# Bölüm 4: Threads

**Öğrenim Hedefleri** Bu bölümü çalıştıktan sonra şunları yapabilmelisiniz:

- **Process** ve **thread** arasındaki ayrımı anlamak.
- **Thread**'ler için temel tasarım meselelerini tanımlamak.
- **User-level threads** (kullanıcı seviyesi thread'ler) ve **kernel-level threads** (çekirdek seviyesi thread'ler) arasındaki farkı açıklamak.
- Windows'taki **thread** yönetim tesisini (yapısını) tanımlamak.
- Solaris'teki **thread** yönetim tesisini tanımlamak.
- Linux'taki **thread** yönetim tesisini tanımlamak.

Bu bölüm, **process management** (süreç yönetimi) ile ilgili, güncel işletim sistemlerinin çoğunda bulunan bazı daha ileri düzey kavramları inceler. **Process** kavramının şimdiye kadar sunulandan daha karmaşık ve incelikli olduğunu; aslında iki ayrı ve potansiyel olarak bağımsız kavramı bünyesinde barındırdığını gösteriyoruz: biri **resource ownership** (kaynak sahipliği) ile ilgili, diğeri ise **execution** (yürütme) ile ilgilidir. Bu ayrım, birçok işletim sisteminde **thread** olarak bilinen yapının geliştirilmesine yol açmıştır.

## 4.1 PROCESSES VE THREADS

Şimdiye kadarki tartışma, **process** kavramını iki karakteristiği barındıran bir yapı olarak sundu:

**1. Resource ownership (Kaynak sahipliği):** Bir **process**, **process image**'ını tutmak için bir **virtual address space** (sanal adres alanı) içerir; Bölüm 3'ten hatırlayacağınız üzere **process image**; program, data, stack ve **process control block** içinde tanımlanan niteliklerin (attributes) bütünüdür. Zaman zaman bir **process**'e **main memory** (ana bellek), I/O kanalları, I/O cihazları ve dosyalar gibi kaynakların kontrolü veya sahipliği **allocate** (tahsis) edilebilir. **OS** (İşletim Sistemi), kaynaklar açısından **process**'ler arasında istenmeyen müdahaleleri önlemek için bir koruma işlevi yerine getirir.

**2. Scheduling/execution (Çizelgeleme/Yürütme):** Bir **process**'in **execution**'ı, bir veya daha fazla program boyunca bir **execution path** (veya **trace**) izler (örn. Şekil 1.5). Bu **execution**, diğer **process**'lerin **execution**'ı ile serpiştirilmiş (interleaved) olabilir. Dolayısıyla, bir **process**'in bir **execution state**'i (Running, Ready, vb.) ve bir **dispatching priority**'si (dağıtım önceliği) vardır; ve bu, **OS** tarafından **schedule** ve **dispatch** edilen varlıktır.

Biraz düşünmek okuyucuyu ikna edecektir ki, bu iki karakteristik birbirinden bağımsızdır ve **OS** tarafından bağımsız olarak ele alınabilir. Bu, özellikle son zamanlarda geliştirilen sistemler olmak üzere birçok işletim sisteminde yapılmaktadır. İki karakteristiği ayırt etmek için:

- **Dispatching** birimi genellikle **thread** veya **lightweight process** olarak adlandırılır.
- **Resource ownership** birimi ise genellikle **process** veya **task** olarak adlandırılır.

**Dipnot 1:** Ne yazık ki, bu tutarlılık derecesi bile her zaman korunmamıştır. IBM'in mainframe işletim sistemlerinde, **address space** ve **task** kavramları, sırasıyla bizim bu bölümde tanımladığımız **process** ve **thread** kavramlarına kabaca karşılık gelir. Ayrıca literatürde, **lightweight process** terimi şu anlamlarda kullanılır: (1) **thread** terimine eşdeğer olarak, (2) **kernel-level thread** olarak bilinen belirli bir **thread** türü olarak, veya (3) Solaris örneğinde olduğu gibi, **user-level thread**'leri **kernel-level thread**'lere eşleyen bir varlık olarak.

**Multithreading**

**Multithreading**, bir **OS**'in (İşletim Sisteminin) tek bir **process** içinde birden fazla, eşzamanlı **execution** (yürütme) yolunu destekleme yeteneğini ifade eder. **Thread** kavramının tanınmadığı, her **process** başına tek bir **thread of execution** içeren geleneksel yaklaşım, **single-threaded** yaklaşım olarak adlandırılır. **Figure 4.1**'in sol yarısında gösterilen iki düzenleme, **single-threaded** yaklaşımlardır. **MS-DOS**, tek bir kullanıcı **process**'ini ve tek bir **thread**'i destekleyen bir **OS** örneğidir. Bazı **UNIX** varyantları gibi diğer işletim sistemleri, birden fazla kullanıcı **process**'ini destekler, ancak **process** başına yalnızca bir **thread** destekler.

**Figure 4.1**'in sağ yarısı **multithreaded** yaklaşımları tasvir eder. Bir Java **runtime environment**, birden fazla **thread** içeren tek bir **process** sisteminin örneğidir. Bu bölümde ilgi odağı olan şey, her biri birden fazla **thread** destekleyen birden fazla **process** kullanımıdır. Bu yaklaşım, diğerlerinin yanı sıra Windows, Solaris ve **UNIX**'in birçok modern versiyonunda benimsenmiştir. Bu bölümde, **multithreading**'in genel bir tanımını veriyoruz; Windows, Solaris ve Linux yaklaşımlarının detayları bu bölümde daha sonra tartışılacaktır.

**Figure 4.1: Threads ve Processes**

Bir **multithreaded** ortamda, bir **process**; **resource allocation** (kaynak tahsisi) birimi ve bir koruma birimi olarak tanımlanır. Aşağıdakiler **process**'ler ile ilişkilendirilir:

- **Process image**'ı tutan bir **virtual address space**.
- **Processor**'lara, diğer **process**'lere (**interprocess communication** için), dosyalara ve I/O kaynaklarına (cihazlar ve kanallar) korumalı erişim.

Bir **process** içinde, her biri aşağıdakilere sahip olan bir veya daha fazla **thread** olabilir:

- Bir **thread execution state** (Running, Ready, vb.).
- Çalışmadığı zaman kaydedilmiş bir **thread context**; bir **thread**'e bakmanın bir yolu, onu bir **process** içinde çalışan bağımsız bir **program counter** olarak görmektir.
- Bir **execution stack**.
- Yerel değişkenler için bazı **per-thread static storage**.
- Kendi **process**'inin belleğine ve kaynaklarına erişim (o **process**'teki diğer tüm **thread**'lerle paylaşılır).

**Figure 4.2**, **process management** bakış açısından **thread**'ler ve **process**'ler arasındaki ayrımı gösterir. Bir **single-threaded process** modelinde (yani, belirgin bir **thread** kavramının olmadığı

durumda), bir **process**'in temsili; onun **process control block**'unu ve **user address space**'ini, ayrıca **process**'in **execution**'ının call/return davranışını yönetmek için **user** ve **kernel stack**'lerini içerir. **Process** çalışırken (**running**), **processor register**'larını kontrol eder. Bu **register**'ların içeriği, **process** çalışmadığında kaydedilir.

Bir **multithreaded** ortamda, **process** ile ilişkili hala tek bir **process control block** ve **user address space** vardır; ancak şimdi her **thread** için ayrı **stack**'ler ve ayrıca **register** değerlerini, önceliği (**priority**) ve diğer **thread** ile ilgili durum bilgilerini içeren, her **thread** için ayrı bir kontrol bloğu (**control block**) bulunur.

**Figure 4.2: Single-Threaded ve Multithreaded Process Modelleri**

[LETW88], tek kullanıcılı bir **multiprocessing** sistemde **thread**'lerin kullanımına dair dört örnek verir:

1. **Foreground ve background work (Ön plan ve arka plan çalışması):** Örneğin bir hesap tablosu (spreadsheet) programında, bir **thread** menüleri gösterip kullanıcı girdisini okurken, diğer bir **thread** kullanıcı komutlarını **execute** edip tabloyu güncelleyebilir. Bu düzenleme, bir önceki komut tamamlanmadan programın sonraki komutu istemesine izin vererek uygulamanın algılanan hızını genellikle artırır.
2. **Asynchronous processing (Asenkron işlem):** Programdaki asenkron elemanlar **thread** olarak implement edilebilir. Örneğin güç kesintisine karşı bir koruma olarak, bir kelime işlemci (word processor) **RAM** (random access memory) buffer'ını dakikada bir diske yazacak şekilde tasarlanabilir. Tek işi periyodik yedekleme olan ve kendini doğrudan **OS** ile **schedule** eden bir **thread** oluşturulabilir; böylece ana programda zaman kontrolleri sağlamak veya girdi/çıktıyı koordine etmek için karmaşık kodlara gerek kalmaz.
3. **Speed of execution (Yürütme hızı):** Bir **multithreaded process**, bir cihazdan sonraki veri partisini (batch) okurken, elindeki veri partisini hesaplayabilir (compute). Bir **multiprocessor** sistemde, aynı **process**'in birden fazla **thread**'i eşzamanlı olarak **execute** edilebilir. Böylece, bir **thread** veri okumak için bir I/O işleminde **blocked** (bloklanmış) olsa bile, başka bir **thread execute** ediliyor olabilir.
4. **Modular program structure (Modüler program yapısı):** Çeşitli aktiviteler veya çeşitli I/O kaynak ve hedefleri içeren programlar, **thread**'ler kullanılarak daha kolay tasarlanabilir ve implement edilebilir.

**Thread** destekleyen bir **OS**'te, **scheduling** ve **dispatching** işlemi **thread** bazında yapılır; bu yüzden **execution** ile ilgili durum bilgilerinin çoğu **thread-level** veri yapılarında tutulur. Ancak, bir **process**'teki tüm **thread**'leri etkileyen ve **OS**'in **process** seviyesinde yönetmesi gereken bazı eylemler vardır. Örneğin, **suspension** (askıya alma), bir **process**'in **address space**'inin başka bir **process**'e yer açmak için **main memory**'den dışarı **swap** edilmesini (swapping out) içerir. Bir **process**'teki tüm **thread**'ler aynı **address space**'i paylaştığı için, hepsi aynı anda **suspend** edilir. Benzer şekilde, bir **process**'in **termination**'ı (sonlandırılması) o **process** içindeki tüm **thread**'leri sonlandırır.

## Thread Functionality (Thread İşlevselliği)

**Process**'ler gibi, **thread**'lerin de **execution state**'leri vardır ve birbirleriyle **synchronize** olabilirler. **Thread** işlevselliğinin bu iki yönüne sırayla bakacağız.

## Thread States (Thread Durumları)

**Process**'lerde olduğu gibi, bir **thread** için anahtar **state**'ler **Running**, **Ready** ve **Blocked**'dır. Genellikle, **suspend state**'lerini **thread**'lerle ilişkilendirmek mantıklı değildir çünkü bu **state**'ler **process-level** kavramlardır. Özellikle, eğer bir **process swap out** edilirse, tüm **thread**'leri de zorunlu olarak **swap out** edilir çünkü hepsi **process**'in **address space**'ini paylaşır.

**Thread state**'indeki bir değişiklikle ilişkili dört temel **thread** operasyonu vardır [ANDE04]:

1. **Spawn:** Tipik olarak yeni bir **process spawn** edildiğinde, o **process** için bir **thread** de **spawn** edilir. Daha sonra, bir **process** içindeki bir **thread**, aynı **process** içinde başka bir **thread spawn** edebilir; yeni **thread** için bir **instruction pointer** ve argümanlar sağlar. Yeni **thread**'e kendi **register context**'i ve **stack space**'i sağlanır ve **Ready queue**'ya yerleştirilir.
2. **Block:** Bir **thread** bir olayı (event) beklemesi gerektiğinde **block**lanır (kendi **user register**'larını, **program counter**'ını ve **stack pointer**'larını kaydederek). **Processor** daha sonra aynı veya farklı bir **process**'teki başka bir **ready thread**'in **execution**'ına geçebilir.
3. **Unblock: Thread**'in **block**lanmasına neden olan olay gerçekleştiğinde, **thread Ready queue**'ya taşınır.
4. **Finish:** Bir **thread** tamamlandığında, **register context**'i ve **stack**'leri **deallocate** edilir.

Önemli bir mesele, bir **thread**'in **block**lanmasının tüm **process**'in **block**lanmasıyla sonuçlanıp sonuçlanmadığıdır. Yani, bir **process**'teki bir **thread block**landığında, bu durum aynı **process**'teki diğer **thread**'lerin (hazır durumda, **ready state**'te olsalar bile) çalışmasını engeller mi? Açıkçası, eğer **block**lanan tek bir **thread** tüm **process**'i bloklarsa, **thread**'lerin esnekliğinin ve gücünün bir kısmı kaybolur.

Bu konuya daha sonra **user-level** ile **kernel-level thread** tartışmamızda döneceğiz, ancak şimdilik tüm **process**'i bloklamayan **thread**'lerin performans faydalarını düşünelim. **Figure 4.3** ([KLEI96]'dakine dayanan), birleşik bir sonuç elde etmek için iki farklı host'a iki uzak prosedür çağrısı (**RPCs - Remote Procedure Calls**) yapan bir programı gösterir. **Single-threaded** bir programda sonuçlar sırayla alınır, bu yüzden program sırayla her sunucudan yanıt beklemek zorundadır. Programı her **RPC** için ayrı bir **thread** kullanacak şekilde yeniden yazmak önemli bir hız artışı (speedup) sağlar. Not: Eğer bu program bir **uniprocessor** üzerinde çalışıyorsa, istekler sırayla üretilmeli ve sonuçlar sırayla işlenmelidir; ancak program iki yanıtı **concurrently** (eşzamanlı olarak) bekler.

> **Dipnot 2: RPC**, farklı makinelerde **execute** edilebilen iki programın **procedure call/return** sözdizimi ve semantiği kullanarak etkileşime girdiği bir tekniktir. Hem

çağrılan hem de çağıran programlar, ortak program aynı makinede çalışıyormuş gibi davranır. **RPC**'ler genellikle **client/server** uygulamaları için kullanılır ve Bölüm 16'da tartışılacaktır.

**Figure 4.3 Thread'ler Kullanılarak Remote Procedure Call (RPC)**

Bir **uniprocessor** üzerinde, **multiprogramming**, birden çok **process** içindeki birden çok **thread**'in serpiştirilmesini (interleaving) sağlar. **Figure 4.4**'teki örnekte, iki **process** içindeki üç **thread**, **processor** üzerinde serpiştirilmiştir. **Execution**, halihazırda çalışan **thread** **block**landığında veya **time slice**'ı (zaman dilimi) tükendiğinde bir **thread**'den diğerine geçer.

> **Dipnot 3:** Bu örnekte, **Thread** C, **Thread** B de çalışmaya hazır (**ready**) olmasına rağmen, **Thread** A **time quantum**'unu tükettikten sonra çalışmaya başlar. B ve C arasındaki seçim bir **scheduling** kararıdır; bu konu Kısım Dört'te işlenecektir.

**Figure 4.4 Uniprocessor Üzerinde Multithreading Örneği**

## Thread Synchronization (Thread Senkronizasyonu)

Bir **process**'in tüm **thread**'leri aynı **address space**'i ve açık dosyalar gibi diğer kaynakları paylaşır. Bir kaynağın bir **thread** tarafından değiştirilmesi, aynı **process** içindeki diğer **thread**'lerin ortamını etkiler. Bu nedenle, birbirlerine müdahale etmemeleri veya veri yapılarını bozmamaları için çeşitli **thread**'lerin aktivitelerini **synchronize** etmek (eşzamanlamak) gereklidir. Örneğin, iki **thread** aynı anda **doubly linked list**'e bir eleman eklemeye çalışırsa, bir eleman kaybolabilir veya liste bozuk (malformed) hale gelebilir.

**Thread**'lerin **synchronization**'ında ortaya çıkan sorunlar ve kullanılan teknikler, genel olarak, **process**'lerin **synchronization**'ı ile aynıdır. Bu sorunlar ve teknikler Bölüm 5 ve 6'nın konusu olacaktır.

---

## 4.2 TYPES OF THREADS (Thread Tipleri)

### User-Level ve Kernel-Level Threads

İki geniş **thread implementation** (uygulama/gerçekleştirim) kategorisi vardır: **User-level threads (ULTs)** ve **kernel-level threads (KLTs)**. İkincisi literatürde **kernel-supported threads** veya **lightweight processes** olarak da anılır.

> **Dipnot 4:** ULT ve KLT kısaltmaları yaygın olarak kullanılmaz ancak özlü olması için tanıtılmıştır.

### User-Level Threads (Kullanıcı Seviyesi Thread'ler)

Saf bir **ULT** tesisinde, **thread management** işinin tamamı uygulama tarafından yapılır ve **kernel**, **thread**'lerin varlığından haberdar değildir. **Figure 4.5a**, saf **ULT** yaklaşımını gösterir. Herhangi bir uygulama, **ULT** yönetimi için bir rutin paketi olan bir **threads library** (thread kütüphanesi) kullanılarak **multithreaded** olacak şekilde programlanabilir. **Threads library**; **thread** oluşturma ve yok etme, **thread**'ler arasında mesaj ve veri iletme, **thread execution**'ını **schedule** etme ve **thread context**'lerini kaydetme ve geri yükleme (restore) kodlarını içerir.

**Figure 4.5 User-Level ve Kernel-Level Threads**

Varsayılan olarak, bir uygulama tek bir **thread** ile başlar ve o **thread** içinde çalışmaya başlar. Bu uygulama ve onun **thread**'i, **kernel** tarafından yönetilen tek bir **process**'e tahsis edilir (**allocated**). Uygulamanın çalıştığı (**process**'in **Running state**'de olduğu) herhangi bir zamanda, uygulama aynı **process** içinde çalışacak yeni bir **thread spawn** edebilir. **Spawning**, **threads library** içindeki **spawn** yardımcı programı (utility) çağrılarak yapılır. Kontrol, bir **procedure call** ile o yardımcı programa geçer. **Threads library**, yeni **thread** için bir veri yapısı oluşturur ve ardından bir **scheduling algorithm** kullanarak bu **process** içindeki **Ready state**'de olan **thread**'lerden birine kontrolü devreder.

Kontrol kütüphaneye geçtiğinde, mevcut **thread**'in **context**'i kaydedilir ve kontrol kütüphaneden bir **thread**'e geçtiğinde, o **thread**'in **context**'i geri yüklenir. **Context** esasen **user register**'ların, **program counter**'ın ve **stack pointer**'ların içeriğinden oluşur.

Önceki paragrafta açıklanan tüm aktivite **user space**'te ve tek bir **process** içinde gerçekleşir. **Kernel** bu aktiviteden habersizdir. **Kernel**, **process**'i bir birim olarak **schedule** etmeye devam eder ve o **process**'e tek bir **execution state** (Ready, Running, Blocked, vb.) atar. Aşağıdaki örnekler, **thread scheduling** ve **process scheduling** arasındaki ilişkiyi netleştirmelidir. **Process** B'nin **thread** 2'sinde **execute** edildiğini varsayalım; **process**'in ve **process**'in parçası olan iki **ULT**'nin durumları **Figure 4.6a**'da gösterilmiştir. Aşağıdakilerin her biri olası bir durumdur:

**Figure 4.6 User-Level Thread Durumları ve Process Durumları Arasındaki İlişkilerin Örnekleri**

1. **Thread** 2'de **execute** edilen uygulama, B'yi **block**layan bir **system call** yapar. Örneğin, bir I/O çağrısı yapılır. Bu, kontrolün **kernel**'a transfer olmasına neden olur. **Kernel**, I/O eylemini başlatır, **Process** B'yi **Blocked state**'e koyar ve başka bir **process**'e geçer. Bu arada, **threads library** tarafından tutulan veri yapısına göre, **Process** B'nin **Thread** 2'si hala **Running state**'dedir. **Thread** 2'nin aslında bir **processor** üzerinde yürütülme anlamında çalışmadığını; ancak **threads library** tarafından **Running state**'de olarak algılandığını not etmek önemlidir. Karşılık gelen durum diyagramları **Figure 4.6b**'de gösterilmiştir.
2. Bir **clock interrupt** kontrolü **kernel**'a geçirir ve **kernel**, o anda çalışan **process**'in (B) **time slice**'ını tükettiğini belirler. **Kernel**, **Process** B'yi **Ready state**'e koyar ve başka bir **process**'e geçer. Bu arada, **threads library** tarafından tutulan veri yapısına göre,

**Process** B'nin **Thread** 2'si hala **Running state**'dedir. Karşılık gelen durum diyagramları **Figure 4.6c**'de gösterilmiştir.

3. **Thread** 2, **Process** B'nin **Thread** 1'i tarafından gerçekleştirilecek bir eyleme ihtiyaç duyduğu bir noktaya ulaşmıştır. **Thread** 2 bir **Blocked state**'e girer ve **Thread** 1 **Ready**'den **Running**'e geçer. **Process**'in kendisi **Running state**'de kalır. Karşılık gelen durum diyagramları **Figure 4.6d**'de gösterilmiştir.

Önceki üç maddenin her birinin, **Figure 4.6**'nın (a) diyagramından başlayan alternatif bir olayı önerdiğine dikkat edin. Yani diğer üç diyagramın her biri (b, c, d), (a)'daki durumdan bir geçişi gösterir. 1. ve 2. durumlarda (**Figures 4.6b** ve **4.6c**), **kernel** kontrolü tekrar **Process** B'ye geçirdiğinde, **execution Thread** 2'de devam eder. Ayrıca, bir **process**'in **threads library** içindeki kodu çalıştırırken, ya **time slice**'ını tüketerek ya da daha yüksek öncelikli bir **process** tarafından **preempt** edilerek (kesilerek) durdurulabileceğini (**interrupted**) unutmayın. Dolayısıyla, bir **process**, bir **thread**'den diğerine **thread switch** işleminin ortasındayken **interrupted** olabilir. O **process** devam ettirildiğinde (**resumed**), **execution threads library** içinde devam eder, bu da **thread switch** işlemini tamamlar ve o **process** içindeki başka bir **thread**'e kontrolü devreder.

KLT'ler (Kernel-Level Threads) yerine ULT'lerin (User-Level Threads) kullanımının, aşağıdakiler de dahil olmak üzere bir dizi avantajı vardır:

1. **Thread switching**, **kernel-mode privileges** gerektirmez çünkü tüm **thread management data structure**'ları tek bir **process**'in **user address space**'i içindedir. Bu nedenle, **process**, **thread management** yapmak için **kernel mode**'a geçmez. Bu, iki **mode switch** (user'dan kernel'a; kernel'dan tekrar user'a) **overhead**'inden tasarruf sağlar.
2. **Scheduling**, **application specific** (uygulamaya özgü) olabilir. Bir uygulama en çok basit bir **round-robin scheduling algorithm**'dan fayda sağlarken, bir başkası **priority-based scheduling algorithm**'dan faydalanabilir. **Scheduling algorithm**, alttaki **OS scheduler**'ı bozmadan uygulamaya göre uyarlanabilir (tailored).
3. **ULT**'ler herhangi bir **OS** üzerinde çalışabilir. **ULT**'leri desteklemek için alttaki **kernel**'da herhangi bir değişiklik gerekmez. **Threads library**, tüm uygulamalar tarafından paylaşılan bir dizi **application-level** fonksiyondur.

KLT'lere kıyasla ULT'lerin iki belirgin dezavantajı vardır:

1. Tipik bir **OS**'te, birçok **system call blocking**'dir (bloklayıcıdır). Sonuç olarak, bir **ULT** bir **system call execute** ettiğinde, sadece o **thread blocked** olmaz, aynı zamanda **process** içindeki tüm **thread**'ler de **blocked** olur.
2. Saf bir **ULT** stratejisinde, **multithreaded** bir uygulama **multiprocessing**'den faydalanamaz. Bir **kernel**, bir **process**'i aynı anda yalnızca bir **processor**'a atar (**assigns**). Bu nedenle, bir **process** içinde aynı anda yalnızca tek bir **thread execute** edilebilir. Aslında, tek bir **process** içinde **application level multiprogramming**'e sahibizdir. Bu **multiprogramming**, uygulamanın önemli ölçüde **speedup** (hızlanma)

kazanmasını sağlasa da, kodun bölümlerini **simultaneously** (eşzamanlı) **execute** etme yeteneğinden faydalanacak uygulamalar vardır.

Bu iki sorunun etrafından dolaşmanın yolları vardır. Örneğin, her iki sorun da uygulamayı birden fazla **thread** yerine birden fazla **process** olarak yazarak aşılabilir. Ancak bu yaklaşım **thread**'lerin ana avantajını ortadan kaldırır: Her **switch**, bir **thread switch** yerine bir **process switch** haline gelir ve bu da çok daha büyük bir **overhead** ile sonuçlanır.

**Blocking thread** sorununu aşmanın bir başka yolu da **jacketing** olarak bilinen bir tekniği kullanmaktır. **Jacketing**'in amacı, bir **blocking system call**'u bir **nonblocking system call**'a dönüştürmektir. Örneğin, doğrudan bir **system I/O routine**'ini çağırmak yerine, bir **thread application-level I/O jacket routine** çağırır. Bu **jacket routine** içinde, I/O cihazının meşgul (**busy**) olup olmadığını kontrol eden kod bulunur. Eğer meşgulse, **thread Blocked state**'e girer ve kontrolü (**threads library** aracılığıyla) başka bir **thread**'e geçirir. Bu **thread**'e daha sonra tekrar kontrol verildiğinde, **jacket routine** I/O cihazını tekrar kontrol eder.

## Kernel-Level Threads (KLTs)

Saf bir **KLT** tesisinde (**facility**), **thread management** işinin tamamı **kernel** tarafından yapılır. **Application level**'da **thread management** kodu yoktur, sadece **kernel thread facility**'ye yönelik bir **application programming interface (API)** bulunur. Windows bu yaklaşımın bir örneğidir.

**Figure 4.5b**, saf **KLT** yaklaşımını tasvir eder. **Kernel**, **process**'in bütünü ve **process** içindeki bireysel **thread**'ler için **context information** (bağlam bilgisi) tutar. **Kernel** tarafından yapılan **scheduling**, **thread** bazında yapılır. Bu yaklaşım, **ULT** yaklaşımının iki temel dezavantajını aşar. Birincisi, **kernel** aynı **process**'ten birden fazla **thread**'i birden fazla **processor** üzerinde **simultaneously** (eşzamanlı) **schedule** edebilir. İkincisi, bir **process**'teki bir **thread blocked** olursa, **kernel** aynı **process**'in başka bir **thread**'ini **schedule** edebilir. **KLT** yaklaşımının bir başka avantajı da, **kernel routine**'lerinin kendilerinin **multithreaded** olabilmesidir.

**KLT** yaklaşımının **ULT** yaklaşımına kıyasla temel dezavantajı, aynı **process** içinde bir **thread**'den diğerine kontrol transferinin **kernel**'a bir **mode switch** gerektirmesidir. Farklılıkları göstermek için, **Table 4.1**, UNIX benzeri bir **OS** çalıştıran bir **uniprocessor** VAX bilgisayarında alınan ölçümlerin sonuçlarını gösterir. İki benchmark şöyledir:

- **Null Fork: null procedure**'ü çağıran bir **process/thread**'i **create** etme (oluşturma), **schedule** etme, **execute** etme ve **complete** etme (tamamlama) süresi (yani, bir **process/thread forking** işleminin **overhead**'i).
- **Signal-Wait:** Bir **process/thread**'in bekleyen bir **process/thread**'e **signal** göndermesi ve ardından bir **condition** (koşul) üzerinde beklemesi (**wait**) için geçen süre (yani, iki **process/thread**'i birlikte **synchronize** etmenin **overhead**'i).

**ULT**'ler ve **KLT**'ler arasında ve benzer şekilde **KLT**'ler ve **process**'ler arasında bir **order of magnitude** (bir kat) veya daha fazla fark olduğunu görüyoruz.

**Table 4.1 Thread ve Process Operasyon Gecikmeleri (Latencies)**

| Operation | User-Level Threads (μs) | Kernel-Level Threads (μs) | Processes (μs) |
|---|---|---|---|
| Null Fork | 34 | 948 | 11,300 |
| Signal Wait | 37 | 441 | 1,840 |

Böylece, görünüşte, **single-threaded process**'lere kıyasla **KLT multithreading** kullanarak önemli bir **speedup** (hızlanma) olsa da, **ULT**'leri kullanarak ek bir önemli **speedup** daha vardır. Ancak, bu ek **speedup**'ın gerçekleşip gerçekleşmemesi ilgili uygulamaların doğasına bağlıdır. Eğer bir uygulamadaki **thread switch**'lerinin çoğu **kernel-mode access** gerektiriyorsa, o zaman **ULT-based** bir şema, **KLT-based** bir şemadan çok daha iyi performans göstermeyebilir.

## Combined Approaches (Birleşik Yaklaşımlar)

Bazı işletim sistemleri combined (birleşik) bir **ULT/KLT facility** sağlar (bkz. **Figure 4.5c**). Combined bir sistemde, **thread creation** tamamen **user space**'te yapılır; bir uygulama içindeki **thread**'lerin **scheduling** ve **synchronization** işlemlerinin büyük kısmı da böyledir. Tek bir uygulamadan gelen çoklu **ULT**'ler, belirli (daha az veya eşit) sayıda **KLT**'ye **map** edilir (eşlenir). Programcı, en iyi genel sonuçları elde etmek için belirli bir uygulama ve **processor** için **KLT** sayısını ayarlayabilir.

Combined bir yaklaşımda, aynı uygulama içindeki çoklu **thread**'ler çoklu **processor**'lar üzerinde paralel olarak çalışabilir ve bir **blocking system call** tüm **process**'i bloklamak zorunda kalmaz. Eğer düzgün tasarlanırsa, bu yaklaşım saf **ULT** ve **KLT** yaklaşımlarının avantajlarını birleştirirken dezavantajlarını en aza indirmelidir.

Solaris, bu combined yaklaşımı kullanan bir **OS**'in iyi bir örneğidir. Mevcut Solaris versiyonu, **ULT/KLT** ilişkisini bire-bir (**one-to-one**) olacak şekilde sınırlar.

