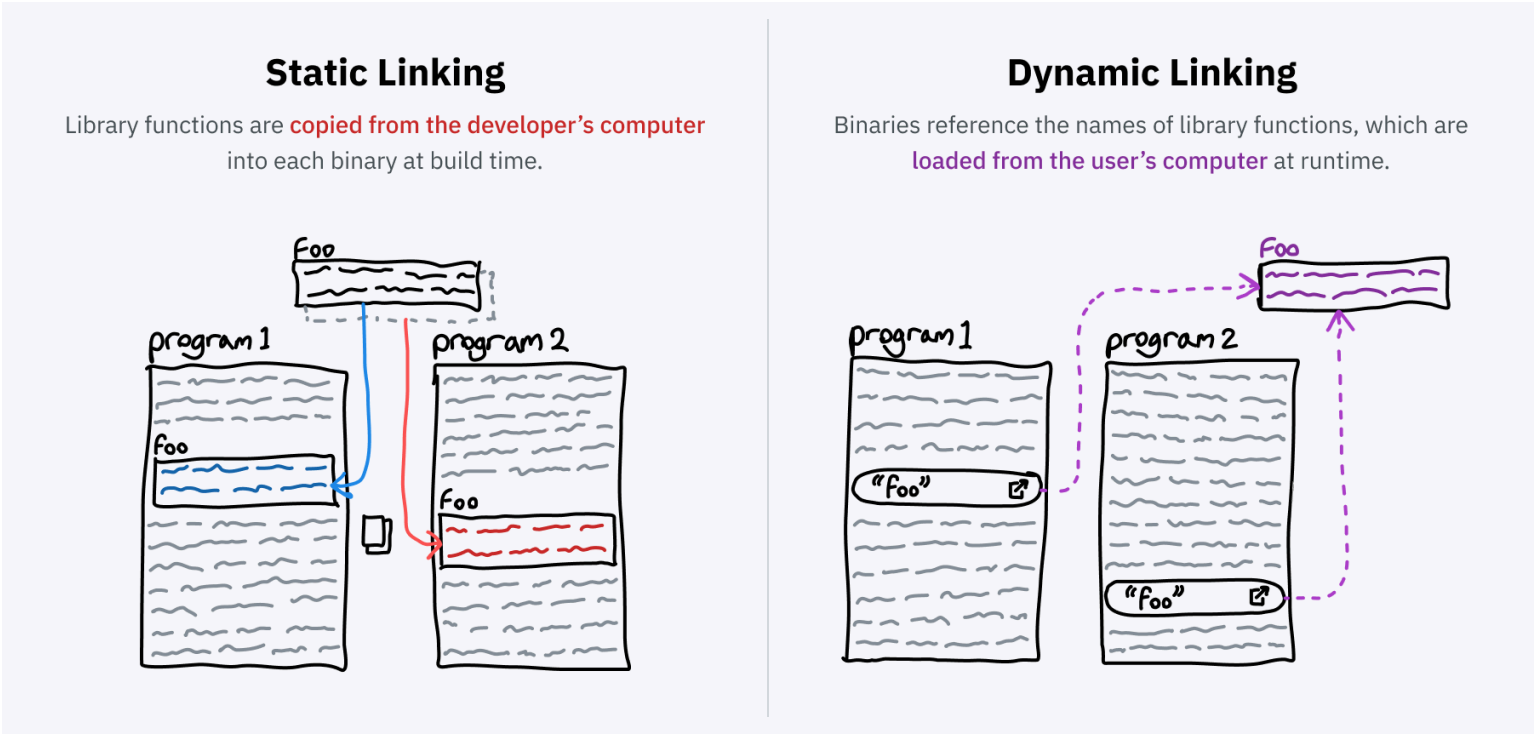
Kernel'ın önem verdiği diğer program header türü ise `PT_INTERP`'tir; bu alan, "dynamic linking runtime"ı ya da daha pratik adıyla dynamic loader'ı işaret eder.

Dynamic linking'den önce genel olarak "linking"den söz edelim. Programcılar, programlarını yeniden kullanılabilir kütüphaneler üzerine kurar; biraz önce adı geçen libc bunun klasik örneğidir. Kaynak kodunu executable binary'ye dönüştürürken linker adlı program, ihtiyaç duyulan library kodunu bulur ve bunları binary'ye ekler. Dış kodun doğrudan dağıtılan dosyaya dâhil edildiği bu yönteme *static linking* denir.

Ama bazı kütüphaneler aşırı yaygındır. Libc, sistemle konuşmanın standart yolu olduğu için neredeyse her programda vardır. Bilgisayardaki her programa ayrı bir libc kopyası gömmek hem alan israfıdır hem de bakım açısından kötüdür. Tek bir yerde güncelleme yapıp bunu her programa yansıtabilmek çok daha iyidir. Dynamic linking bu derdin çözümüdür.

Static linked bir programın, `bar` adlı bir kütüphaneden `foo` fonksiyonuna ihtiyacı varsa, binary kendi içinde `foo`'nun bir kopyasını taşır. Dynamic linked durumda ise binary yalnızca "Benim `bar` kütüphanesindeki `foo`'ya ihtiyacım var" diyen bir referans tutar. Program çalıştırıldığında

`bar` kütüphanesinin sistemde yüklü olduğu varsayılır ve `foo`'nun makine kodu gerektiğinde belleğe alınır. Sistemindeki `bar` güncellenirse, programın kendisini yeniden derlemeye gerek kalmadan bir sonraki çalıştırmada yeni kod kullanılabilir.



Gerçek Hayatta Dynamic Linking

Linux'ta `bar` gibi dynamic link edilebilen kütüphaneler genelde `.so` (Shared Object) uzantılı dosyalar hâlinde paketlenir. Bu `.so` dosyaları da programlar gibi ELF dosyalarıdır; ELF header'ın dosyanın executable mı yoksa library mi olduğunu belirten bir alan taşıdığını hatırlarsın. Shared object'lerde ayrıca `.dynsym` adlı bir tablo bulunur; dışarıya hangi sembolleri açtığını burada listeler. Dikkat et: az önce section header table'ın silinebileceğini ve programın yine de çalıştığını söylemiştik. Dynamic linking bundan etkilenmez, çünkü linker bu tabloyu section header'lardan değil, `PT_DYNAMIC` program header'ının gösterdiği adresten bulur. Kural şu: **section header'lar araçlar için, program header'lar çalışma zamanı içindir.**

Windows'ta `bar` gibi kütüphaneler `.dll` (**d**ynamic **l**ink **l**ibrary) dosyaları olarak paketlenir. macOS ise `.dylib` (**dy**namically linked **lib**rary) uzantısını kullanır. Bunlar, tıpkı macOS

uygulamaları ve Windows `.exe` dosyaları gibi ELF'den biraz farklı biçimlendirilmiştir; ama temel fikir aynıdır.

Static linking ile dynamic linking arasındaki ilginç farklardan biri şudur: Static linking'de kütüphanenin yalnızca gerçekten kullanılan bölümleri binary'ye girer ve programın parçası olur. Dynamic linking'de ise kütüphane binary'nin içine hiç girmez; program çalışırken kütüphane dosyası, o programın bellek görüntüsüne iliştirilir. İlk bakışta bu daha savurgan görünebilir ama tersi olur: aynı kütüphaneyi kullanan yirmi program, kütüphanenin salt okunur kod bölgelerinin RAM'de **tek bir kopyasını** paylaşır. Yazılabilir veriler paylaşılmaz, her programın kendine ait olur. Bunun bellekte tam olarak nasıl mümkün olduğunu **[bir sonraki bölümde]** göreceğiz.

GOT ve PLT: Tembel Bağlamanın Sırrı

Dynamic linking'in çalışma zamanında nasıl işlediğini anlamak için iki kritik yapıyı bilmek gerekir: **GOT** (Global Offset Table) ve **PLT** (Procedure Linkage Table).

PLT, her paylaşımlı kütüphane fonksiyonu için bir "trambolin" görevi görür:

1. Program `printf` çağırdığında, aslında PLT'deki bir stub'a atlar.
2. İlk çağrıda PLT stub'ı, GOT'taki girişi kontrol eder — henüz çözümlenmemiştir.
3. Dynamic linker'a (`ld.so`) geri döner. `ld.so`, `printf`'in gerçek adresini bulur ve GOT'a yazar.
4. Sonraki tüm `printf` çağrılarında PLT artık GOT'taki gerçek adrese gider — resolver maliyeti kalkar, yalnızca küçük bir dolaylı atlama maliyeti kalabilir.

Buna **lazy binding** (tembel bağlama) denir. Program başlarken tüm fonksiyon adreslerini çözmek yerine, sadece gerçekten çağrılan fonksiyonlar çözümlenir. Bu davranış her sistemde ve her binary'de açık olmak zorunda değildir; `LD_BIND_NOW`, linker bayrakları ve güvenlik sıkılaştırma ayarları binding zamanını program başlangıcına çekebilir.

GOT yalnızca fonksiyonlar için değil, global değişkenler için de çalışır. İçinde tuttuğu şey her zaman aynı: bir adres. Kod, bir global değişkene doğrudan adresiyle değil, "GOT'un şu kaçıncı satırındaki adres" diyerek erişir.

Bu dolaylılık her şeyin anahtarı. Sayesinde aynı makine kodu, kütüphane bellekte hangi adrese düşerse düşsün çalışır — çünkü kod hiçbir mutlak adres içermez, yalnızca GOT'a göre sabit offset'ler içerir. Buna **PIC** (Position Independent Code) denir ve her shared library'nin kendi GOT'u vardır.

Ama burada bir bedel var, üstelik güvenlik bedeli. GOT'un çalışma anında doldurulabilmesi için **yazılabilir** olması gerekir. Bir saldırgan bellekte istediği yere yazabiliyorsa, GOT'taki bir fonksiyon adresini kendi kodunun adresiyle değiştirerek programı ele geçirebilir; bu saldırıya **GOT overwrite** denir.

Savunması da lazy binding'den vazgeçmektir: program başlarken bütün adresleri hemen çöz, sonra GOT'u **salt okunur** yap. Bu moda **Full RELRO** denir ve `-Wl,-z,relro,-z,now` bayraklarıyla açılır; ELF tarafındaki karşılığı da bölümün başındaki şemada gördüğün `PT_GNU_RELRO` segmentidir. Modern dağıtımların çoğu bugün sistem binary'lerini bu şekilde derliyor — yani anlattığım tembel bağlama, kendi makinende çoğu program için fiilen kapalı. Farkı görmek istersen bir programı `LD_BIND_NOW=1` ile çalıştırıp `LD_DEBUG=bindings` çıktısındaki değişikliğe bak.

Kütüphaneler Nerede Aranır?

Dynamic linker bir kütüphane adı gördüğünde onu sabit bir yerde aramaz; belirli bir sıra izler ve bu sıra hem hata ayıklamada hem güvenlikte önemlidir:

1. `DT_RPATH` — binary'ye gömülü eski usul yol. Yalnızca `DT_RUNPATH` yoksa dikkate alınır.
2. `LD_LIBRARY_PATH` — ortam değişkeniyle verilen dizinler.
3. `DT_RUNPATH` — binary'ye gömülü yeni usul yol.
4. `/etc/ld.so.cache` — `ldconfig` tarafından üretilen önbellek; asıl hızlı yol budur.
5. `/lib` **ve** `/usr/lib` gibi varsayılan sistem dizinleri.

İlk üç maddedeki sıra tesadüf değil ve `RPATH` ile `RUNPATH` arasındaki tek gerçek fark burada: `RPATH` **ortam değişkenini ezer,** `RUNPATH` **ezmez.** Yani `RPATH` gömülmüş bir binary'nin kütüphane seçimini `LD_LIBRARY_PATH` ile değiştiremezsin. Hata ayıklamayı zorlaştırdığı için `RPATH` artık önerilmiyor.

Bir programın hangi kütüphaneleri, hangi dosyalardan çözdüğünü görmek için:

```
ldd /bin/ls
```

Çözümlemenin adım adım nasıl ilerlediğini görmek istersen dynamic linker'ın kendi izleme kipini açabilirsin:

```
LD_DEBUG=libs ls /tmp
```

Çıktı, her kütüphane için hangi dizinlerin denendiğini ve hangisinde bulunduğunu satır satır yazar. "Kütüphane bulunamadı" hatalarını teşhis etmenin en doğrudan yolu budur.

LD_PRELOAD ve Sembol Araya Girmesi

Arama sırasının başına, herhangi bir kütüphaneden **önce** yüklenecek bir nesne koyabilirsin:

```
LD_PRELOAD=./benim.so ./program
```

Önce yüklenen nesnedeki semboller kazanır. Yani kendi `malloc`'unu yazıp preload edersen, program ve kullandığı tüm kütüphaneler artık seninkini çağırır — kaynak koda dokunmadan, yeniden derlemeden. Buna **symbol interposition** denir.

Bu, hata ayıklama ve profilleme araçlarının temel numarasıdır: bellek sızıntısı bulucular, sahte zaman fonksiyonları ve ağ trafiği izleyicileri çoğunlukla bu yolla çalışır. Aynı güç, kötü amaçlı kullanıldığında bir programın davranışını sessizce değiştirmenin de yoludur; bu yüzden setuid bir program çalıştırıldığında `LD_PRELOAD` ve `LD_LIBRARY_PATH` yok sayılır. Aksi hâlde ayrıcalıklı bir programa istediğin kodu enjekte etmek önemsiz bir iş olurdu.

Dikkat: bunu yapan kernel değildir. Kernel yalnızca "bu process ayrıcalık kazandı" bilgisini `AT_SECURE` bayrağıyla stack'e bırakır — birazdan göreceğimiz auxiliary vector'ün içinde. O bayrağı görüp tehlikeli ortam değişkenlerini görmezden gelen `ld.so`'nun kendisidir.

DERİNLEŞME **"GLIBC_2.34 not found": sembol sürümlemenin hikâyesi**

Yeni bir dağıtımda derlediğin binary'yi eski bir sunucuya kopyaladın ve şu hatayı aldın:

```
./program: /lib/x86_64-linux-gnu/libc.so.6: version `GLIBC_2.34' not found
```

Bu hata, dinamik bağlamanın en görünür pratik sonucudur ve arkasında zarif bir mekanizma vardır: **sembol sürümleme** (symbol versioning).

Sorun şu: glibc otuz yıldır geriye dönük uyumluluk sözü veriyor. Ama bazen bir fonksiyonun davranışını değiştirmek gerekir. Adını değiştirirsen eski programlar bozulur; değiştirmezsen davranışı değişen fonksiyon eski programları bozar. Çıkmaz gibi görünür.

Çözüm, aynı isimden **birden fazla sürümü aynı anda** dışa açmaktır. `libc.so.6` içinde hem `memcpy@GLIBC_2.2.5` hem `memcpy@GLIBC_2.14` bulunur; ikisi farklı makine kodudur. Her binary, bağlandığı anda hangi sürümü istediğini kaydeder ve çalışma zamanında tam olarak onu alır.

Klasik örnek `memcpy`'nin kendisidir. glibc 2.14'te performans için kopyalama yönü değiştirildi ve bu, üst üste binen bölgelerde `memcpy` kullanan kodu bozdu — ki bu kullanım zaten tanımsız davranıştı ama yaygındı. En görünür kurbanı Adobe Flash oldu; ses bozuk çıkmaya başladı. Sürümleme sayesinde eski binary'ler eski davranışı almaya devam etti.

Buradan asimetrik bir kural doğar: **yeni glibc eski binary'yi çalıştırır, eski glibc yeni binary'yi çalıştıramaz.** Çünkü ikincisinin istediği sürüm henüz mevcut değildir. Bu yüzden dağıtım paketleri en eski desteklenen sisteme yakın bir ortamda derlenir.

Bir binary'nin ne istediğini görmek için:

```
objdump -T ./program | grep GLIBC_ | sort -u
```

Çıktıdaki en yüksek sürüm numarası, o programın çalışabileceği en eski glibc'yi belirler.

Bu kısıt, ekosistemdeki birkaç eğilimi doğrudan açıklar: container imajlarının kendi libc'lerini taşıması, Go'nun varsayılan olarak statik bağlaması, Rust'ın `musl` hedefi ve Zig'in glibc sürümünü hedef olarak seçebilmesi. Hepsi aynı soruna verilmiş farklı cevaplardır: *hangi libc ile karşılaşacağımı bilmiyorsam, hiç karşılaşmayayım.*

> **Bir yana: statik bağlamanın geri dönüşü**
>
> Dinamik bağlama disk ve bellek tasarrufu için tasarlandı, ama container çağında bu denklem değişti. Her imaj kendi kütüphanelerini taşıdığı için paylaşım avantajı büyük ölçüde kayboldu; buna karşılık "hangi glibc sürümüyle çalışır" sorunu canlı kaldı. Go varsayılan olarak statik bağlar, Rust `musl` hedefiyle statik bağlayabilir ve sonuç tek dosyalık, hiçbir sistem kütüphanesine ihtiyaç duymayan bir binary olur. Sıfırdan (`FROM scratch`) container imajları tam olarak bunu kullanır.

Yürütme

Şimdi ELF dosyalarını çalıştıran kernel'a geri dönelim: Eğer yürütülen binary dynamic linked ise, işletim sistemi onun koduna doğrudan atlayamaz; çünkü ihtiyaç duyduğu kodun bir kısmı henüz yerinde değildir. Unutma: dynamic linked programlar, ihtiyaç duydukları library fonksiyonlarına yalnızca referans taşır.

Programı gerçekten başlatabilmek için dynamic linker (ld.so) hangi kütüphanelere ihtiyaç duyulduğunu bulmalı, onları yüklemeli, isim olarak duran referansları gerçek adreslere çözmeli ve *sonra* gerçek program kodunu başlatmalıdır. Bu iş, ELF formatının ayrıntılarıyla yoğun şekilde uğraşan karmaşık bir süreçtir; bu yüzden kernel'ın ana işi değildir. Kernel bu işi kendisi üstlenmez, taşeronluk verir. Yaptığı şey basit: `PT_INTERP`'te adı yazan dosyayı — yani dynamic linker'ı — programın belleğine, programın kendisinin yanına yerleştirir. Sonra "ilk çalışacak kod" olarak programın kendi başlangıcını değil, dynamic linker'ın başlangıcını işaretler. Yani CPU user space'e döndüğünde önce linker uyanır, kütüphaneleri toplar ve işi bittiğinde sırayı asıl programa devreder. Tipik interpreter örneklerinden biri `/lib64/ld-linux-x86-64.so.2`'dir.

Kernel, ELF header'ı okuyup program header table'ı taradıktan sonra yeni programın bellek düzenini hazırlayabilir. İlk iş olarak tüm `PT_LOAD` segmentlerini sanal belleğe yerleştirir;

böylece programın statik verileri, BSS alanı ve makine kodu adreslenebilir olur. Program dynamic linked ise, kernel ayrıca ELF interpreter'ı (`PT_INTERP`) da map eder; yani dynamic loader'ın verileri, BSS alanı ve kodu da yeni process'in adres alanında görünür hâle gelir.

Sonrasında kernel, user space'e dönerken CPU'nun geri yükleyeceği instruction pointer'ı ayarlar. Executable dynamic linked ise, instruction pointer ELF interpreter'ın bellekteki giriş noktasına konur. Aksi takdirde doğrudan executable'ın entry point'ine ayarlanır.

Kernel artık syscall'dan dönmeye neredeyse hazırdır. Unutma, hâlâ `execve` içindeyiz. Program başlarken okuyabilsin diye `argc`, `argv` ve environment değişkenlerini stack'e yazar.

> Kernel stack'e yalnızca `argc`/`argv`/`envp` değil, bir de auxiliary vector yazar. Bu vektör `AT_PHDR` (program header adresi), `AT_ENTRY` (entry point), `AT_PAGESZ` (sayfa boyutu), `AT_HWCAP` (CPU özellikleri) gibi sistem bilgilerini taşır.

Bir de register temizliği var. Bir syscall'a girerken kernel, user space register'larının değerlerini kendi **kernel stack**'ine kopyalar ve dönüşte oradan geri yükler. (Bu, biraz önce `argc`/`argv` yazdığı user space stack'i değil; ikisi ayrı yerler.) Ama `execve` özel bir durum: dönülecek yer artık eski program değil. Bu yüzden kernel, geri yükleyeceği o kopyayı sıfırlar. Böylece yeni program, kendisinden önce çalışan programın register'larda ne bıraktığını asla göremez — hem temiz bir başlangıç, hem de bilgi sızıntısına karşı bir önlem.

Sonunda syscall biter ve kernel tekrar user space'e döner. Kayıtları geri yükler ve saklı instruction pointer'a atlar. Bu instruction pointer artık yeni programın ya da gerekiyorsa ELF interpreter'ın giriş noktasıdır; yani mevcut süreç fiilen değiştirilmiş olur.

> Program aslında `main()`'den başlamaz. Entry point `_start`'tır; CRT (C Runtime) başlatma kodu `__libc_start_main`'i çağırır, o da gerekli kurulumları yapıp `main()`'e geçer — bu yüzden `main()`'den önce global değişkenler ilklendirilir.

Özet

> **Peki, ne öğrendik?**
>
> - **ELF**, Linux'ta çalıştırılabilir dosyaların, paylaşılan kütüphanelerin ve nesne dosyalarının ortak formatıdır.
> - Aynı dosya iki farklı gözle okunur: **program header**'lar kernel'a "belleğe neyi nereye koyacaksın" der, **section header**'lar ise linker'a "hangi parça nerede" der.
> - **Static linking** her şeyi dosyanın içine kopyalar; **dynamic linking** çalışma anında bulur. İkisi arasındaki seçim boyut, güncellenebilirlik ve başlangıç maliyeti arasındaki bir takastır.
> - Dinamik bağlamada çağrılar **PLT** üzerinden gider ve gerçek adres **GOT**'a yazılır. Bu dolaylılık, adresin ancak ilk çağrıda çözülmesini (lazy binding) mümkün kılar.
> - Bir programın giriş noktası `main` değil `_start`'tır; aradaki çalışma zamanı kodu global değişkenleri hazırlayıp `main`'e geçer.

Peki kernel bu segmentleri belleğe yüklerken nasıl bir dünya kuruyor? Her program aynı sanal adresleri kullanıyor gibi görünüyor ama neden çakışmıyor? Sıradaki bölümde bilgisayarındaki o gizli çevirmeni tanıyacağız.

Bölüm 13: Bellek Aslında Sanal

Şimdiye kadar bellek konusunu kasten basitleştirerek anlattım; bu bölümde perdeyi kaldırıyoruz. ELF dosyaları verilerin yükleneceği belirli bellek adreslerini söylüyor. Peki farklı process'lerde aynı adresleri kullanmaya çalışan yapılar neden çakışmıyor? Neden her process'in kendine ait ayrı bir bellek ortamı varmış gibi görünüyor?

Bir de buraya nasıl geldik? `execve`'nin *mevcut process'i yeni bir programla değiştiren* bir syscall olduğunu artık biliyoruz, ama bu hâlâ birden fazla process'in nasıl ortaya çıktığını açıklamıyor. Daha da önemlisi, ilk programın nasıl başladığını hiç açıklamıyor. Diğer bütün yumurtaları yumurtlayan ilk tavuk hangi process?

> Bu soruların cevabını **[16. bölümde]** bulacaksın: `fork` ile process klonlama, Copy-on-Write ve bilgisayarın açılışında `init process`'in nasıl doğduğunu orada ele alıyoruz. Şimdi önce bellek meselesini halledelim.

Yolculuğun sonuna yaklaşıyoruz. Bellek konusunu çözdükten sonra bilgisayarının açılıştan şu anda kullandığın yazılıma kadar nasıl geldiğine dair neredeyse tam bir resmimiz olacak.

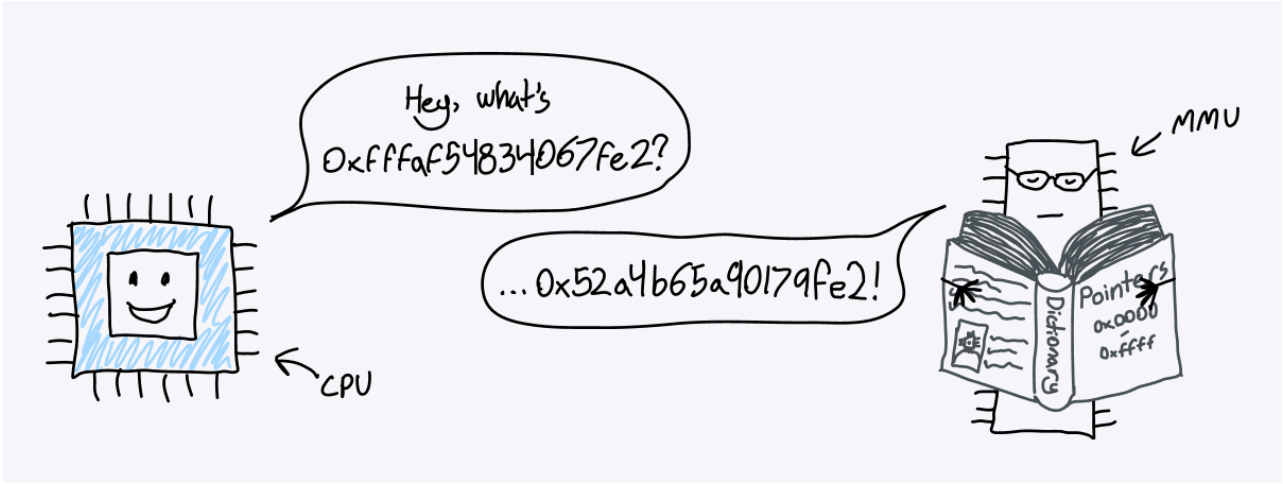
> **Bu bölümde neyi çözüyoruz?**
>
> - RAM'in neden her process'e ayrı bir dünya gibi göründüğünü anlayacağız.
> - Page table, MMU, TLB ve page fault kavramlarını aynı hikâyeye bağlayacağız.
> - Bellek izolasyonu, ASLR, demand paging ve swap'ın hangi probleme çözüm olduğunu göreceğiz.

Bellek Aslında Sanal

Gelelim belleğe. Bunu önce adres defteri gibi düşün: Program "0x400000 adresindeki veriyi oku" dediğinde, bu adres çoğu zaman RAM çipindeki gerçek konum değildir. Process'in elinde kendine ait bir adres defteri vardır; MMU bu deftere bakıp "bu sanal adres fiziksel RAM'de şu sayfaya karşılık geliyor" çevirisini yapar.

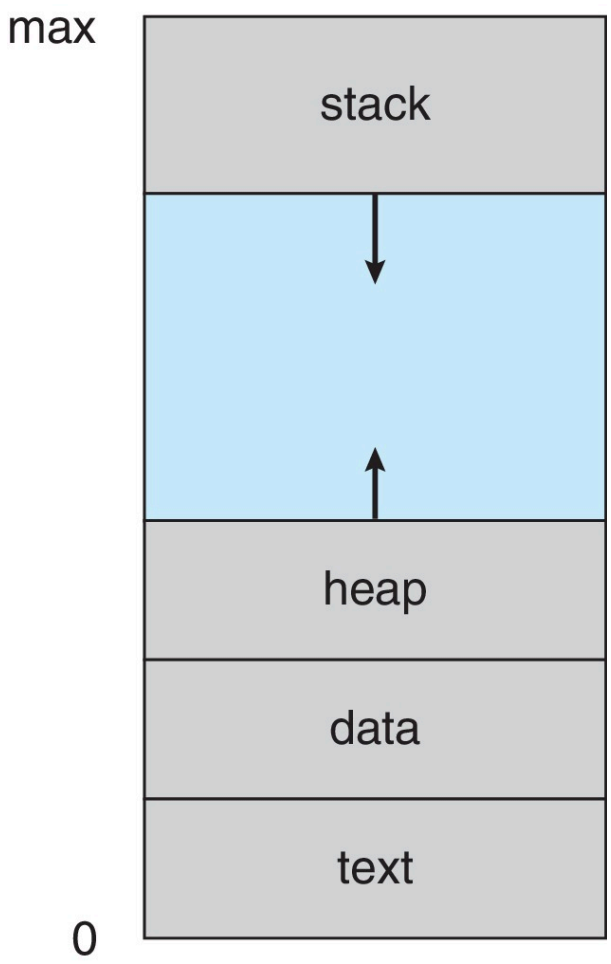
CPU bir bellek adresinden okuma ya da o adrese yazma yaptığında aslında doğrudan *physical memory*'deki (RAM'deki) o noktaya gitmez; önce *virtual memory* alanındaki bir konuma erişir.

CPU, *memory management unit* (MMU) denen bir birimle konuşur. MMU, sanal adresleri RAM'deki fiziksel adreslere çeviren bir sözlük gibi davranır. CPU'ya `0xfffaf54834067fe2` adresinden okuma yap denildiğinde, CPU önce MMU'ya gidip "bunu çevir" der. MMU eşleşen fiziksel adresin `0x53a4b64a90179fe2` olduğunu bulur ve bu sonucu CPU'ya verir. CPU da artık RAM'deki gerçek konuma erişebilir.



Bilgisayar ilk açıldığında bellek erişimleri doğrudan fiziksel RAM'e gider. Çok geçmeden işletim sistemi bu çeviri sözlüğünü kurar ve CPU'ya MMU üzerinden çalışmaya başlamasını söyler.

Her process'in sanal bellek alanı, farklı amaçlara hizmet eden bölgelere ayrılır. Aşağıdaki diyagram tipik bir process'in bellek düzenini gösteriyor: en altta **text** (program kodu) ve **data** (global/sabit değişkenler) segmentleri bulunur; yukarı doğru **heap** dinamik olarak büyür (`malloc`/`mmap` ile); en üstte ise **stack** yerel değişkenler ve fonksiyon çağrıları için kullanılır. Stack ve heap birbirine doğru büyür ama asla çakışmaz — aralarında geniş bir sanal adres boşluğu vardır. Kernel allocator, heap için `mmap` veya `sbrk` syscall'larını kullanarak sanal sayfaları fiziksel RAM'e eşler.



Kaynak: Silberschatz, Galvin, Gagne — Operating System Concepts, Ch. 3 — Processes, Slide 6.

Bu sözlüğe aslında *page table* denir; her bellek erişimini çeviren mekanizmanın adına da *paging* denir. Page table içindeki girdiler PTE (Page Table Entry — Sayfa Tablosu Girdisi) olarak adlandırılır ve her biri sanal bellek alanındaki belirli bir parçanın RAM'de nereye karşılık geldiğini söyler. Bu parçalar sabit boyuttadır ve boyut mimariye göre değişir. x86-64'ün varsayılan page boyutu 4 KiB'dir; yani her PTE, 4096 baytlık bir blok için eşleme tutar. Page = belleğin 4 KiB'lık bloğu; PTE = page table girdisi; Page Frame = fiziksel RAM'deki karşılık.

Başka bir deyişle, 4 KiB paging kullanıldığında bir adresin en düşük 12 biti MMU çevirisinden önce de sonra da aynı kalır. Bunun sebebi basit: 4096 baytlık bir page içindeki konumu göstermek için 12 bit gerekir.

x86-64 ayrıca işletim sistemlerinin 2 MiB veya 1 GiB gibi daha büyük page boyutlarını etkinleştirmesine de izin verir. Bu, adres çevirisini hızlandırabilir ama bellek parçalanmasını

ve israfı artırabilir. Page ne kadar büyükse, adresin MMU tarafından çevrilmesi gereken kısmı o kadar küçülür.

Bir Adresi Bit Bit Sökelim

Yukarıdaki şema fikri veriyor ama asıl iş sayılarda. Somut bir adres alalım ve gerçekten parçalayalım. x86-64'ün 4 seviyeli düzeninde bir sanal adresin 48 biti kullanılır ve şu şekilde bölünür:

Bit aralığı	Genişlik	Adı	Ne yapar
47-39	9 bit	PML4 indeksi	En üst tabloda hangi girdi?
38-30	9 bit	PDPT indeksi	İkinci seviyede hangi girdi?
29-21	9 bit	PD indeksi	Üçüncü seviyede hangi girdi?
20-12	9 bit	PT indeksi	Son tabloda hangi girdi?
11-0	12 bit	Offset	Bulunan sayfanın içinde kaçıncı bayt?

Toplam: 9 + 9 + 9 + 9 + 12 = 48 bit.

Şimdi asıl güzel soru: **neden 9 bit?** Rastgele seçilmiş bir sayı değil, kendi kendini doğuran bir tasarım.

9 bit ile $2^9 = 512$ farklı girdiyi adresleyebilirsin. Her page table girdisi 8 bayt tutar. $512 \times 8 =$ **4096 bayt** — yani tam olarak bir sayfa. Sonuç şu: **her page table'ın kendisi tam olarak bir sayfaya sığar.** Kernel, page table'ları da sıradan sayfalar gibi tahsis edebilir, sayfalayabilir,

hatta gerekirse diske atabilir. Sayfa boyutu tablo boyutunu, tablo boyutu da indeks genişliğini belirliyor; üçü birbirine kilitlenmiş durumda.

Offset'in 12 bit olması da aynı hesabın diğer ucu: $2^{12} = 4096$, bir sayfanın içindeki her baytı adreslemek için tam olarak yeterli. Bu yüzden bir adresin **en düşük 12 biti çeviriden hiç etkilenmez**; MMU yalnızca üstteki 36 biti çevirir, alttaki 12 bit olduğu gibi sonuca yapışır.

Yürüyüşü Adım Adım İzleyelim

Peki MMU bu beş parçayla ne yapıyor? İşleme **page table walk** deniyor ve gerçekten bir yürüyüş.

Bir kütüphanede kitap arıyormuşsun gibi düşün. Önce girişteki kat planına bakarsın, doğru katı bulursun. O katta koridor levhasına bakarsın, doğru koridoru bulursun. Koridorda raf numarasına bakarsın. Rafta kitabın sırasını bulursun. Ve nihayet kitabı açıp doğru sayfaya gidersin. Beş bakış, tek kitap.

MMU'nun yürüyüşü birebir aynı:

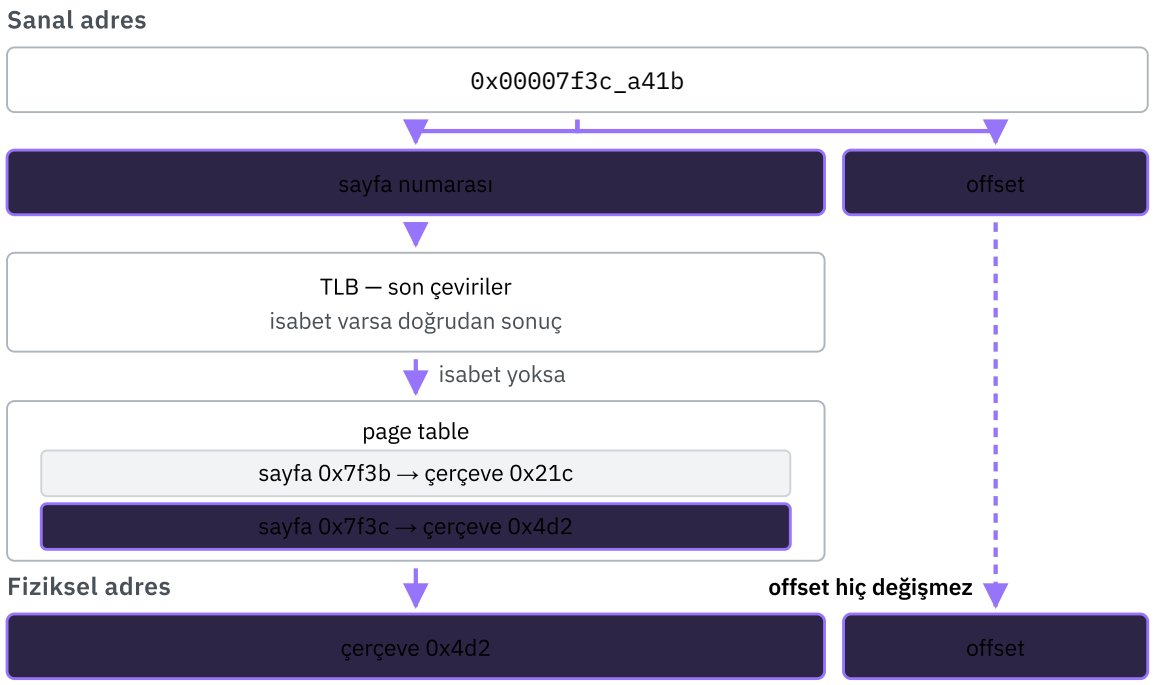
1. **CR3'ten başla.** CPU'da `CR3` adında özel bir register vardır ve içinde o anki process'in en üst tablosunun (PML4) fiziksel adresi durur. Yürüyüşün başlangıç noktası burasıdır.
2. **Bit 47-39 ile PML4'ü indeksle.** Çıkan girdi, bir sonraki tablonun (PDPT) fiziksel adresini verir.
3. **Bit 38-30 ile PDPT'yi indeksle.** Çıkan girdi PD'nin adresini verir.
4. **Bit 29-21 ile PD'yi indeksle.** Çıkan girdi PT'nin adresini verir.
5. **Bit 20-12 ile PT'yi indeksle.** Çıkan girdi nihayet aradığın **fiziksel sayfa çerçevesinin** adresini verir.
6. **Offset'i ekle.** Çerçeve adresinin sonuna, hiç değişmemiş olan 12 bitlik offset eklenir. Fiziksel adres hazır.

Buradaki maliyeti fark ettin mi? Bu tabloların hepsi RAM'de duruyor. Yani **tek bir bayt okumak için önce dört ayrı RAM erişimi yapman gerekiyor**; asıl veriyle birlikte toplam beş. Her bellek erişimini beşe katlayan bir sistem hiçbir işe yaramazdı — birazdan geleceğimiz TLB'nin var olma sebebi tam olarak bu.

> **CR3 nedir?** Tek cümleyle: process'in adres defterinin kapak sayfasının adresi. **[7. bölümde]** context switch sırasında "CPU'ya farklı bir page table kullanmasını söylemek" derken kastettiğim şey buydu — kernel yalnızca bu tek register'ı değiştirir ve process'in gördüğü bütün bellek haritası bir anda değişir.

Çevirinin tamamını tek bir akış hâlinde izlemek istersen:

SANAL ADRESTEN FİZİKSEL ADRESE

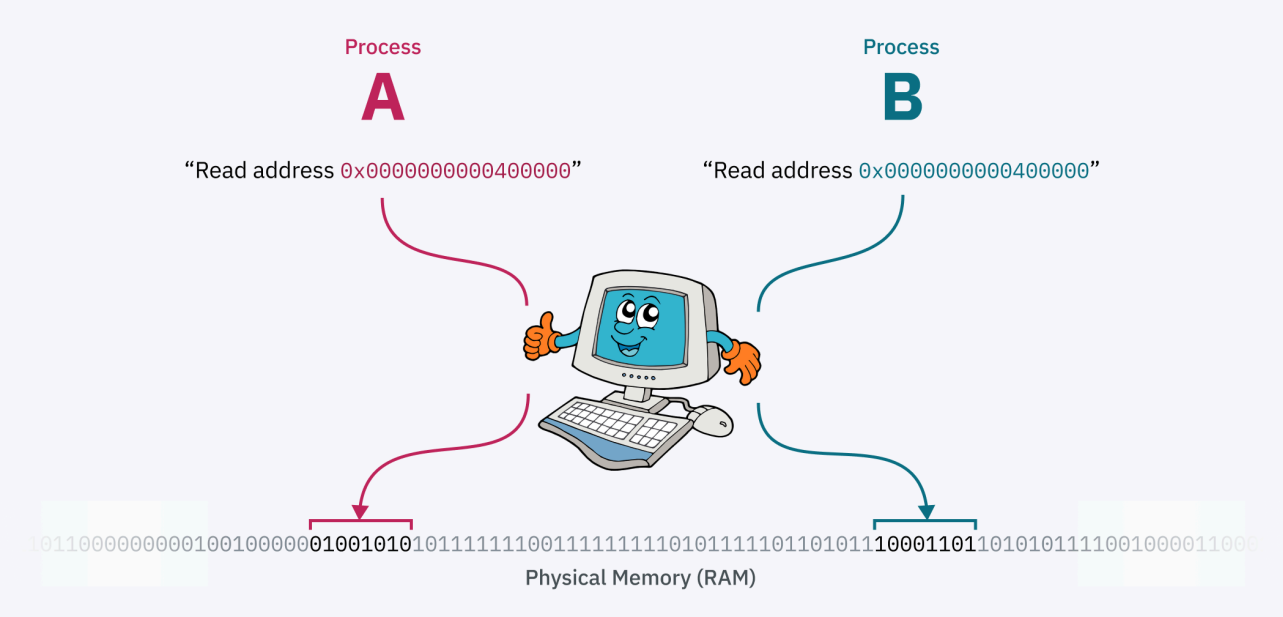


1. **Program bir adres ister.** Kod sanal bir adrese erişir. Bu adres process'e özeldir; fiziksel RAM'de nerede olduğunu program bilmez.
2. **Adres ikiye ayrılır.** Üst bitler hangi sayfaya ait olduğunu, alt 12 bit ise o sayfa içindeki konumu (offset) söyler.
3. **TLB'ye bakılır.** Bu çeviri yakın zamanda yapıldıysa TLB'de hazırdır ve iş burada biter. Isabet edilemezse page table yürünür.
4. **Page table yürünür.** MMU, sayfa numarasını kullanarak tabloyu okur ve karşılık gelen fiziksel çerçeveyi bulur.
5. **Fiziksel adres kurulur.** Bulunan çerçeve numarasının sonuna, hiç değişmemiş olan offset eklenir. Erişim artık gerçek RAM üzerinde yapılır.

Page table'ın kendisi de RAM'de tutulur. Milyonlarca girdi içerebilse bile her bir girdi yalnızca birkaç bayt civarında olduğundan, page table tek başına korkunç boyutlara ulaşmaz.

Boot sırasında paging'i etkinleştirmek için kernel önce RAM'de page table'ı kurar. Ardından page table'ın başlangıç adresini, page table base register (PTBR) denen register'a yazar. Son adımda da tüm bellek erişimlerinin MMU üzerinden çevrilmesi için paging'i etkinleştirir. x86-64'te bu yapı büyük ölçüde CR3 ve ilgili kontrol bitleri üzerinden yönetilir.

Paging'in asıl büyüsü, bilgisayar çalışırken page table'ın değiştirilebilmesidir. Her process'in izole bir bellek alanına sahip olması tam da böyle mümkün olur. İşletim sistemi context switch yaparken yaptığı kritik işlerden biri, sanal bellek alanını fiziksel bellekte başka bir yere yeniden eşlemektir. Diyelim iki process var: A process'inin kodu ve verileri `0x0000000000400000` adresinden erişiliyor olabilir; B process'i de kendi kodunu ve verisini aynı sanal adresten görüyor olabilir. Bu iki process aslında aynı adres aralığı için kavga etmez, çünkü bu sanal adresler fiziksel bellekte farklı yerlere çözülür. Kernel, process değişiminde bu eşlemeyi değiştirir.



> **Bir not: lanetli bir ELF gerçeği**
>
> Belirli koşullarda `binfmt_elf`, belleğin ilk page'ini sıfırlarla eşlemek zorunda kalır. ELF'yi destekleyen ilk sistemlerden biri olan 1988 tarihli UNIX System V Release 4.0 (SVr4) için yazılmış bazı programlar, null pointer'ın okunabilir olmasına dayanır. Ve bir şekilde bazı programlar hâlâ bu davranışı bekliyor.
>
> Görünüşe göre bunu uygulayan Linux kernel geliştiricisi pek de mutlu değildi:
>
> *"Bunun nedenini soruyorsunuz??? Çünkü SVr4, page 0'ı salt okunur şekilde eşliyor ve bazı uygulamalar buna 'bağımlı'. Bunları yeniden derleme şansımız olmadığı için SVr4 davranışını taklit ediyoruz. İç çek."*
>
> Evet. İç çek.

Paging ile Güvenlik

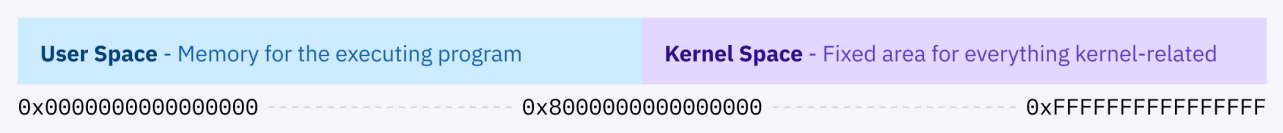
Paging'in sağladığı process izolasyonu yalnızca kod ergonomisini iyileştirmez; aynı zamanda güçlü bir güvenlik katmanı oluşturur. Process'lerin birbirinin belleğine erişememesi, kitabın başındaki önemli sorulardan birini cevaplar:

> Programlar doğrudan CPU üzerinde çalışıyorsa ve CPU doğrudan RAM'e erişebiliyorsa, neden başka process'lerin belleğine ya da aman kernel belleğine erişemiyorlar?

Bunu sanki haftalar önce sormuşuz gibi geliyor, değil mi?

Peki kernel belleği ne olacak? Öncelikle kernel'ın, çalışan tüm process'leri ve page table'ın kendisini takip etmek için kendi verilerine ihtiyacı var. Bir hardware interrupt, software interrupt ya da syscall CPU'yu kernel mode'a geçirdiği anda, kernel kodunun bu belleğe erişebiliyor olması gerekir.

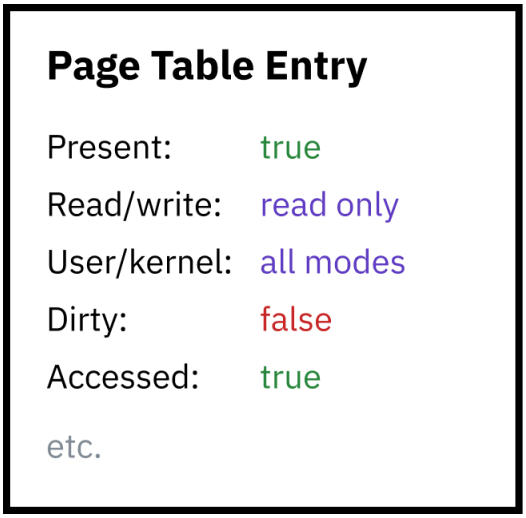
Linux'un yaklaşımı, sanal bellek alanının üst yarısını kalıcı olarak kernel'a ayırmaktır; bu yüzden Linux için *higher-half kernel* ifadesi kullanılır. Windows da benzer bir yaklaşım izler. macOS tarafı ise Mach mirası yüzünden belirgin biçimde daha karmaşık; temel fikir aynı olsa da katman sayısı fazla.



User-space process'lerin kernel belleğini okuyabilmesi ya da yazabilmesi çok kötü olurdu; bu yüzden paging ikinci bir güvenlik katmanı daha sağlar: her page için izin bayrakları tutulur. Bir bayrak, page'in yazılabilir mi yoksa yalnızca okunabilir mi olduğunu söyler. Başka bir bayrak ise bu page'e yalnızca kernel mode'dan erişilebileceğini belirtir. Kernel space'in tamamı, işte bu izinler sayesinde user-space programları için fiilen erişilemez hâle gelir. Teknik olarak bazı sistemlerde eşleme vardır, ama izin yoktur. NX (No-eXecute) biti, belirli bellek sayfalarında kod çalıştırılmasını engeller — W^X (write XOR execute) güvenlik politikasının temelidir.

DERİNLEŞME **Meltdown sonrası: kernel sayfa tablosu izolasyonu**

Eskiden performans için kernel adreslerinin user process page table'larında görünmesi yaygındı; izin bitleri user mode erişimini engelliyordu. Meltdown gibi donanım açıklarından sonra Linux tarafında KPTI/PTI gibi tekniklerle user ve kernel page table'ları daha sert ayrılabildi. Ana fikir değişmiyor: user programı kernel belleğini normal yollarla okuyamaz; sadece implementasyon ayrıntısı ve performans maliyeti değişiyor.



Bellek güvenliğinin bir diğer temel taşı da **ASLR**'dir (Address Space Layout Randomization). Stack, heap ve kütüphanelerin başlangıç adresleri her çalıştırmada rastgele kaydırılır; böylece

bir saldırganın bellekteki belirli adresleri tahmin etmesi zorlaşır.

Page table'ın kendisi de aslında kernel belleğinde durur. Timer chip bir hardware interrupt tetikleyip context switch başlattığında, CPU ayrıcalık seviyesini kernel mode'a çıkarır ve Linux kernel koduna atlar. Kernel mode'da olduğu için CPU artık korumalı kernel bellek bölgesine erişebilir. Kernel, page table'ı güncelleyip sanal belleğin alt yarısını yeni process için yeniden eşler. User mode'a geri dönüldüğünde bu erişim tekrar kapanır.

Kısacası, neredeyse her bellek erişimi MMU'dan geçer. **[2. bölümde]** kernel'ın hazırladığı o numaralı olay listesindeki adresler bile fiziksel adres değil, kernel'ın sanal adres alanındaki adreslerdir.

Burada bir nefes alalım.

> **Peki, ne öğrendik?**
>
> - Bir program **hiçbir zaman fiziksel RAM'i görmez.** Gördüğü her adres sanaldır ve MMU tarafından çevrilir.
> - Bellek sabit boyutlu **sayfalara** bölünür; RAM tarafındaki karşılıklarına **çerçeve** denir. İkisi arasındaki eşleme sayfa tablosunda durur.
> - İki process aynı sanal adresi kullanabilir, çünkü o adresler farklı çerçevelere çözülür. Process izolasyonunun temeli budur.
> - Her eşlemenin **izin bitleri** vardır: yazılabilir mi, çalıştırılabilir mi, user mode'dan erişilebilir mi.
> - Linux, sanal adres alanının üst yarısını kalıcı olarak kernel'a ayırır (**higher-half kernel**). User programı oraya eşlenmiş olsa bile izin bitleri yüzünden erişemez.
> - **ASLR**, adresleri her çalıştırmada kaydırarak tahmin edilebilirliği kırar.