## Diğer Düzenlemeler (Other Arrangements)

Daha önce de belirttiğimiz gibi, **kaynak tahsisi (resource allocation)** ve **dağıtım birimi (dispatching unit)** kavramları geleneksel olarak tek bir kavramda, yani **süreç (process)** kavramında somutlaşmıştır; bu, thread'ler ve process'ler arasında **1:1 (bire bir)** bir ilişki olduğu anlamına gelir. Son zamanlarda, tek bir process içinde birden fazla thread sağlamaya yönelik büyük bir ilgi oluşmuştur; bu da **many-to-one (çoktan bire)** bir ilişkidir. Ancak, Tablo 4.2'de gösterildiği gibi, diğer iki kombinasyon da araştırılmıştır: **many-to-many (çoktan çoğa)** ilişki ve **one-to-many (birden çoğa)** ilişki.

**Tablo 4.2: Thread'ler ve Process'ler Arasındaki İlişki**

| İlişki | Açıklama | Örnek Sistemler |
|---|---|---|
| **1:1** | Her yürütme thread'i (iş parçacığı), kendi adres alanına ve kaynaklarına sahip benzersiz bir process'tir (süreçtir). | Geleneksel UNIX uygulamaları |
| **M:1** | Bir process, bir adres alanı ve dinamik kaynak sahipliği tanımlar. O process içinde birden fazla thread oluşturulabilir ve yürütülebilir. | Windows NT, Solaris, Linux, OS/2, OS/390, MACH |
| **1:M** | Bir thread, bir process ortamından diğerine göç edebilir (migrate). Bu, bir thread'in farklı sistemler arasında kolayca taşınmasına olanak tanır. | Ra (Clouds), Emerald |
| **M:N** | M:1 ve 1:M durumlarının niteliklerini birleştirir. | TRIX |

---

## Çoktan Çoğa İlişki (Many-to-Many Relationship)

Thread'ler ve process'ler arasında çoktan çoğa bir ilişki kurma fikri, deneysel işletim sistemi **TRIX**'te araştırılmıştır [PAZZ92, WARD80]. TRIX'te **domain (etki alanı)** ve **thread** kavramları vardır.

- **Domain:** Bir adres alanı ve mesajların gönderilip alınabildiği "port"lardan (kapılardan) oluşan statik bir varlıktır.
- **Thread:** Bir yürütme yığını (execution stack), işlemci durumu ve zamanlama (scheduling) bilgisine sahip tek bir yürütme yoludur.

Şu ana kadar tartışılan multithreading (çoklu iş parçacığı) yaklaşımlarında olduğu gibi, tek bir domain içinde birden fazla thread çalışabilir ve bu da daha önce bahsedilen verimlilik kazanımlarını sağlar. Ancak, tek bir kullanıcı aktivitesinin veya uygulamasının birden fazla domain'de gerçekleştirilmesi de mümkündür. Bu durumda, bir domain'den diğerine geçebilen bir thread mevcuttur.

Tek bir thread'in birden fazla domain'de kullanılması, temel olarak programcıya yapılandırma araçları sağlama isteğiyle motive edilmiş görünmektedir. Örneğin, bir I/O (Giriş/Çıkış) alt programını kullanan bir programı düşünelim. Kullanıcı tarafından oluşturulan process'lere izin veren bir multiprogramming (çoklu programlama) ortamında:

1. Ana program, I/O işlemini halletmesi için yeni bir process oluşturabilir ve ardından çalışmaya devam edebilir.
2. Ancak, ana programın gelecekteki ilerlemesi I/O işleminin sonucuna bağlıysa, ana program diğer I/O programının bitmesini beklemek zorunda kalacaktır.

Bu uygulamayı hayata geçirmenin birkaç yolu vardır:

1. **Tüm program tek bir process olarak uygulanabilir.** Bu makul ve basit bir çözümdür. Ancak bellek yönetimiyle ilgili dezavantajları vardır. Process bir bütün olarak verimli çalışmak için önemli miktarda ana bellek gerektirebilirken, I/O alt programı I/O'yu tamponlamak (buffer) ve nispeten küçük miktardaki program kodunu işlemek için nispeten küçük bir adres alanına ihtiyaç duyar. I/O programı daha büyük programın adres alanında çalıştığı için, I/O işlemi sırasında ya tüm process ana bellekte kalmalıdır ya da I/O işlemi "swapping"e (disk ile bellek arası takas) maruz kalır. Ana program ve I/O alt programı aynı adres alanında iki thread olarak uygulansaydı da bu bellek yönetimi etkisi var olacaktı.
2. **Ana program ve I/O alt programı iki ayrı process olarak uygulanabilir.** Bu, alt process'i oluşturma maliyetini (overhead) getirir. Eğer I/O aktivitesi sıksa, ya yönetim kaynaklarını tüketen alt process'i canlı bırakmalısınız ya da verimsiz olan alt programı sık sık oluşturup yok etmelisiniz.
3. **Ana programı ve I/O alt programını tek bir thread olarak uygulanacak tek bir aktivite olarak ele almak.** Ancak, ana program için bir adres alanı (domain) ve I/O alt programı için bir başka adres alanı oluşturulabilir. Böylece, yürütme devam ederken thread iki adres alanı arasında taşınabilir. İşletim Sistemi (OS) iki adres alanını bağımsız olarak yönetebilir ve process oluşturma maliyeti (overhead) oluşmaz. Ayrıca, I/O alt programı tarafından kullanılan adres alanı diğer basit I/O programları tarafından da paylaşılabilir.

TRIX geliştiricilerinin deneyimleri, üçüncü seçeneğin değerli olduğunu ve bazı uygulamalar için en etkili çözüm olabileceğini göstermektedir.

---

## Birden Çoğa İlişki (One-to-Many Relationship)

Dağıtık işletim sistemleri alanında (dağıtık bilgisayar sistemlerini kontrol etmek için tasarlanmıştır), thread kavramına öncelikle adres alanları arasında hareket edebilen bir varlık olarak ilgi duyulmuştur. Bu araştırmanın dikkate değer bir örneği **Clouds** işletim sistemi ve özellikle **Ra** [DASG92] olarak bilinen çekirdeğidir (kernel). Bir diğer örnek ise **Emerald** sistemidir [STEE95].

> *Dipnot 5: Process'lerin veya thread'lerin farklı makinelerdeki adres alanları arasında hareketi veya thread göçü (thread migration), son yıllarda popüler bir konu haline gelmiştir. Bölüm 18 bu konuyu inceleyecektir.*

Clouds sisteminde bir thread, kullanıcının bakış açısına göre bir aktivite birimidir. Bir process ise, ilişkili bir process kontrol bloğuna (PCB) sahip sanal bir adres alanıdır. Oluşturulduğunda,

bir thread o process içindeki bir programa giriş noktasını çağırarak bir process içinde çalışmaya başlar. Thread'ler bir adres alanından diğerine geçebilir ve aslında bilgisayar sınırlarını aşabilir (yani, bir bilgisayardan diğerine geçebilir). Bir thread hareket ettiğinde, kontrol eden terminal, global parametreler ve zamanlama rehberliği (örneğin öncelik/priority) gibi belirli bilgileri de beraberinde taşımalıdır.

Clouds yaklaşımı, hem kullanıcıları hem de programcıları dağıtık ortamın detaylarından yalıtmak (soyutlamak) için etkili bir yol sağlar. Bir kullanıcının aktivitesi tek bir thread olarak temsil edilebilir ve o thread'in bilgisayarlar arasındaki hareketi, uzak bir kaynağa erişim ihtiyacı veya yük dengeleme (load balancing) gibi çeşitli sistemle ilgili nedenlerle İşletim Sistemi (OS) tarafından dikte edilebilir.

## 4.3 Çok Çekirdekli (Multicore) ve Çoklu İş Parçacığı (Multithreading)

Tek bir uygulamanın (örneğin ağır bir video oyunu veya iş istasyonu yazılımı) çok çekirdekli bir sistemde birden fazla thread ile çalıştırılması, hem performans hem de tasarım sorunlarını beraberinde getirir.

### 1. Çok Çekirdekli Sistemlerde Yazılım Performansı

Bir yazılımın çok çekirdekten ne kadar verim alacağı, o yazılımın ne kadarının **paralel** hale getirilebildiğine bağlıdır. Burada karşımıza ünlü **Amdahl Yasası** çıkar.

- Amdahl Yasası: Bir programın hızlanma potansiyeli, programın "seri" (tek tek yapılması gereken) ve "paralel" (aynı anda yapılabilen) kısımlarına bağlıdır.
  $$Speedup = \frac{\text{Tek işlemcide geçen süre}}{\text{Çoklu işlemcide geçen süre}}$$
- **Sorun (Seri Kodun Etkisi):** Eğer kodun sadece %10'u bile seri olmak zorundaysa (yani paralelleştirilemiyorsa), 8 çekirdekli bir işlemcide bile performans artışı en fazla 4.7 katına çıkabilir.[1] Asla 8 katına çıkmaz.
- **Overhead (Genel Gider):** İşlemciler arası iletişim ve önbellek tutarlılığı (cache coherence) yüzünden bir "yönetim yükü" oluşur. Bu yüzden, çekirdek sayısı arttıkça performans bir noktada zirve yapar, sonra düşüşe geçebilir.

### 2. Çok Çekirdekten En İyi Yararlanan Uygulamalar

Yazılım mühendisleri bu performans sorununu aşmak için bazı özel uygulama türlerine odaklanmıştır:

- **Veritabanı Uygulamaları:** Donanım ve yazılım katmanlarında seri kısımlar azaltıldığı için çok çekirdekli sistemlerde çok verimli çalışırlar.
- **Çok İş Parçacıklı (Multithreaded) Yerel Uygulamalar:** Az sayıda ama çok yoğun thread kullanan işlemler (Örn: Lotus Domino, CRM yazılımları).
- **Çok Süreçli (Multiprocess) Uygulamalar:** Birçok tek thread'li sürecin aynı anda çalıştığı sistemler (Örn: Oracle veritabanı, SAP).

- **Java Uygulamaları:** Java, doğası gereği thread kullanımını destekler.[2] **Java Sanal Makinesi (JVM)**, thread yönetimi ve zamanlamayı kendi içinde yaptığı için çok çekirdeği verimli kullanır (Örn: Tomcat, WebLogic).
- **Çok Örnekli (Multi-instance) Uygulamalar:** Uygulamanın kendisi threadlere bölünemese bile, uygulamanın birden fazla kopyasını (instance) sanallaştırma teknolojisi ile paralel çalıştırarak performans kazanılır.

---

**3. Gerçek Hayat Örneği: Valve Oyun Yazılımı (Source Motoru)**

Half-Life 2 gibi oyunların yapımcısı **Valve**, kendi oyun motoru olan **Source Engine**'i çok çekirdekli işlemciler (Intel/AMD) için yeniden tasarladığında üç farklı yöntem denemiştir:[3]

**A. Kaba (Coarse) Threading:**

- **Mantık:** Her büyük modülü ayrı bir işlemciye vermek. (Örn: Render bir işlemcide, Yapay Zeka (AI) diğerinde, Fizik diğerinde).
- **Sonuç:** Kodlaması kolaydır ama performans artışı düşüktür. Gerçek oyun senaryolarında sadece 1.2 kat hızlanma sağlamıştır.

**B. İnce (Fine-grained) Threading:**

- **Mantık:** Benzer küçük görevleri (örneğin bir döngü içindeki işlemleri) parçalayıp işlemcilere dağıtmak.
- **Sonuç:** Çok zordur. Zamanlamayı yönetmek ve kodun karmaşıklığı ile başa çıkmak büyük sorundur.

**C. Melez (Hybrid) Threading (Kazanan Yöntem):**

- **Mantık:** Bazı sistemler tek işlemcide kalır, bazıları parçalanır.
- **Örnek:**
    - **Ses Miksajı:** Kullanıcı etkileşimi azdır ve bağımsız veridir -> Tek bir işlemciye atanır.
    - **Sahne Renderlama (Rendering):** Hiyerarşik bir yapıda thread'lere bölünür. Üst seviye thread'ler, alt görevleri doğurur (spawn eder).

Render Modülü Nasıl Çalışıyor?

Valve mühendisleri render modülünü hızlandırmak için şu stratejileri kullanmıştır:

1. **Paralellik:** Dünyanın kendisi ve sudaki yansıması gibi farklı sahnelerin listelerini aynı anda hazırlar.
2. **Karakter Kemik Dönüşümleri:** Tüm karakterlerin hareket hesaplamaları paralel yapılır.
3. **Veri Kilitleme Stratejisi:** Veritabanını (World List) tamamen kilitlemek yerine, **"Single-Writer-Multiple-Readers" (Tek Yazıcı - Çok Okuyucu)** modelini kullanırlar.

Çünkü oyun motoru %95 oranında okuma, sadece %5 oranında yazma yapar. Bu sayede okuma yapan thread'ler birbirini beklemek zorunda kalmaz.

Bu bölüm, Windows işletim sisteminin uygulamaları nasıl çalıştırdığına dair temel taşları ve modern Windows (Windows 8 ve sonrası) ile gelen "uygulama yaşam döngüsü" değişikliklerini anlatıyor.

Senin için metni teknik terimleri koruyarak, anlaşılır bir Türkçe ile çevirdim ve ders notu formatında özetledim.

---

## 4.4 Windows Süreç (Process) ve İş Parçacığı (Thread) Yönetimi

Bu bölüm, Windows'ta uygulama yürütmeyi destekleyen temel nesnelere ve mekanizmalara genel bir bakışla başlar.

**Temel Kavramlar**

Aşağıdaki kavramlar Windows mimarisinin temel taşlarıdır. Bunları bir hiyerarşi gibi düşünebilirsin:

1. **Application (Uygulama):** Bir veya daha fazla **Process**'ten oluşur.
2. **Process (Süreç):** Bir programı çalıştırmak için gereken kaynakları sağlayan kutudur.
    - **Sahip oldukları:** Sanal adres alanı (Virtual Address Space), çalıştırılabilir kod, sistem nesnelerine açık "handle"lar (tutaçlar), güvenlik bağlamı, benzersiz bir işlem kimliği (PID), ortam değişkenleri, öncelik sınıfı, minimum ve maksimum **working set** (çalışma kümesi - bellekteki alan) boyutları ve **en az bir Thread**.
    - Her process tek bir thread (genellikle **primary thread** olarak adlandırılır) ile başlar, ancak daha sonra başka thread'ler oluşturabilir.
3. **Thread (İş Parçacığı):** Bir process içinde yürütülmek üzere zamanlanabilen (schedule edilebilen) varlıktır.
    - Bir process'in tüm thread'leri, o process'in sanal adres alanını ve sistem kaynaklarını paylaşır.
    - Her thread'in kendine ait şunları vardır: İstisna işleyicileri (exception handlers), zamanlama önceliği, **Thread Local Storage (TLS)**, benzersiz bir thread kimliği ve thread çalışmadığında durumunu kaydetmek için kullanılan yapılar (Context).
    - Çok işlemcili bir bilgisayarda, sistem işlemci sayısı kadar thread'i aynı anda çalıştırabilir.

**İleri Seviye Yönetim Birimleri**

- **Job Object (İş Nesnesi):** Süreç gruplarının (process groups) tek bir birim olarak yönetilmesini sağlar. Process'lerin niteliklerini kontrol eden; isimlendirilebilen, güvenliği sağlanabilen ve paylaşılabilen nesnelerdir.

    - Örneğin: Bir "Job"a bağlı tüm process'lerin bellek kullanımını sınırlamak veya hepsini tek seferde sonlandırmak (terminate) için kullanılır.
- **Thread Pool (İş Parçacığı Havuzu):** Uygulama adına asenkron geri aramaları (callbacks) verimli bir şekilde yürüten işçi thread'ler koleksiyonudur. Amaç, sürekli yeni thread oluşturup yok etmek yerine, eldeki thread'leri tekrar kullanarak maliyeti düşürmektir.

**Fiber ve UMS (Daha Hafif Birimler)**

Burada işletim sisteminin "kernel" (çekirdek) seviyesinden ziyade, uygulamanın kendi içinde yönettiği yapılar anlatılıyor:

- **Fiber:** Uygulama tarafından manuel olarak zamanlanması gereken bir yürütme birimidir.
    - Fiber'lar, onları zamanlayan thread'lerin bağlamında çalışır.
    - İyi tasarlanmış multithreaded bir uygulamaya göre genelde bir avantaj sağlamazlar; ancak kendi thread yönetimini yapan eski uygulamaları Windows'a taşımayı (port etmeyi) kolaylaştırır.
    - **Önemli Fark:** Fiber'lar **preemptive** (kesintili/öncelikli) zamanlanmaz. Yani işletim sistemi "Süren bitti, sıra diğerinde" demez; bir fiber çalışmayı bitirip sırayı diğerine kendisi vermelidir.
- **User-mode Scheduling (UMS):** Uygulamaların kendi thread'lerini zamanlamak için kullanabileceği hafif bir mekanizmadır.
    - Uygulama, sistem zamanlayıcısını (kernel scheduler) devreye sokmadan, kullanıcı modunda (user mode) thread'ler arasında geçiş yapabilir.
    - Eğer bir UMS thread'i kernel'de bloklanırsa (takılırsa), işlemcinin kontrolü tekrar uygulamaya verilir.
    - Kısa süreli ve çok sayıda iş parçacığı gerektiren yüksek performanslı uygulamalar (örneğin veritabanı sunucuları) için Thread Pool'dan daha verimlidir.

---

## Arka Plan Görevleri ve Uygulama Yaşam Döngülerinin Yönetimi

Windows 8 ve Windows 10 ile birlikte, process yönetimi felsefesinde büyük bir değişiklik oldu. Eskiden masaüstü uygulamalarında "Kapat" tuşuna basmak kullanıcının sorumluluğundaydı. Modern (Metro/UWP) arayüzde ise ipler İşletim Sisteminin elindedir.

**Yeni Yaşam Döngüsü Modeli**

1. **Tek Aktif Uygulama:** Modern arayüzde (Metro UI), genellikle aynı anda sadece bir "Store" uygulaması çalışır (SnapView ile yan yana çalıştırma hariç).
2. **Askıya Alma (Suspended Mode):** Ön planda olmayan diğer tüm uygulamalar askıya alınır. İşlemci, ağ veya disk kaynaklarına erişemezler.
    - **Live Tiles (Canlı Kutucuklar):** Uygulama çalışıyormuş gibi görünse de, aslında sadece "push notification" (anlık bildirim) alır ve sistem kaynağı tüketmez.

3. **Sonlandırma (Termination) ve Veri Kaybı Riski:**
    - Windows, kaynak yetersizliğinde veya zaman aşımında, arka plandaki (askıdaki) bir uygulamayı **haber vermeden** kapatabilir (terminate).
    - **Geliştiricinin Sorumluluğu:** Uygulama askıya alındığında (suspended), geliştirici kullanıcının verilerini ve ayarlarını kaydetmelidir.
    - Uygulama tekrar açıldığında, kullanıcı sanki hiç kapanmamış gibi kaldığı yerden devam etmelidir. Geliştirici, uygulamanın "kapandığını" mı yoksa "askıya alındıktan sonra sistem tarafından mı öldürüldüğünü" kod ile kontrol edip ona göre eski durumu (state) geri yüklemelidir.

**Özetle Geliştirici Ne Yapmalı?**

Eskiden "kullanıcı kaydetmezse veri giderdi". Şimdi ise Windows uygulamanı arka planda çat diye kapatabileceği için, uygulama her "askıya alındığında" (ekrandan gittiğinde) durumu (state) diske yazmalıdır. Kullanıcı geri döndüğünde, sanki uygulama hiç kapanmamış gibi hissetmelidir.

Bu durum, Windows uygulama geliştirmede **durum yönetimini (state management)** kritik bir başarı faktörü haline getirir.

Harika bir devam bölümü. Burası, Windows'un modern mobil ve masaüstü hibrit yapısını (arka plan görevleri) ve işletim sisteminin kalbi olan **Process (Süreç)** ve **Thread (İş Parçacığı)** nesnelerinin "Object-Oriented" (Nesne Yönelimli) yapısını anlatıyor.

Metni senin için teknik terimleri koruyarak ve ders notu formatında özetleyerek çevirdim.

---

## Arka Plan Görevlerinin Yönetimi (Background Task Management)

Windows, uygulamaların ön planda (ekranda) değilken bile küçük işleri yapabilmesi için bir **Background Task API** (Arka Plan Görev API'si) sunar. Ancak bu ortam oldukça kısıtlıdır:

- **Push Notifications (Anlık Bildirimler):** Uygulamalar sunucudan bildirim alabilir. Bunlar şablon XML dizileridir ve **Windows Notification Service (WNS)** adlı bulut servisi tarafından yönetilir.
- **Kaynak Kısıtlaması:** WNS, güncellemeleri ara sıra yollar ve API bunları sıraya alıp işlemci müsait olduğunda işler. Ancak arka plan görevleri işlemci kullanımında ciddi şekilde sınırlandırılmıştır: **Her işlemci saati başına sadece 1 saniye işlemci süresi alırlar.**
- **Amaç:** Bu kısıtlama, kritik görevlerin kaynak garantisini sağlar ve pil ömrünü korur. Ancak, bir arka plan uygulamasının çalışacağı asla garanti edilmez.

---

## Windows Process (Windows Süreci)

Windows process'lerinin en önemli karakteristik özellikleri şunlardır:

1. **Nesne (Object) Yapısı:** Windows process'leri birer "Object" olarak uygulanır.

2. **Yaratılış:** Bir process, yeni bir process olarak veya var olan bir process'in kopyası olarak yaratılabilir.
3. **İçerik:** Bir çalıştırılabilir process, bir veya daha fazla thread (iş parçacığı) içerebilir.
4. **Senkronizasyon:** Hem process hem de thread nesneleri yerleşik senkronizasyon yeteneklerine sahiptir.

**Güvenlik ve Erişim (Access Token)**

Şekil 4.10'da (kitabındaki referans) belirtildiği gibi, her process kontrol ettiği veya kullandığı kaynaklarla ilişkilidir.

- **Access Token (Erişim Jetonu):** Her process'e atanan bir güvenlik kimliğidir.
- **Nasıl Çalışır?** Kullanıcı sisteme giriş yaptığında (login), Windows kullanıcının güvenlik ID'sini içeren bir token oluşturur. Bu kullanıcı adına çalışan her process, bu token'ın bir kopyasına sahip olur.
- **İşlevi:** Windows, process'in güvenli nesnelere erişip erişemeyeceğini veya kısıtlı fonksiyonları çalıştırıp çalıştıramayacağını bu token ile doğrular. Process'in kendi özelliklerini değiştirip değiştiremeyeceği bile bu token'a bağlıdır.

---

## Process ve Thread Nesneleri (Process and Thread Objects)

Windows'un nesne yönelimli yapısı, genel amaçlı bir işlem tesisi geliştirmeyi kolaylaştırır. İki temel nesne türü vardır:

1. **Process (Süreç):** Kaynaklara (bellek, açık dosyalar vb.) sahip olan, kullanıcı işine veya uygulamasına karşılık gelen varlıktır.
2. **Thread (İş Parçacığı):** Sırayla yürütülen, kesintiye uğratılabilen (interruptible) ve işlemcinin başka bir thread'e geçebildiği dağıtılabilir iş birimidir.

**Tablo 4.3: Windows Process Nesnesi Nitelikleri (Attributes)**

Bir process oluşturulduğunda, "Process Class" şablon olarak kullanılır ve aşağıdaki niteliklere sahip yeni bir "Instance" (örnek) oluşturulur:

| Nitelik | Açıklama |
|---|---|
| **Process ID** | İşletim sisteminin process'i tanımasını sağlayan benzersiz değer (Kimlik Numarası). |
| **Security Descriptor** | Nesneyi kimin oluşturduğunu, kimin erişebileceğini ve kime erişimin reddedildiğini tanımlar. |

| | |
|---|---|
| **Base Priority** | Process'in thread'leri için temel yürütme önceliği. |
| **Default Processor Affinity** | Process'in thread'lerinin üzerinde çalışabileceği varsayılan işlemci kümesi (Hangi çekirdeklerde çalışabilir?). |
| **Quota Limits** | Kullanıcı process'lerinin kullanabileceği maksimum belleği (paged/nonpaged), disk alanını ve işlemci süresini belirler. |
| **Execution Time** | Process içindeki tüm thread'lerin toplam çalışma süresi. |
| **I/O Counters** | Process thread'lerinin gerçekleştirdiği G/Ç (Giriş/Çıkış) işlemlerinin sayısını ve türünü tutan değişkenler. |
| **VM Operation Counters** | Sanal bellek işlemlerinin sayısını ve türünü tutan değişkenler. |
| **Exception/Debugging Ports** | Bir thread istisna (hata) verdiğinde process yöneticisinin mesaj gönderdiği iletişim kanalları. |
| **Exit Status** | Process'in sonlanma nedeni. |

**Tablo 4.4: Windows Thread Nesnesi Nitelikleri (Attributes)**

Bir Windows process'i çalışmak için en az bir thread içermelidir. Thread niteliklerinin bazıları (örneğin Affinity) process'ten türetilir.

| **Nitelik** | **Açıklama** |
|---|---|
| **Thread ID** | Thread'i tanımlayan benzersiz değer. |

| | |
|---|---|
| **Thread Context** | Thread'in son çalıştığı andaki yürütme durumunu tanımlayan **Register (Yazmaç)** değerleri ve diğer uçucu veriler seti. |
| **Dynamic Priority** | Thread'in o anki yürütme önceliği (Sistem tarafından anlık değiştirilebilir). |
| **Base Priority** | Thread'in dinamik önceliğinin alt limiti. |
| **Thread Processor Affinity** | Thread'in üzerinde çalışabileceği işlemciler kümesi (Process Affinity'nin bir alt kümesi veya tamamı olmak zorundadır). |
| **Thread Execution Time** | Thread'in User Mode ve Kernel Mode'da geçirdiği toplam kümülatif süre. |
| **Alert Status** | Bekleyen bir thread'in asenkron bir prosedür çağrısını yürütüp yürütemeyeceğini gösteren bayrak (flag). |
| **Suspension Count** | Thread'in devam ettirilmeden (resume) önce kaç kez askıya alındığının sayısı. |
| **Impersonation Token** | Thread'in başka bir process adına (örneğin bir sistem servisi) işlem yapmasına izin veren geçici erişim jetonu. |
| **Termination Port** | Thread sonlandığında mesaj gönderilen iletişim kanalı. |
| **Thread Exit Status** | Thread'in sonlanma nedeni. |

**Önemli Not:** Thread nesnesinin en kritik niteliklerinden biri **Context**'tir. Bu, thread en son çalıştığında işlemci register'larının (yazmaçlarının) aldığı değerleri içerir. Bu

bilgi sayesinde thread askıya alınabilir (suspended) ve daha sonra kaldığı yerden devam ettirilebilir (resumed).

Harika bir bölüm daha. Burası işletim sistemleri dersinin en "baba" konularından biridir: **Thread Durumları (States)** ve **Solaris**'in kendine has (ve oldukça zekice olan) çok katmanlı yapısı.

Özellikle Windows'taki "Transition" durumu ve Solaris'teki "LWP" kavramı sınavlarda çok can yakar, o yüzden buraları senin için "salağa anlatır gibi" ama teknik terimleri koruyarak özetledim.

---

## Windows Multithreading (Çoklu İş Parçacığı) Yapısı

Windows, "concurrency" (eş zamanlılık) kavramını çok iyi uygular.

- **Concurrent (Eş Zamanlı Gibi):** Farklı process'lerin thread'leri sırayla çalışarak sanki aynı anda çalışıyormuş gibi hissettirir.
- **Simultaneous (Gerçekten Aynı Anda):** Aynı process içindeki birden fazla thread, eğer bilgisayarda çok çekirdek varsa, farklı çekirdeklerde *gerçekten* aynı anda çalışabilir.

Bu yapı sayesinde sunucu (server) uygulamaları çok verimli olur. Tek bir sunucu process'i, process açıp kapatma maliyetine girmeden binlerce müşteriye thread'ler sayesinde cevap verebilir.

---

## Windows Thread Durumları (Thread States)

Windows'ta bir thread şu 6 durumdan birinde olabilir. Bu döngüyü anlamak hayati önem taşır:

1. **Ready (Hazır):** Thread çalışmaya hazırdır. Kernel dağıtıcısı (dispatcher), hazır olan tüm thread'leri öncelik sırasına göre takip eder. "Hoca beni tahtaya kaldırsa da soruyu çözsem" diye bekleyen öğrenci gibidir.
2. **Standby (Beklemede - Eli Kulağında):** Bu durum Windows'a özgüdür ve genelde karıştırılır.
    - Bir thread belirli bir işlemci (çekirdek) üzerinde çalışmak üzere **seçilmiştir**.
    - O işlemcinin boşa çıkmasını bekliyordur.
    - Eğer Standby'daki thread'in önceliği çok yüksekse, o an çalışan thread işlemciden atılır (preempted) ve yerini buna bırakır.
3. **Running (Çalışıyor):** Kernel dağıtıcısı (context switch) işlemini yapar ve thread resmen çalışmaya başlar. Şunlar olana kadar çalışır:
    - Daha yüksek öncelikli biri gelir (Preemption).
    - Kendisine ayrılan süre (Time slice / Quantum) biter.
    - Bloklanır (Bir şey beklemeye başlar).
    - Sonlanır (İşi biter).

4. **Waiting (Bekliyor/Bloklanmış):** Thread şu nedenlerle durur:
    - Bir G/Ç (I/O) işleminin bitmesini bekliyordur (Diskten veri okuma vb.).
    - Senkronizasyon için gönüllü bekliyordur (Mutex/Semaphore bekliyordur).
    - Sistem tarafından askıya alınmıştır.
    - *Not: Beklediği şey gerçekleşince direkt "Running"e geçmez, sıraya girip "Ready" olur.*
5. **Transition (Geçiş):** Burası çok kritiktir. Thread çalışmaya hazırdır (Ready olmak ister) AMA kaynakları (örneğin bellek yığını/stack) RAM'de değildir, diske takaslanmıştır (paged out).
    - Kaynaklar RAM'e geri yüklendiğinde "Ready" durumuna geçer.
6. **Terminated (Sonlanmış):** Thread işini bitirmiştir veya öldürülmüştür. Temizlik işlemleri yapıldıktan sonra sistemden tamamen silinir.