Şimdiye kadar çeviriyi bir kutu gibi anlattım: sanal adres giriyor, fiziksel adres çıkıyor. Sıradaki bölümde o kutuyu açacağız — ve içeride tek bir tablo olmadığını göreceğiz.

Bölüm 14: Adres Çevirisi ve TLB

Bir önceki bölümde her sanal adresin MMU tarafından çevrildiğini gördük. Şimdi o çevirinin içine gireceğiz.

Soru masum görünüyor: bir sanal adresi fiziksel adrese çevirmek için bir tabloya bakmak yeterli değil mi? Değil — çünkü o tablo, tutulacağı makinenin belleğine sığmıyor.

Bu bölümde önce bu imkânsızlığın nasıl aşıldığını, sonra da aşmanın bedelinin nasıl ödendiğini göreceğiz.

> **Bu bölümde neyi çözüyoruz?**
>
> - Tek bir düz sayfa tablosunun neden imkânsız olduğunu ve ağaç yapısının bunu nasıl çözdüğünü göreceğiz.
> - Bir sanal adresin bit bit nasıl parçalandığını izleyeceğiz.
> - TLB'nin ne olduğunu ve context switch'in neden görünmeyen bir maliyeti olduğunu anlayacağız.
> - Page fault'un tek değil üç ayrı olay olduğunu ve `malloc`'un neden hiç başarısız olmadığını çözeceğiz.

Hierarchical Paging ve Diğer Optimizasyonlar

64-bit sistemlerde bellek adresleri 64 bit uzunluğunda olabilir; bu da teorik sanal adres alanının 16 exbibyte gibi akıl almaz bir büyüklüğe çıkabileceği anlamına gelir. Bu sayı, PiB ölçekli büyük makinelerin bile çok ötesindedir. Yani sorun "gerçek makinelerde bu kadar RAM var mı?" değil; CPU ve işletim sisteminin böylesine büyük bir adres uzayını pratik ve verimli biçimde nasıl temsil edeceğidir.

Sanal adres alanındaki her 4 KiB blok için page table'da ayrı bir girdi gerektiğini düşün. Bu durumda 4.503.599.627.370.496 page table girdisine ihtiyacın olurdu. Her girdi 8 bayt ise, sadece page table'ı saklamak için 32 pebibyte RAM gerekirdi. Evet, bu sayı gerçek makinelerdeki toplam RAM miktarından bile absürt derecede büyük.

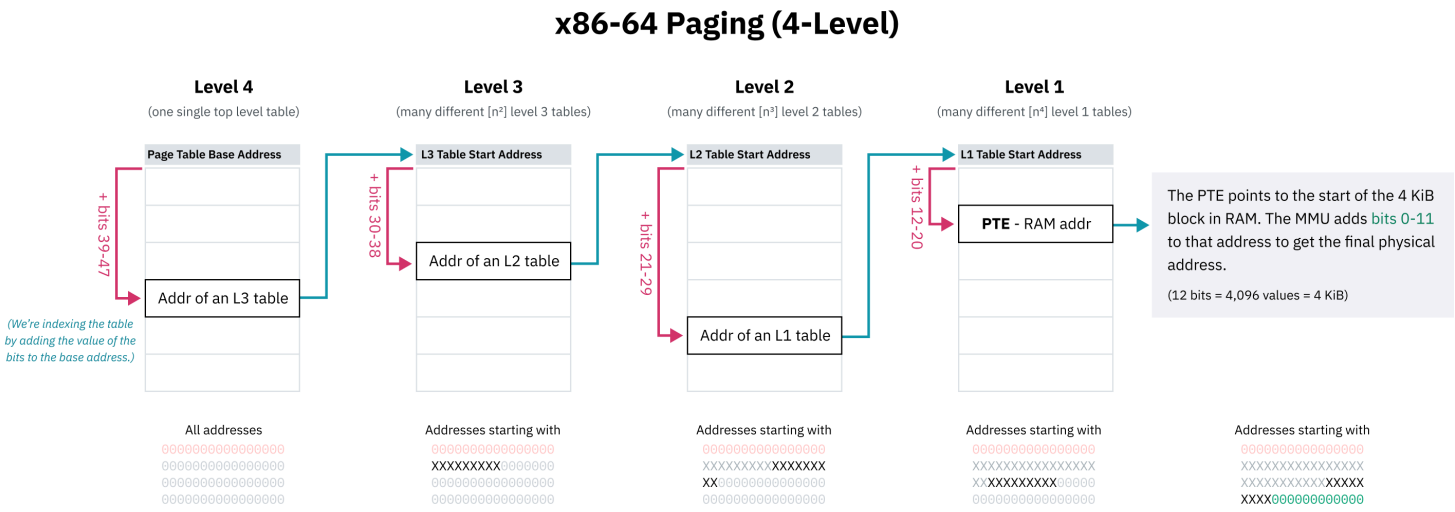
> **Bir not: neden bu kadar tuhaf birimler kullanıyorum?**
>
> Bunun nadir ve biraz çirkin göründüğünü biliyorum, ama ikili tabanlı bayt boyutlarıyla (2'nin kuvvetleri) onluk SI birimlerini ayırmanın önemli olduğunu düşünüyorum. Bir kilobyte, yani kB, 1000 bayttır. Bir kibibyte, yani KiB, 1024 bayttır. CPU'lar ve bellek adresleri bağlamında sayılar çoğu zaman 2'nin kuvvetleriyle ilerlediği için bu ayrım bana anlamlı geliyor.

Sanal bellek alanının tamamı için düz bir page table tutmak imkânsız, ya da en azından korkunç derecede verimsiz olacağı için, CPU mimarileri *hierarchical paging* kullanır. Bu modelde, tek bir dev tablo yerine farklı ayrıntı seviyelerinde birden çok page table katmanı bulunur. Üst katmanlar büyük bellek bloklarını kapsar ve daha küçük aralıklar için alt tablolara işaret eder. 4 KiB page'lere karşılık gelen tek tek girdiler ağacın yapraklarıdır.

x86-64 tarihsel olarak 4 seviyeli hierarchical paging kullandı. Bu düzende, her page table girişi adresin bir kısmıyla indekslenir. En üst anlamlı bitler bir önek gibi davranır; böylece o girdi bu bitlerle başlayan tüm adres aralığını temsil eder. Sonra sıradaki bit kümesi alt tablonun içinde bir sonraki adımı belirler ve bu böyle devam eder.

x86-64'ün 4 seviyeli paging tasarımında sanal işaretçilerin sadece 48 biti çeviri için kullanılır. Ancak üstteki 16 bit donanım tarafından göz ardı edilmez; **Canonical Addressing** kuralı gereği bu bitler 47. bitin kopyası olmak zorundadır. Eğer 47. bit 0 ise üst 16 bit tamamen 0 olmalıdır; 1 ise tamamen 1 olmalıdır. Bu, toplam 256 TiB canonical sanal adres alanı demektir; pratik anlatıda düşük yarı ve yüksek yarı kabaca 128 TiB + 128 TiB gibi düşünülebilir. (Tam 64 bit kullansan sayı 16 EiB olurdu ki, gereksiz derecede büyüktür.)

İlk 16 bitin atlanması şu anlama gelir: page table'ın ilk seviyesini indeksleyen "en yüksek anlamlı bitler" aslında 63. bitten değil, 47. bitten başlar. Bu yüzden bu bölümün başındaki higher-half kernel diyagramı aslında tam doğruyu söylemiyor; işleri anlaşılır tutmak için basitleştirilmiş bir çizim. Orta nokta, tam 64-bit uzayın değil, fiilen kullanılan çok daha dar adres uzayının orta noktasıdır.



Peki bu ağaç gerçekte ne kadar yer kaplıyor? Az önceki 32 pebibyte'ın karşı sayısını verelim.

"Merhaba dünya" yazan sıradan bir programın eşlemesi tipik olarak bir PML4 tablosu ve birkaç alt tablodan ibarettir: toplam **20-30 KiB civarı**. Otuz iki pebibyte'tan yirmi dört kibibyte'a. Aradaki oran milyarda bir mertebesinde.

Üstelik bu tasarruf her process için ayrı ayrı kazanılıyor. Düz bir tablo kullansaydın 32 PiB'lık maliyeti *her process için* ödemen gerekirdi; makinede yüz process varsa 3200 pebibyte. Hiyerarşik tabloyla aynı yüz process birkaç megabayt tutar.

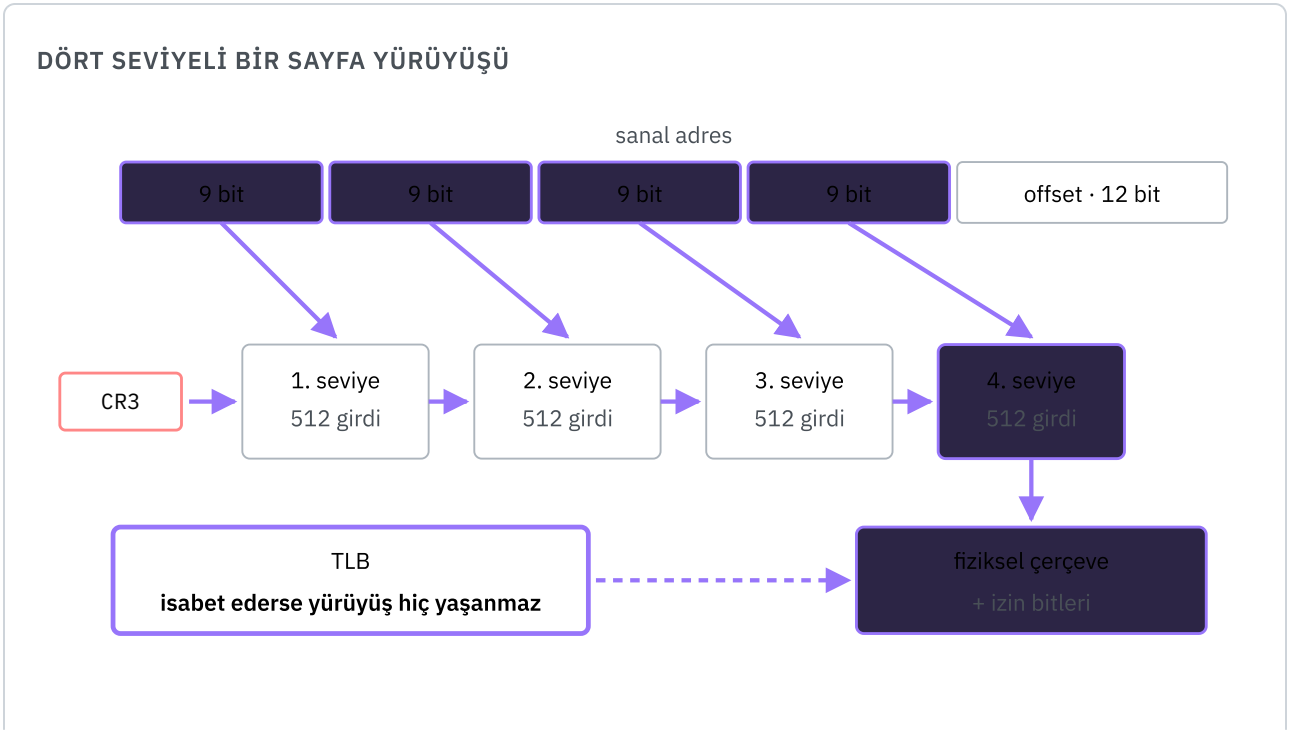
Hierarchical paging alan sorununu çözer; çünkü ağacın herhangi bir seviyesinde, bir sonraki tabloya işaret eden pointer boş (`0x0`) olabilir. Bu sayede page table'ın tüm alt ağaçları atlanabilir; yani eşlenmemiş sanal adres alanı RAM'de yer kaplamaz. CPU, ağacın üst seviyelerinde boş bir giriş gördüğünde erişimi hızlıca başarısız sayabilir. Page table girdilerinde ayrıca "adres geçerli görünse bile kullanılamaz" demeye yarayan bayraklar da vardır.

Bir başka avantaj da, büyük sanal adres bölgelerini topluca değiştirebilmektir. Kernel, bir process için bir fiziksel bellek alanını, başka bir process içinse başka bir alanı hazır tutabilir. Process değiştirirken ağacın en üst düzeyindeki birkaç pointer'ı güncellemek yeterli olur. Eğer tüm eşleme düz bir dizi şeklinde saklansaydı, kernel'ın çok daha fazla girdiyi güncellemesi gerekirdi.

Biraz önce x86-64'ün "tarihsel olarak" 4 seviyeli paging kullandığını söyledim, çünkü yeni işlemciler 5 seviyeli paging destekliyor. Mimari açıdan 5 seviye, linear/canonical adres tarafını 57 bit'e kadar genişleterek toplam 128 PiB sanal alana ulaşmayı sağlar. Ama Linux bu geniş alanı kendiliğinden dağıtmaz: eski programların 47 bitten uzun adres beklemediğini bildiği için tahsisleri varsayılan olarak yine alt 128 TiB'ın içinde tutar. Bir program daha yüksek bir adres istediğini `mmap`'e açıkça söylerse kernel sınırı açar.

> **Bir not: fiziksel adres alanı sınırları**
>
> Tıpkı işletim sistemlerinin sanal adresler için tüm 64 biti kullanmaması gibi, CPU'lar da fiziksel adresler için tam 64 biti kullanmaz. 4 seviyeli paging döneminde x86-64 CPU'lar genelde 46 bitten fazlasını kullanmıyordu; bu da fiziksel adres alanını 64 TiB ile sınırlıyordu. 5 seviyeli paging ile bu destek 52 bite çıktı ve 4 PiB fiziksel adres alanı mümkün hâle geldi.
>
> İşletim sistemi perspektifinden bakarsan, sanal adres alanının fiziksel adres alanından daha büyük olması avantajlıdır. Linus Torvalds'ın dediği gibi, en az iki kat büyük olması gerekir; on kat büyük olması ise daha da iyidir.

Bütün yürüyüşü tek bir şemada, adım adım izleyelim:



1. **Adres beş parçaya bölünür.** Sanal adresin alt 12 biti sayfa içindeki konumdur (offset). Kalan 36 bit ise dokuzarlı dört dilime ayrılır; her dilim bir tablo seviyesinde indeks olarak kullanılır.
2. **`CR3` başlangıcı verir.** MMU nereden başlayacağını bir register'dan öğrenir: `CR3`, o anki process'in en üst sayfa tablosunun fiziksel adresini tutar. Context switch'te değişen tek şey budur.
3. **Ağaç seviye seviye inilir.** Her seviyede o seviyenin 9 biti bir girdi seçer ve o girdi bir alt tablonun adresini verir. Dört seviye, dört ayrı bellek erişimi demektir.
4. **Son tablo çerçeveyi verir.** Dördüncü seviyedeki girdi artık bir tabloyu değil, RAM'deki gerçek çerçeveyi gösterir. İzin bitleri de bu girdinin içindedir: yazılabilir mi, çalıştırılabilir mi?
5. **TLB bütün bu yolu atlatır.** Bu yürüyüş her erişimde yapılsaydı tek bir bayt için beş bellek erişimi gerekirdi. TLB, sonucu saklayarak yürüyüşü tamamen ortadan kaldırır; isabet ederse çeviri neredeyse bedavadır.

TLB: Çevirinin Önbelleği

Az önce hesabı yaptık: TLB olmasaydı tek bir bayt okumak beş RAM erişimi ederdi. Böyle bir sistem kullanılamaz. Bu yüzden MMU'nun içinde **TLB** (Translation Lookaside Buffer) adı verilen küçük ama son derece hızlı bir önbellek bulunur ve son yapılan sanal → fiziksel çevirileri saklar.

Bir bellek erişiminde MMU önce buraya bakar:

- **TLB hit:** Çeviri önbellekte varsa yürüyüş hiç yapılmaz; fiziksel adres anında elde edilir.
- **TLB miss:** Çeviri yoksa MMU tam page table walk yapar, sonucu TLB'ye yazar ve yoluna devam eder.

Sayılara bakmadan bu anlatı eksik kalıyor. Tipik bir modern x86-64 çekirdeğinde durum kabaca şöyle (mimariden mimariye değişir, mertebe olarak oku):

Katman	Girdi sayısı	Maliyet
L1 dTLB (veri)	~64 girdi	~1 saat çevrimi — L1 cache erişimiyle *paralel* yürür
L2 STLB (ortak)	~1500-2000 girdi	birkaç saat çevrimi
TLB miss	—	~10-100+ saat çevrimi (yürüyüşün kendisi)

İlk satırdaki "paralel" kelimesi önemli: TLB isabeti neredeyse bedavadır, çünkü CPU veriyi cache'te ararken adresi de aynı anda çeviriyordur. Maliyet ancak ıskalayınca ortaya çıkar.

TLB Kapsama Alanı: Asıl Sınır

Şimdi bu bölümün en kullanışlı sayısına geliyoruz. TLB'nin kaç girdi tuttuğu tek başına bir şey ifade etmez; asıl soru **ne kadar bellek alanını kapsadığıdır.**

2000 girdilik bir TLB, 4 KiB'lık sayfalarla toplam 2000 × 4 KiB ≈ **8 MiB** kapsar. Yani programın aktif olarak dokunduğu bellek 8 MiB'ı aşarsa TLB yetmemeye başlar ve sürekli ıskalarsın. 100 MiB'lık bir veri yapısında rastgele gezinen bir program için bu tam bir felakettir: her erişim neredeyse garantili bir yürüyüş demektir.

Aynı TLB, 2 MiB'lık büyük sayfalarla 2000 × 2 MiB ≈ **4 GiB** kapsar. Beş yüz kat.

Huge page'lerin asıl kazancı işte burada; "daha az girdi gerekir" cümlesinin altındaki gerçek sayı bu. Ama bu kazancın bir bedeli olduğunu da unutma — az önceki Derinleşme panelinde konuştuğumuz iç parçalanma ve THP duraklamaları aynı madalyonun diğer yüzü.

Context Switch'in Gizli Maliyeti

TLB'nin bir yan etkisi daha var. Her process'in page table'ı farklı olduğuna göre, context switch sırasında TLB'deki çeviriler geçersiz hâle gelir; en basit çözüm TLB'yi tamamen boşaltmaktır. Ama bu, **[7. bölümde]** konuştuğumuz "ısınma maliyeti"nin en pahalı bileşenidir: yeni process ilk binlerce erişiminde sürekli yürüyüş yapar.

Modern CPU'lar bunu hafifletmek için TLB girdilerini bir process kimliğiyle etiketler. x86-64'te bu mekanizmaya **PCID** (Process Context ID) denir ve 12 bitlik, yani 4096 farklı kimlik tutabilir; ARM tarafındaki karşılığı **ASID**'dir. Etiket sayesinde context switch'te TLB'yi boşaltmak gerekmez; başka process'in girdileri orada durur ama eşleşmediği için kullanılmaz.

Bu mekanizma **[Meltdown'dan]** sonra çok daha kritik hâle geldi. KPTI, her syscall'da kernel ile user adres alanı arasında geçiş yaptığı için TLB baskısını fiilen ikiye katladı; PCID de tam olarak bu bedeli hafifletmek için yeniden önem kazandı.

DERİNLEŞME **Neden hâlâ 4 KiB? Sayfa boyutunun kırk yıllık takası**

4 KiB sayfa boyutu 1980'lerin başında yerleşti. O günden bugüne tipik RAM miktarı yüz binlerce kat arttı; sayfa boyutu ise aynı kaldı. Bu bir gözden kaçma değil, sürekli yeniden verilen bir karar.

Büyük sayfa neyi kazandırır? Aynı belleği daha az girdiyle adreslersin. 1 GiB'lık bir alan 4 KiB sayfalarla 262.144 PTE ister; 2 MiB sayfalarla yalnızca 512. Bunun iki sonucu var: page table'ın kendisi küçülür ve — asıl önemlisi — TLB kapsama alanı yüzlerce kat büyür. Büyük veri kümesiyle rastgele çalışan bir programda tek başına bu, ölçülebilir hızlanma verir.

Neyi kaybettirir?

- **İç parçalanma.** 4 KiB isteyen bir tahsis 2 MiB alırsa 2044 KiB boşa gider. Küçük ve çok sayıda tahsis yapan programlarda bu hızla toplanır.
- **Gecikme sıçraması.** Kernel yeni bir sayfayı programa vermeden önce sıfırlamak zorundadır (aksi hâlde başka bir process'in eski verisi sızardı). 4 KiB sıfırlamak mikrosaniyenin altındadır; 2 MiB sıfırlamak fark edilir bir duraklamadır ve bu duraklama tam da page fault anında, yani programın beklediği anda gerçekleşir.
- **Kaba taneli kararlar.** Swap, COW ve bellek geri kazanımı sayfa birimiyle çalışır. Büyük sayfada tek bir byte'a yazmak, **[16. bölümde]** göreceğimiz Copy-on-Write mantığında 2 MiB'lık bir kopyalamayı tetikler.

Linux bu takası otomatikleştirmeyi denedi: **Transparent Huge Pages (THP)**, uygun bellek bölgelerini arka planda büyük sayfalara terfi ettirir. Fikir güzel, pratikte tartışmalı. Terfi için kernel'ın bitişik 2 MiB'lık fiziksel alan bulması gerekir; bellek parçalanmışsa bunu *sıkıştırma* (compaction) yaparak açar ve bu iş bazen tahsisi isteyen thread'in içinde, senkron olarak yapılır. Sonuç, milisaniyelerce süren beklenmedik duraklamalardır.

Bu yüzden veritabanı dünyasının neredeyse tamamı — MongoDB, Redis, Oracle, birçok Java kurulumu — belgelerinde THP'yi kapatmayı önerir. Kapatmadan çok, `madvise` kipine almak bugün daha yaygın bir tavsiye: kernel kendiliğinden terfi etmesin, isteyen program açıkça istesin.

Peki 4 KiB neden hiç değişmiyor? Cevap donanım değil, uyumluluk. On yıllardır yazılan kod `4096` sayısını varsaymış durumda: bellek eşleme hizalamaları, dosya biçimleri, ELF segment hizaları. ARM64 mimarisi 16 KiB ve 64 KiB sayfaları da destekler ve Apple Silicon **16 KiB** kullanır — nitekim x86'dan taşınan bazı yazılımların Apple Silicon'da sayfa boyutu varsayımı yüzünden bozulması bunun somut sonucudur. Doğrusu, sayfa boyutunu hiç varsaymamak ve `getpagesize()` / `sysconf(_SC_PAGESIZE)` sormaktır.

Swapping ve Demand Paging

Bellek erişimi birkaç sebeple başarısız olabilir: adres geçersiz olabilir, page table'da hiç eşlenmemiş olabilir ya da girdi "mevcut değil" olarak işaretlenmiş olabilir. Bu durumların herhangi birinde MMU, sorunu kernel'ın ele alabilmesi için *page fault* adlı bir donanım exception'ı üretir.

Ama "page fault" tek bir olay değil ve aralarındaki maliyet farkı beş büyüklük mertebesine varıyor. Üçünü ayırmak gerekiyor:

Tür	Ne oldu?	Maliyet
Minor fault	Eşleme var, sayfa zaten RAM'de (ya da paylaşılan sıfır sayfası). Disk yok.	~mikrosaniyenin altı
Major fault	Sayfanın diskten okunması gerekiyor (dosya ya da swap).	SSD'de ~100 µs, diskte milisaniyeler
Invalid	Böyle bir eşleme yok, olmamalı da.	Program `SIGSEGV` ile ölür

Aradaki uçurumu bir kez daha söyleyeyim: bir major fault, minor fault'un yüz katından fazla sürer. Bir programın "aniden takılması" çoğu zaman minor fault'ların major'a dönüşmesidir — yani makinenin belleği dolmuş, sayfalar diske gitmeye başlamıştır.

Kendi makinende bakabilirsin:

Shell oturumu

```
$ ps -o min_flt,maj_flt,cmd -p $$
 MINFL  MAJFL CMD
  4821      3 -bash
```

Muhtemelen binlerce minor, bir avuç major göreceksin. Bu oran sağlıklı bir sistemin imzasıdır; major sayısı hızla artıyorsa makinen swap'e düşmüş demektir.

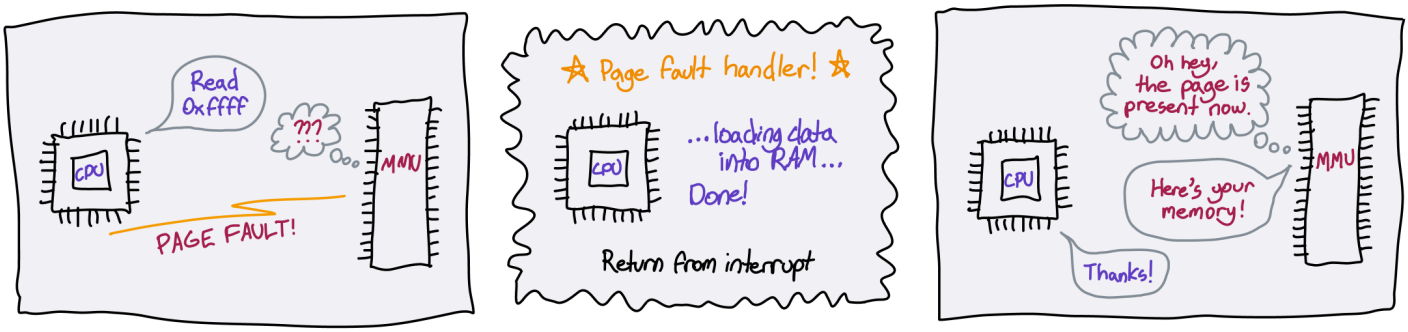
Bazı durumlarda erişim gerçekten geçersizdir ya da yasaktır. Böyle olduğunda kernel büyük ihtimalle programı segmentation fault ile sonlandırır.

Shell oturumu

```
$ ./program
Segmentation fault (core dumped)
$
```

> **Bir not: segfault ontolojisi**
>
> "Segmentation fault" farklı bağlamlarda biraz farklı şeyler ifade eder. MMU, yetkisiz bellek erişiminde donanım seviyesinde bir hata üretir; aynı ad, işletim sisteminin bu tür geçersiz erişimler yüzünden programlara gönderdiği sinyal için de kullanılır.

Başka durumlarda ise bellek erişiminin *bilerek* başarısız olmasına izin verilir; böylece işletim sistemi eksik veriyi yükleyip sonra kontrolü CPU'ya geri verebilir. Örneğin işletim sistemi bir dosyayı, içeriğini daha RAM'e taşımadan sanal belleğe eşleyebilir. Adrese gerçekten erişildiğinde page fault oluşur; kernel de ilgili veriyi diskten RAM'e yükler. Buna *demand paging* denir.



Bu mekanizma sayesinde `mmap` gibi syscall'lar tüm dosyaları tembel biçimde sanal belleğe eşleyebilir. Eğer LLaMa.cpp'yi tanıyorsan, Justine Tunney kısa süre önce yükleme mantığını mmap temelli hâle getirerek bunu ciddi biçimde optimize etti. Eğer bilmiyorsan, işlerine bir göz at; Cosmopolitan Libc ve APE epey ilginç.

> *Bu değişiklik etrafında internette hatırı sayılır bir tartışma, devamı ve bir dalga daha döndü. Tartışmanın büyük kısmı teknik değil, lisans ve atıf üzerineydi. Buradaki teknik ders yerinde duruyor: mmap temelli yükleme, modeli RAM'e kopyalamadan çalıştırabiliyor.*

Bir programı ve kütüphanelerini çalıştırdığında kernel aslında her şeyi baştan RAM'e kopyalamaz. Çoğu durumda yaptığı şey, dosya için bir `mmap` oluşturmak olur; CPU kodu gerçekten yürütmeye çalıştığında page fault oluşur ve kernel o page'i fiziksel bellekle doldurur.

Aynı numaranın bir başka kullanımını **[16. bölümde]** göreceğiz ve mekanizması birebir aynı: **Copy-on-Write da aslında bir page fault türüdür.** `fork` sonrası kernel, paylaşılan sayfaları okunabilir ama yazılamaz işaretler. Process'lerden biri o sayfaya yazmaya kalktığında MMU bir fault üretir, kernel devreye girer, sayfayı kopyalar ve yeni kopyaya yazma izni verir. Program hiçbir şeyin farkında değildir; kendi belleğine yazdığını sanır.

> **Paylaşılan sıfır sayfası.** Aynı fikrin bir başka uygulaması daha var. `malloc` ile yeni bellek istediğinde kernel sana gerçek RAM vermez; bütün o sayfaları, sistemde tek bir kopyası bulunan salt okunur bir **sıfır sayfasına** eşler. Okuduğunda hep sıfır görürsün ve tek bir bayt bile RAM harcanmamıştır. Ancak ilk *yazma* denemesinde fault üretilir ve kernel sana gerçek bir sayfa ayırır. Yani gigabaytlarca sıfır, RAM'de tek bir sayfa yer kaplar.

`malloc` Neden Hiç Başarısız Olmuyor?

Buradan çok tanıdık bir tuhaflığa varıyoruz. 16 GiB RAM'i olan bir makinede 100 GiB'lık `malloc` çağrısı yaparsan ne olur?

Başarılı döner. Anında, üstelik.

Sebebi artık tahmin edebilirsin: kernel sana RAM vermedi, yalnızca adres alanında 100 GiB'lık bir aralığı "bu senin" diye işaretledi. Fiziksel sayfa, ancak o adreslere gerçekten dokunduğunda ayrılacak. Kernel'ın verdiğinden fazlasını vaat etmesine **overcommit** deniyor ve varsayılan davranış budur.

Bu, üzerinde durup düşünmeye değer bir tercih. Programların çoğu istediği belleğin tamamını kullanmaz — büyük tamponlar ayırıp bir kısmına hiç dokunmamak son derece yaygındır. Kernel bunu bildiği için cömert davranır ve karşılığında makineden fiilen daha fazla iş çıkarır.

Bedeli ise sıra dışı bir hata anı doğurmasıdır. Vaatler gerçekten toplandığında, yani herkes belleğine aynı anda dokunmaya kalktığında, kernel'ın verecek sayfası kalmaz. `malloc` çoktan başarılı dönmüştü; hata bildirebileceği bir yer yok. Bu durumda devreye **OOM killer** girer: kernel çalışan process'leri puanlar ve en yüksek puanlıyı öldürür. Programın kendi hatası olmadan, hiçbir uyarı almadan sonlanmasının sebebi budur.

`dmesg` çıktısında `Out of memory: Killed process ...` satırını gördüysen, olan tam olarak buydu.

Demand paging, muhtemelen "swap" ya da "paging" adıyla duyduğun başka bir tekniği de mümkün kılar. İşletim sistemi bellek page'lerini diske yazıp sonra fiziksel RAM'den çıkarabilir; ama bunların sanal adreslerini page table'da tutmaya devam eder. Sonra aynı veri tekrar istenirse, diskten geri yükleyip erişimi yeniden mümkün kılar. Gerekirse diskten gelen sayfaya yer açmak için RAM'deki başka bir page'i swap out etmek zorunda kalabilir. Disk I/O yavaş olduğu için işletim sistemleri iyi page replacement algoritmaları ile bunu mümkün olduğunca az yapmaya çalışır.

Düşünmesi eğlenceli bir hack de şudur: page table içindeki fiziksel adres alanlarını, dosyaların diskteki konumlarını saklamak için kullanabilirsin. MMU zaten "present" biti kapalı bir girdi

gördüğünde page fault üreteceği için, o alanların gerçek RAM adresi olmaması bazı tasarımlarda sorun yaratmaz. Her yerde pratik değildir ama akılda tutması keyiflidir.

Bu bölüm uzundu; toparlayalım.

- **Program hiçbir zaman fiziksel RAM'i görmez.** Gördüğü her adres sanaldır ve MMU tarafından çevrilir. İki process aynı adresi kullanabilir çünkü o adresler farklı yerlere çözülür.
- **Sanal adres beş parçaya bölünür:** dört tablo indeksi (her biri 9 bit) ve bir offset (12 bit). 9 bit tesadüf değil — 512 girdi × 8 bayt tam bir sayfa eder, yani her page table'ın kendisi bir sayfaya sığar.
- **Çeviri bir yürüyüştür ve pahalıdır.** `CR3`'ten başlayıp dört tablo dolaşmak, tek bir bayt için dört ek RAM erişimi demektir.
- **TLB bu yürüyüşü ortadan kaldırır** ama kapsama alanı sınırlıdır: 4 KiB sayfalarla birkaç megabayt. Büyük sayfaların asıl kazancı bu alanı yüzlerce kat büyütmesidir.
- **Page fault üç ayrı olaydır:** minor (ucuz), major (yüz kat pahalı, diske gider), invalid (segfault).
- **Kernel tembeldir ve bu bir erdemdir.** Demand paging, copy-on-write, paylaşılan sıfır sayfası ve overcommit — hepsi aynı ilkenin uygulamaları: *işi gerçekten gerekene kadar erteler.*

Sanal bellek, bir process'in başka bir process'in belleğine uzanmasını engelliyor. Peki ya tehdit dışarıdan değil, programın *kendi içinden* gelirse? Sıradaki bölümde bir programın kendi belleğini nasıl bozabildiğini ve buna karşı yıllar içinde kurulmuş savunma katmanlarını göreceğiz.

Bölüm 15: Bellek Güvenliği ve Sertleştirme

Bir önceki bölümde sanal belleğin process'leri birbirinden nasıl ayırdığını gördük: her process kendi adres alanında yaşıyor, komşusunun belleğini okuyamıyor. Bu güçlü bir sınır, ama tek bir process'in **kendi içinde** olan biteni hiç denetlemiyor. Program yanlışlıkla kendi stack'ini ezdiğinde MMU bir şey söylemez; adres o process'e aittir, erişim meşrudur.

İşte saldırganların onlarca yıldır kullandığı boşluk tam burası. Şimdi seninle bu boşluğun etrafına örülmüş savunma katmanlarını tek tek sökeceğiz.

Baştan söyleyeyim: bu katmanların hiçbiri açığı kapatmıyor. Hepsi yalnızca sömürmeyi pahalılaştırıyor. Her birinde neyi durdurduğunu değil, **neyi durdurmadığını** da göstereceğim — asıl öğretici olan kısım orası.

> **Bu bölümde neyi çözüyoruz?**
>
> - Bir buffer overflow'un program akışını nasıl ele geçirdiğini adım adım göreceğiz.
> - NX, stack canary, ASLR, PIE ve RELRO korumalarının her birinin hangi saldırı adımını kırdığını anlayacağız.
> - Kendi sistemimizdeki bir binary'nin hangi korumalarla derlendiğini `checksec` ile okuyacağız.

Sorunun Kökeni: Sınır Kontrolü Olmayan Bellek

C ve C++ gibi diller bellek erişiminde sınır kontrolü yapmaz. `char tampon[64]` diye ayırdığın alana 100 byte yazarsan derleyici seni durdurmaz, CPU da durdurmaz. Fazlalık, stack'te o tamponun hemen ardındaki ne varsa onun üzerine yazılır.

Peki tamponun ardında ne var? **[2. bölümde]** gördüğümüz gibi bir fonksiyon çağrıldığında CPU, geri dönüş adresini stack'e iter. Yerel değişkenler de aynı stack çerçevesindedir. Tipik bir yerleşimde tampon aşağı, dönüş adresi yukarıdadır ve tampon taştığında yazma yönü tam olarak o adrese doğrudur.

savunmasiz.c

```
void selamla(const char *ad) {
  char tampon[64];
  strcpy(tampon, ad);           // ad 64 byte'tan uzunsa taşar
  printf("Merhaba %s\n", tampon);
}
```

`strcpy` kopyalamayı yalnızca kaynak dizede bir null byte gördüğünde bırakır. Hedefin ne kadar büyük olduğunu ise hiç bilmez — kimse ona söylememiştir, söyleyecek bir yer de yoktur. Saldırgan 64 byte'lık tamponu doldurup üzerine kendi seçtiği bir adresi yazarsa, fonksiyon `ret` yaptığında CPU o adrese atlar. Program akışı artık saldırganındır.

Gönderilen Byte'lar Tam Olarak Neye Benziyor?

"Dönüş adresini ez" cümlesi soyut kalıyor; somut yerleşimi görmeden mekanizma oturmuyor. 64 baytlık tamponu olan bir fonksiyonun stack çerçevesi, düşük adresten yükseğe doğru kabaca şöyle dizilir:

```
dusuk adres
+------------------------+
| tampon[0..63]          |  64 bayt
+------------------------+
| stack canary           |   8 bayt  (koruma aciksa)
+------------------------+
| kaydedilmis rbp        |   8 bayt
+------------------------+
| donus adresi           |   8 bayt  <- hedef bu
+------------------------+
yuksek adres
```

Saldırganın göndereceği dizi, bu haritanın üstüne birebir oturur: önce 64 bayt dolgu (içerik önemsiz, sadece yer doldurur), sonra kanarya alanı için 8 bayt, sonra `rbp` için 8 bayt ve nihayet gerçekten önemsediği son 8 bayt — atlamak istediği adres.

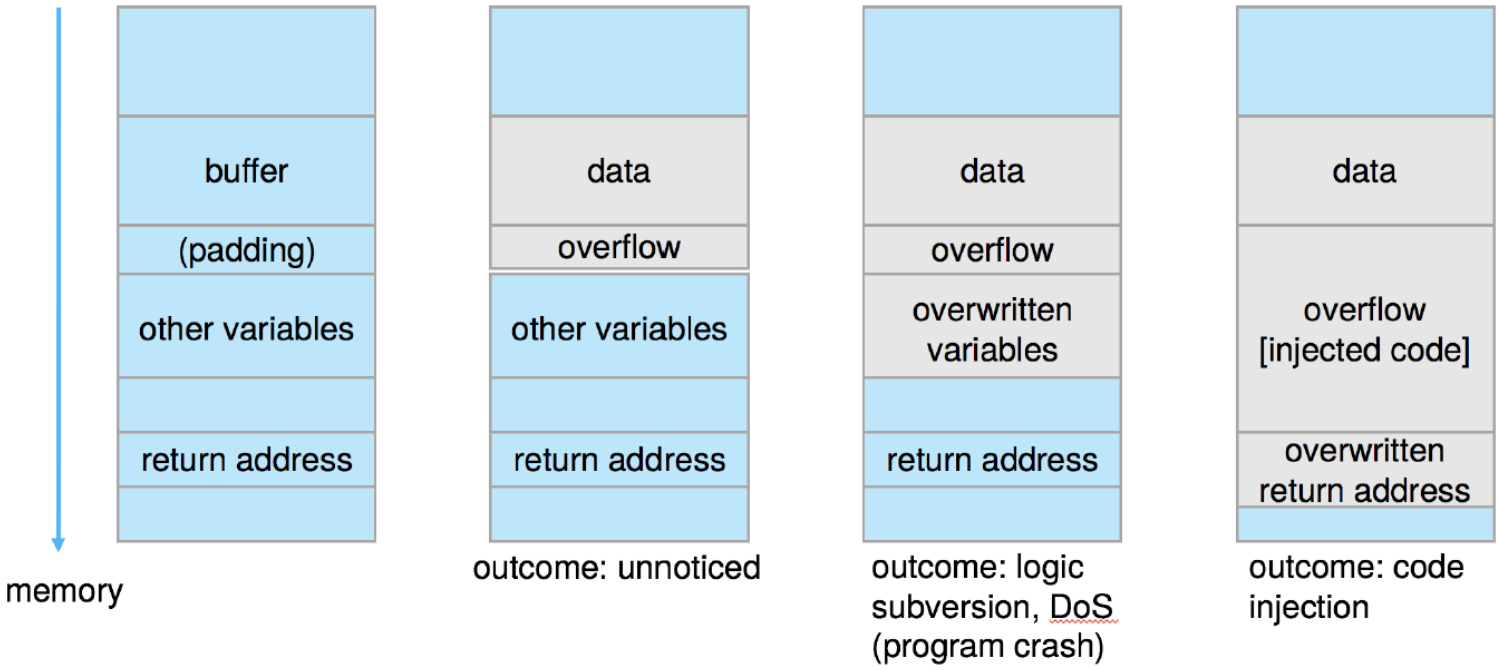
Yani "taşma" dediğimiz şey aslında dikkatlice ölçülmüş bir yerleştirme. Saldırganın bilmesi gereken tek sayı, tamponun başlangıcı ile dönüş adresi arasındaki mesafe. Bunu bulmak için

genelde artan uzunlukta girdiler denenir ve programın hangi uzunlukta çöktüğüne bakılır.

Bir de **NOP sled** numarası var. Saldırgan atlayacağı adresi tam tutturamayabilir; birkaç bayt sapma ihtimali her zaman vardır. Bu yüzden kendi kodunun önüne yüzlerce `nop` talimatı koyar — `nop` hiçbir şey yapmayan tek baytlık bir talimattır. Bu bloğun herhangi bir yerine düşerse CPU boş boş ilerleyip asıl koda varır. Yani tek bir adresi tutturmak yerine geniş bir hedef tahtası oluşturur.

Bu son cümleyi aklında tut: **saldırının tamamı bir adres tahmin etmeye dayanıyor.** Birazdan geleceğimiz ASLR'nin varlık sebebi tam olarak bu tahmini imkânsız kılmak.

Bir taşmanın nereye varacağı, ne kadar taştığına bağlıdır. Aşağıdaki dört sahne aynı tamponun farklı taşma miktarlarını gösteriyor:



Kaynak: Silberschatz, Galvin, Gagne — Operating System Concepts, 10. baskı, Ch. 16 — Security.

Soldan sağa giderek ciddiyet artıyor. İlkinde taşma yalnızca boş hizalama alanına (padding) düşüyor ve **hiç fark edilmiyor** — program normal çalışmaya devam ediyor, hata bir gün başka bir bağlamda patlamak üzere orada duruyor. İkincisinde komşu değişkenler eziliyor: program

yanlış değerlerle çalışıyor ya da çöküyor. Sonuncusunda ise taşma **dönüş adresine** ulaşıyor ve iş tamamen değişiyor; fonksiyon döndüğünde CPU artık saldırganın seçtiği yere atlıyor.

Bu son sütun, klasik saldırının üç adımını da içinde barındırıyor: **çalıştırılacak kodu belleğe yerleştir** (enjekte edilen kod), **o kodun adresini öğren**, **dönüş adresini o adresle değiştir**. Bundan sonra göreceğimiz korumaların her biri bu üç adımdan birini kırmayı hedefler.

NX: Veri Sayfaları Yürütülemez

İlk savunma en doğrudan olanı: saldırganın yazdığı byte'ların hiç çalıştırılamamasını sağlamak.

[13. bölümde] sayfa tablosu girdilerinin izin bitleri taşıdığını gördük. Bu bitlerden biri sayfanın yürütülebilir olup olmadığını söyler. AMD bu bite **NX** (No-eXecute), Intel **XD** (eXecute Disable) der; Politikanın genel adı ise **W^X**'tir (write XOR execute): bir sayfa ya yazılabilir ya yürütülebilir olsun, ikisi birden olmasın. Terim OpenBSD'den gelir; 2003'te bu politikayı sistem genelinde uygulayan ilk işletim sistemi oydu ve diğerleri sonradan izledi.

NX etkinken stack ve heap yazılabilir ama yürütülemez sayfalardır. Saldırgan stack'e makine kodu yazmayı başarsa bile CPU o adrese atladığı anda page fault üretir ve process sonlanır.

Bir process'in bellek haritasında bu ayrımı doğrudan görebilirsin:

```
cat /proc/self/maps | head -5
```

Çıktıdaki dört karakterlik izin alanı `r-xp` ise okunabilir ve yürütülebilir (kod), `rw-p` ise okunabilir ve yazılabilir (veri). `rwxp` kombinasyonunu modern bir binary'de neredeyse hiç görmezsin.

> **NX neyi çözmez?** Saldırgan kendi kodunu çalıştıramaz, ama programda **zaten var olan** kod parçalarını zincirleyebilir. NX gerekli bir katmandır, yeterli değildir.

ROP: Kod Yazmadan Kod Çalıştırmak

NX'in neden yetmediğini anlamak, bölümün geri kalanının anahtarı — çünkü ASLR, PIE ve RELRO'nun tamamı bu boşluğu kapatmak için var.

Numara `ret` talimatının ne kadar basit olduğunu fark etmekle başlıyor. `ret` tek bir iş yapar: **stack'in tepesindeki değeri al ve oraya atla.** Yani stack'i kontrol eden, programın akışını kontrol eder. Saldırganın taşmayla ele geçirdiği şey de tam olarak stack'ti.

Şimdi ikinci gözlem: bir programda ve bağlı kütüphanelerinde milyonlarca bayt makine kodu var. Bu kodun içinde, birkaç faydalı talimat yapıp hemen ardından `ret` ile biten minik parçalar bulmak şaşırtıcı derecede kolay. Bu parçalara **gadget** deniyor. `ret` ile bitmeleri şart, çünkü zinciri ilerleten motor o.

Zincir şöyle kuruluyor. Saldırgan stack'e üst üste gadget adresleri diziyor:

```
donus adresi   -> gadget 1 adresi   ; pop rdi ; ret
                  "/bin/sh" adresi   ; gadget 1'in rdi'ye cekecegi deger
                  gadget 2 adresi    ; pop rsi ; ret
                  0                  ; gadget 2'nin rsi'ye cekecegi deger
                  system() adresi    ; artik argumanlar hazir
```

Fonksiyon `ret` yaptığında birinci gadget'a atlar. O gadget `pop rdi` yapıp `ret` eder — ve `ret`, stack'teki bir sonraki adrese, yani ikinci gadget'a atlar. Zincir kendiliğinden yürür. Stack artık veri değil, **bir program**: her satırı bir sonraki adımı söyleyen bir talimat listesi.

Kritik nokta şu: bu zincirin hiçbir aşamasında NX ihlal edilmiyor. Çalıştırılan her bayt, zaten yürütülebilir olarak işaretlenmiş sayfalarda duruyor — programın kendi kodu. Saldırgan tek bir yeni talimat yazmadı; var olanları yeni bir sırayla dizdi.

Bu tekniğin en basit hâline **ret2libc** denir: gadget zincirlemekle uğraşmak yerine doğrudan `system()` fonksiyonuna atlanır. ROP ise aynı fikrin genelleştirilmiş hâli ve pratikte neredeyse her şeyi yapabilecek kadar güçlü.

Buradan çıkan ders net: **kodu yazılamaz yapmak yetmiyor, saldırganın kodun nerede olduğunu bilmesini de engellemek gerekiyor.** Sıradaki koruma tam olarak bunu deniyor.

Stack Canary: Kanaryayı Madene İndir

İkinci savunma taşmayı, zarar verdikten sonra ama iş işten geçmeden önce fark etmeye çalışır.

Derleyici, fonksiyon girişinde yerel değişkenlerle dönüş adresi arasına rastgele bir değer yerleştirir. Fonksiyon dönmeden hemen önce bu değeri kontrol eder: değişmişse aradaki bölge ezilmiş demektir ve program `__stack_chk_fail` ile derhal sonlandırılır. Madenciler zehirli gazı erken fark etmek için kafeste kanarya taşırdı; adı buradan gelir.

Peki bu rastgele değer nereden geliyor ve saldırgan onu okuyamaz mı? Değer program başlarken bir kez üretilir ve process'in ömrü boyunca sabit kalır. Stack'ten ayrı, programın kendi verilerinin arasında özel bir köşede saklanır — kanarya karşılaştırması her fonksiyon dönüşünde yapıldığı için bu erişimin çok ucuz olması gerekir. Saldırgan taşma yoluyla oraya **yazabilir** ama **okuyamaz**; okuyamadığı için de doğru değeri taşmanın içine kopyalayamaz. Kanaryanın işini yapan şey tam olarak bu asimetri. GCC ve Clang'de üç seviye vardır:

- `-fstack-protector` — yalnızca dizi içeren fonksiyonları korur
- `-fstack-protector-strong` — yerel dizisi veya adresi alınan değişkeni olan fonksiyonları korur; dağıtımların çoğunun varsayılanı budur
- `-fstack-protector-all` — her fonksiyonu korur, en pahalısı

> **Kanarya neyi çözmez?** Yalnızca ardışık (sıralı) taşmaları yakalar. Saldırgan bir indeks hatası sayesinde stack'in tam istediği noktasına yazabiliyorsa kanaryanın üstünden atlayabilir. Ayrıca koruma yalnızca dönüş anında devreye girer; fonksiyon o ana kadar ezilmiş verilerle çalışmaya devam etmiştir.

ASLR: Adresleri Her Çalıştırmada Karıştır

Üçüncü savunma saldırının ikinci adımını hedefler: adres bilmek.

ASLR (Address Space Layout Randomization), process başlatılırken stack, heap ve paylaşılan kütüphanelerin sanal bellekteki başlangıç adreslerini rastgeleleştirir. Saldırgan artık "libc'deki `system` fonksiyonu şu adrestedir" diyemez; adres her çalıştırmada değişir.

Aynı ikili dosyayı iki kez çalıştırıp bellek haritasını karşılaştırarak bunu kendin görebilirsin:

```
cat /proc/self/maps | grep '\[stack\]'
```
```
cat /proc/self/maps | grep '\[stack\]'
```

İki komut farklı stack adresleri yazdırır. Sistem genelindeki ayar `/proc/sys/kernel/randomize_va_space` dosyasındadır: `0` kapalı, `1` stack ve kütüphaneler için açık, `2` bunlara ek olarak heap (brk) için de açık. Modern dağıtımlar `2` ile gelir.

Kernel'ın kendi kod adreslerini rastgeleleştiren karşılığına **KASLR** denir ve kernel'a yönelik saldırıları aynı mantıkla zorlaştırır.

> **ASLR neyi çözmez?** Rastgelelik bir bilgi sızıntısıyla çökebilir. Program bir bellek adresini hata mesajında yazdırıyorsa ya da format string açığı varsa, saldırgan tek bir adresi öğrenip modülün tamamının nereye yerleştiğini geri hesaplayabilir. 32-bit sistemlerde entropi de düşüktür; kaba kuvvetle denemek mümkün olabilir.

PIE: Programın Kendisi de Taşınabilsin

ASLR kütüphaneleri ve stack'i karıştırır, ama programın kendi kodu sabit bir adrese yüklenmişse saldırgan gadget'larını oradan toplayabilir.