---

## İşletim Sistemi Alt Sistemleri Desteği (OS Subsystems)

Windows, üzerinde farklı işletim sistemi kurallarını (Win32, POSIX vb.) çalıştırabilir. Ancak her sistemin process yaratma mantığı farklıdır. Windows Executive (Çekirdek yönetimi) bu farkları şöyle yönetir:

- **Win32 Mantığı:**
    - Win32'de bir process oluşturulduğunda, içinde çalışacak bir thread de *mutlaka* olmalıdır.
    - Bu yüzden Win32 sistemi, önce process'i yaratır, sonra dönüp process yöneticisinden "buna bir de thread ver" der.
- **POSIX Mantığı:**
    - POSIX standardında (Unix benzeri) process ve thread ayrımı Windows gibi değildir.
    - POSIX alt sistemi, Windows'tan bir process ister. Windows ona alttan alta bir thread de verir (çünkü Windows'ta thread'siz process olmaz) ama POSIX uygulaması bunu görmez, sadece process'i görür.
- **Kalıtım (Inheritance) Detayı:**
    - Bir kullanıcı (Client), bir sunucudan (Server) kendisi için bir process yaratmasını istediğinde; yeni process sunucunun özelliklerini değil, **kullanıcının** özelliklerini (erişim jetonu, öncelik vb.) miras alır. "Baban ben değilim, isteği yapan kullanıcı" der.

---

## 4.5 Solaris Thread ve SMP Yönetimi

Solaris (Oracle'ın Unix sistemi), işlemci kaynaklarını sömürmek için 4 katmanlı çok esnek bir yapı kurmuştur.

**Solaris'in 4 Temel Kavramı:**

1. **Process:** Bildiğimiz klasik UNIX süreci. Adres alanı, yığın (stack) ve process kontrol bloğu (PCB) vardır.
2. **User-level Threads (ULT - Kullanıcı Seviyesi Threadler):**
    - Tamamen uygulamanın içinde, bir kütüphane tarafından yönetilir.
    - **İşletim sistemi (Kernel) bunlardan habersizdir.** Görünmezdirler.
    - Çok hızlı yaratılır ve yönetilirler çünkü kernel devreye girmez.
3. **Lightweight Processes (LWP - Hafif Siklet Süreçler):**
    - İşte kilit nokta burasıdır. LWP, kullanıcı seviyesindeki thread (ULT) ile çekirdek (Kernel) arasındaki köprüdür/adapterdir.
    - Her LWP, bir Kernel Thread'e bağlanır.
    - Kernel, LWP'leri bağımsız olarak zamanlayabilir.
4. **Kernel Threads (Çekirdek Threadleri):**
    - İşlemci üzerinde fiziksel olarak çalıştırılan (dispatched) asıl varlıklardır.

**İlişki Nasıl Çalışır? (Şekil 4.12 Özeti)**

- Her **LWP**'nin karşısında mutlaka bir **Kernel Thread** vardır.
- Uygulama (Process) içinde birden fazla **ULT** olabilir.
- Bu ULT'ler, ellerindeki işi yapmak için bir LWP'ye ihtiyaç duyarlar.
- Eşzamanlılık (Concurrency) gerekmiyorsa, klasik Unix gibi 1 ULT = 1 LWP yapısı kullanılır.
- Ancak genelde: Bir process içinde çok sayıda ULT vardır, bunlar az sayıdaki LWP'yi paylaşarak Kernel'e ulaşırlar.

**Özetle Solaris:** "Ben Kernel'i çok yormayayım, kullanıcı kendi içinde ULT'ler ile oynasın, ne zaman gerçekten işlemciye ihtiyaç duyarsa LWP üzerinden bana gelsin" der.

Harika bir bölüm daha. Burası Solaris'in neden "Mühendislik Harikası" olarak görüldüğünü anlatan kısım. Özellikle **"Kesilmelerin (Interrupts) Thread Olarak İşlenmesi"** fikri, modern işletim sistemlerinde devrim niteliğindedir.

Senin için bu karmaşık yapıyı parçalara ayırdım ve teknik terimleri koruyarak özetledim.

---

## Motivasyon: Solaris Neden Böyle Yapıyor?

Solaris'in 3 katmanlı yapısı (ULT -> LWP -> Kernel Thread) kafa karıştırıcı görünse de amacı şudur: **Yönetimi Kolaylaştırmak.**

- **ULT (Kullanıcı Thread'i):** Programcı standart kütüphanelerle kodunu yazar.
- **LWP (Hafif Süreç):** İşletim sistemi devreye girer, ULT'yi alır ve bir LWP'ye haritalar (eşler).
- **Kernel Thread:** Asıl işi yapan işçidir. LWP buna bağlanır.

Böylece, eş zamanlılık (concurrency) ve yürütme işleri tamamen **Kernel Thread** seviyesinde yönetilir. Uygulamalar ise donanımla (disk, bellek vb.) konuşmak için **API (Sistem Çağrıları)** kullanır. "Bana bellek ver", "Dosyayı oku" gibi emirleri bu API üzerinden verirler.

---

## Süreç (Process) Yapısı: Unix vs. Solaris

Geleneksel Unix ile Solaris arasındaki farkı anlamak için "tekli koltuk" ile "minibüs" arasındaki farkı düşünebilirsin.

**Geleneksel UNIX Yapısı**

Bir Unix süreci şunları içerir:

- Process ID ve User ID (Kimlikler).
- Sinyal Tablosu (Dürtüldüğünde ne yapacak?).
- Dosya Tanımlayıcıları (Hangi dosyalar açık?).
- Bellek Haritası (Adres alanı).
- **İşlemci Durum Yapısı (Processor State):** *Tek bir* çekirdek yığını (kernel stack) içerir. Çünkü geleneksel Unix'te bir process tek bir iş yapardı.

**Solaris Yapısı**

Solaris temel yapıyı korur AMA "İşlemci Durum Yapısı"nı değiştirir. Tek bir blok yerine, **her LWP için bir veri bloğu içeren bir liste** tutar. Yani bir process içinde birden fazla çalışan parça (LWP) olabilir.

**Bir LWP Veri Yapısının İçinde Ne Var?**

- LWP ID (Kimlik).
- Öncelik (Priority).
- Sinyal Maskesi (Hangi sinyalleri kabul ediyor?).
- Kullanıcı Seviyesi Register'lar (LWP çalışmıyorken değerlerin saklandığı yer).
- Kernel Stack (Sistem çağrısı argümanları ve sonuçları için).
- Kaynak Kullanım Verisi.
- **Pointerlar:** Bağlı olduğu Kernel Thread'e ve Process'e işaret eden oklar.

---

## Thread Yürütme Durumları (Execution States)

Solaris'te bir thread'in (ve ona bağlı LWP'nin) yaşam döngüsü 6 durumdan oluşur.

1. **RUN (Koşmaya Hazır):** Thread hazır, "Biri beni işlemciye koysa da çalışsam" diye bekliyor.

2. **ONPROC (İşlemcide):** Thread şu an işlemci üzerinde aktif olarak çalışıyor.
3. **SLEEP (Uyuyor/Bloklanmış):** Thread bir olay bekliyor (örneğin diskten veri gelmesini). Olay gerçekleşene kadar çalışamaz.
4. **STOP (Durdu):** Thread durduruldu (Genelde hata ayıklama/debug işlemleri için).
5. **ZOMBIE (Zombi):** Thread sonlandı (terminate), işi bitti ama işletim sistemi henüz temizliğini yapmadı.
6. **FREE (Özgür/Boş):** Thread'in tüm kaynakları serbest bırakıldı, sistemden silinmeyi bekliyor.

**Döngü Nasıl Çalışır?**

- **ONPROC -> RUN:** Thread çalışırken süresi biterse veya daha öncelikli biri gelirse (Preemption), işlemciden atılır ve tekrar sıraya (RUN) geçer.
- **ONPROC -> SLEEP:** Thread bir sistem çağrısı yapar (örn: "Dosyayı oku") ve cevabı beklemek zorundaysa uyku moduna geçer.

---

## Devrimsel Özellik: Kesilmelerin (Interrupts) Thread Olarak İşlenmesi

Çoğu işletim sistemi dünyayı ikiye ayırır: **Süreçler** ve **Kesilmeler (Interrupts)**.

- Normalde bir Interrupt gelince (örn: Klavye tuşuna basıldı), işlemci elindeki işi bırakır, "Panik Modu"na geçer ve o kesilmeyi işler. Bu sırada veri karışmasın diye diğer kesilmeleri engeller (blocklar). Bu durum çok çekirdekli sistemlerde yavaşlığa neden olur.

**Solaris'in Çözümü: Interrupt = Thread** Solaris, kesilmeleri özel birer "Kernel Thread"e dönüştürür.

1. **Her Şey Thread:** Bir kesilme geldiğinde, onu işlemek için özel bir kimliği, önceliği ve yığını olan bir "Interrupt Thread" atanır.
2. **Senkronizasyon:** Veri koruması için tüm sistemi kilitlemek yerine, normal thread'lerde kullanılan **Mutex** (Kilit) mekanizmalarını kullanırlar.
3. **Yüksek Öncelik:** Bu thread'ler en yüksek önceliğe sahiptir.

**Nasıl Çalışır?** Bir kesilme geldiğinde:

1. O an çalışan normal thread **"Pinned" (Sabitlenmiş)** duruma geçer. Yani işlemciden atılmaz, olduğu yerde dondurulur.
2. Havuzda hazır bekleyen (yeni yaratılmasına gerek olmayan) bir Interrupt Thread devreye girer ve işi halleder.
3. Eğer bu Interrupt Thread kilitli bir veriye ulaşmaya çalışırsa, o da "Bekle" durumuna geçebilir (Eskiden olsa sistemi kilitlerdi).

**Sonuç:** Bu yöntem, özellikle çok işlemcili sistemlerde geleneksel yöntemden çok daha yüksek performans sağlar.

Harika, Linux kısmına geldik. Burası işletim sistemleri dünyasının en pratik ve modern kısmıdır. Özellikle "Container" (Docker vb.) teknolojilerinin temelini burada göreceksin.

Metni senin için teknik terimleri koruyarak (Türkçe açıklamalarıyla) çevirdim ve özetledim.

---

## 4.6 Linux Süreç (Process) ve İş Parçacığı (Thread) Yönetimi

### Linux Görevleri (Linux Tasks)

Linux'ta bir process (süreç) veya task (görev), `task_struct` adı verilen bir veri yapısı ile temsil edilir. Bu yapı, process hakkında her şeyi içeren bir "kimlik kartı" gibidir. İçinde şu kategorilerde bilgiler bulunur:

1. **State (Durum):** Process'in o anki yürütme durumu (Çalışıyor, durdu, zombi vb.). Detayları aşağıda.
2. **Scheduling Information (Zamanlama Bilgisi):** Linux'un bu process'i ne zaman çalıştıracağına karar vermesi için gerekenler.
    - Process'ler "Normal" veya "Gerçek Zamanlı (Real-time)" olabilir.
    - Gerçek zamanlı olanlar her zaman önceliklidir.
3. **Identifiers (Kimlikler):** Her process'in kendine ait bir **PID** (Process ID), kullanıcı ve grup kimlikleri (UID/GID) vardır.
4. **Interprocess Communication (IPC):** Process'ler arası haberleşme mekanizmaları.
5. **Links (Bağlantılar):** Process'in ailesiyle olan bağları.
    - Parent (Ebeveyn - Onu yaratan).
    - Siblings (Kardeşler - Aynı ebeveynden olanlar).
    - Children (Çocuklar - Kendi yarattıkları).
6. **Times and Timers (Zamanlar ve Zamanlayıcılar):** Yaratılma zamanı ve harcadığı işlemci süresi. Ayrıca process'e sinyal yollayan zamanlayıcılar (alarm gibi).
7. **File System:** Açık dosyaların ve şu anki klasörün (directory) adresleri.
8. **Address Space (Adres Uzayı):** Process'e ayrılan sanal bellek alanı.
9. **Processor-specific Context:** Process o an durdurulursa, kaldığı yerden devam edebilmesi için saklanan Register (yazmaç) ve Stack (yığın) bilgileri.

### Şekil 4.15: Linux Process Yürütme Durumları

Linux'ta bir process şu durumlardan birinde olabilir:

1. **Running (Çalışıyor):** Bu durum iki şeyi kapsar:
    - Process şu an gerçekten işlemcide çalışıyor.
    - Veya çalışmaya hazır (Ready), sıra bekliyor.
    - *Linux, "Ready" ve "Running" ayrımını diğer sistemler gibi keskin yapmaz, ikisini de bu statüde tutar.*

2. **Interruptible (Kesilebilir):** Bu bir "bekleme/bloklanma" durumudur. Process bir olay bekliyordur (örneğin klavyeden tuşa basılması). Sinyal gelirse uyanır.
3. **Uninterruptible (Kesilemez):** Bu da bir bekleme durumudur ama çok kritiktir. Process doğrudan donanım bekliyordur (örneğin diske veri yazıyordur).
    - **Fark:** Bu durumdaki process'e sinyal (örneğin "Öldür/Kill" emri) yollasanız bile cevap veremez. "Şu an ameliyattayım, kimse beni dürtmesin" modudur.
4. **Stopped (Durduruldu):** Process başka bir process tarafından (genelde Debugger) durdurulmuştur.
5. **Zombie (Zombi):** Process ölmüştür (terminate olmuştur) ama ruhu (task yapısı) hala process tablosundadır.
    - Neden? Ebeveyn process henüz "Çocuğum öldü mü?" diye bakıp (wait çağrısı yapıp) raporunu almamıştır.

---

**Linux Thread'leri (Linux Threads)**

İşte burası Linux'un en "trick" noktasıdır ve sınavlarda çok çıkar.

- **Geleneksel UNIX:** Process ve Thread tamamen ayrı dünyalardır.
- **Linux Yaklaşımı:** Linux çekirdeği (kernel), **Process ve Thread arasında bir ayrım tanımaz.**
    - Linux için hepsi birer **"Task" (Görev)** dir.
    - Thread dediğimiz şey, sadece **bazı kaynaklarını (bellek, dosyalar vb.) ebeveyniyle paylaşan bir process'tir.**

**Nasıl Yaratılır? (`clone` vs `fork`)** Normalde Unix'te process yaratmak için `fork()` kullanılır. Linux'ta ise bunun yerine çok daha yetenekli olan `clone()` komutu vardır.

- **`fork()`:** Her şeyi kopyalar. Yeni process, eskisinin birebir kopyasıdır ama tamamen ayrı evde yaşar.
- **`clone()`:** Hangi kaynakların paylaşılacağını seçebildiğiniz bir kopyalama yapar.
    - Eğer `clone()` ile "belleği paylaş (`CLONE_VM`)" derseniz, teknik olarak elinizde iki process olur ama aynı belleği kullandıkları için bunlar pratikte **Thread** gibi davranır.

**Önemli `clone()` Bayrakları (Flags):** Bu bayraklar, yeni yaratılan process'in ebeveyniyle neleri paylaşacağını belirler:

- `CLONE_VM`: Sanal belleği paylaşır (Thread olmanın temel kuralı).
- `CLONE_FS`: Dosya sistemi bilgilerini (hangi klasörde olduğu vb.) paylaşır.
- `CLONE_FILES`: Açık dosya tanımlayıcılarını paylaşır (Biri dosyayı kapatırsa, diğerinde de kapanır).
- `CLONE_PARENT`: Ebeveyni paylaşır (Kardeş olurlar).

- `CLONE_THREAD`: Aynı thread grubuna dahil eder.

---

**Linux İsim Alanları (Namespaces)**

Bu konu modern "Container" (Docker, Kubernetes) teknolojisinin temelidir.

- **Namespace Nedir?** Bir process'e (veya gruba) sistemin sadece belirli bir kısmını gösteren bir soyutlamadır.
- **Amaç:** Hafif sanallaştırma (Lightweight Virtualization). Process'e "Sistemde tek başına sensin" ilüzyonu yaratır.
- **Türleri (6 Adet):** `mnt` (dosya sistemi), `pid` (process ID'leri), `net` (ağ), `ipc`, `uts` (host adı), `user`.

Örneğin `CLONE_NEWPID` bayrağı ile bir process yaratırsanız, o process kendi dünyasında PID'si 1 olan (sistemin yöneticisi gibi) bir process olduğunu sanar. Docker konteynerleri işte tam olarak böyle çalışır; `clone()` komutunu özel bayraklarla kullanarak izole ortamlar yaratırlar.

---

## 1. Namespaces (İsim Alanları) = "Görünmez Duvarlar"

Namespace'ler, bir process'e "Sistemde sadece sen varsın, başka kimse yok" yalanını söyleyen mekanizmadır. Process'i izole bir odaya kapatır.

İşte Linux'taki 6 temel Namespace ve görevleri:

1. **Mount Namespace (Dosya Sistemi Görünümü):**
    - **Olayı:** Process'e özel bir disk/dosya yapısı gösterir.
    - **Örnek:** Sen C: sürücüsüne baktığında Windows dosyalarını görürsün, ama bu namespace içindeki process C:'ye baktığında tamamen boş veya farklı dosyalar görebilir.
2. **UTS Namespace (UNIX Timesharing - İsimlendirme):**
    - **Olayı:** Bilgisayarın ağ üzerindeki adını (Hostname) farklı gösterir.
    - **Örnek:** Gerçek bilgisayarın adı "AnaSunucu" olsa bile, namespace içindeki process kendini "WebSunucusu-1" sanar.
3. **IPC Namespace (İletişim İzolasyonu):**
    - **Olayı:** Process'ler arası haberleşme (mesaj kuyrukları, semaforlar) kanallarını ayırır.
    - **Örnek:** Yan odadaki process bağırsa bile (sinyal yollasa), bu odadaki onu duymaz.
4. **PID Namespace (Kimlik İzolasyonu):**
    - **Olayı:** Process ID (PID) numaralarını izole eder.

    - **Örnek:** İşletim sisteminde PID 1 her zaman "init" (sistemi başlatan) sürecidir. PID Namespace sayesinde, bir konteyner içindeki process de kendini PID 1 (patron) sanabilir. Dışarıdan bakıldığında ise PID 4532'dir.
    - **Kritik Özellik (CRIU):** Bu özellik sayesinde çalışan bir programı "dondurup" (Save Game yapar gibi) dosyaya kaydedebilir, sonra başka bir bilgisayarda kaldığı yerden devam ettirebilirsin.
5. **Network Namespace (Ağ İzolasyonu):**
    - **Olayı:** Her process grubuna kendine ait sanal ağ kartı, IP adresi ve port numaraları verir.
    - **Örnek:** Bilgisayarda tek bir 80 portu vardır ama bu özellik sayesinde 50 farklı process kendi 80 portunu kullanıyormuş gibi çalışabilir.
6. **User Namespace (Kullanıcı İzolasyonu):**
    - **Olayı:** İçerideki "Yönetici (Root)", dışarıdaki "Standart Kullanıcı" olabilir.
    - **Örnek:** Bir process konteynerin içinde her şeyi silme yetkisine (Root) sahipken, gerçek sistemde hiçbir yetkisi olmayan bir stajyer gibidir. Bu, güvenlik için hayati önem taşır.

---

## 2. Linux Cgroups (Kontrol Grupları) = "Bütçe Kontrolü"

Namespace'ler process'i gizliyordu (Duvar örüyordu). **Cgroups** ise process'in ne kadar kaynak tüketeceğini sınırlar (Musluğu kısar).

- **Görevi:** İşlemci (CPU), Bellek (RAM), Disk (I/O) ve Ağ kullanımını sınırlar ve raporlar.
- **Tarihçesi:** Google mühendisleri tarafından 2006'da "Process Containers" adıyla başlatıldı.
- **Nasıl Çalışır?** İşletim sistemine yeni bir sistem çağrısı eklemez. Bunun yerine "Sanal Dosya Sistemi" (VFS) kullanır. Yani /sys/fs/cgroup klasöründeki dosyalara yazarak ayar yaparsın.
- **Kullanım Alanı:** Gömülü sistemlerden dev sunuculara kadar her yerde kaynak yönetimi için kullanılır.

Örnek:

Bir process'e "Sen en fazla 512 MB RAM ve %20 CPU kullanabilirsin" diyebilirsin. Eğer sınırı aşarsa Linux ensesine vurur (Process'i durdurur veya yavaşlatır).

---

## Büyük Resim: Konteyner (Container) Nedir?

Hani Docker, Kubernetes diyoruz ya; işte formülü şudur:

$$\text{Container} = \text{Namespaces (İzolasyon)} + \text{Cgroups (Kaynak Sınırlandırma)}$$

- **Namespaces** sayesinde process kendini tek başına sanır.
- **Cgroups** sayesinde process bilgisayarı sömüremez.
- Sonuç: Sanal makine kurmadan, çok hafif ve hızlı sanal ortamlar elde edersin.

## 4.7 Android Süreç (Process) ve İş Parçacığı (Thread) Yönetimi

Android'in process yönetimini anlamak için önce **Uygulama (Application)** ve **Aktivite (Activity)** kavramlarını bilmemiz gerekir.

### 1. Android Uygulama Bileşenleri

Bir Android uygulaması, 4 temel bileşenden oluşur. Her biri bağımsız çalışabilir ve hatta başka uygulamalar tarafından tetiklenebilir:

1. **Activities (Aktiviteler):**
    - Kullanıcının gördüğü her bir "ekran" bir Activity'dir.
    - **Örnek:** E-posta uygulamasında "Gelen Kutusu" bir aktivitedir, "E-posta Yazma Ekranı" başka bir aktivitedir.
    - **Özellik:** Dışarıya açık (exported) aktiviteler başka uygulamalarca başlatılabilir. (Örn: Kamera uygulamasının, e-posta uygulamasındaki "Mail Yaz" ekranını açması).
2. **Services (Servisler):**
    - Uzun süren işlemleri arka planda yapan görünmez işçilerdir.
    - **Amaç:** Ana ekranı (UI Thread) dondurmamak.
    - **Örnek:** Müzik çalarken başka uygulamada gezebilmen, Spotify'ın bir "Service" olarak arka planda çalışması sayesindedir.
3. **Content Providers (İçerik Sağlayıcılar):**
    - Uygulamalar arası veri paylaşımını yöneten arayüzdür.
    - **Örnek:** Rehber (Contacts) verisi bir Content Provider'dır. WhatsApp bu sayede senin rehberine erişebilir (izin verirsen).
4. **Broadcast Receivers (Yayın Alıcıları):**
    - Sistemden veya diğer uygulamalardan gelen "anonsları" dinler.
    - **Örnek:** "Pil Düşük", "Ekran Kapandı" veya "Wifi Bağlandı" uyarılarını yakalayıp ona göre işlem yapar.

### 2. Sandboxing (Kum Havuzu) Modeli

Android, güvenlik için **Sandboxing** modelini kullanır:

- Her uygulama kendi **Özel Sanal Makinesi (Virtual Machine - ART/Dalvik)** üzerinde ve kendi **Linux Process**'i içinde çalışır.
- Her uygulama ayrı bir Linux kullanıcısı (User ID) gibi davranır.
- **Sonuç:** Bir uygulama, izin verilmediği sürece diğerinin verisine veya belleğine dokunamaz. Tam izolasyon sağlar.

---

### 3. Aktiviteler (Activities) ve Back Stack

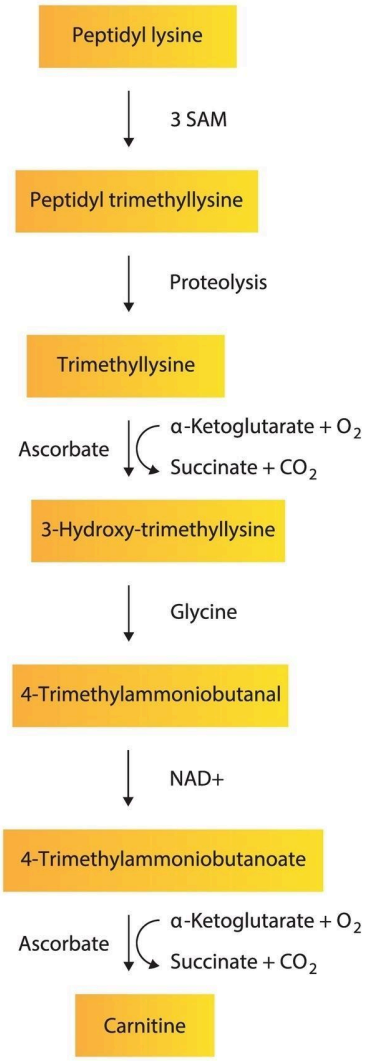
- **Aktivite:** Kullanıcının etkileşime girdiği penceredir. Genelde tam ekrandır.

- **Back Stack (Geri Yığını):** Aktiviteler LIFO (Last-In, First-Out / Son Giren İlk Çıkar) mantığıyla bir yığın (stack) içinde tutulur.
- **Nasıl Çalışır?**
    - Yeni bir ekran açtığında, o ekran yığının tepesine (Top) gelir ve odak ona geçer.
    - "Geri" tuşuna bastığında, en üstteki ekran yığından atılır (pop edilir) ve bir alttaki ekran tekrar görünür olur.

---

**4. Aktivite Yaşam Döngüsü (Activity Lifecycle)**

Bir aktivitenin durumu sürekli değişir. Bu döngü, Android programlamanın en temel konusudur.




Shutterstock

**Temel Durumlar ve Geçişler:**

1. **Başlangıç (Launch):**
    - `onCreate()`: Statik kurulumlar yapılır (Butonları yerleştir vb.).
    - `onStart()`: Aktivite kullanıcıya görünür hale gelir.
    - `onResume()`: Kullanıcı artık etkileşime geçebilir (Dokunabilir).
    - **Resumed State:** Aktivite şu an en önde ve çalışıyor.
2. **Duraklatma (Pause):**

    - Kullanıcı başka bir eyleme geçerse (örneğin yarım ekran bir bildirim gelirse veya başka bir aktivite açılırsa), mevcut aktivite `onPause()` çağrısı ile **Paused** durumuna geçer.
    - **Not:** Paused durumunda aktivite hala kısmen görünür olabilir ama odak onda değildir.
3. **Durdurma (Stop):**
    - Kullanıcı tamamen başka bir uygulamaya geçerse veya "Home" tuşuna basarsa, aktivite `onStop()` ile **Stopped** durumuna geçer.
    - Aktivite artık görünmezdir ama bilgileri hala hafızadadır.
4. **Yok Etme (Destroy):**
    - Kullanıcı "Geri" tuşuyla uygulamadan çıkarsa veya sistem bellek kazanmak isterse `onDestroy()` çağrılır ve aktivite tamamen ölür.

**Kritik Bilgi:** Kullanıcı uygulamadan çıkıp geri döndüğünde, yığının (Stack) en üstündeki aktivite `onRestart()` -> `onStart()` -> `onResume()` sırasıyla tekrar canlanır. Bu yüzden, uygulama arka plana atıldığında verileri kaydetmek (save state) geliştiricinin sorumluluğundadır.

Harika bir devam bölümü. Android'in **"Bellek Yönetimi"** (RAM dolunca kimi öldüreceğine nasıl karar verdiği) ve Apple'ın yazılımcıların hayatını kurtaran **GCD (Grand Central Dispatch)** teknolojisi anlatılıyor.

Senin için metni teknik terimleri koruyarak, Türkçe açıklamalarıyla ve ders notu formatında özetledim.

---

## 1. Android: Uygulamayı Öldürmek (Killing an Application)

Android sisteminde RAM (Ana Bellek) dolarsa, sistemin nefes alabilmesi için birilerini feda etmesi gerekir.

- **Mekanizma:** Sistem, bellek kazanmak için aktiviteleri veya doğrudan o uygulamanın sürecini (process) sonlandırır (kill).
- **Kullanıcı Deneyimi:** Kullanıcı uygulamanın "öldüğünü" fark etmez. Geri döndüğünde, sistem o aktiviteyi yeniden yaratır.
- **Kimi Öldürür?:** Yığın tabanlı (Stack-oriented) çalışır. Genelde **"En Az Son Kullanılan" (Least Recently Used - LRU)** uygulamalar ilk kurban olur. Ön planda (Foreground) çalışanlar en güvendedir.

---

## 2. Android Süreç Hiyerarşisi (Kim Önce Gider?)

Sistem sıkıştığında hangi sürecin (process) önce öldürüleceğini belirleyen bir "Öncelik Hiyerarşisi" vardır. En alttakiler ilk ölür.

Sırasıyla (En önemliden en önemsize):

1. **Foreground Process (Ön Plan Süreci - DOKUNULMAZ):**
    - Kullanıcının şu an etkileşimde olduğu, parmağının ucundaki uygulamadır. (Örn: Şu an yazdığın WhatsApp ekranı).
2. **Visible Process (Görünür Süreç):**
    - Ön planda değil ama hala ekranda görünüyor.
    - **Örnek:** Ön planda küçük bir diyalog penceresi açıktır, arkadaki uygulama hala flu şekilde görünüyordur. Onu öldürürsen görüntü bozulur.
3. **Service Process (Servis Süreci):**
    - Ekranda görünmez ama arka planda iş yapar.
    - **Örnek:** Arka planda müzik çalan Spotify veya dosya indiren Chrome. Müzik kesilirse kullanıcı kızar, o yüzden kolay kolay öldürülmez.
4. **Background Process (Arka Plan Süreci - KURBAN ADAYI):**
    - Kullanıcının "Home" tuşuna basıp aşağı indirdiği, **Stopped** durumundaki aktivitelerdir. Bunlardan çok sayıda vardır ve RAM dolunca ilk bunlar temizlenir.
5. **Empty Process (Boş Süreç - İLK GİDEN):**
    - İçinde aktif hiçbir bileşen yoktur.
    - **Neden var?** Sırf "Cache" (Önbellek) amaçlı tutulur. Kullanıcı uygulamayı tekrar açarsa sıfırdan RAM ayırmakla uğraşmamak için kabuğu hazır bekletilir. İlk gözden çıkarılan budur.

---

## 3. Mac OS X: Grand Central Dispatch (GCD)

Burası çok önemli. Apple, çok çekirdekli işlemcileri (Multicore) programlamayı kolaylaştırmak için **GCD** adında devrimsel bir teknoloji geliştirdi.