PIE (Position Independent Executable), **[12. bölümde]** gördüğümüz paylaşılan kütüphane mantığını çalıştırılabilir dosyanın kendisine uygular. Kod mutlak adres kullanmak yerine instruction pointer'a göreli adresleme yapar (`RIP-relative`), böylece herhangi bir adrese yüklenebilir. ELF başlığındaki tür alanı bu durumda `ET_EXEC` yerine `ET_DYN` olur.

```
file /bin/ls
```

Çıktıda "pie executable" ifadesini görüyorsan binary PIE olarak derlenmiştir. Bunun bir bedeli vardır: göreli adresleme küçük bir performans maliyeti getirir ve bu yüzden bazı performans-kritik yazılımlar hâlâ PIE'siz derlenir.

RELRO: Fonksiyon Tablosunu Kilitle

[12. bölümde] dinamik bağlamanın **GOT** (Global Offset Table) üzerinden çalıştığını görmüştük: kod, `printf` gibi bir fonksiyonu çağırırken GOT'taki adrese dolaylı olarak atlar. Bu tablo yazılabilir bir bölgede durursa saldırgan için nefis bir hedeftir; oraya kendi adresini yazan biri, programın bir sonraki `printf` çağrısını ele geçirir.

RELRO (RELocation Read-Only) bu tabloyu yazılamaz hâle getirir. İki seviyesi vardır:

- **Partial RELRO** — ELF'in yeniden konumlandırma verilerinin bir kısmı salt okunur yapılır, ama fonksiyon adreslerinin tutulduğu bölüm (`.got.plt`) yazılabilir kalmak zorundadır. Çünkü lazy binding, sembolü ilk çağrıda çözüp adresi oraya **yazmak** ister; tabloyu kilitlersen dinamik bağlayıcı işini yapamaz.
- **Full RELRO** — tüm semboller program başlarken peşinen çözülür (`BIND_NOW`), sonra GOT salt okunur işaretlenir. Açılış birkaç milisaniye yavaşlar, buna karşılık GOT saldırısı tamamen kapanır.

Full RELRO, lazy binding'den vazgeçmek demektir. Bu takası tereddütsüz güvenliğin lehine kapatırım: açılışta kaybedilen birkaç milisaniye, programın fonksiyon tablosunu saldırganın kalemine açık bırakmaya değmez. Dağıtımların çoğu da aynı kararı verip paketlerini `-Wl,-z,relro,-z,now` ile derliyor.

FORTIFY_SOURCE: Derleyici Boyutu Biliyorsa Kontrol Etsin

Bazı durumlarda derleyici, hedef tamponun boyutunu derleme anında bilir. `_FORTIFY_SOURCE` bu bilgiyi kullanır: `memcpy`, `strcpy`, `sprintf` gibi çağrıları, boyut kontrolü yapan `__memcpy_chk` gibi güvenli varyantlarıyla değiştirir. Taşma tespit edilirse program çalışma anında sonlandırılır.

Şunu aklında tut: `_FORTIFY_SOURCE` kendiliğinden açılmaz. Optimizasyon açık olmalıdır (`-O1` ve üstü), çünkü derleyici tampon boyutunu ancak optimize ederken hesaplar — ama korumayı asıl açan şey derlerken `-D_FORTIFY_SOURCE=2` (yeni GCC'lerde `=3`) tanımlamandır. Dağıtımların çoğu bunu varsayılan derleme bayraklarına koyduğu için paket olarak kurduğun binary'lerde genellikle açıktır; kendi elinle `gcc dosya.c` dediğinde ise kapalıdır.

Bedeli neredeyse sıfır — çalışma anında yapılan fazladan iş, zaten bilinen bir sayıyı bir sayıyla karşılaştırmaktan ibaret. Ama sınırını da gör: derleyici hedefin boyutunu **derleme anında** bilemiyorsa, mesela tampon `malloc` ile ve çalışma anında hesaplanmış bir boyutla ayrıldıysa, koruma hiç devreye giremez. Bu yüzden FORTIFY_SOURCE'u bir ağ olarak değil, ucuz olduğu için hiç reddetmemen gereken bir bonus olarak düşün.

Hepsini Birlikte Görmek

Bir binary'nin hangi korumalarla derlendiğini `checksec` ile tek bakışta okuyabilirsin:

```
checksec --file=/bin/ls
```

Beklenen çıktıya örnek:

```
RELRO           STACK CANARY      NX            PIE
Full RELRO      Canary found      NX enabled    PIE enabled
```

Aracın kurulu değilse `readelf` ile aynı bilgilere daha ham biçimde ulaşabilirsin:

```
readelf -l -d -W /bin/ls | grep -E 'GNU_STACK|GNU_RELRO|BIND_NOW'
```

`GNU_STACK` satırındaki izinlerde `E` yoksa NX etkindir; `GNU_RELRO` segmentinin varlığı RELRO'yu, `BIND_NOW` bayrağı ise Full RELRO'yu gösterir.

Katmanlar Neden Hepsi Birden Gerekli

Bu korumaların hiçbiri tek başına yeterli değil; her biri saldırı zincirinin farklı bir halkasını kesiyor:

Koruma	Kırdığı adım	Kim uygular?
NX / W^X	Enjekte edilen kodun çalıştırılması	CPU + kernel
Stack canary	Dönüş adresinin sessizce ezilmesi	Derleyici
ASLR	Hedef adresin bilinmesi	Kernel

Koruma	Kırdığı adım	Kim uygular?
PIE	Program kodunun sabit adreste olması	Derleyici + linker
RELRO	GOT üzerinden akış kaçırma	Linker
FORTIFY_SOURCE	Bilinen boyutta taşma	Derleyici + libc

Savunma derinliği fikri tam olarak budur: saldırganın başarılı olması için hepsini birden aşması gerekir. Buna karşılık hiçbiri **bellek güvenliği açığının kendisini** ortadan kaldırmaz; sadece sömürülmesini pahalılaştırır. Use-after-free, double-free ve tam sayı taşmasından doğan hatalar bu listedeki hiçbir korumanın doğrudan hedefi değildir.

Asıl çözüm sınıfın kökünü kurutmaktan geçiyor: Rust gibi sahiplik modeli olan diller, Go gibi çöp toplayıcılı diller ya da C/C++ tarafında `std::span` ve sınır kontrollü kapsayıcılar. Nitekim Linux kernel'ında Rust desteğinin gelişi de büyük ölçüde bu motivasyonla ilerliyor.

> **Bir yana: Spectre ve Meltdown bu tabloda nerede?**
>
> **[8. bölümde]** gördüğümüz speculative execution, tamamen farklı bir sınıf açar. Buradaki korumaların hepsi yazılımın belleğe **yazmasını** denetler; Spectre ailesi ise hiçbir şey yazmadan, yalnızca CPU'nun tahmin ederken bıraktığı cache izlerini ölçerek veri sızdırır. Bu yüzden savunması da farklıdır: mikrokod güncellemeleri, kernel sayfa tablosu izolasyonu (KPTI) ve derleyici tarafında retpoline gibi teknikler.

Özet

- Sanal bellek process'leri birbirinden ayırır ama bir process'i kendi içindeki bellek hatalarından korumaz.
- Buffer overflow saldırısı üç adımdır: kodu yerleştir, adresi öğren, akışı oraya çevir.
- NX enjekte edilen kodun çalışmasını, stack canary sessiz ezilmeyi, ASLR ve PIE adres bilmeyi, RELRO GOT üzerinden kaçırmayı engeller.
- `checksec` ve `readelf` ile herhangi bir binary'nin hangi katmanlarla derlendiğini görebilirsin.
- Hiçbiri açığın kendisini kapatmaz; bellek güvenli diller sorunu sınıf düzeyinde çözer.

Burada bir soru asılı kalıyor. Kitap boyunca hep tek bir process'i takip ettik: dosyadan doğdu, belleğe yerleşti, korundu. Peki o ilk process nereden geldi? Bilgisayarını açtığında birinci process'i kim başlattı ve o process kendinden sonrakileri nasıl doğurdu? Sıradaki bölümün konusu bu.

Bölüm 16: Fork'lar ve COW'lar Hakkında Konuşalım

Son soru: Buraya nasıl geldik? İlk process'ler nereden çıktı?

Bu kitabın son düzlüğündeyiz. Buraya kadar CPU'nun tek bir instruction pointer'ından başlayıp process'lerin birbirinden nasıl yalıtıldığına kadar geldik. Geriye tek bir soru kaldı ve bu bölüm onun cevabı: madem her process başka bir process tarafından başlatılıyor, bu zincirin ilk halkası nereden çıktı?

> **Bu bölümde neyi çözüyoruz?**
>
> - `fork` ile yeni process üretmenin neden "aynı yerden iki kez devam etmek" gibi göründüğünü anlayacağız.
> - Copy-on-Write sayesinde büyük belleklerin neden baştan kopyalanmadığını göreceğiz.
> - Boot sonunda ilk user-space process'in nasıl doğduğunu ve geri kalan programları nasıl başlattığını bağlayacağız.

`execve`, mevcut `process`'i değiştirerek yeni bir program başlatıyorsa, tamamen ayrı bir `process` içinde yeni bir programı nasıl başlatıyorsun? Bilgisayarda birden fazla şey yapmak istiyorsan bu kritik bir yetenek. Bir uygulamaya çift tıkladığında, onu açan program kaybolmaz; yeni uygulama ondan bağımsız şekilde yaşamına devam eder.

Cevap başka bir syscall: `fork`. Çoklu işlem dünyasının temel taşı budur. `fork`, mevcut `process`'i ve belleğini klonlar, kaydedilmiş instruction pointer'ı olduğu yerde bırakır ve ardından iki `process`'in de normal akışına devam etmesine izin verir. Müdahale edilmezse, iki `process` birbirinden bağımsız ama aynı koddaki farklı yürütmeler olarak devam eder.

Yeni ortaya çıkan `process`'e "child", onu başlatana da "parent" denir. Bir `process`, `fork`'ü birden fazla kez çağırabilir ve böylece birden çok child yaratabilir. Her `process`'in, 1'den başlayarak verilen bir *process ID*'si (PID) vardır.

Tamamen aynı kodu körlemesine iki kopya hâline getirmek tek başına çok faydalı olmazdı. Bu yüzden `fork`, parent ve child tarafında farklı değer döndürür. Parent tarafında yeni child `process`'in PID'sini döndürür, child tarafında ise `0` döner. Böylece her iki taraf da kendi rolüne göre farklı iş yapabilir.

> Linux'ta C kütüphanesinin `fork()` wrapper'ı kernel tarafında `clone` ailesinin özel bir kullanımına denk gelir; klasik fork davranışı kabaca `SIGCHLD` ile yeni bir child process üretmektir. `clone` ve modern `clone3` ise hangi kaynakların paylaşılacağını (`CLONE_VM`, `CLONE_FILES` gibi) bayraklarla belirlemeye izin verir. Thread oluşturma da bu ailenin başka bir kullanım biçimidir. `fork`'u temel taşı olarak anlatmak pedagojik olarak doğrudur, ancak kernel/C tarafında resim `clone` etrafında şekillenir.

main.c

```
pid_t pid = fork();
// Bu satirdan sonrasi iki process'te birden akar.
// Tek ayirt edici isaret, fork'un dondurdugu deger.

if (pid == 0) {
        // Child process'teyiz.
} else {
        // Parent process'teyiz; pid, child'in numarasi.
}
```

`fork`'un garip yanı şu: tek bir çağrı yapıyorsun, o çağrıdan iki kez dönülüyor.

Kafanda oturtmanın en kolay yolu, `fork` satırını bir çatallanma noktası gibi görmek. O satıra kadar tek bir yürütme akışı vardı; o satırdan sonra aynı koda bakan, aynı değişkenlere sahip, aynı satırdan devam eden iki akış var. İkisi arasındaki tek fark `fork`'un döndürdüğü sayı: parent'ta child'ın PID'si, child'ta `0`. Programın geri kalanı işte bu tek sayıya bakarak hangi tarafta olduğuna karar veriyor.

Aynı fikri bol şemayla görmek istersen şu eski ders notu iyi bir tekrar sunuyor.

Child, Sonucu Parent'a Nasıl Veriyor?

Yukarıdaki koddaki "sonucu parent'a ver" yorumu bir soruyu açıkta bırakıyor. Child ayrı bir adres alanında yaşıyor; bir değişkene yazarak parent'a hiçbir şey iletemez. Peki nasıl iletiyor?

İki klasik yol var.

Çıkış kodu. Child `exit(5)` ile sonlanır, parent `wait()` ile bu değeri okur. Basit ama dar bir kanal: yalnızca tek bir bayt taşır. **[11. bölümde]** konuştuğumuz çıkış kodu sözleşmesi tam olarak bu mekanizmadır.

Pipe. Gerçek veri taşımak gerekiyorsa, `fork`'tan **önce** bir pipe açılır. Burada devreye kitabın en işlevsel kurallarından biri giriyor: **child, parent'ın açık file descriptor'larını devralır.** `fork` anında parent'ta açık olan her fd, child'da da açık olarak belirir. Bu yüzden fork'tan önce açılmış bir pipe, iki tarafın da elinde hazır bekler.

Bu kural sandığından daha geniş bir alanı açıklıyor. **[11. bölümdeki]** `>` yönlendirmesi ve `|` pipeline'ı da tam olarak buna dayanıyordu: shell fork ediyor, child devraldığı fd'leri `dup2` ile yeniden bağlıyor ve `execve` çağırıyor. Aynı kuralın bir de güvenlik tarafı var — bölümün sonundaki `O_CLOEXEC` tartışması oradan çıkıyor.

Her neyse: Unix programları yeni bir program başlatmak istediklerinde tipik olarak önce `fork` çağırır, ardından child `process` içinde hemen `execve` çalıştırır. Buna *fork-exec modeli* denir. Bir programı başlattığında bilgisayarının yaptığı şey kabaca şuna benzer:

launcher.c

```
pid_t pid = fork();

if (pid == 0) {
        // Child process'i hemen yeni programla değiştir.
        execve(...);
}

// Buraya geldiysek process değiştirilmedi.
// Parent process'teyiz. İstersek yeni child'ın PID'si de elimizde.

// Parent program burada devam eder...
```

> Child process sonlandığında, parent onu `wait()` ile temizlemezse bir zombie olarak kalır. Zombie, çıkış kodunu parent'a iletmek için process tablosunda yer tutan, ancak artık çalışmayan bir process'tir. Parent ölürse child genellikle aynı PID namespace içindeki init'e veya varsa subreaper olarak atanmış başka bir process'e evlat edinilir; o process de `wait()` ile temizleme sorumluluğunu alır.

Moooo!

Fark etmiş olabilirsin: Başka bir program yüklerken bir process'in belleğini yalnızca biraz sonra çöpe atmak üzere tamamen kopyalamak kulağa verimsiz geliyor. Neyse ki elimizde bir MMU var. En pahalı şey fiziksel RAM'deki veriyi çoğaltmaktır; `fork()` bunun yerine task/process yapıları ve page table eşlemeleri üzerinde çalışır. Page table kopyalamak da bedava değildir, ama devasa veri bloklarını baştan sona RAM'de çoğaltmaktan çok daha ucuzdur. Bu yüzden ilk anda hiç veri kopyalamayız. Yeni process için eski process'in page table eşlemelerinin bir kopyasını çıkarır ve her iki tarafın da aynı fiziksel bellek bloklarına bakmasını sağlarız.

Ama child process'in parent'tan bağımsız ve yalıtılmış olması gerekir. Child'ın parent belleğine yazabilmesi ya da tam tersi kabul edilebilir değil.

İşte burada `COW` (*copy on write*) page'leri devreye girer. `COW` page'lerinde iki `process` de aynı fiziksel bellekten okuyabilir; ama içlerinden biri yazmaya kalktığı anda o page RAM içinde kopyalanır. Böylece her iki `process` de, baştan bütün bellek alanını çoğaltmanın maliyetine katlanmadan ayrı birer bellek görüntüsüne kavuşur. Fork-exec modelinin verimli olmasının nedeni tam olarak budur: Yeni binary yüklenmeden önce eski belleğe yazılmadığı için çoğu durumda gerçek kopyalama hiç gerekmez.

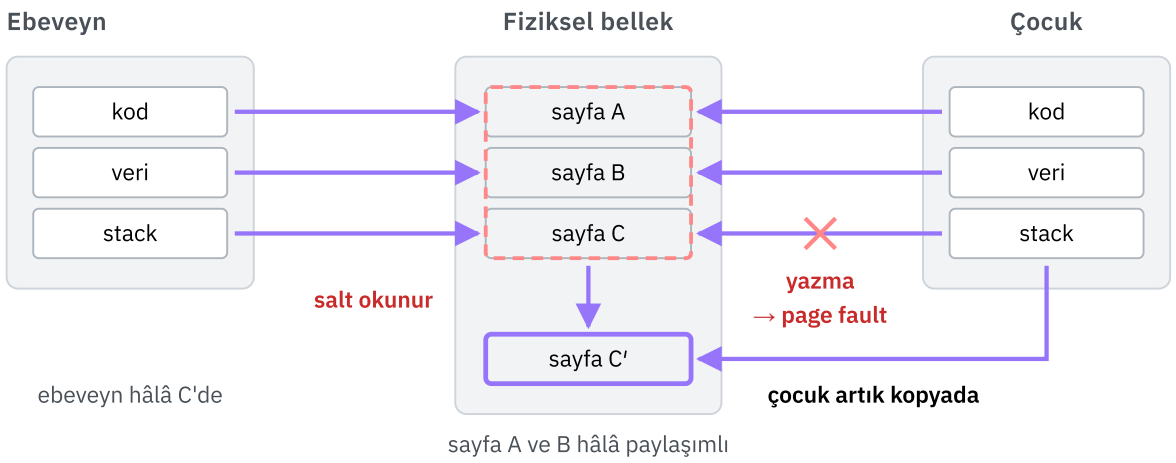
COW veriyi kopyalamaz ama **page table'ı kopyalar**. Adres alanı büyüdükçe bu tablonun kendisi de büyür: 100 GB kullanan bir process'te milyonlarca PTE (page table entry — bir sanal sayfayı bir fiziksel çerçeveye bağlayan tek satır) demektir ve `fork`'un kendisi gözle görülür biçimde yavaşlar. Bu maliyetten kaçınmanın yollarını bölümün sonundaki Derinleşme panelinde konuşacağız.

`COW`, pek çok eğlenceli şey gibi, page fault'lar üzerinden uygulanır. `fork`, parent'ı klonladıktan sonra iki process'in de paylaşılan sayfalarından **yazılabilir olanlarını** geçici olarak salt okunur

işaretler. Zaten salt okunur olan sayfalara (örneğin programın kendi kodu) dokunmasına gerek yoktur; kasıtlı olarak paylaşılan sayfalara ise dokunmaması gerekir — sebebini birazdan kernel kaynağında göreceğiz. Bir program belleğe yazmaya kalktığında, sayfa salt okunur olduğu için yazma başarısız olur. Bu da kernel'ın ele aldığı bir page fault tetikler. Kernel ilgili page'i kopyalar, yazılabilir hâle getirir ve sonra kesmeden çıkıp yazmayı tekrar dener.

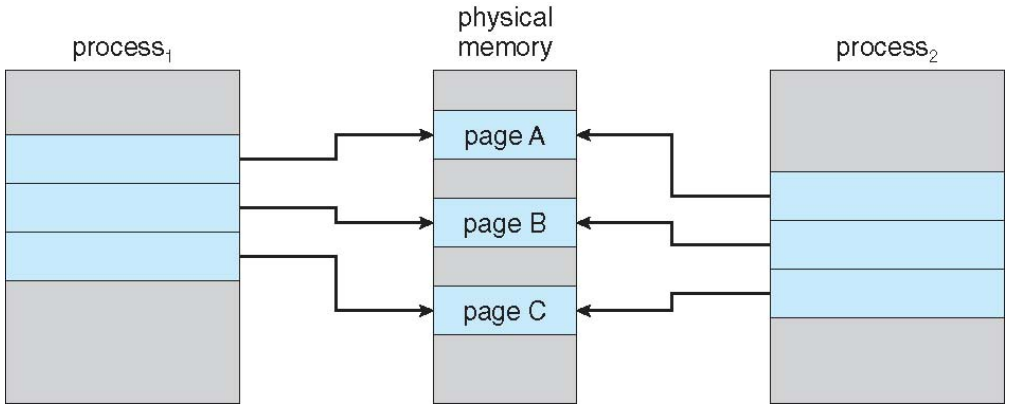
Bu "önce paylaş, yazınca kopyala" mantığını adım adım izleyebilirsin:

FORK VE COPY-ON-WRİTE

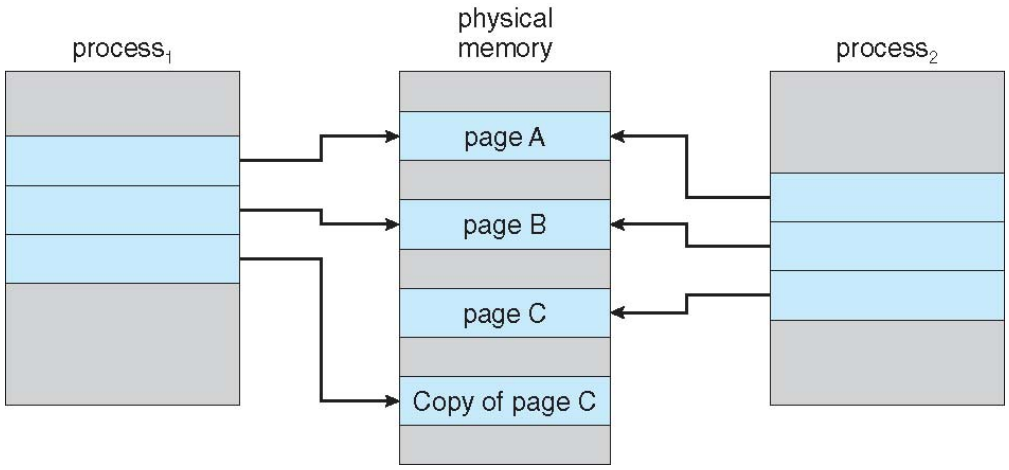


1. **Tek process.** Ebeveyn process'in sayfaları kendisine ait ve yazılabilir durumda.
2. **fork çağrılır.** Kernel yeni bir process yaratır ama sayfaların içeriğini kopyalamaz; iki process de aynı fiziksel sayfalara bakar.
3. **Sayfalar salt okunur işaretlenir.** Paylaşılan her sayfa iki tarafta da yazma iznini kaybeder. Okumalar serbest, yazma denemesi kernel'a düşecek.
4. **Çocuk bir sayfaya yazmaya çalışır.** Salt okunur sayfaya yazma girişimi page fault üretir ve kontrol kernel'a geçer.
5. **Yalnızca o sayfa kopyalanır.** Kernel sayfanın bir kopyasını çıkarır, çocuğa yazılabilir olarak verir ve programı kaldığı yerden sürdürür. Diğer sayfalar hâlâ paylaşımlıdır.

Aynı akışı ders kitabı diyagramlarıyla da görelim. İlk diyagram `fork()` sonrasını gösteriyor: parent ve child aynı fiziksel bellek sayfalarını paylaşıyor ve yazılabilir sayfalar geçici olarak salt okunur işaretlenmiş durumda. Henüz hiçbir kopyalama yapılmadı.



İkinci diyagram ise bir process sayfaya yazdığında ne olduğunu gösteriyor: kernel o sayfayı kopyalar, yazan process'i yeni kopyaya yönlendirir ve diğer process orijinal sayfayı kullanmaya devam eder.



Kaynak: Silberschatz, Galvin, Gagne — Operating System Concepts, Ch. 10 — Virtual Memory, Slide 26-27.

> *A: Tak tak!*
> *B: Kim o?*
> *A: İneğin sözünü kesen.*
> *B: İneğin sözünü kesen ki —*
> *C:* ***MOOOOO!***

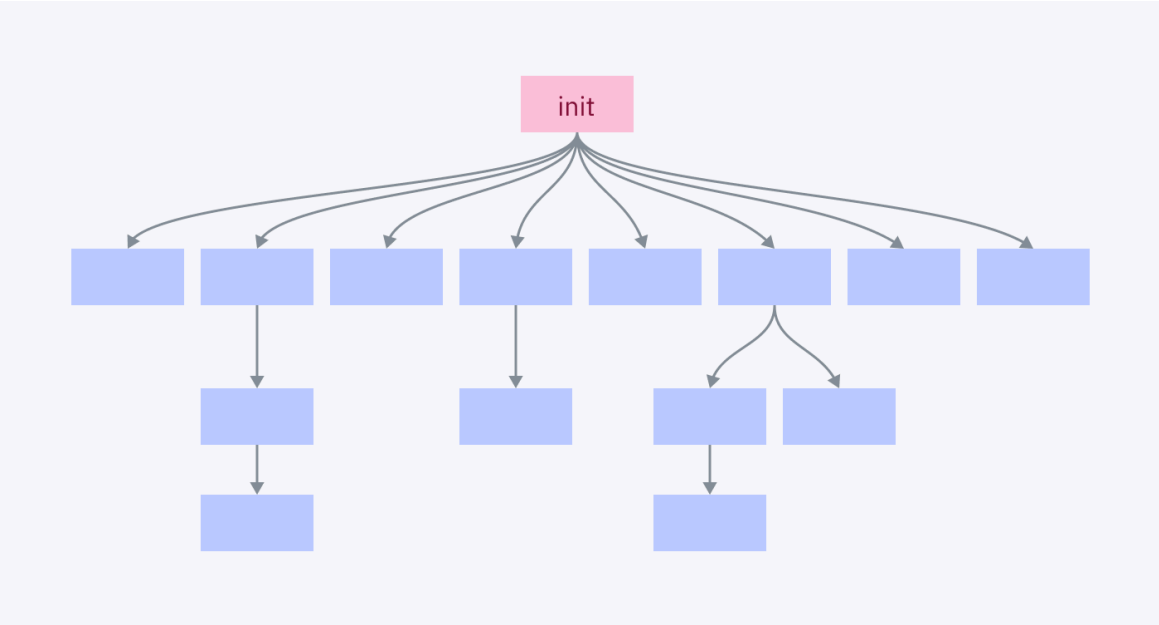
Başlangıçta (Yaratılış 1:1 Olan Değil)

Normal bir Unix benzeri sistemde bilgisayarındaki process'lerin büyük çoğunluğu, zincirin bir yerinde başka bir program tarafından `fork` edilip çalıştırılmıştır. Bu ağacın kökünde özel bir process durur: `init process`. `Init process`, doğrudan kernel tarafından hazırlanır. Bu, user space'te çalışan ilk programdır ve sistem kapanırken en son ayrılanlardan biridir.

`init process`'in ne kadar hayati olduğunu anlamak için onu öldürmeye çalışmak gerçek makinelerde iyi bir fikir değildir. Linux kernel'ı PID 1'e gelen bazı ölümcül sinyalleri normal process'lerden farklı ele alır; `init`/`systemd` gibi PID 1 programları da SIGTERM gibi sinyalleri sistem kapanışı başlatmak için yorumlayabilir. PID 1 gerçekten çöker veya çıkarsa meşhur "Kernel panic — not syncing: Attempted to kill init!" ailesinden hatalarla karşılaşabilirsin. Kısacası: bunu üretim ya da günlük kullandığın makinede deneme.

> *Yazar notu: Buradaki `init process` anlatısı Linux ve macOS gibi Unix benzeri sistemler için geçerli. Windows'un NT kernel'ı aynı işi bambaşka bir mimariyle çözer; bu bölümü okuduktan sonra Windows'un açılışını da anladığını varsayma.*
>
> *Tıpkı `execve` bölümünde olduğu gibi sınırı açıkça çiziyorum: NT kernel'ın boot ve process modeli başlı başına ayrı bir kitabın konusu, bu kitabın değil.*

`Init process`, işletim sistemini oluşturan servislerin ve programların büyük bölümünü başlatmaktan sorumludur. Bu programların çoğu da daha sonra kendi çocuklarını üretir.



`Init process`'in kaybolması, process ağacının kökünü ve servis yönetimini kaybetmek anlamına gelir. Bu yüzden pratik sonuç çoğu sistemde oturumun kapanması, servislerin durması veya kernel panic gibi dramatik bir arıza olur.

Kernel'a Dönüş

Linux kernel kodunu kurcalamak **[Bölüm 10]**'da bayağı eğlenceliydi; biraz daha yapalım. Bu kez kernel'ın `init process`'i nasıl başlattığına bakacağız.

Bilgisayarın açılışı kabaca şu sırayla ilerler:

1. Anakartın üzerinde, diskten bağımsız duran küçük bir yazılım vardır: **firmware** (eski adıyla BIOS, modern makinelerde UEFI). Firmware RAM'de durmaz; anakarta lehimli, elektrik kesilince silinmeyen bir flash yongasında yaşar — **[2. bölümdeki]** Derinleşme panelinde bunun neden zorunlu olduğunu konuşmuştuk. Firmware önce donanımı ve RAM'i kullanılabilir hâle getirir, sonra bağlı disklerde bootloader denen küçük programı arar, onun makine kodunu RAM'e kopyalar ve o koda atlar.
2. Unutma: Henüz tam anlamıyla çalışan bir işletim sistemi dünyasında değiliz. Kernel bir `init process` başlatana kadar çoklu işlem, syscall ve benzeri üst seviye soyutlamalar aslında ortada yoktur. Boot öncesi bağlamda bir programı "çalıştırmak", çoğu zaman RAM'deki makine koduna doğrudan atlamak demektir.

3. Bootloader, kernel'ı bulmak, RAM'e yüklemek ve çalıştırmaktan sorumludur. GRUB gibi bootloader'lar yapılandırılabilir ve hatta birden fazla işletim sistemi arasında seçim yapmana izin verebilir. Windows tarafında Windows Boot Manager bu rolün önemli parçasıdır; modern macOS'ta ise Intel ve Apple Silicon makinelerde boot zinciri farklı ayrıntılara sahiptir, ama temel fikir yine firmware'in işletim sistemi yükleyicisine ve kernel'a yolu açmasıdır.
4. Kernel artık çalışır durumdadır ve interrupt handler'ları kurmak, driver'ları yüklemek, ilk bellek eşlemelerini oluşturmak gibi geniş bir başlatma rutinine girer. Sonunda `init` programını başlatır — dikkat et, kernel kendi ayrıcalık seviyesini düşürmez; kernel mode'da kalmaya devam eder ve *yeni başlattığı process*'i user mode'da çalıştırır.
5. İşte şimdi gerçekten bir işletim sisteminin kullanıcı alanındayız. Init programı init script'lerini çalıştırır, servisleri başlatır ve shell ile grafik arayüz gibi başka programları devreye sokar.

Linux'un Başlatılması

Yukarıdaki 4. adım, yani kernel'ın kendi kendini kurması, `init/main.c` içindeki `start_kernel` fonksiyonunda geçer. Bu fonksiyon yüzlerce satır boyunca sırayla her alt sistemi ayağa kaldırır: bellek yöneticisi, scheduler, interrupt tablosu, zamanlayıcılar, dosya sistemi katmanı.

Sonunda ilginç bir devir teslim olur. Kernel, kendi başlatma kodunun içinden çıkıp **PID 1'i doğuracak bir görev** yaratır ve kontrolü scheduler'a bırakır. Yani boot'un son adımı "kernel bir programı çalıştırdı" değildir; kernel kendini sıradan bir işletim sistemine dönüştürmüş ve ilk kullanıcı programını sıraya koymuştur.

O ilk program aranırken kernel bir yedek listesi üzerinde yürür:

kernel_init @ init/main.c

```
/* Sırayla dener; ilk başarılı olan PID 1 olur.
 * Gerçekten bozuk bir makineyi kurtarmak için
 * init yerine Bourne shell de kullanılabilir. */
if (!try_to_run_init_process("/sbin/init") ||
    !try_to_run_init_process("/etc/init")  ||
    !try_to_run_init_process("/bin/init")  ||
    !try_to_run_init_process("/bin/sh"))
       return 0;

panic("No working init found.  Try passing init= option to kernel.");
```

Bu birkaç satır, kernel'ın kişiliği hakkında beklediğinden fazlasını söylüyor. Listenin sonundaki `/bin/sh` bir kurtarma kapısıdır: init sistemin bozulduysa makine yine de açılsın, eline çıplak bir shell versin, sen düzelt. Ve hiçbiri bulunamazsa kernel kibarca pes etmez, `panic` eder. Çünkü çalıştıracak tek bir user-space programı olmayan bir kernel'ın yapabileceği anlamlı hiçbir şey yoktur.

Linux'ta `init` neredeyse her zaman `/sbin/init`'tir ya da oraya bir sembolik bağdır; arkasında genelde systemd, OpenRC veya runit durur. macOS'un karşılığı `launchd`'dir. Kernel olmadığın için onu terminalden elle çalıştırmayı denemeni önermem.

Artık boot sürecinin sonundayız: `init process` user space'te çalışıyor ve fork-exec modeliyle geri kalan her şeyi başlatıyor.

Çatal Bellek Eşlemesi

`fork`'ün asıl işi `kernel/fork.c` içinde döner ve dosyanın açılış yorumu, kernel'ın kendine dürüstlüğünün güzel bir örneğidir: fork'ün kendisi bir kez kavradın mı basittir, asıl belaya bellek yönetiminde girilir.

Gerçekten de öyle. Bellek eşlemesini kopyalayan kod, her bölge için tek bir soru sorar: bu alanın Copy-on-Write olması gerekiyor mu? Cevap kernel kaynağında tek satırlık bir bit kontrolüdür ve okunuşu şu: **yazılabilir ama paylaşılmıyorsa** COW gerekir. Paylaşılan bellek zaten ortak kullanılmak için oradadır, onu kopyalamak isteneni bozmak olur. Salt okunur

bellek ise hiç değişmeyeceği için kopyalanmasına gerek yoktur. Geriye tek bir durum kalır ve tam olarak o durum COW'dur.

Bu fonksiyon `copy_page_range` → `dup_mmap` → `dup_mm` → `copy_mm` zinciriyle en sonunda `copy_process`'e bağlanır. Yani Unix'te bir programın başlatılması kökeninde hep aynı şeydir: var olan bir process'in kopyalanıp uyarlanması.

DERİNLEŞME **`fork()` iyi bir soyutlama mı?**

`fork` yarım asırdır Unix'in temel taşı. Buna karşılık 2019'da bir grup sistem araştırmacısı (Baumann, Appavoo, Krieger, Roscoe) HotOS'ta *"A fork() in the road"* başlıklı bir bildiri yayımladı. Tezleri şuydu: `fork` bugün artık kötü bir soyutlamadır ve büyük ölçüde alışkanlıktan yaşamaktadır.

Argümanları kayda değer:

- **Thread'lerle bileşmiyor.** `fork` yalnızca çağıran thread'i kopyalar, diğerlerini bırakır. Ama onların tuttuğu kilitler kopyaya *kilitli hâlde* geçer ve açacak kimse kalmaz. Bu yüzden POSIX, `fork` ile `exec` arasında yalnızca async-signal-safe çağrılara izin verir; pratikte `malloc` bile yasaktır. Çok thread'li bir programda `fork` sonrası `printf` çağırmak tanımsız davranıştır.
- **Ölçeklenmiyor.** COW veriyi kopyalamaz ama page table'ları kopyalar. 100 GB'lık bir adres alanı milyonlarca PTE demektir; `fork` çağrısının kendisi yüz milisaniyelerce sürebilir. Redis'in arka planda anlık görüntü alırken yaşadığı gecikme sıçramalarının kaynağı budur.
- **Güvenli değil.** Child, parent'ın **her şeyini** devralır: adres alanı, açık file descriptor'lar, ASLR düzeni. Ayrıcalık düşürmek "neyi devrettiğini hatırlamak" işine dönüşür ve unutulan her şey bir açıktır.
- **Modüler değil.** `fork`'ün davranışı, programın kullandığı her kütüphaneyi ilgilendirir. Bir kütüphane arka planda thread açtıysa senin `fork` çağrın onun yüzünden bozulabilir; üstelik bunu önceden bilmenin bir yolu yoktur.

Önerilen alternatif yeni değil: yeni process'i klonlamak yerine **tarif ederek** yaratmak. `posix_spawn` tam olarak bunu yapar, Windows'un `CreateProcess`'i baştan böyle tasarlanmıştır, `vfork` da aynı derdin eski bir yamasıdır.

Peki `fork` neden hâlâ her yerde? Çünkü tarife dayalı API'ler *ifade gücünden* ödün verir. `fork` sonrası, `exec` öncesi o küçük pencerede çocuğun içinde istediğin her şeyi yapabilirsin: fd'leri yeniden bağla, namespace değiştir, cgroup'a gir, capability düşür. **[18. bölümde]** göreceğimiz container runtime'larının işi tam olarak o pencerede yaşar ve `posix_spawn`'ın sabit eylem listesine sığmaz.

Yani tartışma “hangisi doğru” değil; klasik bir tasarım gerilimi: **ifade gücü mü, öngörülebilirlik mi?** Bu bölümde gördüğün mekanizma gerilimin bir tarafı. Hangi tarafın kazanması gerektiği hâlâ açık bir soru.

> **İzlemelik:** `fork` ve `exec` ikilisinin neden bu kadar merkezi olduğunu görsel olarak anlatan bir video için bu kaydı izleyebilirsin.

Özetle...

Peki... programlar nasıl çalışır?

En düşük seviyede cevap şu: işlemciler aptaldır. Ellerinde bellekte bir işaretçi vardır ve onlara başka yere atlamalarını söyleyen bir talimata rastlamadıkları sürece talimatları art arda yürütürler.

Ama bu akış sadece jump talimatlarıyla değişmez; hardware ve software interrupt’lar da yürütmeyi, önceden belirlenmiş başka bir konuma sıçratarak bozabilir. Tek bir işlemci çekirdeği aynı anda birden fazla programı gerçekten çalıştıramaz, ama timer’lar aracılığıyla tekrar tekrar interrupt üretip kernel’ın farklı instruction pointer’lar arasında geçiş yapmasını sağlayarak bunu ikna edici biçimde taklit edebiliriz.

Programlar, aslında olduklarından çok daha düzenli bir ortamda çalıştıklarına inandırılır. User mode, sistem kaynaklarına doğrudan erişimi keser; sayfalama, bellek alanını yalıtır; syscall’lar ise process’lerin altında yatan gerçek yürütme bağlamını bilmeden genel G/Ç yapabilmesini sağlar. Syscall dediğimiz şey, CPU’ya “kernel’ın önceden belirlediği şu kod yoluna git” diyen talimatlardır.

Ama... programlar nasıl çalışır?

Bilgisayar açıldıktan sonra kernel, `init process`’i başlatır. Bu, makine kodunun çok fazla donanım ayrıntısıyla uğraşmak zorunda olmadığı ilk yüksek seviyeli programdır. `Init`, bilgisayarının grafik ortamını ve geri kalan servisleri başlatır; yani diğer yazılımların dünyaya gelmesinden sorumlu ana ebeveyndir.

Bir programı başlatmak için `init` ya da başka bir `process` önce `fork` ile kendini klonlar. Bu klonlama verimlidir; çünkü page'ler `COW` olarak paylaşılır ve fiziksel RAM'i baştan sona kopyalamak gerekmez. Linux tarafında bu hikâyenin büyük kısmı `copy_process` çevresinde akar.

Ardından child process, gerekirse parent'tan farklı yola sapar. Yeni programı gerçekten başlatmak istediğinde exec ailesinden bir syscall çağırır ve kernel'dan mevcut process'i yeni programla değiştirmesini ister.

Bu yeni program çoğu zaman bir ELF dosyasıdır. Kernel, ELF'yi ayrıştırıp kodun ve verinin yeni sanal bellek düzeninde nereye yükleneceğini belirler. Program dynamic linked ise, gerekirse ELF interpreter'ı da devreye girer.

Son adımda kernel yeni sanal bellek eşlemesini kurar ve user space'e geri döner. Pratikte bu, CPU'nun instruction pointer'ını yeni programın kodunun başlangıcına ayarlamak demektir. Ve işte, program gerçekten çalışmaya başlar.

> **Şimdi başta sorduğumuz sorulara dönelim:**
>
> - *CPU birden fazla process'i takip etmiyorsa nasıl çoklu görev yapıyor?* → Timer interrupt'ları ve context switch ile.
> - *Programlar neden birbirinin belleğine erişemiyor?* → Sanal bellek ve sayfa tabloları ile izole ediliyorlar.
> - *Her process kendi dünyasını nasıl görüyor?* → MMU her process için farklı sanal→fiziksel eşleme yapıyor.
>
> Hepsinin cevabını şimdi biliyoruz.
>
> Sıradaki bölümde bu dünyanın en çok kullanılan ama en az konuşulan arayüzüne bakacağız: dosyalar.

Bölüm 17: Dosya Sistemi ve I/O

Buraya kadar programın nasıl başladığını, belleğe nasıl yerleştiğini ve nasıl çoğaldığını gördük. Ama bir programın işe yarayabilmesi için dış dünyayla konuşması gerekir: dosya okumak, ağdan veri almak, terminale yazmak.

Linux bunların hepsini tek bir soyutlamaya indirger ve bu, sistemin en zarif tasarım kararlarından biridir.

> **Bu bölümde neyi çözüyoruz?**
>
> - "Her şey dosyadır" felsefesinin pratikte ne anlama geldiğini göreceğiz.
> - Bir `read` çağrısının file descriptor'dan VFS'e, oradan inode ve page cache'e uzanan yolunu izleyeceğiz.
> - Bloklanan I/O'nun neden ölçeklenmediğini ve `epoll` ile `io_uring`'in bunu nasıl çözdüğünü anlayacağız.

Her Şey Dosyadır

Linux'ta sıradan dosyalar da, dizinler de, klavyen de, ağ soketleri de, hatta çalışan process'lerin kendisi de aynı arayüzün arkasındadır: `open`, `read`, `write`, `close`.

Bu, sıradan bir kolaylık değil. Aynı `read` çağrısı bir metin dosyasında, bir TCP bağlantısında ve bir donanım aygıtında çalışır. Shell'deki yönlendirme ve pipe'lar da tam olarak bu tekdüzelik sayesinde mümkündür; **[11. bölümde]** gördüğümüz `komut > dosya` yapısı, aslında process'in 1 numaralı file descriptor'ını başka bir nesneye bağlamaktan ibarettir.

Bunu somut görmek için `/proc` dosya sistemine bakabilirsin. Diskte hiçbir karşılığı olmayan, kernel tarafından anlık üretilen sanal bir dosya sistemidir:

```
cat /proc/cpuinfo | head -3
```

Burada okuduğun şey bir dosya değil; kernel'ın o anda ürettiği metin. Ama `cat` bunu bilmez, bilmesine de gerek yoktur.

File Descriptor: Küçük Bir Tam Sayı

Bir process bir dosya açtığında kernel ona küçük bir tam sayı verir: **file descriptor** (fd). Bu sayı, process'in kendi fd tablosundaki bir indekstir.

Her process üç fd ile başlar: `0` standart girdi, `1` standart çıktı, `2` standart hata. Yeni açılan her nesne, kullanılabilir en küçük numarayı alır — bu davranış tesadüfi değil, standartta garanti edilmiştir ve shell'in yönlendirme numaraları buna dayanır.

Çalışan bir process'in açık fd'lerini doğrudan görebilirsin:

```
ls -l /proc/self/fd
```

Kernel tarafında üç katmanlı bir yapı vardır ve bu ayrım **[16. bölümdeki]** `fork` davranışını açıklar:

1. **fd tablosu** — process'e özeldir, fd numarasını bir açık dosya tanımına bağlar.
2. **Açık dosya tanımı** — dosya konumunu (offset) ve erişim kipini tutar; `fork` sonrasında ebeveyn ve çocuk **aynı** tanımı paylaşır.
3. **inode** — dosyanın kendisidir; birden fazla açık dosya tanımı aynı inode'a bakabilir.

Bu yüzden `fork` sonrası çocuk bir şey okuduğunda ebeveynin okuma konumu da ilerler: offset paylaşılan tanımdadır. Buna karşılık `open` ile aynı dosyayı iki kez açarsan iki ayrı tanım, dolayısıyla iki ayrı offset elde edersin.

VFS: Ortak Dil

Peki `read` çağrısı ext4, Btrfs, XFS, NFS ve `/proc` üzerinde nasıl aynı şekilde çalışır? Cevap **VFS** (Virtual File System): kernel içinde, gerçek dosya sistemlerinin üzerinde duran bir soyutlama katmanı.

VFS'in bütün modeli dört nesne üzerine kurulu ve dördünü tanımak, `/home/emir/notlar.txt` gibi bir yolun nasıl gerçek veriye dönüştüğünü de açıklıyor:

- **superblock** — bağlı (mount edilmiş) dosya sisteminin kendisini temsil eder. Blok boyutu, toplam kapasite, hangi sürücünün yönettiği gibi bilgiler buradadır. Her mount noktası

için bir tane vardır.

- **inode** — bir dosyanın kimliği. Az önce konuştuğumuz künye. Dikkat: inode'un **adı yoktur**.
- **dentry** (directory entry) — ad ile inode arasındaki bağ. `notlar.txt` adının hangi inode'a karşılık geldiğini söyleyen şey budur.
- **file** — açık bir dosyayı temsil eder. Aynı inode'a bakan iki ayrı `open` çağrısı iki ayrı `file` nesnesi üretir; her birinin kendi okuma konumu (offset) vardır.

İsimlerin inode'da değil dentry'de olması ilk bakışta tuhaf gelebilir ama bir şeyi doğrudan açıklıyor: **aynı inode'a birden fazla ad bağlanabilir.** Hard link dediğimiz şey tam olarak budur ve dosyayı silmenin neden "adı kaldırmak" anlamına geldiğini de bu açıklar.

Yol çözümleme de bu nesneler üzerinde yürüyen bir zincirdir. `/home/emir/notlar.txt` için VFS önce kök dizinin inode'unu alır, içinde `home` dentry'sini arar, bulduğu inode'a geçer, orada `emir`'i arar, sonra `notlar.txt`'yi. Her adım bir dizin okuması demektir.

> Bu yürüyüşün her seferinde diske inmesi felaket olurdu. Kernel bu yüzden **dentry cache** tutar: ad → inode eşleşmelerini bellekte saklar. Sık kullandığın yolların çözümlenmesi neredeyse bedavaya gelir; `ls` komutunu aynı dizinde ikinci kez çalıştırdığında hissettiğin hız farkının bir kısmı buradan gelir.

Peki VFS, tek bir `read` çağrısını ext4 ile NFS arasında nasıl doğru yere yönlendiriyor? Her dosya sisteminden bir **fonksiyon işaretçisi** tablosu ister. Fonksiyon işaretçisi, bir fonksiyonun adresini değişkende tutmak demektir; tabloya hangi adresi koyarsan `read` çağrısı oraya gider. Nesne yönelimli bir arayüzün C ile yazılmış hâli gibi düşünebilirsin.

Kernel kaynağında bu tablonun adı `file_operations`'dır ve içinde `read`, `write`, `open` gibi her işlem için bir satır bulunur. ext4 o satırlara kendi fonksiyonlarının adresini yazar, NFS bambaşka adresler yazar, `/proc` bir üçüncüsünü. Syscall geldiğinde VFS yalnızca ilgili satırdaki adrese atlar; hangi dosya sistemine gittiğini bilmesine bile gerek yoktur.

Yeni bir dosya sistemi yazmak, bu tabloyu doldurmak demektir. Kullanıcı tarafındaki hiçbir program değişmez.

inode: Dosyanın Kimliği

Bir dosyanın adı, dosyanın kendisi değildir. Ad, dizinde tutulan bir girdidir ve bir **inode** numarasına işaret eder. Dosyanın künyesi — boyut, izinler, sahip, zaman damgaları ve veri bloklarının diskte nerede olduğu — inode'da durur. Dikkat et: **verinin kendisi inode'un içinde değildir**; inode yalnızca o blokların adresini tutar. (Çok küçük dosyalarda bazı dosya sistemleri veriyi doğrudan inode'a sığdırır — buna *inline data* denir — ama bu bir istisnadır, kural değil.)

```
ls -li /etc/hostname
```

Baştaki sayı inode numarasıdır. Bu ayrım birkaç davranışı bir anda açıklar:

- **Hard link**, aynı inode'a ikinci bir addır. İki ad da eşit derecede "gerçektir"; birini silmek dosyayı yok etmez, yalnızca bağlantı sayacını azaltır.
- **Silinen ama açık dosya** yer kaplamaya devam eder. `rm` yalnızca dizin girdisini kaldırır; inode, onu açık tutan process kapatana kadar yaşar. Log dosyasını sildiğin hâlde diskin bir türlü boşalmamasının nedeni tam olarak budur.
- **Dosyayı taşımak** aynı dosya sistemi içindeyse ucuzdur, çünkü veri kopyalanmaz; yalnızca dizin girdisi değişir.

Page Cache: Diskin Önündeki Bellek

[5. bölümde] diskin RAM'e göre binlerce kat yavaş olduğunu görmüştük. Kernel bu farkı **page cache** ile kapatır: diskten okunan her sayfa, kullanılmayan RAM'de saklanır. Aynı veri tekrar istendiğinde disk hiç dönmez.

Bu yüzden bir dosyayı ikinci kez okumak dramatik biçimde hızlıdır ve bu yüzden `free -h` çıktısında "kullanılan" bellek beklediğinden yüksek görünür. Boş RAM boşa harcanmış RAM'dir; kernel onu cache olarak değerlendirir ve bir program bellek istediğinde cache'i anında geri verir.

Yazma tarafında varsayılan davranış **write-back**'tir: `write` çağrısı veri page cache'e kopyalandığında döner, disk yazımı sonra yapılır. Hızlıdır, ama güç kesilirse son yazılanlar kaybolabilir. Verinin gerçekten diskte olduğundan emin olmak gerektiğinde `fsync` çağrılır — veritabanlarının her commit'te ödediği bedel tam olarak budur.