- **Sorun:** Eskiden "Thread" (İş parçacığı) yaratmak, yönetmek, kilitlemek (lock) çok zordu.
- **Çözüm (GCD):** Sen "Ne yapılacağını" söylüyorsun (Block), GCD ise "Nasıl ve Hangi Thread'de yapılacağını" kendisi hallediyor.

**Temel Kavramlar**

1. **Block (Blok):** C, C++ veya Objective-C dillerine eklenmiş bir eklentidir. Kendi kendine yeten, kod ve veriyi paketleyen bir iş birimidir. (Senin C# veya JS'deki "Lambda Expression" `{ () => ... }` mantığına benzer).
2. **Queue (Kuyruk):** Bloklar bu kuyruklara atılır. FIFO (İlk giren ilk çıkar) mantığıyla çalışır.
    - **Serial Queue:** İşleri sırayla yapar (Biri bitmeden diğeri başlamaz).
    - **Concurrent Queue:** İşleri aynı anda (paralel) yapar.

**GCD Nasıl Çalışır?**

Geliştirici manuel olarak Thread yaratmaz. İşi (Block'u) bir kuyruğa atar. İşletim sistemi (OS), çekirdek sayısına ve yoğunluğa bakarak "Thread Pool" (İş parçacığı havuzu) içinden en uygun thread'i seçer ve işi ona verir. Bu yöntem, manuel thread yönetiminden çok daha verimli ve az maliyetlidir.

**Gerçek Hayat Örneği (Belge Analizi)**

Metindeki örnek, bir butona basıldığında büyük bir belgeyi analiz eden bir uygulamayı anlatıyor.

**Eski/Kötü Yöntem:** Analizi "Main Thread"de yaparsan, işlem bitene kadar arayüz donar. Kullanıcı butona basar, 3 saniye ekran kilitlenir (Mac'teki o meşhur dönen renkli top çıkar).

**GCD Yöntemi (Kodun Mantığı):**

```
C
// 1. İşi Arka Plana At (Global Queue)
dispatch_async(dispatch_get_global_queue(...), ^{
   // Burası arka planda çalışır, arayüzü dondurmaz.
   AnalyzeDocument();

   // 2. Sonucu Ekrana Yazmak için Ana Kuyruğa Dön (Main Queue)
   dispatch_async(dispatch_get_main_queue(), ^{
      // Burası arayüzü günceller.
      UpdateUI();
   });
});
```

- **Adım 1:** Ağır işi (Analyze) alıp global bir kuyruğa (arka plana) atıyor. Böylece arayüz donmuyor.
- **Adım 2:** İş bitince, sonucu ekrana yazdırmak için tekrar "Main Queue"ya (arayüz thread'ine) dönüyor. Çünkü arayüz güncellemeleri *sadece* ana thread'de yapılmalıdır.

## 4.9 ÖZET (Summary)

İşletim sistemleri **Process (Süreç)** ve **Thread (İş Parçacığı)** kavramlarını birbirinden ayırır.

- **Process:** Kaynakların sahibi olan yapıdır (Ev sahibi).
- **Thread:** Programın yürütülmesinden sorumlu olan yapıdır (Evdeki çalışanlar).

Bu ayrım verimlilik sağlar. Tek bir process içinde birden fazla eş zamanlı (concurrent) thread olabilir. İki ana yöntem vardır:

1. **User-Level Threads (ULT - Kullanıcı Seviyesi Threadler):**
    - İşletim sistemi (OS) bunları bilmez. Bir kütüphane tarafından yönetilir.
    - **Avantajı:** Çok hızlıdır (mod değişimi gerekmez).

    - **Dezavantajı:** Biri bloklanırsa (takılırsa), tüm process durur. OS sadece tek bir iş görüp onu durdurur.
2. **Kernel-Level Threads (KLT - Çekirdek Seviyesi Threadler):**
    - OS tarafından yönetilir.
    - **Avantajı:** Biri bloklansa bile diğerleri çalışmaya devam eder. Çok işlemcili sistemlerde paralel çalışabilirler.
    - **Dezavantajı:** Thread değiştirmek için Kernel'e girip çıkmak (Mode Switch) gerekir, bu da biraz daha yavaştır.

---

## BÖLÜM PROBLEMLERİ VE ÇÖZÜMLERİ

Sınavlarda ve mülakatlarda karşına çıkabilecek en önemli soruları ve cevaplarını senin için derledim:

**Soru 4.1: Mode Switch Maliyeti**

**Soru:** Aynı process içindeki iki thread arasında geçiş yapmak (Mode Switch), farklı process'lerdeki iki thread arasında geçiş yapmaktan daha mı az iş gerektirir? **Cevap: Evet.**

- Çünkü aynı process içindeki thread'ler aynı **adres alanını (Address Space)** paylaşır.
- Farklı process'lere geçerken bellek haritalarını, önbellekleri (cache) ve TLB'yi temizleyip yeniden yüklemek gerekir. Aynı process içinde bu veriler zaten hazırdır, sadece işlemci register'larını (yazmaçlarını) değiştirmek yeterlidir.

**Soru 4.2: ULT Dezavantajı**

**Soru:** Bir ULT sistem çağrısı yapıp bloklandığında neden tüm process bloklanır? **Cevap:**

- İşletim sistemi (Kernel), içeride thread'ler olduğunu bilmez; sadece tek bir process görür.
- Process bir sistem çağrısı yapıp "beklemeye" geçtiğinde, Kernel "Tamam, bu process bekliyor" der ve onu işlemciden atar. İçeride çalışmaya hazır başka thread'ler olsa bile Kernel bunu göremez.

**Soru 4.3: OS/2 ve Oturum (Session) Kavramı**

**Soru:** OS/2'de Session, Process ve Thread ayrımı vardır. Session'ları kaldırıp sadece Process ve Thread bıraksaydık ne kaybederdik? **Cevap:**

- **a. Kayıp:** Bir kullanıcının aynı anda birden fazla process kullanan bir uygulamayı (örneğin bir ofis paketi) tek bir grup olarak yönetme yeteneğini kaybederdik. Session, klavye ve mouse girdisinin doğru pencere grubuna gitmesini sağlar.
- **b. Kaynak Atama:** Kaynakları (bellek, dosyalar) **Process** seviyesinde atamak en mantıklısıdır. Thread'ler sadece yürütme birimidir, kaynak sahibi değildir.

**Soru 4.4: Tek İşlemcide Multithreading Hızı**

**Soru:** Tek işlemcili (Uniprocessor) bir bilgisayarda bile multithreaded (1:1 model) programlar neden single-threaded olanlardan daha hızlı çalışabilir? **Cevap:**

- **G/Ç (I/O) Beklemeleri Yüzünden.**
- Tek thread'li bir program diskten veri beklerken işlemci tamamen boş durur.
- Multithread'li programda ise, bir thread disk beklerken, diğer thread işlemciyi kullanıp hesaplama yapmaya devam edebilir. İşlemci boş yatmaz.

**Soru 4.5: Process Ölürse Thread Yaşar mı?**

**Soru:** Bir process sonlanırsa (exit), ona ait çalışan thread'ler çalışmaya devam eder mi? **Cevap: Hayır.**

- Thread'ler, process'in kaynaklarını (bellek, dosya tutaçları) kullanır. Process ölünce kaynaklar gider, ev yıkılırsa içindekiler de ölür.

**Soru 4.6: Global (ASCB) ve Yerel (TCB) Kontrol Blokları**

**Soru:** OS/390'da kontrol bilgilerini Global (ASCB - Bellekte sabit) ve Yerel (TCB - Adres alanında) olarak ikiye ayırmanın avantajı nedir? **Cevap:**

- İşletim sistemi, bir process **swapped out** (RAM'den diske atılmış) olsa bile onun hakkında bilgiye ihtiyaç duyabilir (önceliği nedir, ne kadar RAM kullanıyor vb.).
- **ASCB (Global):** Her zaman RAM'dedir, OS diske atılan process'i bile buradan yönetebilir.
- **TCB (Yerel):** Sadece process çalışırken lazımdır, process ile birlikte diske gidip gelebilir. RAM tasarrufu sağlar.

**Soru 4.7 & 4.8: C Dili ve Concurrency (Kod Analizi)**

**Senaryo:** `update()` fonksiyonu `count` değişkenine 1 ekliyor. `count += 1;` işlemi aslında makine dilinde 3 adımdır: (1) Oku, (2) Ekle, (3) Yaz.

**Soru:** İki thread aynı anda `update()` çağırırsa ne olur? **Cevap:**

- **Yarış Durumu (Race Condition):** İkisi de aynı anda `count` değerini (diyelim ki 5) okuyabilir.
    - Thread A: 5 okudu, 1 ekledi (6 yaptı), yazdı.
    - Thread B: (A yazmadan önce) 5 okudu, 1 ekledi (6 yaptı), yazdı.
- Sonuç: `count` 7 olması gerekirken 6 olur. Veri kaybı yaşanır. C dili veya derleyiciler (compiler) varsayılan olarak thread güvenliğini (thread-safety) garanti etmez. `Mutex` veya `Lock` kullanmak zorunludur.

**Soru 4.9: Pthreads (Kod Analizi)**

**Soru:** Verilen kodda `fork()` yerine `pthread_create()` kullanılmış ve bir global değişken (`myglobal`) arttırılmış. **a. Ne yapıyor?** Ana thread ve yeni yaratılan thread, global değişkeni 20'şer kere arttırıyor. **b. Çıktı Neden 21? (Beklenen 40 veya 42 olmalıydı):**

- Yine **Race Condition**. Thread'ler aynı değişkene korumasız saldırdığı için bazı artırma işlemleri birbirini eziyor (overwrite). Çıktı her çalıştırmada farklı ve hatalı olabilir.

**Soru 4.10: Solaris Yield (Yol Verme)**

**Soru:** Bir thread `yield()` dediğinde (sırasını saldığında), neden kendisinden daha yüksek öncelikli birine değil de "aynı öncelikli" birine sıra verir? **Cevap:**

- Çünkü daha yüksek öncelikli bir thread hazır olsaydı, sistem zaten **Preemption** (Öncelikli Kesme) yaparak o anki thread'i atıp yükseği çalıştırırdı. `Yield` sadece "Benim acilem yok, aynı seviyedeki arkadaşlarım beklemesin" demektir.

**Soru 4.12: Linux Kesilemez (Uninterruptible) Durumu**

**Soru:** Linux'ta neden "Kesilemez" (Uninterruptible) bir durum var? **Cevap:**

- Bazı donanım işlemleri (örneğin disk sürücüsüne bir komut yollamak) yarıda kesilirse donanım kararsız hale gelebilir veya veri bozulabilir.
- Bu işlemler çok kısa sürer ve "Lütfen bitene kadar beni öldürme veya sinyal yollama" demek için bu mod kullanılır.

## Giriş: Neden Eşzamanlılık (Concurrency)?

İşletim sistemlerinin temel olayı şudur: Kaynakları (CPU, RAM, Disk) yönetmek.

- **Multiprogramming:** Tek işlemci var ama birden fazla program çalışıyor gibi görünüyor (sırayla hızlıca geçiş yaparak).
- **Multiprocessing:** Birden fazla işlemci var, gerçekten aynı anda birden fazla iş yapılıyor.
- **Distributed Processing:** İşler birden fazla bilgisayara dağıtılmış (Cluster sistemler).

Hangi sistem olursa olsun, temel sorun şudur: **"Aynı kaynağa (örneğin bir değişkene veya dosyaya) iki kişi aynı anda saldırırsa ne olur?"**

Cevap: Kaos olur. Veriler bozulur. İşte bu yüzden **Mutual Exclusion (Karşılıklı Dışlama)** kavramına ihtiyacımız var. Yani, "Ben yazarken sen bekle, sen yazarken ben bekleyeyim."

---

## Tablo 5.1: Eşzamanlılık Sözlüğü (Sınavda %100 Çıkar)

Bu terimler işletim sistemlerinin alfabesidir, çok iyi bilmen lazım:

1. **Atomic Operation (Atomik İşlem):**
    - Ya hep ya hiç işlemidir. İşlem sırasında kimse araya giremez.
    - *Örnek:* Bankada para transferi. Hesabından para düşmesi ve karşıya gitmesi tek bir atomik işlemdir. Para düşüp karşıya gitmezse olmaz. Yarıda kesilemez.
2. **Critical Section (Kritik Bölge):**
    - Kodun içinde paylaşılan kaynağa (shared resource) erişilen kısımdır.
    - *Kural:* Burası mayın tarlasıdır. Aynı anda sadece bir process burada olabilir.
3. **Deadlock (Ölümcül Kilitlenme):**
    - İki inatçı keçinin köprüde karşılaşması.
    - Process A, B'nin elindekini bekliyor; Process B, A'nın elindekini bekliyor. İkisi de sonsuza kadar bekler.
4. **Livelock (Canlı Kilitlenme):**
    - Deadlock'a benzer ama process'ler durmaz, sürekli durum değiştirir ama iş yapamaz.
    - *Örnek:* Koridorda biriyle karşılaştın, sen sağa geçtin o da senin sağına geçti, sola geçtin o da sola geçti. Hareket var ama ilerleme yok.
5. **Mutual Exclusion (Karşılıklı Dışlama):**
    - Tekillik ilkesi. "Kritik Bölgeye" aynı anda sadece bir kişi girebilir kuralı.
6. **Race Condition (Yarış Durumu):**
    - Birden fazla process'in aynı veriyi okuyup yazmaya çalışması ve sonucun "kimin önce bitirdiğine" bağlı olarak şansa kalması durumu. (Bunu istemeyiz, sonuç tutarlı olmalı).
7. **Starvation (Açlık):**
    - Bir process'in sürekli sırasını başkalarına kaptırması ve bir türlü çalışamaması. Kaynak var ama ona verilmiyor.

---

## 5.1 Yazılımsal Çözümler (Software Approaches)

Donanımın özel desteği olmadan, sadece kod yazarak **Mutual Exclusion**'ı nasıl sağlarız?

Burada **Busy Waiting (Meşgul Bekleme)** dediğimiz bir yöntem kullanılır.

- **Mantık:** Process, sıranın kendisine gelip gelmediğini sürekli kontrol eder.
- **Dezavantaj:** İşlemciyi boş yere yorar. "Sıra bende mi? Hayır. Sıra bende mi? Hayır..." diye sürekli sorar.

**Dekker Algoritması: İlk Deneme (First Attempt)**

Dijkstra abimiz, Dekker'in algoritmasını aşama aşama anlatır. İlk denemede **"Turn" (Sıra)** değişkeni kullanılır.

**Nasıl Çalışır?** Ortak bir global değişkenimiz var: `turn`.

- Eğer `turn = 0` ise, Process 0 (P0) kritik bölgeye girer.
- Eğer `turn = 1` ise, Process 1 (P1) kritik bölgeye girer.

**Kodun Mantığı (P0 için):**

1. Bak bakalım `turn` 0 mı? Değilse, `turn` 0 olana kadar dönüp dur (Busy wait).
2. `turn` 0 olunca içeri gir (Kritik Bölge). İşini yap.
3. Çıkarken `turn` değerini 1 yap (Sırayı P1'e ver).

**Bu Yöntemin Sorunları (Neden Kullanmıyoruz?):** Bu çözüm çalışır ama iki devasa sorunu vardır:

1. **Strict Alternation (Katı Sıralama):** Process'ler sırayla çalışmak zorundadır (P0 -> P1 -> P0 -> P1...).
    - *Sorun:* Diyelim ki P0 çok yavaş (saatte 1 kez işi var), P1 ise çok hızlı (saatte 1000 kez işi var). P1 işini bitirince P0'ı beklemek zorundadır. P0 gelip sırayı almazsa P1 çalışamaz. Hızlı olan yavaş olana mahkum olur.
2. **Failure (Çökme):**
    - Eğer bir process kritik bölgenin dışında bile olsa çökse, sırayı diğerine devredemez.
    - P0 çöktü diyelim. Sıra P0'daydı. P0 sırayı P1'e devredemediği için P1 sonsuza kadar bekler. Sistem kilitlenir.

## 2. Deneme: "Herkesin Kendi Bayrağı Olsun"

İlk denemede "Tek bir anahtar (Turn)" vardı ve bozulursa sistem kilitleniyordu. Bu sefer diyoruz ki: **"Herkesin kendi bayrağı (flag) olsun. Ben girmek istiyorsam bayrağımı kaldırayım."**

- **Mantık:**
    - Ben (P0) girmek istersem `flag[0] = true` yaparım.
    - Sonra senin bayrağına (`flag[1]`) bakarım. Eğer seninki inikse (`false`), içeri girerim.
- **Neden Patladı? (Mutual Exclusion Yok):**
    - Tam ben bayrağımı kaldıracağım sırada sen de bayrağını kaldırdın.
    - Ben baktım seninki inik (henüz işlemci yazmadı), sen baktın benimki inik.
    - **BUM!** İkimiz de içeri girdik. Veriler birbirine girdi. (Yarış Durumu).

## 3. Deneme: "Önce Bayrak Kaldır, Sonra Kontrol Et"

2. denemedeki sorunu çözmek için sıralamayı değiştiriyoruz.

- **Mantık:** Önce **kesinlikle** bayrağımı kaldırıyorum, sonra senin bayrağına bakıyorum.
- **Neden Patladı? (Deadlock - Ölümcül Kilitlenme):**
    - Ben bayrağımı kaldırdım. (Tam bu sırada işlemci sana geçti).
    - Sen bayrağını kaldırdın.

    - Ben senin bayrağına baktım: "Ooo bayrak havada, bekleyeyim."
    - Sen benim bayrağıma baktın: "Ooo bayrak havada, bekleyeyim."
    - İkimiz de birbirimizi sonsuza kadar bekleriz.

## 4. Deneme: "Kibarlık Yapalım (Livelock)"

3. denemedeki inatlaşmayı çözmek için "Kibar" olmaya karar veriyoruz.

- **Mantık:** Bayrağımı kaldırırım. Baktım seninki de havada; "Ayıp olmasın" diye bayrağımı indirip beklerim, sonra tekrar kaldırırım.
- **Neden Patladı? (Livelock - Canlı Kilitlenme):**
    - Koridorda karşılaşan iki kişi gibi:
    - Ben bayrağı indirdim (Sen geç diye).
    - Sen bayrağı indirdin (Ben geçeyim diye).
    - Ben tekrar kaldırdım. Sen de tekrar kaldırdın.
    - Sürekli "Siz buyrun", "Yok siz buyrun" döngüsüne gireriz. İşlemci çalışır ama iş yapılmaz.

---

## 5. Mutlu Son: Dekker Algoritması (Doğru Çözüm)

Dekker abimiz demiş ki: **"Hem Bayrak (İstek) hem de Turn (Sıra) değişkenini beraber kullanalım."**

**Mantık Şudur:**

1. İkimizin de "İstek Bayrağı" var.
2. Bir de ortada "Hakem Kartı (Turn)" var.
3. Ben girmek istediğimde bayrağımı kaldırırım.
4. Eğer senin bayrağın da havadaysa (Çakışma varsa), **Hakem Kartına (Turn)** bakarız.
5. Kart kimdeyse o devam eder, diğeri bayrağını indirip (kibarlık yapıp) bekler.

Bu algoritma hem **Deadlock**'u hem de **Livelock**'u çözer.

---

## Peterson Algoritması (Daha Şık Çözüm)

Dekker'in çözümü doğruydu ama kodlaması biraz karışıktı. 1981'de Peterson, "Bunu daha kısa yazarım" diyerek efsanevi algoritmasını çıkardı.

**Mantık (İki satırlık nezaket):**

1. Ben girmek istiyorum (`flag[benim] = true`).
2. Ama öncelik senin olsun (`turn = senin`).

3. **Bekleme Şartı:** Eğer sen de girmek istiyorsan VE sıra sendeyse ben beklerim. (`while (flag[senin] && turn == senin) Wait();`)

Bu kod çok daha kısa ve anlaşılırdır.

- Eğer sadece ben istiyorsam, direkt girerim.
- İkimiz de istiyorsak, `turn` değişkenini en son kim değiştirdiyse (örneğin ben "sıra senin" dediysem), o bekler, diğeri girer.

**Özetle:** Peterson algoritması, yazılımsal karşılıklı dışlama (Software Mutual Exclusion) için bilinen en zarif ve klasik yöntemdir.

## 5.2 Eşzamanlılık (Concurrency) İlkeleri

İster tek işlemcili (sırayla çalışan) bir sistem olsun, ister çok işlemcili (aynı anda çalışan) bir sistem olsun; sorun hep aynıdır: **Process'lerin çalışma hızı tahmin edilemez.**

Bu "tahmin edilemezlik" şu 3 büyük derdi başımıza açar:

1. **Paylaşılan Kaynak Tehlikesi:** İki process aynı anda aynı değişkene (variable) yazmaya çalışırsa ne olur? Kimin yazdığı geçerli olur? Sıra bozulursa veri çöp olur.
2. **Kaynak Yönetimi Zorluğu:** Bir process bir kaynağı (örn: yazıcıyı) alıp sonra uykuya dalarsa, diğer process'ler sonsuza kadar bekler mi? (Deadlock riski).
3. **Hata Ayıklama (Debugging) İşkencesi:** Hatalar "bazen olur bazen olmaz". Çünkü her çalıştırdığında process'lerin çalışma sırası ve hızı değişir. Hatayı tekrar oluşturmak (reproducible) çok zordur.

---

## Basit Bir Örnek: `echo()` Fonksiyonu

Diyelim ki ekrana basılan her tuşu yansıtan (echo) basit bir fonksiyonumuz var.

C

```
procedure echo() {

   chin = getchar(); // Klavyeden karakter al

   chout = chin;     // Alınan karakteri çıktı değişkenine koy

   putchar(chout);   // Ekrana bas

}
```

Bu fonksiyonu iki farklı process (P1 ve P2) aynı anda kullanmaya çalışırsa ne olur? Diyelim ki `chin` değişkeni **Global (Paylaşılan)** bir değişken.

**Senaryo 1: Korumasız Erişim (Felaket)**

1. **P1:** Klavyeden "A" tuşuna bastın. `chin = 'A'` oldu.
2. **P1:** Tam ekrana basacakken İŞLETİM SİSTEMİ P1'i DURDURDU (Interrupt).
3. **P2:** Çalışmaya başladı. Klavyeden "B" tuşuna bastın. `chin = 'B'` oldu. (P1'in 'A'sı silindi, üzerine yazıldı).
4. **P2:** Ekrana "B" bastı ve bitti.
5. **P1:** Kaldığı yerden devam etti. Ekrana basacağı değişken (`chin`) artık 'B' olduğu için o da ekrana "B" bastı.

**Sonuç:** Sen "A" ve "B" bastın ama ekranda "BB" gördün. "A" kayboldu. Bu bir veri kaybıdır.

**Senaryo 2: Korumalı Erişim (Mutual Exclusion)**

Bu sorunu çözmek için `echo()` fonksiyonunu **"Tek Kişilik Oda"** haline getiriyoruz.

1. **P1:** `echo()` fonksiyonuna girdi. "A" tuşunu okudu.
2. **P2:** `echo()` fonksiyonuna girmek istedi. AMA kapı kilitli (P1 içeride). P2 kapıda beklemeye (Blocked) başladı.
3. **P1:** İstediği kadar yavaş olsun veya dursun fark etmez. İşini bitirdi, ekrana "A" bastı ve çıktı.
4. **P2:** Kapı açıldı. İçeri girdi, "B" okudu ve ekrana "B" bastı.

**Sonuç:** Ekranda "AB" gördün. Her şey doğru çalıştı.

---

## Özetle Ders

- Sorun tek işlemcili sistemde **"Interrupt" (Kesilme)** yüzünden olur (tam iş yaparken yarıda kesilme).
- Çok işlemcili sistemde ise **"Simultaneous Execution" (Aynı Anda Çalışma)** yüzünden olur.
- Ama ikisinin de çözümü aynıdır: **Shared Resource (Paylaşılan Kaynak)** erişimini kontrol altına almak. Yani **Mutual Exclusion (Karşılıklı Dışlama)** uygulamak.

## 1. Race Condition (Yarış Durumu)

Yarış durumu, birden fazla process veya thread'in aynı veriyi okuyup yazdığı ve **sonucun "kimin önce bitirdiğine" bağlı olduğu** tehlikeli durumdur.

**Kitaptaki Örnekler:**

1. **Son Yazan Kazanır:**
    - Global değişken $a$ var.
    - P1, $a = 1$ yapmak istiyor.
    - P2, $a = 2$ yapmak istiyor.
    - **Sonuç:** Eğer P2 milisaniye farkla P1'den sonra yazarsa, sonuç 2 olur. P1'in yaptığı iş boşa gider.
2. **Hesap Karışıklığı:**
    - Değişkenler: $b$ ve $c$ (Başlangıçta $b=1, c=2$).
    - P3: $b = b + c$ işlemi yapıyor.
    - P4: $c = b + c$ işlemi yapıyor.
    - **Yarış:** Eğer P3 önce çalışırsa sonuç farklı ($b=3, c=5$), P4 önce çalışırsa sonuç bambaşka ($c=3, b=4$) çıkar.

**Özetle:** Yazılımda sonucun "şansa" kalması kabul edilemez.

---

## 2. İşletim Sisteminin Dertleri (OS Concerns)

Eşzamanlılık (Concurrency) olduğunda İşletim Sisteminin (OS) çözmesi gereken 4 temel dert vardır:

1. **Takip (Tracking):** Process Control Block (PCB) kullanarak kimin ne yaptığını bilmek zorunda.
2. **Kaynak Dağıtımı (Allocation):** İşlemci, Bellek, Dosya ve I/O cihazlarını kimseye haksızlık etmeden dağıtmalı.
3. **Koruma (Protection):** Bir process'in hatası diğerini patlatmamalı. (Process Isolation).
4. **Hız Bağımsızlığı (Speed Independence):** En önemlisi bu. **Programın sonucu, işlemcinin hızına veya process'in ne kadar hızlı çalıştığına bağlı olmamalıdır.** Program yavaş da çalışsa hızlı da çalışsa çıktı aynı olmalıdır.

---

## 3. Process Etkileşim Türleri (Process Interaction)

Process'ler birbirlerini ne kadar tanıyor? Bu farkındalık (Awareness) seviyesine göre 3 tür ilişki vardır:

| İlişki Türü | Farkındalık | Durum | Olası Sorunlar |
|---|---|---|---|
| | | | |

| | | | |
|---|---|---|---|
| **Rekabet (Competition)** | **Hiç Yok** | İki yabancı process aynı yazıcıyı kullanmak istiyor. Birbirlerini bilmezler ama kaynak için savaşırlar. | Mutual Exclusion, Deadlock, Starvation |
| **Paylaşım ile İşbirliği (Cooperation by Sharing)** | **Dolaylı** | Birbirlerini tanımazlar ama ortak bir veritabanını veya tamponu (buffer) kullanırlar. | Veri Tutarlılığı (Data Coherence) + Yukarıdakiler |
| **İletişim ile İşbirliği (Cooperation by Communication)** | **Doğrudan** | Birbirlerini PID (Process ID) ile tanırlar ve mesajlaşırlar. | Deadlock, Starvation |

---

## 4. Kaynak Rekabeti ve 3 Büyük Sorun

Process'ler birbirini tanımasa bile (Competition), aynı kaynağa (yazıcı, dosya, bellek) ihtiyaç duyduklarında çatışma çıkar.

Burada iki kavramı ayırmalıyız:

- **Critical Resource (Kritik Kaynak):** Paylaşılamayan kaynak (Örn: Yazıcı, aynı anda iki kağıda basamaz).
- **Critical Section (Kritik Bölge):** Kodun o kaynağı kullandığı kısmı.

Bu rekabet 3 temel sorunu doğurur:

**A. Mutual Exclusion (Karşılıklı Dışlama)**
Sorun: İki process aynı anda yazıcıya veri gönderirse, çıktıdaki satırlar birbirine karışır (Interleaved).

Çözüm: Bir process Kritik Bölgedeyken, diğerleri kapıda beklemelidir.

Mekanizma (Şekil 5.4):

- entercritical(Ra): Kapıyı kilitle.
- [İşini Yap]
- exitcritical(Ra): Kapıyı aç.

**B. Deadlock (Ölümcül Kilitlenme)**

**Senaryo:**

- P1, Kaynak 1'i (R1) almış, R2'yi bekliyor.
- P2, Kaynak 2'yi (R2) almış, R1'i bekliyor.
- İkisi de elindekini bırakmıyor. Sonsuza kadar bakışırlar.

**C. Starvation (Açlık)**
**Senaryo:** P1, P2 ve P3 var. Kaynak sürekli P1 ve P3 arasında gidip geliyor. P2 sırada beklemesine rağmen işletim sistemi (veya algoritma) bir türlü sırayı ona vermiyor. P2 "aç" kalıyor. Deadlock yoktur (işler yürüyor) ama P2 çalışamıyor.

## 1. Paylaşım Yoluyla İşbirliği (Cooperation by Sharing)

Burada process'ler birbirini tanımaz ama aynı "odadaki eşyaları" (değişkenleri, dosyaları, veritabanını) kullanırlar.

**Temel Sorun: Veri Tutarlılığı (Data Coherence)**

Sadece "aynı anda yazmayı engellemek" (Mutual Exclusion) yetmez. İşlemlerin mantık sırasının da korunması gerekir.

Kitaptaki Muhasebe Örneği:

Diyelim ki kuralımız şu: $a = b$ olmalı.

- Başlangıçta: $a=1, b=1$.
- **P1:** İkisini de 1 artırır ($a+1, b+1$).
- **P2:** İkisini de 2 ile çarpar ($2*a, 2*b$).

Eğer bu işlemler karışırsa (Interleaved):

1. P1: a = a + 1 yaptı. (a=2, b=1) -> *Şu an $a=b$ kuralı bozuldu ama geçici.*
2. P2: b = 2 * b yaptı. (a=2, b=2) -> *Düzeldi sanarken...*
3. P1: b = b + 1 yaptı. (a=2, b=3)
4. P2: a = 2 * a yaptı. (a=4, b=3)

Sonuç: $a=4, b=3$. Artık $a=b$ değil. Veri bozuldu (Inconsistent).

Çözüm: Sadece tek bir satırı değil, işlemlerin tamamını bir "Kritik Bölge" içine almalıyız. Yani P1 işi bitmeden P2 asla araya girememeli.