> **Bir yana: `sync` ve düşen elektrik**
>
> Bir editörün “kaydedildi” demesi ile verinin fiziksel olarak diskte olması aynı an değildir. Aradaki pencere genelde milisaniyeler mertebesindedir ve `/proc/sys/vm/dirty_expire_centisecs` gibi ayarlarla yönetilir. Kritik veri yazan yazılımlar bu pencereyi `fsync` ile kapatır; kapatmayanlar ise elektrik kesintisinde tutarsız dosya bırakabilir.

DERİNLEŞME **fsyncgate: "diske yazdım" demek neden bu kadar zor?**

`fsync` çağırdın ve döndü. Verinin diskte olduğundan emin olabilir misin?

Sorunun cevabı 2018'e kadar herkesin sandığından farklı çıktı ve PostgreSQL geliştiricileri bunu zor yoldan öğrendi. Olay literatüre *fsyncgate* olarak geçti.

Mekanizma şöyleydi. `write` çağrısı veriyi page cache'e koyar ve döner; sayfa "kirli" işaretlenir. `fsync` bu kirli sayfaların diske inmesini ister. Peki disk yazımı **başarısız olursa** ne olur?

Linux'un o günkü davranışı şuydu: hata işaretlenir, sayfanın kirli bayrağı temizlenir ve hata **bir sonraki `fsync` çağrısına** bildirilir. Yalnızca bir kez. Sonraki `fsync` çağrıları başarı döner — çünkü kirli sayfa kalmamıştır.

Sonuç felaket bir bileşim yarattı: PostgreSQL bir `fsync` hatası gördüğünde, o zamanki tasarımıyla işlemi yeniden deniyordu. İkinci `fsync` başarı dönüyordu. Veritabanı "tamam, yazıldı" diyor ve checkpoint'i ilerletiyordu. Oysa veri hiçbir zaman diske inmemişti ve artık cache'te de yoktu. Sessiz veri kaybı.

İşin daha tuhaf tarafı: hata, dosyayı **o an açık tutan** file descriptor'lara bildiriliyordu. Yazan process ile `fsync` çağıran process farklıysa — ki PostgreSQL'in mimarisinde tam olarak öyleydi — hata hiç kimseye ulaşmayabiliyordu.

Sonrasında iki şey değişti. Kernel tarafında hata raporlaması sıkılaştırıldı; PostgreSQL tarafında ise mimari karar değişti: `fsync` hatası artık yeniden denenmez, süreç derhal `panic` eder ve veritabanı kurtarma moduna girer. Yani "tekrar dene" yerine "asla devam etme".

Buradan çıkarılacak iki ders var. Birincisi teknik: **kalıcılık (durability) bir çağrının dönüş değeri değil, bir protokoldür.** Veritabanlarının write-ahead log, checksum ve kurtarma prosedürü tutmasının sebebi budur.

İkincisi daha genel: bir soyutlama en çok **hata yolunda** sızar. `write`/`fsync` arayüzü mutlu yolda kusursuz görünüyordu; sözleşmenin belirsiz olduğu yer, işler ters gittiğinde ne

olacağıydı. Bu, sistem tasarımında tekrar tekrar karşına çıkacak bir kalıptır.

Buffered, Direct ve mmap

Aynı dosyayı üç ayrı yoldan okuyabilirsin ve hangisini seçtiğin, programının ne kadar hızlı çalışacağını doğrudan belirler. Üçünü sırayla tanıyalım.

Buffered I/O (varsayılan): veri diskten page cache'e, oradan process'in tamponuna kopyalanır. İki kopya vardır ama cache isabetleri bunu fazlasıyla telafi eder.

Direct I/O (`O_DIRECT`): page cache atlanır, veri doğrudan process'in tamponuna gider. Yalnızca kendi cache'ini yöneten yazılımlar için mantıklıdır — veritabanları genelde bu grubtadır, çünkü hangi sayfanın sıcak olduğunu kernel'dan daha iyi bilirler. Yanlış yerde kullanıldığında performansı çökertir.

Bellek eşleme (`mmap`): dosya doğrudan process'in adres alanına eşlenir. Artık `read` çağrısı yoktur; dosyaya sıradan bir dizi gibi erişilir ve eksik sayfalar **[13. bölümde]** gördüğümüz demand paging ile page fault üzerinden getirilir.

mmap-ornegi.c

```
int fd = open("veri.bin", O_RDONLY);
struct stat st;
fstat(fd, &st);

// Dosya artık bellekteymiş gibi: veri[0] okumak page fault üretir
char *veri = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
```

`mmap` kopya sayısını azaltır ve rastgele erişimde çok rahattır. Buna karşılık her page fault bir kernel geçişidir; sıralı ve büyük okumalarda düz `read` çoğu zaman daha hızlıdır.

Aynı mekanizma **[12. bölümde]** gördüğümüz program yüklemenin de temelidir: kernel bir ELF'i çalıştırırken dosyayı okumaz, `mmap`'ler.

OverlayFS: Katmanları Üst Üste Bindirmek

Şimdiye kadar tek bir dosya sisteminden söz ettik. Peki birden fazla dizini **tek bir dizinmiş gibi** göstermek mümkün mü?

Mümkün ve bunun adı **OverlayFS**. VFS'in en zarif kullanımlarından biridir: gerçek bir depolama sürücüsü değildir, kendi diski yoktur; yaptığı tek şey başka dosya sistemlerinin üstüne oturup onları birleşik bir görünüm hâlinde sunmaktır.

Mount ederken üç dizin verirsin, dördüncüsü sonucun göründüğü yerdir:

Dizin	Rolü
lowerdir	Salt okunur alt katman ya da katmanlar. Birden fazlaysa üst üste bindirilir.
upperdir	Yazılabilir katman. Bütün değişiklikler buraya gider.
workdir	OverlayFS'in kendi iç işlemleri için kullandığı geçici alan.
merged	Kullanıcının gördüğü birleşik görünüm.

Okuma isteği geldiğinde OverlayFS dosyayı önce `upperdir`'de arar, bulamazsa alt katmanlara iner. İlk bulduğu kazanır; bu yüzden üst katman, alttakini "gölgeleyebilir".

Yazma tarafı ise doğrudan **[16. bölümde]** gördüğümüz fikre bağlanıyor. Salt okunur bir katmandaki dosyayı değiştirmek istediğinde OverlayFS onu önce `upperdir`'e kopyalar, sonra değişikliği orada yapar. Bu bir **copy-on-write** işlemidir ve `fork`'un bellek sayfaları için yaptığının dosya düzeyindeki karşılığıdır: *kopyalamayı gerçekten gerekene kadar ertele.*

Sonuç, aynı salt okunur tabanın onlarca farklı yazılabilir katmanla paylaşılabilmesidir. Bir sonraki bölümde göreceğimiz container imajlarının hem az yer kaplamasının hem de saniyeler içinde başlayabilmesinin sebebi tam olarak budur.

Bloklanan I/O ve Ölçek Sorunu

Sıradan bir `read` çağrısı veri hazır değilse **bloklar**: process uyku durumuna geçer, scheduler CPU'yu başkasına verir. Tek bir bağlantı için bu gayet iyidir.

Peki on bin eşzamanlı bağlantıyı dinleyen bir sunucu? Her bağlantıya bir thread ayırmak, on bin stack ve sürekli context switch demektir — **[7. bölümde]** gördüğümüz gibi her geçişin bir bedeli vardır.

Çözüm, "hangisi hazır olursa onu işle" diyebilmektir. Bu fikrin Linux'taki evrimi:

- `select` / `poll` — ilgilendiğin tüm fd'leri her çağrıda kernel'a yeniden verirsin. Kernel hepsini tek tek tarar; maliyet bağlantı sayısıyla doğru orantılı büyür.
- `epoll` — ilgilendiğin fd'leri kernel'da **bir kez** kaydedersin, sonra yalnızca hazır olanların listesini alırsın. Maliyet bağlantı sayısına değil, o an aktif olanların sayısına bağlıdır. Nginx ve Node.js'in altında bu vardır.
- `io_uring` — daha ileri bir adım: process ile kernel arasında paylaşılan iki halka tampon kurulur. Yüksek yükte syscall sayısı neredeyse sıfıra iner.

`epoll` Nasıl Çalışıyor?

Bugün internetin büyük kısmı bu üç çağrının üzerinde dönüyor, o yüzden açalım.

- `epoll_create1()` — kernel'da bir izleme listesi oluşturur ve sana onun fd'sini verir. Evet, listenin kendisi de bir dosya.
- `epoll_ctl()` — listeye fd ekler, çıkarır ya da ilgilendiğin olayı değiştirir. Her fd için **bir kez** çağırırsın.
- `epoll_wait()` — bloklanır ve yalnızca hazır olan fd'lerin listesiyle döner.

Bir olay döngüsünün iskeleti bundan ibarettir:

```
int ep = epoll_create1(0);
struct epoll_event ev = { .events = EPOLLIN, .data.fd = sock };
epoll_ctl(ep, EPOLL_CTL_ADD, sock, &ev);

struct epoll_event olaylar[64];
for (;;) {
  int n = epoll_wait(ep, olaylar, 64, -1);   // sadece hazir olanlar
  for (int i = 0; i < n; i++)
    isle(olaylar[i].data.fd);
}
```

On bin bağlantın olsa ve bunlardan yalnızca üçünde veri gelse, `epoll_wait` sana üç eleman döner. `select` aynı durumda on bin fd'yi baştan sona tarardı. Aradaki fark, C10K probleminin çözümünün adıdır.

İki çalışma kipi var ve aralarındaki fark başa çok iş açar. **Level-triggered** (varsayılan) kipte, fd'de okunmamış veri kaldığı sürece `epoll_wait` onu tekrar tekrar bildirir. **Edge-triggered** kipte ise yalnızca *durum değiştiğinde* bir kez bildirir; veriyi `EAGAIN` alana kadar okumazsan kalan kısım için ikinci bir uyarı gelmez ve bağlantı sessizce asılı kalır. Edge-triggered daha az sistem çağrısı demektir ama hata affetmez.

Son bir uyarı: `epoll` **düzenli dosyalarda işe yaramaz.** Bir disk dosyası kernel'ın gözünde her zaman "hazır"dır; okuma gerekiyorsa zaten bloklanır. `epoll`'ün anlamlı olduğu yer soketler, pipe'lar ve terminaller gibi gerçekten beklenebilen nesnelerdir. Dosya I/O'sunu asenkron yapmak isteyenlerin `io_uring`'e yönelmesinin sebeplerinden biri de bu.

`io_uring` Syscall'ı Nasıl Ortadan Kaldırıyor?

`epoll` çağrı sayısını azaltır ama sıfırlayamaz: her `epoll_wait` ve her `read` yine bir syscall'dır. `io_uring` bu son maliyeti de hedefler.

Fikir, **halka tampon** üzerine kurulu: başı ve sonu birbirine değen sabit boyutlu bir dizi; bir taraf yazar, diğer taraf okur. `io_uring` böyle iki halka kurar — biri istekler (submission), biri sonuçlar (completion) için.

Kritik ayrıntı şu: bu iki halka `mmap` ile hem process'in hem kernel'ın adres alanında görünür. Yani **aynı fiziksel bellek**. İstek eklemek bir syscall değil, sıradan bir bellek yazımıdır. Syscall yalnızca kernel'ı "bak, yeni iş var" diye uyandırmak gerektiğinde atılır. `SQPOLL` kipinde kernel tarafında bir thread halkayı sürekli yokladığı için o son syscall bile ortadan kalkar.

Sonuç, yüksek yükte saniyede yüz binlerce işlemin tek bir syscall bile atılmadan tamamlanabilmesi.

`epoll`'ün "hazır olanı bildir" mantığı, çoğu modern async runtime'ının (libuv, tokio) temelidir. Yani JavaScript'te yazdığın `await fetch(...)` ifadesinin altında, döngünün bir yerinde bu mekanizma çalışıyordur.

Özet

- Linux'ta dosya, aygıt, soket ve process bilgisi aynı `open`/`read`/`write` arayüzünün arkasındadır.
- File descriptor, process'e özel bir indekstir; açık dosya tanımı offset'i tutar ve `fork` sonrasında paylaşılır, inode ise dosyanın kendisidir.
- VFS, farklı dosya sistemlerini tek bir fonksiyon tablosu arkasında birleştirir.
- Page cache disk erişimini RAM hızına yaklaştırır; `write` varsayılan olarak diske değil cache'e yazar, garanti isteyen `fsync` çağırır.
- `mmap` dosyayı adres alanına eşler ve program yüklemenin de temelidir.
- Bloklanan I/O bağlantı başına thread gerektirir; `epoll` ve `io_uring` bu maliyeti ortadan kaldırır.
- **OverlayFS**, birden fazla dizini tek bir görünümde birleştirir ve yazmayı copy-on-write ile üst katmana alır.

Artık tek bir makinede olan biten her şeyi gördük: donanımdan process'e, bellekten dosyaya. Sıradaki bölümde bu resmi bir kez daha katlıyor ve modern altyapının üstünde durduğu iki fikre bakıyoruz: sanal makineler ve container'lar.

Bölüm 18: Sanal Makineler ve Container'lar

Buraya kadar hep tek bir bilgisayardan söz ettik: bir donanım, üstünde bir kernel, onun üstünde process'ler. Şimdi bu resmi bir kez daha katlayacağız.

Buluttan bir sunucu kiraladığında sana verilen şey fiziksel bir makine değil. Bir veri merkezinde duran çok daha büyük bir makinenin, senin için ayrılmış bir dilimi. O dilim kendini gerçek bir bilgisayar sanıyor: kendi kernel'ını açıyor, kendi diskini biçimlendiriyor, kendi ağ kartını görüyor. Hepsi doğru değil — ama hiçbiri de yalan sayılmaz.

Aynı işi yapmanın ikinci bir yolu daha var ve bugün daha yaygın olanı o: **container**. İkisi de sana "kendi makinen" hissi verir, ama sınırı bambaşka yerlerden çizerler. Bu bölümde ikisini de söküp bakacağız.

Güzel tarafı şu: yeni bir mekanizma öğrenmeyeceğiz. Kitap boyunca gördüğümüz üç fikir — ayrıcalık seviyeleri, adres çevirisi ve interrupt — bir kez daha, bu sefer bir kat yukarıda uygulanacak.

> **Bu bölümde neyi çözüyoruz?**
>
> - Bir işletim sisteminin başka bir işletim sisteminin içinde nasıl çalışabildiğini göreceğiz.
> - Hypervisor'ın ne olduğunu ve belleğin neden iki kez çevrildiğini anlayacağız.
> - Container'ın neden bir sanal makine olmadığını, kernel'ın hangi araçlarla bu izolasyonu kurduğunu çözeceğiz.
> - `docker run` yazdığında arkada sırayla ne olduğunu izleyeceğiz.

Temel Fikir: Kernel'a Kernel Olmadığını Söylememek

Bir işletim sistemi, donanımın tek sahibi olduğu varsayımıyla yazılır. Kendini kernel mode'da sanır, sayfa tablolarını kendi kurar, disk denetleyicisine doğrudan komut yazar.

Sanallaştırmanın bütün numarası şu: **bu varsayımı bozmadan yalan söylemek.**

Araya bir katman koyarsın. O katman fiziksel donanımın gerçek sahibidir. Üstünde çalışan işletim sistemi ayrıcalıklı bir iş yapmaya kalktığında — mesela sayfa tablosu register'ını değiştirmeye ya da diske komut yazmaya — işlem gerçekten donanıma gitmez. Araya giren katman onu yakalar, taklit eder ve sonucu sanki gerçekten olmuş gibi geri verir.

Bu araya giren katmanın adı **hypervisor** (ya da virtual machine monitor). Üstünde çalışan işletim sistemine **guest**, altındaki fiziksel makineye **host** denir.

Mekanizmanın çekirdeği **[3. bölümde]** tanıştığımız fikrin ta kendisi. Orada user mode'daki bir programın ayrıcalıklı bir talimat çalıştırmaya kalkması durumunda CPU'nun bunu reddedip kernel'a atladığını görmüştük. Sanallaştırmada aynı şey bir kat yukarıda olur: guest kernel ayrıcalıklı bir talimat çalıştırır, CPU bunu yakalar ve kontrol hypervisor'a geçer. Hypervisor işi taklit eder, guest hiçbir şey fark etmez.

Bu düzene **trap-and-emulate** denir ve sanallaştırmanın tek cümlelik özetidir: *yakala, taklit et, geri ver.*

İki Tür Hypervisor

Hypervisor'ın nerede durduğuna göre iki aile var.

Tip 1 — doğrudan donanımın üstünde. Altında bir işletim sistemi yoktur; hypervisor'ın kendisi zaten bir çeşit kernel'dır. Sunucu ve bulut tarafında kullanılan budur: VMware ESXi, Microsoft Hyper-V, Xen. Aradan bir katman kalktığı için hem daha hızlıdır hem de saldırı yüzeyi daha dardır.

Tip 2 — normal bir işletim sisteminin üstünde. Kendi makinende çalıştırdığın VirtualBox, VMware Workstation ya da UTM bu gruptan. Guest'i sıradan bir uygulama gibi başlatırsın; host işletim sistemi çalışmaya devam eder.

Linux'un **KVM**'i ikisinin arasında duran ilginç bir örnek. KVM ayrı bir hypervisor değil, Linux kernel'ının kendisine eklenen bir modüldür. Yani Linux, modül yüklendiği anda tip 1 hypervisor'a dönüşür; ama aynı anda sıradan bir Linux olmaya da devam eder. Bulut sağlayıcılarının büyük kısmı bu modeli kullanır.

> Windows'ta WSL2 ve Docker Desktop, macOS'ta Docker Desktop — hepsi arka planda sessizce bir sanal makine açar. Linux olmayan bir makinede "container çalıştırıyorum" dediğinde, aslında önce bir Linux sanal makinesi çalışıyor demektir.

x86'nın Sanallaştırmayla İmtihanı

Trap-and-emulate kulağa temiz geliyor ama bir şartı var: **ayrıcalıklı her talimatın gerçekten trap etmesi gerekir.**

x86 uzun süre bu şartı sağlamadı. Mimaride, user mode'da çalıştırıldığında hata vermek yerine sessizce farklı — ve yanlış — davranan talimatlar vardı. Guest kernel böyle bir talimatı çalıştırdığında hypervisor devreye giremiyor, guest de yanlış bir cevap alıp yoluna devam ediyordu. Yakalanamayan bir yalan, yalanın tamamını bozar.

Sektör bu duvarı iki farklı yoldan aştı:

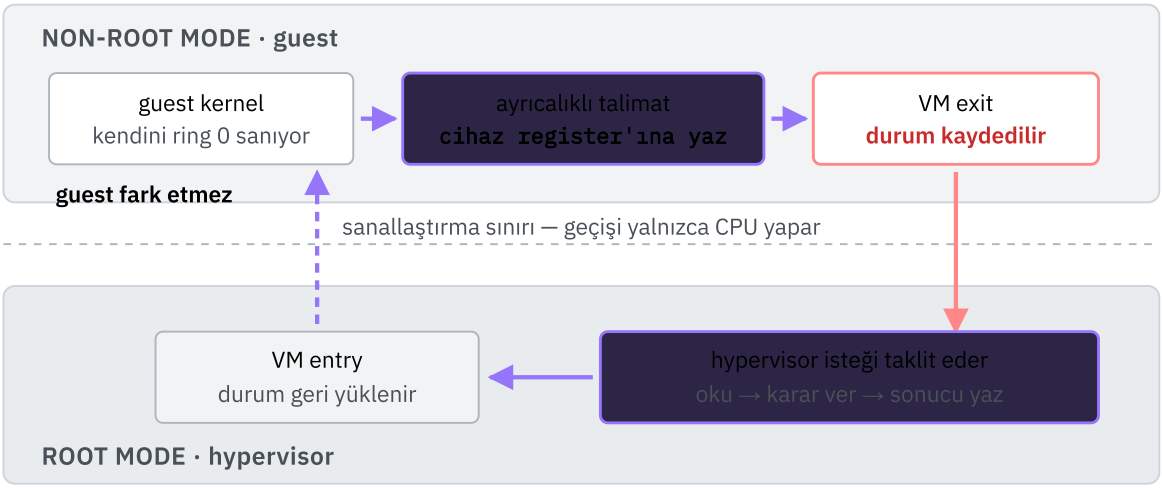
- **Binary translation.** VMware'in yolu buydu: guest kernel'ın makine kodunu çalışmadan önce tarayıp sorunlu talimatları güvenli karşılıklarıyla değiştirmek. İşe yarıyordu ama karmaşık ve pahalıydı.
- **Paravirtualization.** Xen'in yolu: guest kernel'ı değiştirmek. Guest, sanallaştırıldığını *bilir* ve sorunlu talimatları hiç çalıştırmaz; onun yerine hypervisor'a doğrudan çağrı yapar. Hızlıdır ama guest'in kaynak kodunu değiştirebilmeyi gerektirir — kapalı kaynak bir işletim sistemi için işe yaramaz.

Kalıcı çözüm 2005-2006'da donanımdan geldi: Intel **VT-x**, AMD **AMD-V**. İkisi de CPU'ya yepyeni bir eksen ekledi.

Şöyle düşün: artık ring 0 ile ring 3 ayrımının *yanına* ikinci bir ayrım kondu. CPU ya **root mode**'dadır (hypervisor burada çalışır) ya da **non-root mode**'da (guest burada çalışır). Guest kendi içinde hâlâ ring 0 ve ring 3 kullanır — yani guest kernel gerçekten kendini ring 0'da görür, yalana gerek kalmaz. Ama tüm o non-root dünya, hypervisor'ın çizdiği sınırların içindedir.

Guest ayrıcalıklı bir iş yapmaya kalktığında CPU **VM exit** üretir: non-root'tan çıkılır, kontrol hypervisor'a geçer. Hypervisor işini bitirince **VM entry** ile geri döner. Adım adım izleyelim:

BİR VM EXİT VE DÖNÜŞÜ



1. **Guest çalışıyor.** Sanal makinenin içindeki kernel non-root mode'da yürüyor. Kendi içinde ring 0'dadır ve sanallaştırıldığının farkında değildir.
2. **Ayrıcalıklı bir iş isteniyor.** Guest kernel gerçek donanıma dokunmaya kalkıyor — mesela bir cihaz register'ına yazmaya. Fiziksel makinede bu iş gerçekten yapılırdı.
3. **VM exit.** CPU işlemi donanıma geçirmez. Guest'in bütün görünür durumunu bir kenara kaydeder, root mode'a çıkar ve kontrolü hypervisor'a verir.
4. **Hypervisor taklit eder.** İsteği okur, kendi kurallarına göre karşılar ve sonucu guest'in göreceği yere yazar. Gerçek donanıma dokunup dokunmayacağına o karar verir.
5. **VM entry.** Guest'in durumu geri yüklenir ve CPU non-root mode'a döner. Guest için hiçbir şey olmamıştır: istediği işin sonucu elindedir.

Bu iki geçiş, syscall'ın kernel/user geçişinin bir üst kattaki karşılığıdır — ve tıpkı onun gibi, sayısı arttıkça performansı yer.

DERİNLEŞME **VM exit neden performansın anahtarı?**

Sanal makine performansını konuşurken asıl ölçtüğün şey neredeyse her zaman VM exit sayısıdır.

Sebep şu: her exit'te CPU'nun guest'in bütün görünür durumunu bir kenara kaydedip hypervisor'ınkini yüklemesi gerekir. Register'lar, ayrıcalık durumu, adres çevirisi ayarları… **[7. bölümde]** konuştuğumuz context switch'in daha ağır bir versiyonu.

Bu yüzden sanallaştırma tarihinin son yirmi yılı, büyük ölçüde "hangi işi exit üretmeden yapabiliriz?" sorusunun cevabıdır. Adres çevirisini donanıma taşımak, zamanlayıcı okumalarını guest içinde halletmek, ağ ve disk için exit toplayıp tek seferde işlemek — hepsi aynı hedefe hizmet eder.

Pratik sonucu şu: CPU-yoğun bir iş yükü sanal makinede fiziksel makineye çok yakın hızda koşar, çünkü hesap yaparken exit üretmez. I/O-yoğun bir iş yükü ise aradaki farkı acı biçimde hisseder.

Belleğin İki Kez Çevrilmesi

Şimdi işin en güzel kısmına geldik, çünkü doğrudan **[14. bölümde]** öğrendiğimiz şeyin üstüne biniyor.

Bir sanal makinede iki ayrı adres çevirisi vardır:

1. Guest'in içindeki bir program sanal adres kullanır. Guest kernel, kendi sayfa tablolarıyla bunu bir **guest fiziksel adrese** çevirir.
2. Ama o "fiziksel" adres de gerçek değildir. Guest'in RAM sandığı şey, host'un bakış açısından sadece bir bellek bloğudur. Dolayısıyla ikinci bir çeviri daha gerekir: guest fiziksel adres → **host fiziksel adres**.

İlk yıllarda bu ikinci çeviriyi hypervisor yazılımla yapıyordu. **Shadow page table** denen yöntemde hypervisor, guest'in sayfa tablolarını sürekli izler ve ikisini birleştiren gizli bir tablo bakımı yapardı. Guest tablosuna her dokunduğunda bir VM exit; yani sürekli maliyet.