---

## 2. İletişim Yoluyla İşbirliği (Cooperation by Communication)

Burada process'ler birbirini tanır ve doğrudan **Mesajlaşır**. (Örn: Send(P2, message)).

- **Mutual Exclusion Gerekmez:** Çünkü paylaşılan bir kaynak (ortak defter) yoktur, mektuplaşma vardır.
- **Hala Sorun Var mı?** Evet.
    - **Deadlock:** İkisi de birbiriden mesaj beklerse kilitlenirler.[1]
    - **Starvation:** P1 sürekli P2 ile konuşur, garibim P3 sürekli P1'den cevap bekler ama P1 onu takmazsa P3 "aç kalır".

---

## 3. Mutual Exclusion için 6 Altın Kural (Requirements)

Bir sistemde "Karşılıklı Dışlama" (Mutual Exclusion) kuracaksak, şu 6 şartı M U T L A K A sağlamalıdır. (Sınavlarda madde madde sorulur):

1. **Dışlama Zorunlu (Enforced):** Kritik bölgeye aynı anda SADECE TEK BİR process girebilir. İstisna yok.
2. **Dışarıdakiler Karışamaz:** Kritik bölgeyle işi olmayan (non-critical section'da takılan) bir process, diğerlerini asla engellememeli.
3. **Sonsuz Bekleme Yok:** İçeri girmek isteyen biri sonsuza kadar bekletilmemeli (Deadlock veya Starvation olmamalı).
4. **Hemen Giriş:** Eğer içerisi boşsa, kapıya gelen ilk kişi bekletilmeden içeri alınmalıdır.
5. **Hız Bağımsızlığı:** Sistem, process'lerin hızına veya işlemci sayısına göre kafasına göre davranmamalıdır. (Yavaş da olsa hızlı da olsa kod doğru çalışmalı).
6. **Sınırlı Süre:** İçeri giren process, sonsuza kadar içeride kalamaz. İşini bitirip makul bir sürede çıkmalıdır.

---

## Çözüm Yolları

Bu kuralları sağlamak için 3 farklı yaklaşım vardır:

1. **Yazılımsal Çözümler (Software Approaches):** Programcının kendi yazdığı kodlarla (Dekker algoritması vb.). *Zordur ve hata riski yüksektir.*
2. **Donanımsal Çözümler (Machine Instructions):** İşlemcinin özel komutları ile (TestAndSet). *Hızlıdır ama yetersiz olabilir.*
3. **İşletim Sistemi / Dil Desteği:** En popüler olan budur. **Semaforlar (Semaphores)**, **Monitörler** ve **Mesajlaşma**.

## 1. Kesilmeleri Devre Dışı Bırakmak (Interrupt Disabling)

Tek işlemcili (Uniprocessor) sistemlerde en basit ve kaba çözüm budur.

- **Mantık:** Bir process çalışırken araya başkasının girmesinin tek yolu "Interrupt" (Kesilme) gelmesidir (Örneğin süre bitti sinyali).

- **Yöntem:** Process, kritik bölgeye girerken **"Beni kimse bölmesin (Disable Interrupts)"** der. İşini bitirince **"Tamam bölebilirsiniz (Enable Interrupts)"** der.

C

```
// Kaba Kuvvet Yöntemi

while (true) {

   disable_interrupts(); // Kulakları tıka

   /* Kritik Bölge */

   enable_interrupts();  // Kulakları aç

   /* Kritik olmayan işler */

}
```

- **Sorunlar:**
    1. **Çok İşlemcide Çalışmaz:** Sen kendi işlemcinin kulağını tıkasan bile, yanındaki diğer işlemci çalışmaya ve belleğe erişmeye devam eder.
    2. **Tehlikelidir:** Kullanıcı programına bu yetkiyi verirsen, program sonsuz döngüye girip sistemi tamamen kilitleyebilir. İşletim sistemi kontrolü kaybeder.

---

## 2. Özel Makine Komutları (Special Machine Instructions)

Çok işlemcili sistemlerde, işlemciler belleği paylaşır. Donanım üreticileri (Intel, AMD vb.), **"Atomik" (Bölünemez)** işlem yapan özel komutlar tasarlamıştır. Bu komutlar çalışırken bellek yolunu kilitler, başka hiçbir işlemci o adrese erişemez.

### A. Compare & Swap (CAS) Komutu

Bu komut şunu tek bir adımda yapar: "Bellekteki değere bak, eğer beklediğim değerse yenisiyle değiştir."

**Mantık:**

- bolt (sürgü) adında global bir değişkenimiz var. 0 ise açık, 1 ise kilitli.
- Process içeri girmek için CompareAndSwap komutunu bir döngü içinde (Busy Wait) kullanır.
- Komut der ki: "Git bolta bak. Eğer 0 ise (TestVal), onu 1 yap (NewVal). Bana da eski değerini söyle."
- Eğer eski değer 0 ise, process "Yaşasın kilit açıkmış, ben kilitledim ve girdim" der.

- Eğer eski değer 1 ise, process "Tüh içeride biri var" der ve döngüye devam eder.

**B. Exchange (XCHG) Komutu**

Bu komut, bir işlemci yazmacı (register) ile bellekteki bir değişkenin yerini **tek hamlede** değiştirir.

**Mantık:**

- Her process'in cebinde key = 1 anahtarı vardır.
- Ortada tek bir kilit (bolt) vardır. Başlangıçta 0'dır (Açık).
- Process içeri girmek istediğinde exchange yapar. Cebindeki 1 ile kilitteki değeri takas eder.
    - Eğer kilit 0 ise: Takas sonucu cebime 0 gelir (Giriş İzni), kilide 1 gider (Kilitlendi).
    - Eğer kilit zaten 1 ise: Takas sonucu cebime 1 gelir (Hala kilitli), kilide yine 1 gider (Değişen bir şey olmaz).

---

## Donanımsal Yaklaşımın Artıları ve Eksileri

Bu özel komutların avantajları ve dezavantajları mülakatlarda sıkça sorulur:

| **Avantajlar (Pros)** | **Dezavantajlar (Cons)** |
|---|---|
| **Basitlik:** Doğrulaması kolaydır. | **Busy Waiting (Meşgul Bekleme):** Process beklerken sürekli döngüde işlemciyi yorar. Enerji ve zaman kaybıdır. |
| **Evrensellik:** İşlemci sayısından bağımsız çalışır. | **Starvation (Açlık):** Sıra garantisi yoktur. Şanslı olan kapar, şanssız olan sonsuza kadar bekleyebilir. |
| **Çoklu Kilit:** Birden fazla kritik bölgeyi yönetebilir. | **Deadlock (Kilitlenme):** Öncelik terslemesi (Priority Inversion) durumunda kilitlenmeye yol açabilir. |

**Deadlock Örneği:** Düşük öncelikli P1 kilidi alır. Tam o sırada yüksek öncelikli P2 gelir ve işlemciyi P1'den zorla alır (preempt). P2 kilidi almaya çalışır ama kilit P1'dedir. P2 sürekli "Ver kilidi" diye döner durur (busy wait). P1 ise işlemciyi P2 aldığı için çalışıp kilidi bırakamaz. Sistem kilitlenir.

## 1. Genel (Sayılan) Semafor (Counting Semaphore)

Düşün ki bir restorandasın ve tuvalet kapısı kilitli. Tuvalete girmek için kasadaki anahtarı alman lazım.

- **Initialize (Başlangıç):** Kasada **3 tane** anahtar asılı. (Semafor Değeri = 3). Bu, aynı anda 3 kişinin tuvaleti kullanabileceği (veya 3 boş kabin olduğu) anlamına gelir.
- **semWait (Bekle / Azalt):**
    - Tuvalete girmek isteyen müşteri kasaya gelir.
    - Anahtar varsa birini alır (Değeri 1 azaltır: 3 -> 2) ve girer.
    - Anahtar yoksa (Değer 0 veya negatifse), sıraya girip **bekler (Block)**.
- **semSignal (Sinyal Ver / Artır):**
    - İşini bitiren müşteri anahtarı geri getirir (Değeri 1 artırır).
    - Eğer sırada bekleyen (uyuyan) biri varsa, anahtarı direkt ona verir (Onu uyandırır/Unblock eder).
    - Kimse yoksa anahtarı yerine asar.

**Teknik Karşılığı (Şekil 5.6):**

- Semafor bir **Tamsayı (Integer)** değişkendir.
- **Değer Pozitifse (+):** O kadar sayıda kaynak (anahtar) boşta demektir.
- **Değer Sıfırsa (0):** Kaynaklar tükendi, kimse beklemiyor.
- **Değer Negatifse (-):** Kaynaklar tükendi VE sıra var demektir. (Örn: -2 ise, sırada 2 kişi uyuyor).

---

## 2. İkili Semafor (Binary Semaphore)

Bu da restoranın sadece **tek kişilik** tuvaleti olması durumudur.

- Semafor sadece **0 (Dolu)** veya **1 (Boş)** değerini alır.
- **semWaitB:** Değer 1 ise 0 yap ve gir. Değer 0 ise bekle.
- **semSignalB:** İşin bitince değeri 1 yap (veya bekleyen varsa ona devret).

Bu yapı **Mutex** (Mutual Exclusion) kavramına çok benzer ama küçük bir fark vardır: Mutex'te kilidi kilitleyen kişi açmak zorundadır. Semafor'da ise kilidi A process'i kilitleyip, B process'i açabilir (Sinyalleşme/Signaling amaçlı).

---

## Tablo 5.3: Ortak Eşzamanlılık Mekanizmaları Özeti

Kitabın başında verdiği tablo, alet çantamızdaki diğer malzemeleri özetliyor:

| **Mekanizma** | **Açıklama** |
| --- | --- |
| | |

| | |
|---|---|
| **Semaphore** | Sayısal sinyalleşme aracı. (Restoran Anahtarı). En temel yapı taşıdır. |
| **Binary Semaphore** | Sadece 0 ve 1 alabilen özel semafor. |
| **Mutex** | İkili semafora benzer ama "Sahip olma" kuralı vardır. Kilitleyen açmalıdır. |
| **Condition Variable** | Bir process'i belirli bir şart (condition) gerçekleşene kadar bloklar. |
| **Monitor** | Programlama dillerinin (Java, C#) sağladığı, değişkenleri ve kodları tek bir çatı altında koruyan yüksek seviyeli yapı. (Semaforun kullanımı zor olduğu için icat edilmiştir). |
| **Event Flags** | Hafızadaki bir "bit" kümesidir. Process, "3. bit 1 olunca beni uyandır" diyebilir. |
| **Mailboxes/Messages** | İki process'in veri (mesaj) takas etmesini sağlar. |
| **Spinlocks** | Semafor gibi uyutmaz, "açıldı mı?" diye sürekli sorarak işlemciyi meşgul eder (Busy Wait). Kısa süreli beklemeler için kullanılır. |

---

## Semaforların 3 Kritik Sonucu

Dijkstra'nın tanımına göre şunları asla bilemeyiz (ve bilmemize gerek yoktur):

1. semWait yaptığında bloke mi olacaksın yoksa direkt girecek misin? (İçerideki sayıyı bilmediğin için bilemezsin, sadece denersin).
2. semSignal yapıp birini uyandırdığında, işlemci hanginizi çalıştıracak? (Seni mi yoksa yeni uyananı mı? İşletim sisteminin keyfine kalmış).
3. semSignal yaptığında sırada birinin olup olmadığını kesin olarak bilemezsin.

Harika bir noktadayız. Burası genelde sınavların "şaşırtmalı soru" (trick question) kısmıdır. Herkes Semaforu bilir ama "Mutex ile farkı nedir?" denilince takılır.

Bu metin 3 kritik kavramı netleştiriyor: **Mutex vs Binary Semaphore**, **Güçlü vs Zayıf Semafor** ve **Semafor ile Korumak**.

Senin için bu kavramları günlük hayattan örneklerle ve teknik terimleri koruyarak özetledim.

---

## 1. Mutex ve Binary Semaphore Arasındaki Fark

İkisi de sadece 0 ve 1 değerini alır. İkisi de kilit gibi davranır. Ama aralarında hayati bir fark vardır: **Sahiplik (Ownership).**

- **Mutex (Mutual Exclusion Lock):**
  - **Kural:** Kilidi kim kilitlediyse, **SADECE O** açabilir.
  - **Metafor:** Spor salonundaki kişisel dolabın. Asma kilidi sen taktıysan, anahtar sendedir, başkası gelip açamaz. Sen işini bitirince kilidi sen açarsın.
- **Binary Semaphore:**
  - **Kural:** Sahiplik yoktur. A process'i kilitleyip (Wait), B process'i kilidi açabilir (Signal).
  - **Metafor:** Bayrak yarışı. Bayrağı (kilidi) sen alırsın, koşarsın, sonra bayrağı arkadaşına verirsin (Signal). Başlatan ve bitiren farklı olabilir. Genelde senkronizasyon (haberleşme) için kullanılır.

**Not:** Linux ve Windows gibi işletim sistemlerindeki Mutex yapısı, kitaptaki bu "Sahiplik" tanımına sıkı sıkıya uyar.

---

## 2. Güçlü (Strong) ve Zayıf (Weak) Semafor

Semaforun değeri negatife düştüğünde process'ler kuyruğa (Queue) girip uyur demiştik. Peki, kilit açılınca (Signal gelince) kuyruktan **kim** uyanacak?

- **Güçlü (Strong) Semafor:**
  - **Politika: FIFO (First-In, First-Out / İlk Giren İlk Çıkar).**
  - **Mantık:** Banka kuyruğu gibidir. En uzun süredir bekleyen kişi uyanır.
  - **Sonuç:** En adil yöntemdir. **Starvation (Açlık)** olmaz, herkesin sırası elbet gelir. İşletim sistemleri genelde bunu kullanır.
- **Zayıf (Weak) Semafor:**
  - **Politika:** Sıra garantisi yoktur. Kuyruktan rastgele biri seçilebilir.
  - **Sonuç:** Adaletsizdir. Şanssız bir process sonsuza kadar kuyrukta kalabilir (Starvation).

---

## 3. Semafor ile Mutual Exclusion (Karşılıklı Dışlama) Nasıl Sağlanır?

Bir kaynağı (örneğin Yazıcıyı) korumak için semaforu nasıl kullanırız?

Adım 1: Başlangıç

Semaforu (s) 1 olarak başlatırız. (1 boş koltuk var).

Adım 2: Algoritma (Şekil 5.9)

Tüm process'ler şu kodu çalıştırır:

C

```
semWait(s);        // 1. Adım: Kapıyı çal. (Değer 1 ise 0 yap gir. 0 ise bekle)

/* KRİTİK BÖLGE */ // 2. Adım: İşini yap (Yazıcıyı kullan).

semSignal(s);      // 3. Adım: Çıkarken haber ver. (Değeri 1 artır, bekleyen varsa uyandır).
```

Adım 3: Değerlerin Anlamı

Bu algoritma çalışırken s değerine bakarak sistemin durumunu anlayabilirsin:

- **$s \ge 0$ (Pozitif veya Sıfır):** İçeriye beklemeden girebilecek process sayısı. (Eğer 1 ise içerisi boş, 0 ise içerisi dolu ama kapıda bekleyen yok).
- **$s < 0$ (Negatif):** İçerisi dolu VE kapıda kuyruk var.
    - *Örnek:* s = -3 ise, içeride 1 kişi var, kapıda **3 kişi** (Mutlak değer) uyuyarak sıra bekliyor demektir.

İşte işletim sistemlerinin "Efsanevi Bölüm Sonu Canavarı": **Üretici/Tüketici Problemi (Producer/Consumer Problem)**.

Bu problem, eşzamanlılık (concurrency) konularının anlaşılıp anlaşılmadığını test eden en iyi örnektir. Senin için bu problemi, yanlış çözümlerden başlayıp doğru çözüme doğru giden bir hikaye gibi anlatacağım.

---

## Problem Nedir?

Bir masamız var (Buffer).

- **Üretici (Producer):** Masaya sürekli tabak (veri) koyar.
- **Tüketici (Consumer):** Masadan sürekli tabak alır.

**Kurallar:**

1. **Mutual Exclusion:** Masaya aynı anda sadece bir kişi dokunabilir. (Biri koyarken diğeri alamaz).
2. **Boş Masa Kuralı:** Masa boşsa tüketici beklemelidir (Boş tabak yiyemez).
3. **Dolu Masa Kuralı:** Masa sonsuz değilse, doluyken üretici beklemelidir (Bu senaryoda masayı sonsuz/infinite varsayıyoruz).

---

## 1. Yanlış Çözüm (Binary Semaphore ile Deneme)

İlk denemede (Şekil 5.12) masadaki tabak sayısını bir `n` değişkeni ile takip etmeye çalışıyoruz. İki tane semaforumuz var:

- `s`: Masayı kilitlemek için (Mutex gibi).
- `delay`: Tüketiciyi bekletmek için (Masa boşsa).

**Kodun Mantığı:**

- **Üretici:** Masayı kilitler (`semWait(s)`), tabağı koyar, `n`'i artırır, masayı açar (`semSignal(s)`). Eğer masa boş idiyse (`n==1`), tüketiciye "Uyan, yemek geldi" (`semSignal(delay)`) der.
- **Tüketici:** Önce "Yemek var mı?" diye bekler (`semWait(delay)`). Sonra masayı kilitler, tabağı alır, `n`'i azaltır, masayı açar.

**Neden Patladı? (Race Condition)** Tablo 5.4'te gösterildiği gibi, kritik bir senaryo var:

1. Tüketici son tabağı aldı, `n` değişkenini 0 yaptı.
2. Tüketici tam "Şimdi kendimi uyutayım (`semWait(delay)`)" diyecekken...
3. **Üretici Araya Girdi!** Yeni bir tabak koydu, `n` değişkenini 1 yaptı.
4. Tüketici kaldığı yerden devam etti: "Aa `n` sıfır sanıyordum ama 1 olmuş, demek ki uyumama gerek yok" demesi gerekirken, koddaki `if` kontrolünü çoktan geçtiği için yanlışlıkla uyku moduna (`semWait(delay)`) girer.
5. **Sonuç:** Yemek olduğu halde tüketici uyur. Üretici de "Ben haber vermiştim" der. Sistem karışır.

---

## 2. Doğru Çözüm (Binary Semaphore + Yardımcı Değişken)

Şekil 5.13'teki çözümde, `n` değerini korumak için kritik bölge içinde `m` adında yerel bir yardımcı değişken (auxiliary variable) kullanırız. Bu sayede `n` değerini okuma ve karar verme işini sağlama alırız. Bu çözüm çalışır ama karmaşıktır.

---

## 3. En Temiz Çözüm (Counting Semaphore - Genel Semafor)

Eğer elimizde **Sayılan (Counting) Semafor** varsa hayat çok kolaylaşır.

**Mantık (Şekil 5.14):**

- `n` değişkenini normal bir tamsayı (integer) değil, bir **Semafor** yaparız.
- `s`: Masayı koruyan kapıcı (Mutex). Değeri 1.
- `n`: Masadaki tabak sayısı. Değeri 0.

**Kod:**

C

```
/* ÜRETİCİ */

produce();          // Yemek yap

semWait(s);         // 1. Kapıyı kilitle

append();           // Tabağı koy

semSignal(s);       // 2. Kapıyı aç

semSignal(n);       // 3. Tabak sayısını (n) artır. (Varsa bekleyen tüketici uyanır)


/* TÜKETİCİ */

semWait(n);         // 1. Tabak var mı? Varsa azalt ve geç. Yoksa uyu.

semWait(s);         // 2. Kapıyı kilitle

take();             // Tabağı al

semSignal(s);       // 3. Kapıyı aç

consume();          // Ye
```

**Neden Mükemmel?**

- Tüketici `semWait(n)` dediğinde; eğer `n` (tabak sayısı) 0 ise semafor mantığı gereği otomatikman uyur.
- Üretici `semSignal(n)` dediğinde; `n` artar ve uyuyan tüketici otomatikman uyanır.
- Ekstra `if` kontrollerine, `delay` semaforlarına gerek kalmaz. Semaforun kendisi tabak sayısını tutar.

**Sınav Sorusu İpucu:** Sınavda hoca "Kodda `semWait(s)` ve `semWait(n)` yer değiştirirse ne olur?" diye sorabilir. (Tüketici tarafında).

- **Cevap: Deadlock Olur!**
- Tüketici önce kapıyı kilitler (`semWait(s)`). Sonra tabağa bakar (`semWait(n)`).
- Eğer tabak yoksa, kapıyı kilitlemiş halde uykuya dalar.
- Üretici kapı kilitli olduğu için içeri girip tabak koyamaz.
- Tüketici tabak gelmediği için uyanamaz.
- Sonsuza kadar kilitlenirler.

Harika, şimdi "Masamız Sonsuz Değilse Ne Olur?" (Gerçek Hayat) senaryosuna geçiyoruz.

Ayrıca önceki bölümde bahsettiğimiz "Sıralama Hatası"nın neden ölümcül olduğunu ve bu Semafor denen aletin makine içinde gerçekte nasıl çalıştığını (Kaputun altını) inceleyeceğiz.

Metni senin için teknik terimleri koruyarak ve ders notu formatında özetledim.

---

## 1. Ölümcül Hata: Sıralamayı Karıştırmak

Önceki çözümde (Sonsuz Tampon) şöyle bir kod vardı: `semWait(n)` (Tabak var mı?) -> `semWait(s)` (Kapıyı kilitle).

**Soru:** Tüketici bu ikisinin yerini yanlışlıkla değiştirirse ne olur? **Kod:** `semWait(s)` -> `semWait(n)`.

**Senaryo (Masa Boşken):**

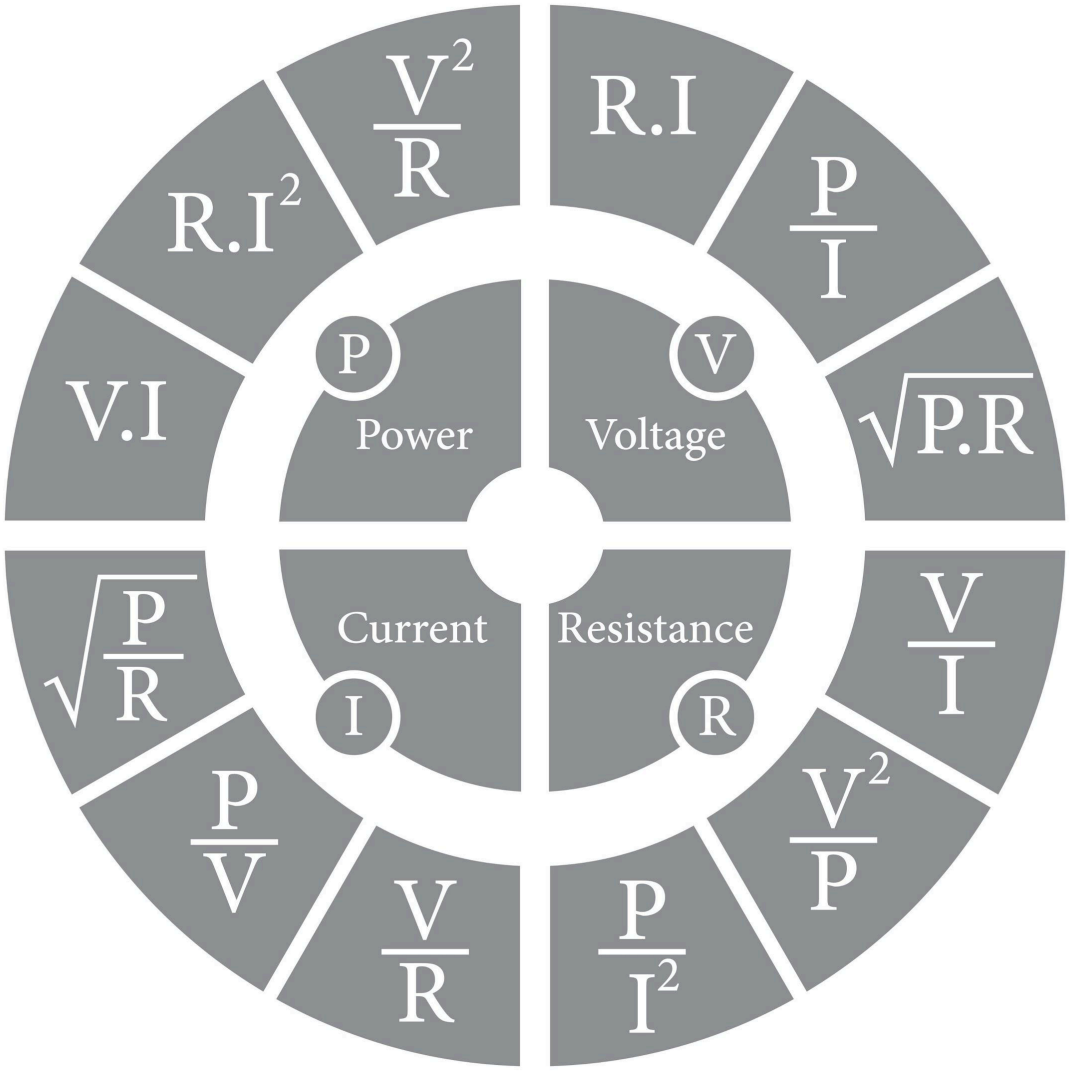
1. **Tüketici:** Önce kapıyı kilitler (`semWait(s)`). Anahtar cebindedir.
2. **Tüketici:** Sonra masaya bakar (`semWait(n)`).
3. **Hata:** Masa boştur (`n=0`). Tüketici semafor kuralı gereği **uykuya dalar.**
4. **Sorun:** Tüketici uyudu AMA **anahtar hala cebinde!** (Kapı kilitli).
5. **Üretici:** Yemek getirdi. Kapıyı açmaya çalışır (`semWait(s)`). Ama kapı kilitli. O da kapıda uyur.
6. **Sonuç: Deadlock.** Kimse kımıldayamaz.

**Ders:** Semafor kullanırken sıralama hayati önem taşır. Önce kaynağı (tabak) kontrol et, sonra kilidi (mutex) al.

---

## 2. Sınırlı Tampon (Bounded-Buffer) Problemi

Gerçek dünyada RAM veya Disk sonsuz değildir. Masamızın sadece belirli sayıda (örneğin 10) tabak kapasitesi vardır.




Shutterstock

**Yeni Kural (Circular Buffer):** Masa yuvarlak bir bant gibidir. Sonu gelince başa döner.

- **Üretici:** Masada "Boş Yer" yoksa beklemek zorundadır.
- **Tüketici:** Masada "Dolu Tabak" yoksa beklemek zorundadır.

**Çözüm (Şekil 5.16):** İki tane sayan semafora ihtiyacımız var:

1. **n:** Dolu tabak sayısı (Başlangıçta 0).
2. **e:** Boş yer sayısı (Başlangıçta Kapasite kadar, örn: 10).
3. **s:** Mutex (Kapı kilidi).

**Algoritma:**

C

```
/* ÜRETİCİ */

produce();

semWait(e);      // 1. Boş yer var mı? (Varsa e'yi azalt, yoksa uyu)

semWait(s);      // 2. Kapıyı kilitle

append();        // Tabağı koy

semSignal(s);    // 3. Kapıyı aç

semSignal(n);    // 4. Dolu tabak sayısını (n) artır. (Tüketici uyanır)


/* TÜKETİCİ */

semWait(n);      // 1. Dolu tabak var mı? (Varsa n'yi azalt, yoksa uyu)

semWait(s);      // 2. Kapıyı kilitle

take();          // Tabağı al

semSignal(s);    // 3. Kapıyı aç

semSignal(e);    // 4. Boş yer sayısını (e) artır. (Üretici uyanır)
```

**Mantık:**

- Üretici yemek koyunca `n` artar (Tüketiciye haber verir).
- Tüketici yemeği yiyince `e` artar (Üreticiye "Yer açıldı, koyabilirsin" diye haber verir).
- Böylece masa ne taşar ne de boşken işlem yapılır. Mükemmel denge!

---

## 3. Semaforun Uygulanması (Kaputun Altı)

Tamam, semaforlar kodumuzu koruyor ama **Semaforun kendisini kim koruyor?** Sonuçta o da hafızada duran bir değişken (`int count`). Ona da aynı anda iki kişi yazarsa bozulur.

Semaforun `semWait` ve `semSignal` operasyonları **Atomik** (Bölünemez) olmak zorundadır. Bunu sağlamak için 3 yol vardır:

1. **Donanım / Firmware (En İyisi):** Bu işi doğrudan işlemcinin içinde gömülü devreler halleder.
2. **Özel Komutlar (Compare&Swap - CAS):**
    - Semaforun kendi içinde minik bir kilidi (`flag`) vardır.
    - İşlemci, semaforun değerini (sayısını) değiştirmeden önce bu minik kilidi **Compare&Swap** komutuyla (önceki bölümde gördüğümüz donanımsal yöntemle) kilitler.
    - Burada çok kısa süreli bir **Busy Waiting** (dönüp durma) olur ama semafor işlemi (sayıyı 1 artırıp azaltmak) mikrosaniyeler sürdüğü için bu bekleme önemsizdir.
3. **Interruptları Kapatmak:**
    - Sadece tek işlemcili sistemlerde çalışır. Semaforu güncellerken "Beni kimse bölmesin" diyerek interruptları kapatır.

## 5.5 Monitörler (Monitors)

Monitör, bir programlama dili yapısıdır (construct). Yani bir kütüphane değil, dilin içine gömülmüş özel bir **Sınıf (Class)** gibidir. C#, Java gibi dillerdeki "Thread-Safe" sınıfların atasıdır.

**Özellikleri:**

1. **Özel Veriler:** Monitörün içindeki değişkenlere (local data) dışarıdan kimse dokunamaz.
2. **Özel Kapı:** Monitöre girmek için onun bir fonksiyonunu (procedure) çağırmak zorundasın.
3. **Tekillik Kuralı (Mutual Exclusion):** Monitörün içinde aynı anda **SADECE BİR** process çalışabilir. Eğer içerisi doluysa, girmek isteyenler kapıda (Entry Queue) bekler.

**Fark:** Semafor kullanırken "kapıyı kilitlemeyi" (wait/signal) programcı kendisi yapmak zorundaydı. Monitörde ise "kapı kilidi" otomatiktir. Fonksiyona giren otomatik kilitler, çıkan otomatik açar.