Donanım burada da imdada yetişti. **EPT** (Intel) ya da **NPT** (AMD) ile MMU artık iki seviyeli çeviriyi kendi başına yapabiliyor: guest'in tablosunu yürüyor, çıkan sonucu ikinci bir tabloda tekrar yürüyor, host fiziksel adresi buluyor. Hypervisor'ın araya girmesine gerek kalmıyor.

Bedeli ise artan yürüyüş maliyeti. Normalde dört seviyeli bir sayfa yürüyüşü, sanallaştırmada her seviyede ikinci bir yürüyüş daha tetikleyebilir. Kötü senaryoda tek bir çeviri için yirmiden fazla bellek erişimi. TLB'nin sanal makinelerde neden bu kadar kritik olduğunu da bu açıklıyor — TLB tuttuğu sürece bu yürüyüşün hiçbiri yaşanmıyor.

> Büyük sayfaların (huge page) sanal makinelerde önerilmesinin sebebi tam olarak bu. Yürüyüş maliyeti iki katına çıkmışken, yürüyüş sayısını azaltmak iki kat değerli hâle geliyor.

Sahte Donanım: Cihazlar Nereden Geliyor?

Guest'in gördüğü disk ve ağ kartı da gerçek değil. Burada üç yaklaşım var ve üçü de farklı bir takas yapıyor.

Öykünme (emulation). Hypervisor, gerçekten var olan eski bir donanımın davranışını yazılımla taklit eder — mesela yıllardır üretilmeyen bir ağ kartını. Avantajı, guest işletim sisteminin sürücüsünün zaten elinde olması; hiçbir şey kurmana gerek kalmaz. Dezavantajı, her register erişiminin bir VM exit üretmesi. Yavaştır.

Paravirtualize cihazlar (virtio). Guest'e "bu gerçek bir donanım değil, o yüzden gerçek donanım gibi davranmayalım" denir. Guest'e virtio sürücüsü kurulur ve hypervisor ile guest, paylaşılan bellek üzerinde bir halka tampon üzerinden konuşur. **[17. bölümde]** `io_uring` için anlattığım fikrin aynısı: isteği belleğe yaz, karşı taraf oradan alsın, syscall'a — burada exit'e — gerek kalmasın. Bugün bulutta gördüğün disk ve ağ neredeyse her zaman virtio'dur.

Doğrudan geçiş (passthrough). Fiziksel bir cihaz tamamen guest'e verilir; hypervisor aradan çıkar. En hızlı yol budur ve GPU ile yüksek performanslı ağ kartlarında kullanılır. Bedeli esneklik: o cihaz artık başka hiçbir guest tarafından kullanılamaz, makineler arası taşıma da zorlaşır.

Container: Donanımı Değil, Görüntüyü Taklit Etmek

Şimdi ikinci yola geçelim ve baştan net bir cümleyle başlayalım: **container bir sanal makine değildir.** Hatta daha da net bir şey söyleyeyim — kernel'da "container" diye bir özellik yoktur. Ne böyle bir syscall vardır, ne böyle bir veri yapısı.

Sanal makinede hypervisor donanımı taklit eder ve guest kendi kernel'ını açar. Bedeli, her sanal makinenin bir bilgisayarın bütün masrafını yeniden ödemesidir: kendi kernel'ı, kendi sürücüleri, kendi açılış süreci.

Container'da böyle bir şey yok. Aynı Linux makinesindeki bütün container'lar **tek bir kernel'ı paylaşır**. Kernel her birine "senin göreceğin process listesi bu, senin ağın bu, senin kök dizinin burası" der ve iş biter. Yeni bir kernel açılmadığı için başlatma maliyeti neredeyse sıfırdır; çalıştırdığın şey en baştan beri sıradan bir Linux process'iydi.

Bunu somutlaştıran şu deneyi hayal et: bir container'ın içinde `ps` çalıştırdığında iki üç process görürsün. Aynı anda host makinede `ps` çalıştırırsan, o container'ın process'lerini diğer bütün process'lerin arasında, sıradan process'ler olarak görürsün. Aynı process'ler, iki farklı pencereden bakıldığında iki farklı dünya. **Container dediğimiz şey tam olarak o pencerenin kendisi.**

Kernel Bu Pencereyi Nasıl Kuruyor?

Pencereyi kuran dört araç var ve dördü de container'lar icat edilmeden önce, bambaşka işler için yazılmıştı.

Namespace: Kim Neyi Görsün?

Namespace (ad alanı), kernel'ın aynı kaynağı farklı process gruplarına farklı görünür hâle getirmesidir. Linux'ta birkaç tür vardır ve her biri tek bir şeyi izole eder:

Namespace	Ne izole eder?	Container içinde sonucu
PID	Process numaraları	İçerideki ilk process kendini PID 1 sanır

Namespace	Ne izole eder?	Container içinde sonucu
NET	Ağ arayüzleri ve portlar	Kendi `eth0`'ı ve kendi IP'si olur
MNT	Mount noktaları	Kendi kök dosya sistemini görür
UTS	Hostname	Kendi makine adını belirleyebilir
IPC	Process'ler arası iletişim	Paylaşılan bellek alanları ayrışır
USER	Kullanıcı ve grup numaraları	İçeride root görünen kullanıcı dışarıda sıradan bir kullanıcıdır

Bir process yeni bir namespace'e `clone` ya da `unshare` ile girer. Yani container başlatmak, aslında **[16. bölümde]** gördüğümüz process klonlamanın birkaç ek bayrakla yapılmış hâlidir.

Son satır özellikle önemli, çünkü modern container güvenliğinin dayandığı yer orası. **USER namespace** sayesinde container içindeki `root`, dışarıda hiçbir ayrıcalığı olmayan sıradan bir kullanıcıya eşlenebilir. İçeride `whoami root` der, dışarıda o process'in host üzerinde hiçbir yetkisi yoktur. "Rootless container" denen şey budur ve container'dan kaçmayı ciddi biçimde zorlaştırır.

cgroup: Kim Ne Kadar Tüketsin?

Namespace "kim ne görür"ü çözer ama "kim ne kadar tüketir"i çözmez. Görünürlüğü kısıtlanmış bir process hâlâ bütün RAM'i yiyebilir.

cgroup (control group) tam olarak bunu sınırlar: bir process grubunun kullanabileceği CPU zamanını, belleği, disk bant genişliğini ve hatta açabileceği process sayısını.

Arayüzü şaşırtıcı derecede sade, çünkü dosya sistemi kılığında sunulur. `/sys/fs/cgroup` altında bir dizin oluşturmak yeni bir grup açmak, o dizindeki bir dosyaya sayı yazmak da sınır koymak demektir. `docker run --memory=512m` yazdığında olan biten tam olarak budur: runtime bir cgroup açar ve `memory.max` dosyasına `536870912` yazar.

> Buradaki `mkdir` alıştığın `mkdir` değil. `/sys/fs/cgroup` diskte yer kaplamaz; kernel'ın kendi veri yapılarını dosya kılığında sunduğu sanal bir dosya sistemidir. Bir dizin oluşturduğunda kernel bellekte yeni bir cgroup nesnesi yaratır. Makine kapandığında hepsi buharlaşır — zaten hiçbiri diske yazılmıyordu.

Capability: Root Yetkisini Parçalara Ayırmak

Geleneksel Unix'te bir kullanıcı ya root'tur (her şeyi yapar) ya değildir (çok az şey yapar). **Capability**'ler bu ikiliği parçalar.

Mekanizma sandığından ince: capability'ler kullanıcıya değil, tek tek **thread'lere** iliştirilmiş bit kümeleridir. Kernel `mount` gibi ayrıcalıklı bir işlem istendiğinde "bu kullanıcı root mu?" diye bakmaz; "bu thread'in yetki kümesinde `CAP_SYS_ADMIN` var mı?" diye bakar. Root olmak, artık bu bitlerin hepsinin açık olmasının kısa adıdır.

Container runtime'ları bir container'ı başlatırken bu bitlerin çoğunu düşürür. Sonuç, içeride kendini root sanan ama sistem saatini değiştiremeyen, kernel modülü yükleyemeyen ve yeni bir dosya sistemi mount edemeyen bir process'tir.

seccomp: Hangi Kapılar Açık Kalsın?

Son katman en dar olanı. **seccomp**, bir process'in hangi syscall'ları yapabileceğini kısıtlar. Runtime, container'ı başlatmadan önce kernel'a bir filtre verir; o filtre bundan sonraki her syscall'dan önce çalışır ve tek bir karar döndürür: izin ver, hata döndür ya da process'i öldür.

İki özelliği tasarımın tamamını belirler. **Tek yönlüdür** — bir kez uygulanan filtre kaldırılamaz, yalnızca daraltılabilir; bu yüzden bir process kendi ayrıcalığını güvenle düşürebilir. Ve **miras alınır** — `fork` ve `execve` sonrası geçerli kalır; bu yüzden runtime filtreyi kurup ardından asıl uygulamayı `exec` ile başlatır, uygulama hiçbir şey yapmadan kısıtlanmış olarak doğar.

Kazanç saldırı yüzeyini daraltmaktır: uygulamanın kendisi ele geçirilse bile kernel'a ulaşabildiği kapı sayısı azalmıştır.

Container İmajı Nedir?

Günlük hayatta en çok karşılaştığın parçayı da yerine oturtalım: **imaj**.

Bir container imajı, bir programın çalışması için gereken her şeyi içeren dondurulmuş bir dosya sistemidir. Programın kendisi, bağlı olduğu kütüphaneler, yapılandırma dosyaları, hatta `/etc` ve `/usr` dizinlerinin tamamı. İçinde olmayan tek şey kernel'dır — çünkü onu host'tan ödünç alacak.

Bunu kurulum paketiyle karıştırmamak önemli. Bir `.deb` ya da `.msi` paketi sisteme *dokunur*: dosyaları sistemin dizinlerine dağıtır ve o andan sonra sistemin hâline bağımlı olur. İmaj ise sisteme hiç dokunmaz; kendi dosya sistemini yanında getirir. "Bende çalışıyordu ama sunucuda çalışmıyor" cümlesinin ortadan kalkmasının sebebi tam olarak bu.

İmajların ikinci özelliği **katmanlı** olmalarıdır. Bir imaj tek bir blok değil, üst üste bindirilmiş birkaç salt-okunur katmandan oluşur: en altta işletim sisteminin temel dosyaları, üstünde kurduğun kütüphaneler, en üstte kendi uygulaman. Bunun iki somut kazancı var:

- **Paylaşım.** Aynı temel katmanı kullanan yirmi imaj, o katmanı diskte bir kez tutar.
- **Artımlı güncelleme.** Uygulamanın tek satırını değiştirdiğinde yalnızca en üst katman yeniden üretilir; alttaki yüzlerce megabaytlık katmanlar olduğu gibi kalır ve ağdan tekrar indirilmez.

Katmanların üst üste nasıl bindirildiğini **[17. bölümde]** OverlayFS başlığında görmüştük. Akılda tutulacak tek cümle şu: **imaj salt okunurdur, container ise onun üstüne eklenmiş ince ve yazılabilir bir katmandır.** Container'ı sildiğinde giden şey yalnızca o ince katmandır; imaj olduğu gibi kalır.

`docker run` Yazdığında Sırayla Ne Oluyor?

Artık bütün parçalar elimizde. Terminale `docker run -it --memory=512m ubuntu bash` yazdığın anda olan biteni baştan sona dizelim:

1. **İmaj bulunur.** Runtime önce imajın yerelde var olup olmadığına bakar; yoksa registry'den katman katman indirir.

2. **Katmanlar birleştirilir.** Salt-okunur katmanların üstüne yazılabilir ince bir katman eklenir ve hepsi tek bir dosya sistemi gibi sunulur.
3. **Yeni namespace'lerle bir process klonlanır.** `clone`, bayraklarına bakarak child'a yeni bir PID, ağ, mount ve hostname dünyası verir.
4. **Kaynak sınırları yazılır.** Runtime bu process için bir cgroup oluşturur ve `--memory=512m` gibi bayrakları o cgroup'un dosyalarına yazar.
5. **Kök dizin değiştirilir.** `pivot_root` ile process'in gördüğü `/`, imajdan açılan dosya sistemine taşınır. Host'un dosya sistemi artık onun için var değildir.
6. **Yetkiler kırpılır.** Gereksiz capability'ler düşürülür ve seccomp filtresi kurulur.
7. **Program çalıştırılır.** Son adım `execve`: process kimliğini değiştirir ve `bash` olur. **[10. bölümde]** adım adım izlediğimiz akışın aynısı çalışır.

Dikkat et: bu yedi adımın hiçbirinde "container oluştur" diye bir çağrı yok. Sıradan bir process başlatılıyor; sadece başlatılırken görebileceği dünya daraltılıyor.

Bu adımlar Docker'a da özgü değil. Tamamı **OCI** adı verilen ortak bir standartta tarif edilir; Docker, Podman ve containerd aynı standardı uygular. Bir imajı Docker ile üretip Kubernetes altında çalıştırabilmenin sebebi bu — ikisi de aynı tarifi okuyor.

DERİNLEŞME **Kubernetes pod'u tam olarak nedir?**

Pod, çoğu zaman "birkaç container'ın bir arada durması" diye anlatılır. Daha kesin bir tanım vermek mümkün: **pod, bir avuç container'ın hangi namespace'leri paylaşacağına dair bir karardır.**

Aynı pod'daki container'lar network, IPC ve UTS namespace'lerini ortak kullanır. Birbirlerine `localhost` üzerinden ulaşabilmelerinin sebebi budur — kernel'ın gözünde aynı ağ dünyasındadırlar. Buna karşılık her biri kendi PID ve mount namespace'inde durur; yani birbirlerinin process'lerini ve dosya sistemlerini görmezler.

Namespace'lerin iç içe geçmediğine dikkat et. Burada olan şey iç içe geçme değil, **paylaşma**. Pod diye ayrı bir kernel nesnesi yoktur; pod, bu paylaşım kararının Kubernetes tarafındaki adıdır.

Sanal Makine mi, Container mı?

İki yolu da gördük. Karşılaştırma tek bir cümleye iniyor: **sanal makine donanımı taklit eder, container işletim sisteminin görüntüsünü.**

	Sanal Makine	Container
İzolasyon sınırı	Donanım (hypervisor)	Kernel (namespace + cgroup)
Kernel	Her guest kendi kernel'ını çalıştırır	Hepsi aynı kernel'ı paylaşır
Başlangıç	Saniyeler (tam bir açılış)	Milisaniyeler
Ek yük	Guest kernel + guest servisleri	Neredeyse yok
Farklı işletim sistemi	Mümkün (Linux üstünde Windows)	Mümkün değil
Bir kernel açığının etkisi	Guest'te kalır	Bütün container'ları etkileyebilir

Seçim de buradan çıkıyor. Farklı bir işletim sistemine ihtiyacın varsa ya da güvenmediğin kod çalıştırıyorsan sanal makine gerekir; çünkü sınır donanımdadır ve aşmak çok daha zordur. Aynı kernel'a güvenebiliyorsan container hem daha hızlı hem daha ucuzdur.

Son yıllarda ikisi birbirine yaklaşıyor. **Firecracker** gibi mikro-VM'ler, guest'e verilen sahte donanımı asgariye indirerek sanal makine açılışını milisaniyelere düşürdü; AWS Lambda'nın altında bu var. Ters yönde ise **gVisor** gibi projeler, container'ın gördüğü kernel arayüzünü araya giren bir katmanla karşılayarak paylaşılan kernel riskini azaltıyor. İkisi de aynı soruya farklı uçlardan yaklaşıyor: *ne kadar izolasyona, ne kadar bedelle?*

Özet

> **Peki, ne öğrendik?**
>
> - Sanallaştırma yeni bir mekanizma değil; ayrıcalık seviyeleri, adres çevirisi ve interrupt fikirlerinin bir kat yukarıda tekrar uygulanmasıdır.
> - **Hypervisor**, guest'in ayrıcalıklı işlemlerini yakalayıp taklit eder. Tip 1 doğrudan donanım üstünde, tip 2 bir işletim sisteminin üstünde çalışır.
> - x86 uzun süre trap-and-emulate için uygun değildi; **VT-x / AMD-V** ile CPU'ya root/non-root ayrımı eklenince sorun donanımda çözüldü.
> - Bellek iki kez çevrilir. **EPT/NPT** bu ikinci çeviriyi donanıma taşıdı; bedeli, sayfa yürüyüşünün belirgin biçimde pahalılaşması.
> - **Container bir sanal makine değildir.** Kernel'da container diye bir özellik yoktur; namespace, cgroup, capability ve seccomp bir araya gelince ortaya çıkan şeye o adı veriyoruz.
> - **Namespace** görünürlüğü, **cgroup** tüketimi, **capability** ve **seccomp** ise yetkiyi kısıtlar.
> - **İmaj** salt okunur ve katmanlıdır; container, onun üstüne eklenen ince ve yazılabilir bir katmandır.
> - Fark izolasyon sınırının nerede çizildiğidir: donanımda mı, kernel'da mı.

Yolun sonuna geldik. Son bölümde bilgisayarın açılışından bir programın ilk talimatını yürüttüğü ana kadar olan zincirin tamamını tek parça hâlinde toparlayacağız.

Bölüm 19: Son Söz

Tebrikler. Artık "sen"i CPU'nun içine epey sağlam bir şekilde yerleştirmiş olduk. Umarım keyif almışsındır.

Kapanırken tekrar vurgulamak istediğim şey şu: Bu kitaptaki bilgiler gerçek ve yaşayan şeyler. Bir dahaki sefere bilgisayarının nasıl aynı anda birden fazla uygulama çalıştırabildiğini düşündüğünde, umarım timer chip'leri ve hardware interrupt'ları gözünün önüne gelir. Süslü bir programlama diliyle bir şey yazıp linker hatası aldığında, o linker'ın aslında ne yapmaya çalıştığını da biraz olsun hissedersin.

> **Bu bölümde neyi kapatıyoruz?**
>
> - Büyük resmi tekrar zihne yerleştiriyoruz.
> - Buradan sonra nereye gidileceğini konuşuyoruz.
> - Ana anlatıya sığmayan birkaç notu paylaşıyoruz.

Büyük Resim: Bilgisayarın Açılışından Program Yürütmeye

Şimdiye kadar öğrendiklerimizi tek bir zincirde özetleyelim. Bilgisayarını açtığında şu sırayla olur:

1. **Firmware (BIOS/UEFI)** donanımı başlatır ve bootloader'ı RAM'e yükler.
2. **Bootloader** (GRUB vb.) kernel'ı diskten bulur, RAM'e yükler ve çalıştırır.
3. **Kernel** (kernel mode'da) kendi sayfa tablolarını kurar, interrupt handler'larını hazırlar, driver'ları yükler.
4. Kernel **init process**'ini (PID 1) başlatır. Artık user space'teyiz.
5. **Init** (systemd, OpenRC vb.) servisleri ve grafik ortamını başlatır.
6. Bir program çalıştırmak istendiğinde çoğu Unix-benzeri akışta parent process `fork()`/`clone()` ailesiyle child üretir; bellek eşlemeleri COW sayesinde baştan kopyalanmaz.
7. Child `execve()` ile yeni programa dönüşür; kernel ELF dosyasını okur, programın kod ve veri parçalarını **sanal belleğe** yerleştirir ve program dinamik kütüphanelere ihtiyaç duyuyorsa dynamic loader'ı devreye sokar.

8. Programın gördüğü **sanal adresler**, **MMU + sayfa tabloları + TLB** üçlüsü tarafından fiziksel RAM adreslerine çevrilir; demand paging sayesinde bazı sayfalar ancak ilk erişildiklerinde RAM'e gelir.
9. CPU, user mode'da talimatları **fetch-execute cycle** ile yürütür; cache, pipeline, branch prediction ve çok çekirdek gibi teknikler bu yürütmeyi hızlandırır.
10. Sistem çağrısı gerektiğinde **syscall** giriş yolu ile kernel mode'a geçilir, iş bitince user mode'a dönülür. Program bir dosya okuduğunda istek **VFS** üzerinden ilgili dosya sistemine iner ve çoğu zaman diske hiç gitmeden **page cache**'ten karşılanır; ağa yazdığındaysa istek socket katmanından ağ yığınına, oradan da ağ kartının sürücüsüne geçer.
11. Birden fazla program aynı anda çalışıyorsa **timer interrupt** ve scheduler başka process/thread'e geçer (**context switch**).
12. Bütün bu resim bir kez daha katlanabilir: **hypervisor** aynı donanımın üstünde birden fazla kernel çalıştırır, **container**'lar ise tek bir kernel'ın üstüne namespace, cgroup, seccomp ve capability katmanları ekleyerek izolasyon kurar.
13. Tüm bu akış boyunca **NX, stack canary, ASLR, PIE ve RELRO** arka planda çalışır; bir bellek hatası olduğunda onu sömürmeyi zorlaştıran katmanlar bunlardır.

> Bu zincirin her halkasını artık biliyorsun. Kendi kendine "Peki ya...?" diye sormaya devam et; işin en güzel yanı her cevabın yeni bir soru getirmesi.

Buradan Sonra Nereye?

Bu kitap bir haritaydı; her kutunun içi kendi başına bir alan. Merakının çektiği yöne göre birkaç somut yol:

- **Ölçmeye devam et.** Bu kitapta öğrendiğin her mekanizmanın kendi makinende bir izi var. `strace` bir programın yaptığı syscall'ları, `perf` işlemcinin donanım sayaçlarını, `/proc` ise kernel'ın o process hakkında bildiği her şeyi gösterir. Bir web sunucusunu `strace -c` ile izlemek, çoğu teoriden daha öğreticidir.
- **Kernel kaynağını aç.** Yazı boyunca bağlantı verdiğimiz dosyalar gerçek: `fs/exec.c` içindeki `do_execveat_common`, `kernel/fork.c` içindeki `copy_process`. Artık bu fonksiyonların ne yaptığını biliyorsun, okumak sandığından kolay.

- **Küçük bir şey yaz.** Kendi shell'ini (`fork` + `execve` + `waitpid`), kendi `malloc`'unu ya da `LD_PRELOAD` ile bir fonksiyonu araya giren küçük bir kütüphane. Yüz satırlık bir deneme, bin satırlık okumadan fazlasını öğretir.
- **Diğer tarafa bak.** Bu anlatı Unix dünyasına ait. Windows'un NT çekirdeği, macOS'un XNU'su ve WebAssembly gibi yeni yürütme modelleri aynı soruları farklı cevaplarla karşılıyor.

Klasik kaynaklar için: Kerrisk'in *The Linux Programming Interface*'i user space tarafının başvuru kitabı, Bovet ve Cesati'nin *Understanding the Linux Kernel*'i kernel tarafının klasiği, Gregg'in *Systems Performance*'ı ise ölçme ve performans konusunda alanın standardıdır.

Bu kitabın aslı Lexi Mattick ve Hack Club'ın eseri; Türkçe adaptasyonu ve güncel notları ben yazdım. İçerikte bir hata, bir eksik ya da daha iyi anlatabileceğim bir yer görürsen Türkçe adaptasyon reposunda issue veya PR aç, istersen doğrudan bana yaz. Özgün metnin kendisiyle ilgili geri bildirimleri ise kaynak projenin kendi iletişim kanallarına iletmen daha doğru olur.

... ama dur, daha bitmedi.

Bonus: Küçük Notlar

Şimdi de ana anlatının akışına sığmayan, ama bilmene değecek birkaç not. Bunları kasten sona sakladım.

> *Linux kullanıcılarının çoğunun yeterince ilginç bir hayatı vardır; page table'ların kernel içinde nasıl temsil edildiğini hayal etmeye fazla zaman ayırmazlar.*
>
> *Jonathan Corbet, LWN*

Hardware interrupt'lar için alternatif bir görselleştirme:

Bazı syscall'lar kernel space'e hiç sıçramaz. Bunun adı **vDSO** (virtual dynamic shared object) ve kitabın syscall bölümünü okuduktan sonra mantığı çok net.

Kernel, her process'in adres alanına küçük bir paylaşımlı kütüphane eşler. İçinde `clock_gettime` ve `gettimeofday` gibi birkaç fonksiyon vardır ve bu fonksiyonların okuduğu veri — mesela o anki zaman — kernel'ın sürekli güncellediği paylaşımlı bir sayfada durur. Yani program saati sorduğunda kernel'a girmez; kendi adres alanındaki bir sayfadan okur ve çıkar.

Kazanç, **[2. bölümde]** konuştuğumuz mode geçişinin tamamen ortadan kalkması. Saniyede milyonlarca kez zaman soran bir programda bu fark ölçülebilir bir hızlanmadır — nitekim vDSO'nun var olma sebebi de tam olarak böyle programlardır.

Ayrıntısına girmek istersen: Wikipedia özeti, `vdso(7)` man sayfası ve linux-insides bölümü.

Unix’e özgü detaylar konusunda son bir dürüstlük notu: anlatının önemli bir kısmı Unix dünyasına ait. macOS ya da Linux kullanıyorsan bu gayet iyi, ama aynı CPU mimarisi üzerinde dursa bile Windows’un programları nasıl yürüttüğü ya da syscall’ları nasıl yönettiği hakkında sana doğrudan aynı resmi vermez. İleride bu işin Windows tarafını anlatan ayrı bir yazı yazmayı çok isterim.