---

## Monitörde Senkronizasyon (Condition Variables)

Monitör, "Aynı anda tek kişi girsin" (Mutual Exclusion) işini otomatik halleder ama "Masa boşsa bekle" (Synchronization) işi için ekstra araca ihtiyaç vardır.

Bunun için **Condition Variables (Koşul Değişkenleri)** kullanılır. İki temel komut vardır:

1. `cwait(c)`:
    - "Ben bu koşul sağlanana kadar bekleyeceğim" der.
    - Process monitörden **dışarı atılmaz**, sadece kenara çekilip uyutulur (Condition Queue).
    - **Önemli:** Uyuyan process, monitörün kilidini bırakır ki başkası içeri girebilsin.
2. `csignal(c)`:
    - "Koşul sağlandı, uyanın!" der.
    - Eğer o koşulu bekleyen biri varsa onu uyandırır.
    - **Semafor Farkı:** Eğer bekleyen kimse yoksa, `csignal` boşa gider (kaybolur). Semaforun sinyali ise sayı olarak saklanırdı.

---

## Monitör ile Üretici/Tüketici Çözümü

Şekil 5.19'da bu problemin Monitör ile nasıl çözüldüğü gösteriliyor.

1. **Monitör:** `BoundedBuffer` adında bir monitörümüz var.
2. **Değişkenler:** `buffer` (dizi), `count` (eleman sayısı), `nextin`, `nextout`.
3. **Koşullar:**
    - `notfull`: Masa tam dolu değil (Üretici için).
    - `notempty`: Masa boş değil (Tüketici için).

**Kodun Mantığı:**

C

```
monitor BoundedBuffer {

   ...

   procedure append(x) {

      if (count == N) cwait(notfull); // Doluysa "notfull" olana kadar bekle

      buffer[nextin] = x;            // Tabağı koy

      count++;
```

```
        csignal(notempty);              // "Hey masa boş değil artık" diye haber ver
    }

    procedure take(x) {
        if (count == 0) cwait(notempty); // Boşsa "notempty" olana kadar bekle
        x = buffer[nextout];            // Tabağı al
        count--;
        csignal(notfull);               // "Hey masada yer açıldı" diye haber ver
    }
}
```

**Semafor ile Farkı Nedir?**

- **Semafor:** Hem kapıyı kilitlemeyi (`semWait(s)`) hem de masayı kontrol etmeyi (`semWait(n)`) programcı elle yapıyordu. Hata riski yüksekti.
- **Monitör:** Kapı kilitleme işini Monitör kendi yapıyor. Programcı sadece "Masa dolu mu?" kontrolünü yazıyor. Hata riski çok daha az.

---

## Hoare'ın Sinyal Mantığı (Urgent Queue)

Bir process `csignal` verip "Uyan!" dediğinde ne olur?

1. Sinyali veren (`Signaler`) hala monitörün içindedir.
2. Uyanan (`Waiter`) da monitöre girmek ister.
3. Kural gereği içeride sadece 1 kişi olabilir.

**Hoare'ın Çözümü:** Sinyali veren process, kibarlık yapıp geçici olarak durur ve **"Acil Durum Kuyruğu"na (Urgent Queue)** geçer. Uyanan process (Waiter) hemen çalışmaya başlar. Waiter işini bitirip çıkınca, Urgent Queue'daki (Sinyal veren) kaldığı yerden devam eder.

> **Concurrent Pascal Dili:** Bu karmaşayı önlemek için "Sinyal verme işi (`csignal`) mutlaka fonksiyonun **en son satırı** olmalıdır" kuralını koymuştur. Böylece sinyal veren zaten çıkacağı için sorun kalmaz.

## 1. Alternatif Monitör Modeli (Mesa / Lampson & Redell)

İlk çıkan (Hoare) monitör modelinde katı bir kural vardı:

- **Hoare Kuralı:** "Ben Signal verip seni uyandırdığım an, ben derhal dururum ve sıra sana geçer."
- **Sorun:** Bu sürekli "bağlam değiştirme" (context switch) gerektirir. İşlemciyi yorar. Ayrıca o an sıra sana geçse bile, araya başkası girip ortamı tekrar bozabilir.

Mesa dilini (ve bugün kullandığın C#, Java'yı) tasarlayanlar dedi ki: **"Sinyal veren (Signaler) işini bitirene kadar çalışmaya devam etsin."**

**Notify (Bildir) ve Broadcast (Yayın Yap)**

Bu yeni modelde Signal yerine Notify kullanılır.

1. **Notify:** "Hey, uyanabilirsin, şartlar oluştu. Ama ben hemen çıkmıyorum, işimi bitirince sen bakarsın."
2. **Kritik Kod Değişikliği (If yerine While):**
    - Eski modelde: if (dolu) wait(); (Uyanınca kesin boştur diye düşünürdüm).
    - Yeni modelde: while (dolu) wait(); (Uyanınca **tekrar kontrol etmem** gerekir. Çünkü ben uyanana kadar başkası gelip doldurmuş olabilir).
3. **Broadcast (Yayın):**
    - Tek bir kişiyi değil, kuyrukta bekleyen **herkesi** uyandırır.
    - **Kullanım Alanı:** Örneğin bellek yöneticisi büyük bir bellek alanını serbest bıraktı. Kimin ne kadar istediğini bilmiyor. "Herkes uyansın, işine yarayan alsın" der.

**Özetle:** Modern dillerde (Java/C#) Monitor.Pulse (Notify) ve Monitor.PulseAll (Broadcast) kullanılır ve her zaman while döngüsü içinde bekleme yapılır.

---

## 2. Mesajlaşma (Message Passing)

Process'ler birbirini görmüyorsa (farklı bilgisayarlardaysa) "Ortak Bellek" kullanamazlar. Mecburen mektuplaşacaklar.

İki temel komut vardır:

1. **send(hedef, mesaj):** Mesajı yolla.
2. **receive(kaynak, mesaj):** Mesajı al.

**Senkronizasyon Türleri (Blocking vs Nonblocking)**

Bir mesaj attığında veya beklediğinde ne yapacaksın? Hayatı durduracak mısın yoksa devam mı edeceksin? 3 ana kombinasyon vardır:

1. **Blocking Send, Blocking Receive (Rendezvous - Randevu):**
    - **Gönderen:** Karşı taraf alana kadar bekler.

    - **Alan:** Mesaj gelene kadar bekler.
    - **Örnek:** Telefon konuşması. İki taraf da hatta olmak zorundadır.
2. **Nonblocking Send, Blocking Receive (En Popüleri):**
    - **Gönderen:** Mesajı postalar ve işine devam eder. (Cevap beklemez).
    - **Alan:** Mesaj gelmeden iş yapamaz, o yüzden bekler.
    - **Örnek:** E-posta (Mail) sunucusu. Sen maili atıp işine bakarsın (Nonblocking), sunucu mail gelene kadar bekler (Blocking). Client-Server mimarisi genelde budur.
3. **Nonblocking Send, Nonblocking Receive:**
    - **Gönderen:** Atar ve gider.
    - **Alan:** "Mesaj var mı?" diye bakar, yoksa işine devam eder.
    - **Risk:** Kontrolsüzdür. Sistem mesaj çöplüğüne dönebilir.

---

## Tablo 5.5: Mesajlaşma Tasarım Kararları

Bir mesajlaşma sistemi tasarlarken şu kararları vermelisin:

| **Özellik** | **Seçenekler** |
|---|---|
| **Senkronizasyon** | Gönderen veya Alan bloklansın mı? (Yukarıdaki 3 model). |
| **Adresleme** | **Direct:** Doğrudan "P1 işlemine yolla".<br><br>**Indirect:** "Posta kutusuna (Mailbox/Port) yolla", kim alırsa alsın. |
| **Kuyruk Disiplini** | Mesajlar sırayla mı (FIFO) yoksa aciliyetle mi (Priority) işlenecek? |
| **Mesaj Formatı** | Sabit uzunlukta mı (Kolay ama kısıtlı), değişken uzunlukta mı (Esnek ama zor)? |

## 1. Adresleme (Addressing)

"Mesajı kime atacağız?" sorusunun iki cevabı vardır:

**A. Doğrudan Adresleme (Direct Addressing)**

- **Mantık:** Gönderen, alıcının adını açıkça belirtir. `Send(P2, mesaj)`.
- **Alıcı Tarafı:**
    - **Açık (Explicit):** Alıcı da kimden beklediğini söyler. `Receive(P1, mesaj)`. (P1 dışında kimseden almam).
    - **Örtülü (Implicit):** Alıcı "Kimden gelirse gelsin" der. `Receive(id, mesaj)`. (Örn: Yazıcı sunucusu, herkesin isteğini kabul eder).

**B. Dolaylı Adresleme (Indirect Addressing) - Posta Kutuları**

- **Mantık:** Mesajlar kişiye değil, bir **Posta Kutusuna (Mailbox / Port)** atılır.
- **Avantaj:** Gönderen ve Alan birbirini tanımak zorunda değildir (Decoupling).
- **İlişki Türleri (Şekil 5.21):**
    - **1:1 (Özel Hat):** Sadece iki process konuşur. Güvenlidir.
    - **M:1 (İstemci/Sunucu):** Birçok kişi bir kutuya atar, bir kişi (Sunucu) okur. (Web sunucusu mantığı).
    - **1:M (Yayın/Broadcast):** Bir kişi atar, birçok kişi okur. (Televizyon yayını).
    - **M:N (Çoklu Sunucu):** Birçok kişi atar, birçok sunucu (işçi) okur.

---

## 2. Mesaj Formatı (Message Format)

Bir mesaj iki kısımdan oluşur (Şekil 5.22):

1. **Header (Başlık):** Zarfın üzerindeki bilgiler. (Kimden, Kime, Uzunluk, Tip, Öncelik, Sıra No).
2. **Body (Gövde):** Mektubun kendisi (Veri).
    - *Sabit Uzunluk:* İşlemesi hızlıdır ama esnek değildir.
    - *Değişken Uzunluk:* Esnektir ama yönetmesi zordur.

---

## 3. Mesajlaşma ile Mutual Exclusion (Jeton Mantığı)

Semafor veya Mutex kullanmadan, sadece mesajlaşarak "Kritik Bölge"yi nasıl koruruz?

**Çözüm: Jeton (Token) Yöntemi** Ortada **`box`** adında bir posta kutusu var.

1. Başlangıçta kutuya **Tek Bir Adet** boş mesaj ("Jeton") atılır.
2. **Giriş:** Kritik bölgeye girmek isteyen process, `Receive(box, msg)` yapar.
    - Kutu boşsa (Jeton başkasındaysa) **Bekler (Block)**.

    - Kutuda mesaj varsa alır ve içeri girer.
3. **Çıkış:** İşini bitiren process, mesajı kutuya geri atar `Send(box, msg)`.

**Sonuç:** Jeton tek olduğu için, aynı anda sadece bir kişi kritik bölgede olabilir.

---

## 4. Mesajlaşma ile Üretici/Tüketici Problemi

Burada çok zekice bir yöntem kullanılır. Mesajlar hem **Sinyal** (Semafor gibi) hem de **Veri Taşıyıcı** (Buffer gibi) olarak kullanılır.

İki tane posta kutumuz var:

1. **`mayconsume` (Tüketilebilir):** İçindeki mesajlar **Dolu Tabakları** temsil eder.
2. **`mayproduce` (Üretilebilir):** İçindeki mesajlar **Boş Tabakları** temsil eder.

**Algoritma (Şekil 5.24):**

- **Başlangıç:** `mayproduce` kutusuna kapasite kadar (örn: 100) boş mesaj atılır. `mayconsume` boştur.
- **Üretici:**
    1. `Receive(mayproduce, msg)`: Boş tabak kutusundan bir mesaj al (Boş yer var mı?). Yoksa bekle.
    2. `msg = data`: Mesajın içine veriyi koy.
    3. `Send(mayconsume, msg)`: Mesajı dolu tabak kutusuna at.
- **Tüketici:**
    1. `Receive(mayconsume, msg)`: Dolu tabak kutusundan mesaj al (Yemek var mı?). Yoksa bekle.
    2. Veriyi ye.
    3. `Send(mayproduce, msg)`: Boşalan mesajı (tabağı) tekrar boş kutuya at.

Bu yöntemle **Shared Memory (Paylaşılan Bellek)** olmadan, tamamen izole process'ler arasında mükemmel bir senkronizasyon sağlanır.

İşte işletim sistemlerinin "Üretici/Tüketici"den sonraki en meşhur ikinci problemi: **Okurlar/Yazarlar Problemi (Readers/Writers Problem)**.

Burası veritabanı yönetim sistemlerinin de temelini oluşturur. Senin için konuyu "Kütüphane Kataloğu" örneğiyle basitleştirip özetledim.

---

## 1. Problem Nedir?

Ortada paylaşılan bir veri alanı (Dosya, Veritabanı veya Kütüphane Kataloğu) var. İki tür işlemci (process) var:

1. **Okurlar (Readers):** Sadece veriyi okur, değiştirmez.
2. **Yazarlar (Writers):** Veriyi değiştirir veya siler.

**Kurallar:**

1. **Çoklu Okuma Serbest:** Aynı anda birden fazla Okur veriyi okuyabilir. (Birbirlerini engellemezler).
2. **Yazar Tek Olmalı:** Bir Yazar işlem yaparken, başka hiçbir Yazar işlem yapamaz.
3. **Okurken Yazmak Yasak:** Bir Yazar işlem yaparken, hiçbir Okur okuyamaz. (Çünkü veri o an değişiyor, tutarsız olabilir).

---

## Üretici/Tüketici ile Farkı Nedir?

Bu ikisi sıkça karıştırılır ama çok temel bir farkları vardır:

- **Üretici/Tüketici:** Tüketici (Consumer) veriyi **yer ve yok eder** (Masadan tabağı alır). Yani veriyi değiştirir.
- **Okurlar/Yazarlar:** Okur (Reader) veriyi **sadece okur**, veriye zarar vermez. Bu yüzden birden fazla okur aynı anda çalışabilir. Tüketiciler ise aynı anda çalışamaz (aynı tabağı iki kişi yiyemez).

---

## 2. Çözüm 1: Okurlar Öncelikli (Readers Have Priority)

Bu yaklaşımda, "Okur sayısı 0 olmadığı sürece Yazarlar beklesin" denir.

- **Senaryo:** Kütüphanede bir öğrenci (Okur) kataloğu inceliyor. O sırada kütüphaneci (Yazar) güncelleme yapmak istiyor.
- **Kural:** Kütüphaneci bekler. O beklerken içeri yeni bir öğrenci girerse, o da okumaya başlar. Kütüphaneci ancak **içeride hiç öğrenci kalmadığında** iş yapabilir.
- **Risk: Yazar Kıtlığı (Writer Starvation):** Eğer öğrenciler sürekli gelmeye devam ederse (içerisi hiç boşalmazsa), kütüphaneci sonsuza kadar bekleyebilir.

**Algoritma Mantığı (Şekil 5.25):**

- `readcount`: İçerideki okur sayısı.
- `x`: `readcount` değişkenini koruyan semafor.
- `wsem`: Yazarı engelleyen semafor.
- **İlk giren okur** kapıyı kilitler (`semWait(wsem)`).

- **Son çıkan okur** kapıyı açar (`semSignal(wsem)`).
- Aradaki okurlar kilitle uğraşmaz, rahatça girer.

---

## 3. Çözüm 2: Yazarlar Öncelikli (Writers Have Priority)

Yazarların sonsuza kadar beklemesini (Starvation) engellemek için önceliği onlara veririz.

- **Senaryo:** Kütüphaneci "Güncelleme yapacağım!" diye bağırır.
- **Kural:**
    1. O an içeride olan öğrenciler işini bitirip çıkar.
    2. Kapıdaki yeni öğrenciler **içeri alınmaz**, dışarıda bekletilir.
    3. İçerisi boşalınca Kütüphaneci girer, işini yapar.
- **Risk: Okur Kıtlığı (Reader Starvation):** Yazarlar sürekli gelirse, okurlar hiç içeri giremeyebilir.

**Algoritma Mantığı (Şekil 5.26):** Burası biraz daha karışıktır, 5 tane semafor kullanılır (`rsem`, `wsem`, `x`, `y`, `z`).

- `rsem`: Yazarların sıraya girmesi ve yeni okurları engellemesi için.
- Bir yazar geldiğinde, okurların giriş kapısını kapatır. İçeridekilerin bitmesini bekler.

---

## 4. Mesajlaşma ile Çözüm (Controller Process)

Semaforlarla uğraşmak yerine (ki hata riski yüksektir), bir **"Kontrolcü Process" (Controller)** görevlendirilebilir.

- **Mantık:**
    - Okur: "Okuyabilir miyim?" mesajı atar.
    - Yazar: "Yazabilir miyim?" mesajı atar.
    - Kontrolcü: Kendi içindeki sayaca (`count`) bakar.
        - `count > 0`: İçeride okurlar var.
        - `count = 0`: Boş.
        - `count < 0`: Yazar var.
    - Kontrolcü, Yazarlara öncelik verecek şekilde "OK" mesajı yollar.

Bu yöntem daha güvenlidir çünkü paylaşılan değişkenlere (`count` vb.) sadece tek bir process (Kontrolcü) dokunur, yarış durumu (Race Condition) oluşmaz.

## 5.8 ÖZET (Bölüm Sonu)

Modern işletim sistemlerinin üç ana teması vardır:

1. **Multiprogramming (Çoklu Programlama):** Tek işlemcide birden fazla iş.
2. **Multiprocessing (Çoklu İşlemci):** Birden fazla işlemci ile gerçek paralel iş.
3. **Distributed Processing (Dağıtık İşlem):** Farklı bilgisayarlarda çalışan işler.

Bu üçünün de ortak noktası ve temel teknolojisi **Concurrency (Eşzamanlılık)** tır.

**Process'lerin Birbiriyle İlişkisi**
Process'ler üç şekilde etkileşime girer:

1. **Habersiz (Unaware) - Rekabet:** Birbirlerini tanımazlar ama işlemci süresi veya I/O cihazı için kavga ederler.
2. **Dolaylı Farkında (Indirectly Aware) - Paylaşım:** Ortak bir dosyayı veya belleği (Memory Block) kullanırlar.
3. **Doğrudan Farkında (Directly Aware) - İletişim:** Birbirlerine mesaj atarak işbirliği yaparlar.

Bu etkileşimlerde başımıza gelen iki büyük bela vardır: **Mutual Exclusion (Karşılıklı Dışlama)** ve **Deadlock (Kilitlenme)**.

**Mutual Exclusion (Karşılıklı Dışlama) Çözümleri**
Bir kaynağa aynı anda sadece bir kişinin erişmesini sağlamak için kullanılan yöntemler:

1. **Donanımsal Çözümler (Machine Instructions):**
    - İşlemcinin özel komutları (Test&Set vb.) kullanılır.
    - **Sorun:** İşlemciyi meşgul ederek bekletir (**Busy Waiting**). Verimsizdir.
2. **İşletim Sistemi Çözümleri:**
    - **Semaforlar (Semaphores):** Trafik ışığı gibi sinyalleşme sağlar. En yaygın yöntemdir.
    - **Mesajlaşma (Messages):** Hem veriyi iletir hem de sıraya sokar (Synchronization). Özellikle dağıtık sistemler için idealdir.
    - **Monitörler (Monitors):** Programlama dillerinin (Java, C#) sunduğu, hataya daha kapalı, yüksek seviyeli yapılardır.

---

## Bölüm 5: Hap Bilgiler Tablosu (Sınav Öncesi Göz At)

| **Kavram** | **Anlamı** | **Metafor** |
|---|---|---|
| **Race Condition** | Sonucun "kimin önce bitirdiğine" bağlı olması. | Aynı koltuğa oturmaya çalışan iki kişi. |

| | | |
|---|---|---|
| **Critical Section** | Paylaşılan kaynağa erişilen kod parçası. | Tek kişilik tuvalet kabini. |
| **Mutual Exclusion** | "Ben içerideyken kimse giremez" kuralı. | Tuvalet kapısının kilidi. |
| **Deadlock** | "Sen bırakmazsan ben de bırakmam" inatlaşması. | Köprüde karşılaşan iki inatçı keçi. |
| **Starvation** | Kaynak var ama sana sıra gelmiyor. | Sürekli başkalarına çay dağıtan garsonun seni görmezden gelmesi. |
| **Semaphore** | Sıra yönetim aracı. | Restoran tuvaleti anahtarı. |
| **Monitor** | Otomatik kilitlenen güvenli oda. | Kartla girilen banka kasası. |

## 6.1 Deadlock İlkeleri (Principles of Deadlock)

**Deadlock Nedir?** Bir grup process'in, asla gerçekleşmeyecek bir olayı beklemesi yüzünden sonsuza kadar bloklanmasıdır.

**En Klasik Örnek: Trafik Kilitlenmesi** Dört yol ağzında 4 araba karşılaşıyor.

- Kural: "Sağındaki arabaya yol ver."
- Durum: 4 araba aynı anda gelirse, herkes sağındakini bekler. Kimse kımıldayamaz.
- Sonuç: **Deadlock.** Kaynaklar (yolun kısımları) orada duruyor ama kimse kullanamıyor.

---

## Ortak İlerleme Diyagramı (Joint Progress Diagram)

Bu grafik biraz soyut ama sınavda çıkabilir. İki process'in (P ve Q) kaynakları (A ve B) nasıl istediğini gösterir.

- **X Ekseni:** P'nin ilerlemesi.

- **Y Ekseni:** Q'nun ilerlemesi.
- **Yasak Bölgeler:** İkisinin aynı anda aynı kaynağa sahip olduğu yerler (Mutual Exclusion yüzünden imkansızdır).
- **Fatal Region (Ölümcül Bölge):** Grafikte gri ile taranmış alandır. Eğer process'lerin çalışması bu alana girerse, oradan çıkış yoktur. Sonuç kaçınılmaz olarak Deadlock'tur.

  **Mantık:** P, A'yı almış B'yi bekliyor. Q, B'yi almış A'yı bekliyor. Bu noktaya (Fatal Region) girersen geri dönüşü yoktur.

---

## Kaynak Türleri (Resource Types)

Deadlock sadece donanımda olmaz, iki tür kaynakta da olabilir:

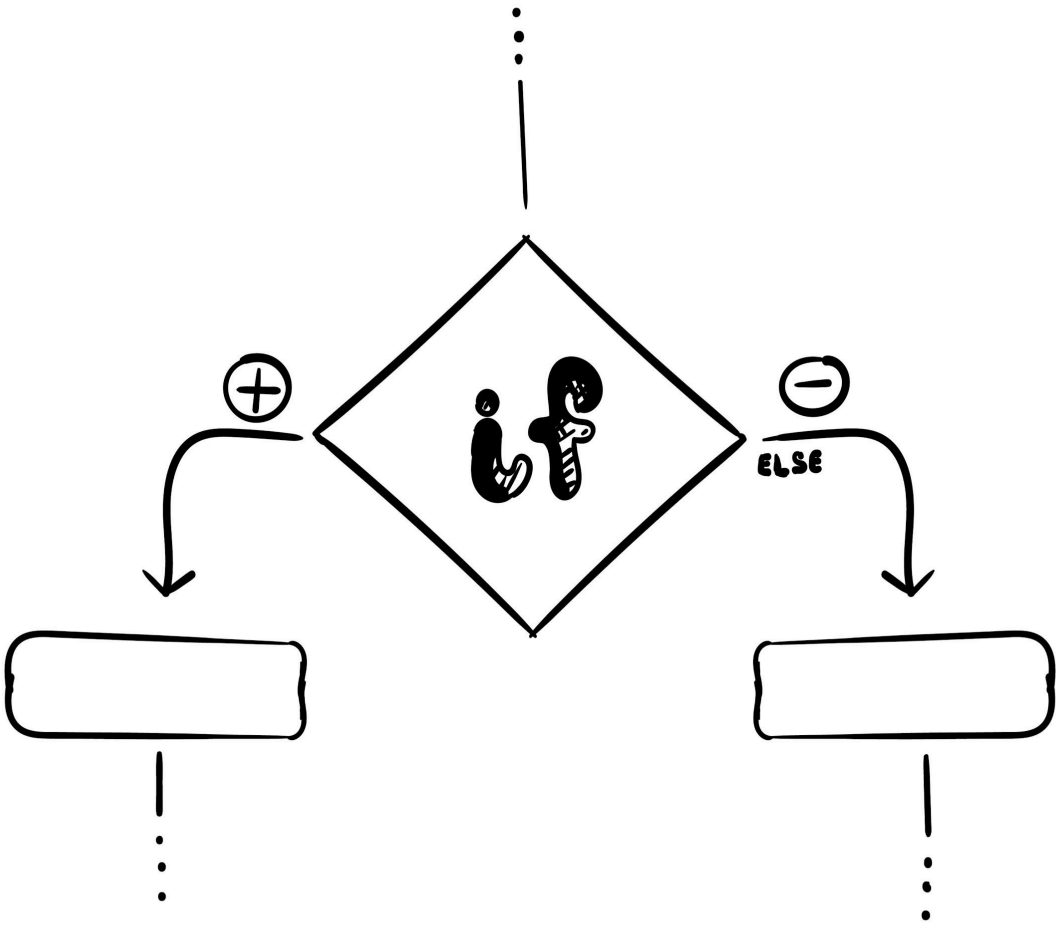
1. **Reusable (Yeniden Kullanılabilir) Kaynaklar:**
   - İşlemci, Bellek, Dosya, Veritabanı kilidi.
   - Process kullanır ve geri bırakır.
   - **Deadlock Örneği:** P1 disk sürücüsünü alır, P2 yazıcıyı alır. Sonra ikisi de diğerinin elindekini ister.
2. **Consumable (Tüketilebilir) Kaynaklar:**
   - Sinyaller, Mesajlar, Kesilmeler (Interrupts).
   - Process üretir ve diğeri tüketir (yok eder).
   - **Deadlock Örneği:** P1, P2'den mesaj bekler (Receive). P2 de P1'den mesaj bekler. İkisi de sonsuza kadar susar.

---

## Kaynak Tahsis Grafikleri (Resource Allocation Graphs - RAG)

Deadlock olup olmadığını anlamak için kullanılan çizim yöntemidir.

- **Yuvarlaklar:** Process'ler.
- **Kareler:** Kaynaklar (İçindeki noktalar, kaç tane kaynak olduğunu gösterir).
- **Oklar:**
  - Process -> Kaynak: "İstiyor" (Request).
  - Kaynak -> Process: "Sahip" (Held by).




Shutterstock

**Kural:** Eğer grafikte bir **Döngü (Cycle)** varsa ve kaynakların sadece birer kopyası varsa, kesinlikle **Deadlock** vardır.

---

## Deadlock İçin 4 Şart (The Conditions for Deadlock)

Burası bölümün en kritik yeridir. Bir sistemde Deadlock olması için şu 4 şartın **AYNI ANDA** gerçekleşmesi gerekir. (Biri bile eksik olsa Deadlock olmaz).

1. **Mutual Exclusion (Karşılıklı Dışlama):** Kaynağı aynı anda sadece 1 kişi kullanabilir. (Paylaşılamaz kaynak).
2. **Hold and Wait (Tut ve Bekle):** Bir process, elinde en az bir kaynak tutarken, gidip başkasının elindeki kaynağı da istiyordur. "Elimdekini bırakmam, yenisini de isterim" mantığı.
3. **No Preemption (Zorla Alma Yok):** Kaynak, sahibi işini bitirmeden zorla elinden alınamaz.
4. **Circular Wait (Döngüsel Bekleme):** P1 P2'yi bekler, P2 P3'ü bekler... Pn de döner P1'i bekler. Bir zincir oluşur.

   **Önemli Not:** İlk 3 madde (Mutual Exclusion, Hold&Wait, No Preemption) sistemin **politikasıdır** (kurallarıdır). 4. madde (Circular Wait) ise bu kuralların sonucunda oluşan **hatadır**.

---

## Deadlock İle Mücadele Yöntemleri

İşletim sistemi tasarımcıları bu beladan kurtulmak için 3 strateji kullanır:

1. **Deadlock Prevention (Önleme):** Yukarıdaki 4 şarttan en az birini *baştan* imkansız hale getirmek. (Örn: "Tut ve Bekle"yi yasaklamak).
2. **Deadlock Avoidance (Kaçınma):** Kaynak isterken "Dur bakalım, bunu sana verirsem ileride sıkışır mıyız?" diye analiz etmek (Banker Algoritması).
3. **Deadlock Detection (Tespit Etme):** Sistemi özgür bırakmak, deadlock oluşursa tespit edip birilerini "öldürerek" (kill) sistemi açmak.

Şimdi Deadlock (Kilitlenme) ile savaşma sanatına giriyoruz. İki ana stratejimiz var: **Önleme (Prevention)** ve **Kaçınma (Avoidance)**. İkisi aynı gibi duyulsa da yöntemleri çok farklıdır.

## 6.2 Deadlock Prevention (Önleme)

Bu strateji, Deadlock'un oluşması için gereken **4 şarttan (Mutual Exclusion, Hold&Wait, No Preemption, Circular Wait) en az birini** baştan imkansız hale getirmeyi hedefler.

1. **Mutual Exclusion (Karşılıklı Dışlama):**
   - **Hedef:** "Aynı anda birden fazla kişi kullansın."
   - **Sorun:** Bunu kaldıramayız. Yazıcıyı aynı anda iki kişi kullanamaz. Bu şart donanımsal bir zorunluluktur.
2. **Hold and Wait (Tut ve Bekle):**
   - **Yöntem:** Process çalışmaya başlamadan önce **ihtiyacı olan her şeyi tek seferde istesin.** Ya hepsini alır başlar ya da hiçbirini alamaz bekler.
   - **Sorun:** Çok verimsizdir. Belki yazıcıya 5 saat sonra ihtiyacım olacak ama şimdiden rezerve edip boş yere tutuyorum.
3. **No Preemption (Zorla Alma Yok):**

    - **Yöntem:** Eğer yeni bir kaynak istedin ve vermedilerse, elindeki her şeyi bırakmak zorundasın.
    - **Sorun:** Yazıcıdan çıktı alırken "Pardon yarıda kesiyoruz" diyemezsin. Sadece İşlemci ve Bellek gibi durumu kaydedilebilen (save/restore) kaynaklarda işe yarar.
4. **Circular Wait (Döngüsel Bekleme) - EN PRATİK ÇÖZÜM:**
    - **Yöntem:** Kaynaklara numara ver (1: Yazıcı, 2: Disk, 3: Tarayıcı...).
    - **Kural:** Kaynakları sadece **artan sıra ile** isteyebilirsin.
    - *Örnek:* Elinde 3 (Tarayıcı) varken 1 (Yazıcı) isteyemezsin. Önce 1'i, sonra 3'ü istemeliydin. Bu kural döngü oluşmasını matematiksel olarak imkansız kılar.

---

## 6.3 Deadlock Avoidance (Kaçınma)

Bu strateji, 4 şartı da kabul eder ama **çok dikkatli davranarak** deadlock noktasına (Fatal Region) girmemeye çalışır.

- **Mantık:** Bir process kaynak istediğinde işletim sistemi durur ve **simülasyon** yapar: *"Sana bunu verirsem sistem güvenli (safe) kalır mı?"*
- **Gereksinim:** Her process, işe başlarken **maksimum ne kadar kaynağa ihtiyacı olacağını** baştan söylemek zorundadır.

**Banker Algoritması (Banker's Algorithm)**

Edsger Dijkstra tarafından bankacılık sisteminden esinlenerek tasarlanmıştır.

- **Bankacı (OS):** Kasada nakit parası (Kaynaklar) var.
- **Müşteriler (Processler):** Her birinin kredi limiti (Maksimum İhtiyaç) var ama parça parça çekiyorlar.

**Kural:** Bankacı, müşteriye kredi verirken şunu kontrol eder: *"Sana bu parayı verirsem, elimde kalan para, en az bir müşterimin tüm borcunu kapatıp işini bitirmesine yetiyor mu?"*

**Güvenli Durum (Safe State) vs Güvensiz Durum (Unsafe State)**

- **Güvenli Durum (Safe State):** Öyle bir sıralama var ki (P2 -> P1 -> P3 gibi), herkes sırayla ihtiyaçlarını karşılayıp işini bitirebilir ve kaynaklarını iade edebilir.
- **Güvensiz Durum (Unsafe State):** Elimizdeki kaynak o kadar azaldı ki, kimse maksimum ihtiyacını karşılayamıyor. Her an Deadlock olabilir (ama kesin olacak diye bir şey yok).

**Örnek (Şekil 6.7):**

- Toplam 9 birim kaynağımız olsun.
- **P1:** Şu an 1 tane almış, bitirmek için 6 tane daha lazım. (Toplam 7)
- **P2:** Şu an 5 tane almış, bitirmek için 2 tane daha lazım. (Toplam 7)

- **Kasadaki:** 3 tane kaldı.

**Senaryo:** P1 gelip "Bana 1 tane daha ver" derse:

- **Simülasyon:** Verirsem kasada 2 kalır.
- **Kontrol:** Bu 2 birimle P1 işini bitirebilir mi? Hayır (5 lazım). P2 bitirebilir mi? Evet (2 lazım).
- **Karar:** P2 bitince kaynakları (5+2=7) geri dönecek. O zaman P1'i de doyurabilirim. O halde bu istek **Güvenlidir**, ver gitsin.

Eğer kasada 2 tane varken P1 gelip 1 tane isteseydi:

- **Simülasyon:** Verirsem kasada 1 kalır.
- **Kontrol:** 1 birimle ne P1 ne P2 işini bitirebilir. İkisi de ortada kalır.
- **Karar: Güvensiz Durum!** İsteği reddet, P1 beklesin.

## 6.4 Deadlock Detection (Kilitlenme Tespiti)

Önleme (Prevention) ve Kaçınma (Avoidance) stratejileri sistemi çok kısıtlıyordu. Tespit etme stratejisi ise şunu der: **"Sistemi özgür bırak, kaynakları istedikçe ver. Sık sık kontrol et, eğer Deadlock varsa o zaman müdahale ederiz."**

**Tespit Algoritması Nasıl Çalışır?**

Mantık "Banker Algoritması"na benzer ama geçmişe değil geleceğe bakar. İşletim sistemi periyodik olarak şu simülasyonu yapar:

1. **İşaretle:** Kaynak tutmayan (masum) process'leri işaretle (Onlar sorun çıkarmaz).
2. **Simülasyon:**
    - Elimizdeki boş kaynaklarla ihtiyacını karşılayıp bitirebilecek bir process var mı?
    - Varsa: "Bu işini bitirir ve elindekileri iade eder" diye varsay. Onun elindeki kaynakları sanal kasaya ekle. Onu işaretle.
    - Yoksa: Dur.
3. **Sonuç:** Algoritma bittiğinde **işaretlenmemiş** process'ler varsa, işte onlar **Deadlock**tadır.

**Kurtarma (Recovery) Yöntemleri**

Deadlock tespit ettik, şimdi ne yapacağız? 4 acı reçete var:

1. **Hepsini Öldür (Abort All):** En yaygın yöntemdir. İşletim sistemi "Burada işler karıştı" der ve kilitlenen tüm process'leri sonlandırır.
2. **Geri Al ve Yeniden Başlat (Rollback & Restart):** Process'leri belirli bir kayıt noktasına (checkpoint) geri döndür. (Bunun için sürekli yedek almak gerekir, maliyetlidir).
3. **Tek Tek Öldür:** Deadlock çözülene kadar process'leri sırayla öldür. Kimi öldüreceğiz? (En az süre harcayanı, en düşük öncelikliy vb.).

4. **Kaynakları Zorla Al (Preempt Resources):** Bir process'in elinden kaynağı zorla alıp diğerine ver. (Kaynağı alınan process bozulabilir).

---

## 6.5 Entegre Strateji (Hepsini Kullanmak)

Tek bir yöntem her yer için uygun değildir. İşletim sistemleri kaynakları sınıflara ayırır ve her sınıf için farklı strateji uygular:

- **Takas Alanı (Swappable Space):** Deadlock Prevention (Hepsini baştan iste).
- **İşlemci Kaynakları (Dosya, Teyp):** Deadlock Avoidance (Banker algoritması).
- **Ana Bellek (RAM):** Preemption (Gerekirse process'i diske at/swap et).

---

## 6.6 Yemek Yiyen Filozoflar Problemi (Dining Philosophers)

Bu problem, eşzamanlılık (concurrency) dünyasının en ünlü senaryosudur (Dijkstra, 1971).

**Senaryo:**

- 5 Filozof yuvarlak bir masada oturuyor.
- Önlerinde makarna var.
- Her filozofun arasında **1 adet çatal** var (Toplam 5 çatal).
- **Kural:** Makarna yemek için **2 çatal** (hem sağdaki hem soldaki) gerekir.

**Problem 1: Deadlock** Tüm filozoflar aynı anda acıkırsa ve hepsi **önce solundaki** çatalı alırsa ne olur?

- Herkesin elinde 1 çatal var.
- Herkes sağındaki çatalı bekliyor.
- Ama sağındaki çatalı, sağındaki komşusu tutuyor.
- **Sonuç:** Kimse yiyemez, hepsi sonsuza kadar bekler.

**Problem 2: Starvation (Açlık)** Deadlock olmasa bile, sağındaki ve solundaki komşuların çok hızlı yiyip çatalı bırakıp tekrar alıyorsa, ortadaki filozof hiç sıra bulamayabilir.

**Çözümler:**

1. **Garson (Attendant) Ekleme (Şekil 6.13):**
    - Masaya en fazla 4 filozofun oturmasına izin verilir.
    - 5 çatal varken 4 kişi otururşa, en az bir kişi 2 çatal bulup yiyebilir. Yiyen bırakınca diğerleri de yer.
2. **Monitör Kullanımı (Şekil 6.14):**
    - Filozof çatalları tek tek değil, ikisini birden "Atomik" (bölünemez) bir işlemle ister.

- Monitör fonksiyonu (`take_forks`) şu kontrolü yapar: "Sağmdaki veya solumdaki komşum yiyor mu? Yemiyorsa iki çatalı da bana ver."

---

## 6.7 UNIX Eşzamanlılık Mekanizmaları

UNIX (ve Linux, MacOS), process'lerin konuşması için çok güçlü araçlar sunar.

1. **Pipes (Borular):**
   - İki process arasında tek yönlü veri akışı sağlar. (Producer-Consumer mantığı).
   - **Örnek:** `ls | grep "txt"` komutunda `ls` çıktısını borudan `grep`e yollar.
   - Bir taraf yazarken diğer taraf okumaya çalışırsa (veya tam tersi) işletim sistemi otomatik bloklar.
2. **Messages (Mesajlar):**
   - Process'ler birbirine etiketli (tipli) veri paketleri yollar. Bir posta kutusu (kuyruk) mantığı vardır.
3. **Shared Memory (Paylaşılan Bellek):**
   - **En Hızlı Yöntemdir.**
   - İşletim sistemi RAM'in bir bölgesini iki process'in de adres uzayına haritalar.
   - **Dikkat:** İşletim sistemi burada senkronizasyon (kilit) sağlamaz! Process'ler kendi aralarında Semafor kullanarak kavga etmeden belleği kullanmalıdır.
4. **Semaphores (Semaforlar):**
   - UNIX semaforları biraz karmaşıktır. Tek bir sayı değil, **Semafor Kümeleri (Sets)** olarak yaratılırlar.
   - İşlemler atomiktir (ya hepsi yapılır ya hiçbiri).
5. **Signals (Sinyaller):**
   - Process'lere olayları bildirmek için kullanılır (Örn: `SIGKILL`, `SIGINT`). Yazılımsal "Interrupt" gibidir.

## 1. Sinyaller (Signals)

Sinyal, bir sürece (process) "Hey, bir şey oldu!" demenin yazılımsal yoludur. Donanımsal kesilmelere (Interrupts) benzerler ama öncelik (priority) kavramları yoktur. Yani "Yangın var" sinyali ile "Telefon çaldı" sinyali aynı muameleyi görür, sıraya konmaz.

**Özellikleri:**

- **Asenkron:** Ne zaman geleceği belli değildir.
- **Kuyruk Yok:** Standart UNIX sinyalleri tek bir bit ile tutulur. Yani aynı anda 5 tane "DUR" sinyali gelse, process sadece 1 tane görür. Diğerleri kaybolur.
- **Tepki:** Bir process sinyal gelince şunlardan birini yapar:
  1. Varsayılanı yap (Genelde kendini kapatmak/terminate).
  2. Sinyal yakalayıcı (Handler) fonksiyonu çalıştır.
  3. Sinyali görmezden gel (Ignore).

**Tablo 6.1: Bilmen Gereken En Önemli UNIX Sinyalleri**
Tabloda çok var ama bir mühendis olarak şunları adın gibi bilmelisin:

| Değer | İsim | Açıklama |
|---|---|---|
| **02** | SIGINT | **Interrupt.** Klavyeden Ctrl+C yaptığında giden sinyaldir. "Kibarca dur" demektir. |
| **09** | SIGKILL | **Kill.** Süreci anında öldürür. Yakalanamaz veya görmezden gelinemez. "Kafasına sıkmak" gibidir. |
| **11** | SIGSEGV | **Segmentation Fault.** Process, kendisine ait olmayan bir belleğe erişmeye çalıştı. (C kodlarken en çok alacağın hata). |
| **14** | SIGALRM | **Alarm.** Belirli bir süre sonra uyanmak isteyen process'e saat gelince yollanır. |
| **15** | SIGTERM | **Termination.** Yazılımsal sonlandırma isteği. SIGKILL'den daha kibardır, process'e "eşyalarını topla ve çık" der. |

---

## 2. Linux Çekirdek (Kernel) Eşzamanlılık Mekanizmaları

Linux, standart UNIX araçlarını (Pipe, Message, Shared Memory) kullanır ama modern ihtiyaçlar için kendine has oyuncaklar da geliştirmiştir.

### A. Gerçek Zamanlı (RT) Sinyaller

Standart sinyallerin (yukarıdakiler) yetersiz kaldığı yerlerde devreye girer.

- **Fark 1 (Kuyruk):** Kuyruklanabilirler. 5 tane gelirse 5'i de işlenir.
- **Fark 2 (Öncelik):** Öncelik sırasına göre işlenir.
- **Fark 3 (Veri):** Standart sinyal sadece "Dürtme"dir. RT sinyali yanında bir tamsayı (integer) veya veri adresi (pointer) de taşıyabilir.

### B. Atomik İşlemler (Atomic Operations)

Linux çekirdeği, basit yarış durumlarını (Race Conditions) önlemek için **"Bölünemez" (Atomik)** işlemler sunar. Bir işlem atomikse, işlemci onu yaparken araya kimse giremez. Ya hep yapılır ya hiç yapılmaz.

İki türü vardır:

1. **Atomik Tamsayı (Integer):** Sadece atomic_t veri tipiyle çalışır.
    - *Neden özel tip?* Derleyici (Compiler) kafasına göre optimize etmesin ve programcı yanlışlıkla normal tamsayı sanıp bozmasın diye.
    - *Örnek:* Sayaç artırma (atomic_inc), çıkarma (atomic_sub).
2. **Atomik Bitmap:** Bellekteki belirli bir bit'i (0 veya 1) değiştirmek için kullanılır.

**C. Spinlocks (Dönen Kilitler)**
Linux çekirdeğinde bir veriyi korumanın en yaygın yolu budur.

- **Mantık:** Bir thread kilitli bir kapıya (Spinlock) gelirse, kapı açılana kadar **olduğu yerde döner durur (Spinning / Busy Wait).**
- **"Kapı açık mı? Hayır. Açık mı? Hayır. Açık mı? Hayır..."**
- **Ne Zaman Kullanılır?** Bekleme süresinin çok çok kısa olacağını bildiğin zaman (2 context switch süresinden az). Çünkü process'i uyutup uyandırmak (Context Switch), olduğu yerde bekletmekten daha maliyetli olabilir.

Spinlock Çeşitleri (Tablo 6.3):

Çekirdek kodlarken hangisini kullanacağını bilmek hayati önem taşır:

1. **spin_lock (Sade):** En temel kilit. Eğer interrupt (kesilme) gelmeyeceğinden eminsen bunu kullan.
2. **spin_lock_irq:** Kilidi alır ve o işlemci üzerindeki **Interruptları kapatır**. (Interrupt handler ile çakışmamak için).
3. **spin_lock_irqsave (En Güvenlisi):** Kilidi alır, interruptları kapatır VE eski interrupt durumunu bir bayrağa (flag) kaydeder. Kilidi açarken eski haline getirir. Mevcut durumu bilmiyorsan bunu kullan.
4. **spin_lock_bh (Bottom Half):** Interruptların "Bottom Half" denen (işin hamallık kısmı) yazılımsal parçalarını devre dışı bırakır.

    **Tek İşlemci vs. Çok İşlemci Farkı:**

    - **Tek İşlemcili (Uniprocessor) sistemde:** Spinlock aslında dönmez (spin atmaz). Çünkü tek işlemci var, sen dönüp durursan kilidi tutan diğer adam ne zaman çalışıp kilidi açacak? O yüzden tek işlemcide spinlock sadece "interruptları kapatma" koduna dönüşür veya tamamen silinir.
    - **Çok İşlemcili (Multiprocessor) sistemde:** Gerçekten dönerek bekler. Çünkü yan odadaki diğer işlemci o sırada kilidi açacak işi yapıyor olabilir.

## 1. Reader-Writer Spinlock (Okur-Yazar Dönen Kilidi)

Temel spinlock "İçeri sadece 1 kişi girebilir" diyordu. Ama ya içerideki veri sadece okunuyorsa? 100 kişi aynı anda okusa veri bozulmaz ki? Boşuna kapıda beklemesinler.

- **Mantık:**
  - **Okurlar:** Kapı açıksa girin. 100 kişi de olsanız girin. (Aynı anda çoklu erişim).
  - **Yazar:** Ben değişiklik yapacağım zaman içeride **KİMSE** olmamalı (ne başka yazar ne de okur).
- **Mekanizma:** 24-bitlik bir sayaç (içeride kaç okur var) ve bir bayrak (yazar var mı) kullanılır.
- **Önemli Detay:** Linux'ta bu kilit **Okurlara Torpil Geçer.** İçeride okur olduğu sürece yeni okurlar gelip girmeye devam edebilir. Kapıda bekleyen gariban yazar, içerisi tamamen boşalana kadar beklemek zorundadır.

---

## 2. Linux Kernel Semaforları

Bunlar kullanıcının kullandığı değil, çekirdeğin kendi içinde kullandığı özel semaforlardır.

**Üç Türü Vardır:**

1. **Binary (Mutex):** 0 veya 1. (Tekillik).
2. **Counting (Sayan):** Birden fazla kaynağı yönetir.
3. **Reader-Writer:** Yukarıdaki spinlock mantığının "uyuyabilen" versiyonudur.

**Kritik Fonksiyonlar:**

- `down` **(Wait):** Kaynağı al (Azalt). Yoksa uyu.
- `up` **(Signal):** Kaynağı bırak (Artır).
- `down_interruptible` **(Çok Önemli):**
  - Normal `down` yaparsan ve kaynak asla gelmezse, process sonsuza kadar uyur (Zombi olur, `kill -9` bile işlemez).
  - `down_interruptible` ise "Kaynak yoksa uyurum AMA biri beni dürterse (Ctrl+C, Signal vb.) uyanırım" der. Device Driver (Sürücü) yazarken hep bu kullanılır ki donanım bozulursa bilgisayar donmasın.
- `down_trylock`**:** "Kaynak var mı? Yoksa hiç bekleme, işine bak." (Non-blocking).

---

## 3. Bariyerler (Barriers)

Burası biraz "Matrix" seviyesi. Derleyiciler (Compiler) ve İşlemciler (CPU) çok zekidir ama bazen "aşırı zeki" davranıp işleri karıştırırlar.

**Sorun: Reordering (Yeniden Sıralama)** Sen kodda şöyle yazdın:

1. `a = 1;`
2. `b = 2;`

İşlemci veya Derleyici diyebilir ki: *"B'yi belleğe yazmak daha kolayıma geldi, önce `b=2` yapayım, sonra `a=1` yaparım. Nasılsa sonuç aynı."* Tek işlemcide sorun yok. Ama çok işlemcili sistemde veya donanımla konuşurken bu sıra hayati olabilir!

**Çözüm: Bariyer Koymak** Kodun arasına "Buradan öteye geçiş yok" tabelası dikmektir.

- `rmb()` (Read Memory Barrier): "Bu satırdan önceki tüm okumalar bitmeden, sonraki okumalara geçme."
- `wmb()` (Write Memory Barrier): "Önceki yazmalar bitmeden sonrakini yazma."
- `mb()`: Hem okuma hem yazma için tam bariyer.

---

## 4. RCU (Read-Copy-Update) - Modern Hız Canavarı

2002'de Linux'a eklenen, performansı uçuran bir tekniktir. Dosya sistemlerinde ve ağ işlemlerinde kullanılır.

**Senaryo:** Bir veriyi 1000 kişi okuyor, 1 kişi çok nadiren güncelliyor. (Örn: IP yönlendirme tablosu). Kilit kullanırsan o 1000 kişi boşuna yavaşlar.

**RCU Çözümü (KİLİTSİZ Okuma):**

1. **Okurlar (Readers):** Hiçbir kilit almazlar! Direkt veriye (Pointer) erişip okurlar. Hız maksimumdur.
2. **Yazar (Writer):**
    - Eski veriyi yerinde değiştirmez (çünkü o an okuyan olabilir).
    - **Kopyala (Copy):** Verinin bir kopyasını oluşturur.
    - **Güncelle (Update):** Değişikliği kopyada yapar.
    - **Değiştir:** İşaretçiyi (Pointer) eski veriden yeni veriye **Atomik** olarak kaydırır.
3. **Çöp Toplama:** Eski veriyi hemen silemez (o an okuyan garibanlar olabilir). Tüm okurlar işini bitirene kadar bekler (`synchronize_rcu`), sonra eskiyi siler.

**Özetle RCU:** Okuması bedava (kilit yok), yazması biraz zahmetli (kopyala-yapıştır) olan bir yöntemdir. Okumanın çok olduğu yerlerde muazzam performans sağlar.

## 6.9 Solaris Eşzamanlılık Mekanizmaları

Solaris, hem çekirdek (kernel) hem de kullanıcı (user) seviyesinde aynı araçları sunar. En büyük özelliği **esnekliğidir**.

Solaris'te 4 temel senkronizasyon aracı vardır:

**1. Mutex (Karşılıklı Dışlama Kilidi)**
Standart kilittir. Sadece kilitleyen thread açabilir.

- **Uyarlanabilir Kilit (Adaptive Lock):** Solaris mutex'leri akıllıdır.
  - Eğer kilidi tutan diğer thread o an başka bir işlemci üzerinde **çalışıyorsa**, Solaris "O zaten çalışıyor, yakında bırakır, ben uyumayayım, kapıda döneyim (Spinlock)" der.
  - Eğer kilidi tutan thread **uyuyorsa**, "Ooo onun uyanacağı yok, ben de gidip uyuyayım (Sleep)" der.
- **mutex_tryenter:** "Kilit boşsa al, doluysa hiç bekleme, bana haber ver işime bakayım" (Non-blocking).

**2. Semaforlar**
Klasik sayan semaforlardır (sema_p, sema_v). Burada da sema_tryp ile beklemeden deneme şansı vardır.

**3. Readers/Writer Lock (Okur/Yazar Kilidi)**
Veritabanları için hayati olan kilit türü.

- **Çoklu Okuma:** Birden fazla okur aynı anda girebilir.
- **Tekli Yazma:** Yazar varken kimse giremez.
- **Downgrade (Rütbe İndirme):** Yazma kilidini elinde tutan bir thread, işi bitince "Ben artık sadece okuyacağım" diyerek kilidini **Okuma Kilidine** çevirebilir. Bu sayede kilidi tamamen bırakıp (araya başkasını sokup) tekrar almaya çalışmak zorunda kalmaz.

**4. Condition Variables (Koşul Değişkenleri)**
Monitör mantığını (Wait/Signal) uygulamak için Mutex ile birlikte kullanılır.

- cv_wait: Mutex'i bırak ve uyu.
- cv_signal: Bir kişiyi uyandır.
- cv_broadcast: Herkesi uyandır.

> **Önemli Kural:** Koşul değişkenleri her zaman bir while döngüsü içinde kullanılmalıdır. Çünkü uyandığında koşulun hala geçerli olup olmadığını tekrar kontrol etmen gerekir (while (!condition) cv_wait()).

---

## 6.10 Windows Eşzamanlılık Mekanizmaları

Windows, "Nesne Yönelimli" (Object-Oriented) bir mimari kullanır. Senkronizasyon araçları da birer **"Dispatcher Object"**tir.

**Bekleme Fonksiyonları (Wait Functions)**
Windows'ta bir thread'in bloklanması (uyuması) için bir **Nesneye** bakması gerekir.

- WaitForSingleObject(Handle): "Şu nesne **İşaretli (Signaled)** duruma gelene kadar beni uyut."
- Thread uyurken işlemci harcamaz.

**Dispatcher Nesneleri (Trafik Işıkları Gibi)**
Windows'ta nesnelerin iki durumu vardır:

1. **Signaled (Yeşil Işık):** Thread geçebilir (uyanabilir).
2. **Unsignaled (Kırmızı Işık):** Thread beklemek zorundadır.

İşte Windows'un alet çantasındaki oyuncaklar (Tablo 6.6):

| **Nesne Türü** | **Ne İşe Yarar?** | **Signaled (Yeşil) Olduğu An** | **Etkisi** |
|---|---|---|---|
| **Notification Event** | Bir olayın olduğunu duyurur (Örn: "Veritabanı yüklendi"). | Thread olayı set ettiğinde. | **HERKESİ** serbest bırakır (Broadcast gibi). |
| **Synchronization Event** | Tek tek izin verir (Örn: Turnike). | Thread olayı set ettiğinde. | Sadece **TEK BİR** thread'i geçirir, sonra otomatik kapanır. |
| **Mutex** | Karşılıklı dışlama. | Sahibi kilidi bıraktığında. | Sadece **TEK BİR** thread kilidi kapabilir. |
| **Semaphore** | Kaynak sayacı. | Sayaç > 0 olduğunda. | Sayaç bitene kadar thread'leri geçirir. |

| | | | |
|---|---|---|---|
| **Waitable Timer** | Zamanlayıcı. | Ayarlanan saat geldiğinde. | Bekleyen herkesi uyandırır. |
| **Process / Thread** | Bir programın çalışması. | Program/Thread sonlandığında (öldüğünde). | Onun bitmesini bekleyenleri (Örn: join) uyandırır. |

**Fark:** "Notification Event" manüel kapı gibidir, açarsın herkes geçer.
"Synchronization Event" ise otomatik turnike gibidir, bir kişi geçince tekrar kilitlenir.

## 1. Windows: Critical Sections (Kritik Bölümler)

Mutex, Event ve Semaphore gibi araçlar "Kernel Object" (Çekirdek Nesnesi) oldukları için ağırdırlar. Process'ler arası çalışabilirler ama sadece tek bir process içindeki thread'leri yönetmek için fazla maliyetlidirler.

Windows bunun için **Critical Section** adında, sadece **tek bir process'in thread'leri** arasında çalışan süper hızlı bir mekanizma sunar.

- **Neden Hızlı?** Çoğu zaman Kernel moduna (çekirdeğe) geçiş yapmaz. Kullanıcı modunda (User Mode) halleder.
- **Çalışma Mantığı:**
  - Önce bir **Spinlock** (dönme) dener. "Belki kilit hemen açılır, boşuna çekirdeği rahatsız etmeyeyim" der. Çok işlemcili sistemlerde bu harika çalışır.
  - Eğer kilit uzun süre açılmazsa, işte o zaman çekirdekten yardım ister ve thread'i uyutur.
- **Kullanımı:**
  - `EnterCriticalSection(&cs)`: Kilidi al.
  - `LeaveCriticalSection(&cs)`: Kilidi bırak.
  - `TryEnterCriticalSection(&cs)`: Beklemeden dene.

**Özetle:** Eğer sadece kendi programının içindeki thread'leri sıraya sokacaksan **Mutex değil, Critical Section kullan.** Çok daha performanslıdır.

---

## 2. Slim Reader-Writer (SRW) Locks ve Condition Variables

Windows Vista ile gelen bu araçlar, modern ve hafiftir.

- **SRW Lock:** "Slim" (İnce) denmesinin sebebi çok az bellek (sadece bir pointer kadar) kaplamasıdır. Okuma/Yazma kilitlerini (daha önce gördüğümüz Reader/Writer mantığı) çok hızlı uygular.
- **Condition Variables:** Monitörlerdeki `Wait/Pulse` mantığının Windows API karşılığıdır.
    - `SleepConditionVariableCS`: Kritik bölümde uyu.
    - `SleepConditionVariableSRW`: SRW kilidinde uyu.
    - `WakeConditionVariable`: Birini uyandır.
    - `WakeAllConditionVariable`: Herkesi uyandır.

---

## 3. Kilit-Siz (Lock-Free) Senkronizasyon

Kilit kullanmak (Mutex, CS) her zaman risklidir (Deadlock olabilir, thread uyuyabilir). Windows, donanımın gücünü kullanarak **kilitsiz** işlem yapmayı sağlar.

- **Interlocked Operations:** Donanım seviyesinde (CPU komutuyla) değişken okuma-yazma işlemini atomik yapar.
    - `InterlockedIncrement`: Bir artır.
    - `InterlockedCompareExchange`: "Eğer değer X ise Y yap." (CAS mantığı).
- **Avantajı:** Asla Deadlock olmaz, thread asla uyumaz. Çok yüksek performanslı kuyruklar (Queue) ve listeler (SList) bu yöntemle yapılır.

---

## 4. Android IPC: Binder

Android, Linux çekirdeği üzerine kuruludur ama Linux'un standart IPC araçlarını (Pipe, Message Queue vb.) pek kullanmaz. Bunun yerine **Binder** adında kendine has, hafif ve güvenli bir mekanizma geliştirmiştir.

**Mantık:** Android'de her uygulama ayrı bir sanal makinede (VM) ve ayrı bir process olarak çalışır. Bir uygulamanın (Client), diğerinin (Service) fonksiyonunu çağırması gerekir. Buna **RPC (Remote Procedure Call)** denir.

**Binder Nasıl Çalışır? (Adım Adım):**

1. **Client (İstemci):** "Harita servisinden şu konumu getir" fonksiyonunu çağırır.
2. **Proxy (Vekil):** Bu çağrıyı paketler (**Marshalling**). Yani veriyi kutuya koyar (Parcel).
3. **Binder Driver:** Çekirdek seviyesindeki bu sürücü, kutuyu alır ve diğer process'e (Servis) iletir. Hedef thread'i uyandırır.
4. **Stub (Taslak):** Kutuyu açar (**Unmarshalling**). Veriyi çıkarır.
5. **Service (Servis):** Gerçek işlemi yapar, sonucu üretir.
6. **Geri Dönüş:** Sonuç aynı yolla (Stub -> Binder -> Proxy -> Client) geri döner.

Bu yapı sayesinde Android uygulamaları birbirinin fonksiyonlarını sanki kendi kodlarıymış gibi güvenle çağırabilir.

---

## 6.12 ÖZET (Bölüm Sonu)

Bu bölümle birlikte **Deadlock (Kilitlenme)** konusunu tamamen bitirdik.

- **Deadlock:** Kaynaklar için yarışan veya haberleşen process'lerin birbirini sonsuza kadar kilitlemesi.
- **Çözüm 1 (Prevention):** Şartlardan birini (özellikle Circular Wait'i) baştan yasakla.
- **Çözüm 2 (Avoidance):** Banker algoritması ile "Güvenli mi?" diye her adımda kontrol et.
- **Çözüm 3 (Detection):** Özgür bırak, kilitlenirse tespit et ve birini öldür (Kill).

## GÖZDEN GEÇİRME SORULARI (Kısa Cevaplar)

**6.1. Tekrar kullanılabilir (Reusable) ve Tüketilebilir (Consumable) kaynaklara örnek ver.**

- **Reusable:** İşlemci, RAM, I/O kanalları, Disk, Veritabanı kilidi. (Kullanılır ve geri bırakılır).
- **Consumable:** Interrupt'lar, Sinyaller, Mesajlar, I/O tamponundaki veriler. (Üretilir ve tüketilir/yok edilir).

**6.2. Deadlock'un *mümkün* olması için gereken 3 şart nedir? (Policy Conditions)**

1. **Mutual Exclusion:** Kaynağı aynı anda tek kişi kullanabilir.
2. **Hold and Wait:** Elindekini tutarken yenisini istemek.
3. **No Preemption:** Kaynağın zorla geri alınamaması.

6.3. Deadlock'u oluşturan 4 şart nedir?

Yukarıdaki 3 şart + Circular Wait (Döngüsel Bekleme).

6.4. "Hold and Wait" nasıl önlenir?

Process'in ihtiyaç duyduğu tüm kaynakları en başta (tek seferde) istemesi zorunlu kılınarak.

**6.5. "No Preemption" nasıl önlenir?**

1. Bir process yeni kaynak istediğinde reddedilirse, elindeki mevcut kaynakları bırakması istenir.
2. İşletim sistemi, öncelik sırasına göre kaynağı zorla alıp (preempt) başkasına verebilir (Sadece durumu kaydedilebilen kaynaklar için).

6.6. "Circular Wait" nasıl önlenir?

Kaynaklara numara verilir (1, 2, 3...). Process'lerin kaynakları sadece artan sırada istemesi zorunlu kılınır. (Elinde 3 varken 1 isteyemez).

**6.7. Deadlock Avoidance, Detection ve Prevention farkı nedir?**

- **Prevention (Önleme):** 4 şarttan birini baştan yasaklayarak deadlock ihtimalini **sıfıra indirmek**.
- **Avoidance (Kaçınma):** Her isteği analiz edip, riskliyse (unsafe) bekletmek. (Banker Algoritması).
- **Detection (Tespit):** Karışmamak, deadlock olursa tespit edip birini öldürmek.

---

## SEÇİLMİŞ PROBLEM ÇÖZÜMLERİ (Sınavlık Sorular)

**SORU 6.5: Banker Algoritması (EFSANE SORU)**
Bu soru tipini çözemeden sınava girmek büyük risktir. Adım adım çözelim.

**Verilenler (T0 Anı):**

- **Kaynaklar:** A(15), B(6), C(9), D(10).
- **Mevcut (Available):** Başlangıçta verilmemiş ama hesaplayabiliriz.

Tablo Analizi:

Önce Need (İhtiyaç) matrisini bulmamız lazım. Formül: Need = Claim (Maksimum) - Allocation (Eldeki).

| Process | Allocation (Eldeki) | Claim (Maksimum) | Need (İhtiyaç) (C-A) |
|---|---|---|---|
| **P0** | 2 0 0 1 | 3 2 1 1 | **1 2 1 0** |
| **P1** | 3 1 2 1 | 5 1 2 1 | **2 0 0 0** |
| **P2** | 2 1 0 3 | 2 1 0 3 | **0 0 0 0** (Bitmiş sayılır) |
| **P3** | 1 3 1 2 | 3 3 2 2 | **2 0 1 0** |

| | | | |
|---|---|---|---|
| **P4** | 1 4 3 2 | 1 4 3 4 | **0 0 0 2** |
| **P5** | 1 2 1 1 | 3 3 2 1 | **2 1 1 0** |

**a. Available (Kasadaki) Değeri Doğrulama:**

- Dağıtılan Toplam (Allocation Sütunlarının Toplamı):
  - A: 2+3+2+1+1+1 = 10
  - B: 0+1+1+3+4+2 = 11 (Burada bir gariplik var, toplam kaynak 6 denmiş ama dağıtılan 11 görünüyor. Soruda basım hatası olabilir veya kaynak sayısı daha fazla. Biz sorudaki tabloya sadık kalalım).
  - *Not:* Genelde soruda "Available" verilir. Biz tabloda verilen Available: **(6, 3, 5, 4)** değerini doğru kabul edip devam edelim.

c. Sistem Güvenli mi? (Safe Sequence Bulma):

Elimizdeki Nakit (Available): A=6, B=3, C=5, D=4.

Sırayla bakalım, kimin Need (İhtiyaç) kısmını karşılayabiliriz?

1. **P2:** İhtiyacı (0,0,0,0). Zaten bitmiş. Kaynaklarını iade eder.
   - Yeni Kasa = (6,3,5,4) + (2,1,0,3) = **(8, 4, 5, 7)**
2. **P1:** İhtiyacı (2,0,0,0). Kasada (8,4,5,7) var. Yeter!
   - P1 çalışır, biter. İade eder: (3,1,2,1).
   - Yeni Kasa = (8,4,5,7) + (3,1,2,1) = **(11, 5, 7, 8)**
3. **P0:** İhtiyacı (1,2,1,0). Kasada yeterince var.
   - P0 çalışır, biter. İade eder: (2,0,0,1).
   - Yeni Kasa = (11,5,7,8) + (2,0,0,1) = **(13, 5, 7, 9)**
4. **P3, P4, P5:** Hepsinin ihtiyacı bu devasa kasadan karşılanabilir.

**Sonuç:** Evet, sistem **Güvenlidir (Safe)**. Örnek sıra: P2 -> P1 -> P0 -> P3 -> P4 -> P5.

**d. P5'in İsteği (3, 2, 3, 3) Karşılanmalı mı?**

- P5 şu anki "Need" (2,1,1,0) değerinden bile fazlasını (3,2,3,3) istiyor.
- **Kural:** Bir process, baştan beyan ettiği Maksimum (Claim) değerini aşamaz.
- **Cevap:** Hayır, reddedilmeli. Çünkü "Maksimum İhtiyaç" sözleşmesine aykırı davranıyor.

---

**SORU 6.17: Filozofların Stratejisi (Livelock Riski)**

**Soru:** Filozoflar şöyle bir algoritma kullansın:

1. Sol çatalı al.
2. Sağ çatal boşsa al ve ye.
3. **Sağ çatal doluysa, sol çatalı da bırak ve tekrar dene.**

Yorum:

Bu çözüm Deadlock (sonsuza kadar kilitlenme) sorununu çözer çünkü kimse çatalı tutup inat etmez, bırakır.

ANCAK, bu çözüm Livelock (Canlı Kilitlenme) riski taşır.

- **Senaryo:** Tüm filozoflar aynı anda sol çatalı alır.
- Hepsi sağa bakar, dolu görür.
- Hepsi aynı anda solu bırakır.
- Hepsi aynı anda tekrar solu alır...
- Bu senkronize dans sonsuza kadar sürerse kimse yemek yiyemez (Starvation).

---

**SORU 6.1: Trafik Örneği ve 4 Şart**
Şekil 6.1a'daki 4 arabanın kilitlenmesi için 4 şartın analizi:

1. **Mutual Exclusion:** Yolun bir karesinde (resource) sadece 1 araba durabilir. (Evet).
2. **Hold and Wait:** Araba kavşağın bir karesine girmiş (Hold), diğer kareye geçmek için bekliyor (Wait). (Evet).
3. **No Preemption:** Arabayı vinçle çekip alamayız. (Evet).
4. **Circular Wait:** Araba 1, Araba 2'yi bekliyor. Araba 2, Araba 3'ü... Zincir tamamlanıyor. (Evet).

---

## Son Tavsiye

Bu bölümden sınavda %90 ihtimalle **"Banker Algoritması tablosunu doldur ve güvenli sırayı bul"** sorusu gelir. Mantığı (Kasa hesabı yapmak) çok basittir, işlem hatası yapmamaya dikkat etmen yeterli.

## Temel Kavramlar (Sözlük)

Metinde geçen ve bu bölümün tamamında karşına çıkacak 3 terimi hemen netleştirelim. Bunları karıştırmaman çok önemli:

| Terim | Türkçe | Anlamı (Analoji) |
|---|---|---|

| | | |
|---|---|---|
| **Frame** | Çerçeve | **RAM'deki Boş Park Yeri.** Fiziksel bellekteki sabit boyutlu (örn: 4KB) kutucuklardır. |
| **Page** | Sayfa | **Araba.** Diskte duran ve o park yerine sığacak sabit boyutlu veri parçasıdır. |
| **Segment** | Bölüm | **Tır veya Motosiklet.** Değişken boyutlu veri parçasıdır. (Fonksiyonlar, değişkenler vb. mantıksal parçalar). |

**Özetle:** İşletim sistemi, diskteki "Sayfaları" (Pages) alır ve RAM'deki boş "Çerçevelere" (Frames) yerleştirir.

---

## Bellek Yönetiminin 5 Şartı (Requirements)

Bir işletim sistemi bellek yönetimi yapacağım diyorsa, şu 5 şartı sağlamak zorundadır:

### 1. Yer Değiştirme (Relocation)

Çoklu programlama sistemlerinde RAM çok dinamiktir. Bir program çalışırken yer kalmazsa diske atılabilir (Swap out), sonra tekrar RAM'e geri yüklenebilir (Swap in).

- **Sorun:** Program diske gidip geri geldiğinde, RAM'de **aynı adrese** (örneğin 1000. satıra) yerleşeceğinin garantisi yoktur. Belki orayı başkası kaptı?
- **Çözüm:** Programın içindeki adresler (pointer'lar, jump komutları) sabit (statik) olmamalıdır. İşletim sistemi, programı RAM'in neresine koyarsa koysun, adresleri ona göre **dinamik olarak çevirebilmelidir.**

### 2. Koruma (Protection)

Herkes kendi çöplüğünde ötmeli. Senin yazdığın Process A, yanlışlıkla (veya kasten) Process B'nin veya İşletim Sisteminin belleğine bir şey yazmamalı.

- **Kim Yapar?** Bu kontrolü **Donanım (İşlemci)** yapmak zorundadır.
- **Neden OS Yapmaz?** Çünkü her bellek erişiminde (milisaniyede milyonlarca kez) işletim sistemine sorarsak bilgisayar kağnı gibi yavaşlar. İşlemci, o an çalışan process'in sınırlarını bilir ve dışına çıkarsa "Hop!" der (Segmentation Fault hatası buradan gelir).

### 3. Paylaşım (Sharing)

Koruma iyidir ama bazen duvarları inceltmek gerekir.

- **Senaryo:** 100 kişi aynı anda "Not Defteri" uygulamasını açtı. RAM'de 100 tane notepad.exe kodu tutmak israf değil mi?
- **Çözüm:** Kodun "Salt Okunur" (Read-Only) kısmını RAM'de bir kere tut, 100 kişi de o tek kopyayı kullansın. Ama herkesin verisi (yazdığı notlar) kendine özel olsun.

**4. Mantıksal Organizasyon (Logical Organization)**

RAM fiziksel olarak dümdüz bir bit dizisidir (0'dan başlar sona kadar gider). Ama biz programcılar kod yazarken böyle düşünmeyiz. Bizim kafamızda:

- Fonksiyonlar (Code)
- Değişkenler (Data)
- Yığın (Stack)
  gibi "Modüller" vardır.
  İşletim sistemi, bizim bu modüler (parça parça) düşünce yapımızı alıp o dümdüz RAM'e sığdırmalıdır (Segmentation tekniği burada devreye girer).

**5. Fiziksel Organizasyon (Physical Organization)**

Bilgisayarda iki katman bellek var:

1. **RAM (Main Memory):** Hızlı, pahalı, elektrik gidince uçar (Volatile).
2. **Disk (Secondary Memory):** Yavaş, ucuz, kalıcı.

Eskiden programcılar RAM yetmeyince "Overlay" denen teknikle kodu parça parça diske yazıp okumak zorundaydı (Bu tam bir işkenceydi). Modern işletim sistemi bunu otomatik yapar. Programcı "RAM yetecek mi?" diye düşünmez, OS RAM ile Disk arasındaki veri akışını yönetir.

## 1. Sabit Bölümlendirme (Fixed Partitioning)

Bellek, sistem açılırken **sabit** boyutlu odalara bölünür. (Örn: 8MB, 8MB, 8MB...). Bu odaların boyutu çalışma sırasında değişmez.

- **Çalışma Mantığı:** Gelen process (program), sığabileceği bir odaya yerleştirilir.
- **Sorun (Internal Fragmentation - İç Parçalanma):**
  - Diyelim ki 8MB'lık bir odamız var.
  - İçeriye 2MB'lık bir program koyduk.
  - Kalan 6MB ne oldu? **Çöp oldu.** Odanın içinde olduğu için başkası kullanamaz.
  - **"Oda büyük, eşya küçük, kalan boşluk israf."** → İşte buna **Internal Fragmentation** denir.

---

## 2. Dinamik Bölümlendirme (Dynamic Partitioning)

Bellek baştan bölünmez. Process geldikçe ona **tam ihtiyacı kadar** yer verilir.

- **Çalışma Mantığı:** 64MB belleğim var. 10MB process geldi, 10MB kestim verdim. 54MB kaldı.
- **Avantajı:** Internal Fragmentation (iç israf) yoktur. Çünkü process'e tam uyan oda verdik.
- **Sorun (External Fragmentation - Dış Parçalanma):**
    - Zamanla process'ler girip çıkar.
    - Belleğin ortasında, başında, sonunda küçük küçük boşluklar (delikler) oluşur.
    - Toplamda 20MB boş yerin var ama hepsi parça parça.
    - 15MB'lık bir process gelse, sığacak **tek parça** yer bulamaz.
    - **"Toplam yer var ama hepsi dağınık."** → İşte buna **External Fragmentation** denir.

**Çözüm: Compaction (Sıkıştırma)**

İşletim sistemi ara sıra tüm process'leri yukarı kaydırıp, aradaki boşlukları aşağıda tek bir büyük boşluk olarak birleştirebilir.

- **Dezavantaj:** Bu işlem ("Defrag" mantığı) işlemciyi çok yorar ve sistemi o an dondurur.

---

## Yerleştirme Algoritmaları (Placement Algorithms)

Dinamik bölümlendirmede, elimizde birçok boşluk varken yeni gelen process'i hangisine koymalıyız? 3 strateji vardır:

1. **Best-Fit (En İyi Uyan):** Tüm belleği tara, process'e en yakın boyuttaki boşluğu bul.
    - *Amaç:* Büyük boşlukları harcamamak.
    - *Sonuç:* Geriye çok minik (kullanışsız) kırıntılar bırakır.
2. **First-Fit (İlk Uyan):** Baştan başla, sığdığı ilk boşluğa koy.
    - *Amaç:* Hızlı olmak. En çok kullanılan yöntemdir.
3. **Next-Fit (Kaldığı Yerden):** En son nereye koyduysan oradan aramaya devam et.
    - *Amaç:* Belleğin sonlarına doğru olan boşlukları da kullanmak.

## 1. Hangi Boşluğa Koyalım? (Placement Algorithms Karşılaştırması)

Önceki adımda bahsettiğimiz Best-Fit, First-Fit ve Next-Fit algoritmalarının hangisi daha iyi?

- **First-Fit (İlk Uyan):** Genelde **en iyisi ve en hızlısıdır.** Belleğin başını biraz kirletir ama hızlı sonuç verir.
- **Next-Fit (Sıradaki):** First-Fit'ten biraz daha kötüdür. Belleğin sonundaki büyük bloğu parçalama eğilimi vardır.
- **Best-Fit (En İyi Uyan):** İsmi havalı ama performansı **en kötüsüdür.** En küçük parçayı ararken hem çok zaman harcar hem de geriye işe yaramayacak kadar küçük (kırıntı) boşluklar bırakır. Bellek çöplüğe döner.

---

## 2. Buddy System (Kanka/Ahbap Sistemi)

Sabit bölümlendirme (Fixed) çok katıydı, Dinamik bölümlendirme (Dynamic) ise yönetmesi zordu (External Fragmentation).

Buddy System, bu ikisinin arasını bulan, 2'nin kuvvetleri ($2^n$) mantığıyla çalışan zekice bir yöntemdir.

Nasıl Çalışır?

Elimizde 1MB ($1024K$) tek parça bellek var diyelim.

Birisi gelip 100K yer istedi.

1. **Böl:** 1024K çok büyük. İkiye böl: **512K - 512K**. (Bunlar birbirinin "Buddy"si yani kankasıdır).
2. **Böl:** 512K hala büyük. Birini tekrar böl: **256K - 256K**.
3. **Böl:** 256K hala büyük. Birini tekrar böl: **128K - 128K**.
4. **Ver:** 128K, 100K için yeterli ve makul. Ver gitsin.

Geri Alma (Coalescing - Birleşme):

Kullanıcı işini bitirip 128K'yı iade ettiğinde sistem bakar:

- "Yanındaki 128K'lık kankası (Buddy) boş mu?"
- Eğer boşsa, onları birleştirip **256K** yapar.
- Sonra bakar, "Bu 256K'nın kankası boş mu?"
- Boşsa birleştirip **512K** yapar...
- Böylece bellek tekrar bütün haline gelir.

> **Kullanım Alanı:** Linux çekirdeği ve paralel sistemler bu yöntemi sever çünkü bölmesi ve birleştirmesi matematiksel olarak çok hızlıdır.

---

## 3. Yer Değiştirme (Relocation) ve Adres Çevirisi

İşletim sistemi bir programı (Process 2) diske attı (Swap out), sonra geri çağırdı (Swap in). Ama bu sefer RAM'in **farklı bir yerine** (örneğin 5000. adrese değil 8000. adrese) koydu.

Programın içindeki kodlar hala "5010. satıra git" diyorsa ne olacak? Program çöker. İşte bunu önlemek için adres türlerini ayırıyoruz:

**Adres Türleri**

1. **Mantıksal/Bağıl Adres (Logical/Relative Address):** Programın kendi dünyasındaki adrestir. "Başlangıçtan itibaren 10. adım" demektir.
2. **Fiziksel Adres (Physical Address):** RAM'deki gerçek, donanımsal adrestir.

**Donanım Desteği (Base & Bounds Registers)**
Bu çeviri işini yazılım yaparsa çok yavaş olur. İşlemci (CPU) üzerinde özel kayıtçılar (register) vardır:

- **Base Register (Taban):** Programın o an RAM'de başladığı adresi tutar. (Örn: 8000).
- **Bounds Register (Sınır):** Programın bitiş adresini veya boyutunu tutar. (Örn: 10000).

Formül:

İşlemci her komut çalıştırdığında şu hesabı otomatik yapar:

$$Fiziksel Adres = Base Register + Bağıl Adres$$

- *Örnek:* Program "10. adrese git" derse, İşlemci: $8000 + 10 = 8010$. adrese gider.
- **Koruma (Protection):** Eğer çıkan sonuç **Bounds Register**'dan büyükse, işlemci "Hop, başkasının alanına girmeye çalışıyorsun!" diyerek programı durdurur (Trap/Interrupt).

İşletim sistemlerinin modern bellek yönetiminin kalbine, **Sayfalama (Paging)** ve **Bölütleme (Segmentation)** konularına geldik. Bu iki yöntem, önceki bölümde gördüğümüz verimsiz "Bölümlendirme" (Partitioning) yöntemlerinin ilacıdır.

Senin için bu iki kavramı ve nasıl çalıştıklarını, teknik terimleri koruyarak ve görselleştirmelerle özetledim.

---

## 7.3 SAYFALAMA (Paging)

Önceki yöntemlerde (Partitioning) belleği ya sabit odalara bölüyorduk (israf oluyordu) ya da dinamik bölüyorduk (parça parça boşluklar kalıyordu).

**Paging** şunu der: **"Belleği de programı da eşit ve küçük parçalara bölelim."**

- **Frame (Çerçeve):** Fiziksel RAM'i sabit boyutlu (örneğin 4KB) bloklara böleriz. Bunlar "park yerleridir".
- **Page (Sayfa):** Çalışacak programı da aynı boyutta (4KB) dilimlere böleriz. Bunlar "arabalardır".

Nasıl Çalışır?

İşletim sistemi, programın sayfalarını (Page 0, Page 1...) alır ve RAM'de bulduğu herhangi bir boş çerçeveye (Frame 5, Frame 12...) yerleştirir.

- **Önemli:** Sayfaların RAM'de yan yana (bitişik) olması gerekmez. Page 1 RAM'in başında, Page 2 sonunda olabilir.

**Avantajı:**

- **External Fragmentation BİTER:** Çünkü her boşluk (Frame) standart boyuttadır ve her sayfa oraya "cuk" diye oturur. Arada boşluk kalmaz.
- **Internal Fragmentation:** Sadece programın **en son sayfasında** biraz boşluk kalabilir (Örn: Program 10.5 sayfa ise, son sayfada yarım boşluk kalır). Bu da ihmal edilebilir.

**Adres Çevirisi (Page Table)**
Program parçalandı ve RAM'e dağıldı. İşlemci "Page 1'deki veriyi getir" dediğinde, o sayfanın RAM'in neresinde (hangi Frame'de) olduğunu nasıl bilecek?

Cevap: Sayfa Tablosu (Page Table).

Her process'in bir haritası vardır.

- Process sorar: "Page 1, Satır 50'ye git."
- Tabloya bakılır: "Page 1 nerede? -> Frame 6'da."
- Fiziksel Adres: "Frame 6, Satır 50."

Teknik Detay (2'nin Kuvvetleri):

Sayfa boyutu 2'nin kuvveti (örn: $1KB = 2^{10}$) seçilirse, adres çevirisi donanım için çocuk oyuncağı olur.

- Adres: 000001 0111011110 (16 bit)
- **Sol Kısım (Page Number):** 000001 (Sayfa 1). Tabloya bak, Frame numarasını bul.
- **Sağ Kısım (Offset):** 0111011110 (Satır 478). Aynen kalır.
- **Sonuç:** Frame numarasını başa, Offset'i sona koy. Bitti.

---

## 7.4 BÖLÜTLEME (Segmentation)

Paging, donanım için harikadır ama programcı için anlamsızdır. Programcı kodunu "Sayfa 1, Sayfa 2" diye düşünmez; "Ana Fonksiyon, Değişkenler, Yığın (Stack)" diye düşünür.

**Segmentation** şunu der: **"Belleği programın mantıksal parçalarına (Segment) göre bölelim."**

- **Segment:** Programın mantıksal bir parçasıdır (Fonksiyon, Dizi vb.). Boyutları değişkendir (Biri 10KB, diğeri 50KB olabilir).
- **Yükleme:** İşletim sistemi her segmenti RAM'de sığacağı bir boşluğa yerleştirir. Yan yana olmaları gerekmez.

**Avantajı:**

- **Mantıksal:** Programcının düşünce yapısına uyar.
- **Koruma:** "Bu segment sadece okunabilir (Read-Only)" veya "Bu segmentte kod var" gibi yetkiler tanımlanabilir.

- **Paylaşım:** İki program aynı kütüphaneyi kullanacaksa, sadece o segmenti paylaşabilirler.

**Dezavantajı:**

- **External Fragmentation:** Segmentler farklı boyutta olduğu için (Dinamik Bölümlendirme gibi), RAM'de yine düzensiz boşluklar oluşur.

**Adres Çevirisi (Segment Table)**
Burada da bir harita (**Segment Table**) vardır ama işleyişi biraz farklıdır. Tabloda her segmentin **Başlangıç Adresi (Base)** ve **Uzunluğu (Limit)** tutulur.

- Adres: Segment No + Offset.
- **Adım 1:** Tablodan Segmentin başlangıç adresini bul.
- **Adım 2 (Koruma Kontrolü):** İstenen yer (Offset), segmentin uzunluğundan (Limit) büyük mü?
    - Büyükse: **HATA!** (Segmentation Fault). Başkasının alanına taşıyorsun.
    - Küçükse: Başlangıç Adresi + Offset işlemini yap ve fiziksel adrese git.

---

## Paging vs. Segmentation (Karşılaştırma)

| **Özellik** | **Paging (Sayfalama)** | **Segmentation (Bölütleme)** |
|---|---|---|
| **Bölme Mantığı** | Fiziksel, sabit boyutlu (Donanım dostu). | Mantıksal, değişken boyutlu (Programcı dostu). |
| **Parçalanma (Fragmentation)** | Dış parçalanma YOK. | Dış parçalanma VAR. |
| **Adres Çevirisi** | Sayfa No -> Çerçeve No. | Segment No -> Başlangıç Adresi + Toplama İşlemi. |
| **Görünürlük** | Programcı fark etmez (Invisible). | Programcı farkındadır (Visible). |

Bölüm 7'nin sonuna geldik. Burası, özellikle **Bellek Hesaplamaları** ve **Adres Çevirimi** konularıyla sınavların en çok puan getiren (veya kaybettiren) kısmıdır.

Senin için **Gözden Geçirme Sorularını** özetledim ve **Problemler** kısmından sınavda kesinlikle karşına çıkacak olan hesaplama sorularını (özellikle 7.12 ve 7.14) detaylı çözdüm.

---

## GÖZDEN GEÇİRME SORULARI (Kısa Cevaplar)

**7.6. Internal vs External Fragmentation farkı nedir?**

- **Internal (İç):** Tahsis edilen alanın (bölmenin) içindeki boşluktur. (Örn: 8MB'lık odaya 2MB'lık process koyarsan 6MB içeride israf olur). Sabit Bölümlendirme ve Sayfalama (son sayfada) görülür.
- **External (Dış):** Tahsis edilen alanların *dışında* kalan, kullanılamayan küçük boşluklardır. (Örn: Toplam 10MB boş yer var ama hepsi 1'er MB parça parça, 5MB'lık process sığmıyor). Dinamik Bölümlendirme ve Bölütlemede görülür.

**7.7. Mantıksal, Bağıl ve Fiziksel Adres farkı nedir?**

- **Mantıksal (Logical):** İşlemcinin (CPU) ürettiği, programın kendi dünyasındaki adrestir (Sayfa No + Offset).
- **Bağıl (Relative):** Programın başlangıcına göre olan mesafedir (Örn: "Başlangıçtan 100 bayt ileri").
- **Fiziksel (Physical):** RAM üzerindeki gerçek, donanımsal adrestir.

**7.8. Sayfa (Page) ve Çerçeve (Frame) farkı nedir?**

- **Page:** Programın (Process) bölündüğü sabit boyutlu parçadır (Mantıksal).
- **Frame:** RAM'in bölündüğü aynı boyutlu parçadır (Fiziksel). Bir Page, bir Frame'e yerleşir.

---

## SEÇİLMİŞ KRİTİK PROBLEMLER VE ÇÖZÜMLERİ

**SORU 7.12: Paging Hesaplaması (Tam Sınav Sorusu)**
**Verilenler:**

- Mantıksal Adres Uzayı: $2^{32}$ bayt (32-bit adresleme).
- Fiziksel Bellek (RAM): $2^{16}$ bayt (16-bit fiziksel adres).
- Sayfa Boyutu (Page Size): $2^{10}$ bayt (1 KB).

**Çözüm Adımları:**

a. Mantıksal Adreste kaç bit var?

Mantıksal uzay $2^{32}$ bayt olduğuna göre adres 32 bittir.

b. Bir Çerçeve (Frame) kaç bayttır?

Çerçeve boyutu = Sayfa boyutu = $2^{10}$ bayt (1024 bayt).

**c. Fiziksel Adreste "Frame Numarası" için kaç bit kullanılır?**

- Fiziksel Bellek = $2^{16}$ bayt.
- Frame Boyutu = $2^{10}$ bayt.
- Toplam Frame Sayısı = $2^{16} / 2^{10} = 2^6 = 64$ Frame.
- 64 Frame'i adreslemek için **6 bit** gerekir. (Fiziksel adresin yapısı: 6 bit Frame No + 10 bit Offset = 16 bit).

**d. Sayfa Tablosunda (Page Table) kaç giriş (entry) vardır?**

- Mantıksal Uzay = $2^{32}$ bayt.
- Sayfa Boyutu = $2^{10}$ bayt.
- Sayfa Sayısı = $2^{32} / 2^{10} = 2^{22}$ Sayfa.
- Yani tabloda **$2^{22}$ (Yaklaşık 4 Milyon)** satır vardır.

**e. Her satırda (Entry) en az kaç bit olmalıdır?**

- Her satırda "Hangi Frame'de olduğu" yazmalıdır.
- Frame numarası 6 bit idi (c şıkkı).
- Bir de geçerli/geçersiz (Valid bit) olsun (+1 bit).
- Toplam **7 bit** (Genelde 8 bit/1 byte'a yuvarlanır).

---

**SORU 7.14: Segmentasyon Adres Çevirimi**
Bu soru tipinde "Segmentation Fault" (Hata) olup olmadığını bulman istenir.

**Segment Tablosu:**

- **Seg 0:** Başlangıç: 660, Uzunluk: 248
- **Seg 1:** Başlangıç: 1752, Uzunluk: 422
- **Seg 2:** Başlangıç: 222, Uzunluk: 198
- **Seg 3:** Başlangıç: 996, Uzunluk: 604

**Sorgular (Mantıksal Adres: Segment No, Offset):**

- **a. (0, 198):**
    - Limit Kontrolü: Offset (198) < Uzunluk (248) mi? **EVET.**
    - Fiziksel Adres: Başlangıç (660) + 198 = **858**.
- **b. (2, 156):**

    - Limit Kontrolü: 156 < 198 mi? **EVET.**
    - Fiziksel Adres: 222 + 156 = **378**.
- **c. (1, 530):**
    - Limit Kontrolü: 530 < 422 mi? **HAYIR!** (Offset, uzunluktan büyük).
    - Sonuç: **SEGMENTATION FAULT (Hata).** Erişim engellenir.
- **d. (3, 444):**
    - Limit Kontrolü: 444 < 604 mü? **EVET.**
    - Fiziksel Adres: 996 + 444 = **1440**.

---

**SORU 7.7: Buddy System (Ahbap Sistemi) Hesaplaması**
**Senaryo:** 1 MB ($1024K$) bellek ile başla.

1. **İstek A (70K):** En yakın $2^n$ boyutu **128K**'dır.
    - 1024'ü böl -> 512, 512.
    - 512'yi böl -> 256, 256.
    - 256'yı böl -> 128, 128.
    - **A'ya 128K ver.**
2. **İstek B (35K):** En yakın boyut **64K**'dır.
    - Boşta kalan 128K vardı. Onu böl -> 64, 64.
    - **B'ye 64K ver.**
3. **Return A:** A (128K) geri döner.
    - Yanındaki 128K'lık "Ahbabı" (Buddy) boş mu? Hayır, oradan B'ye parça verdik. Birleşemezler.
    - 128K boşluk olarak kalır.

Bölüm 8'e (Sanal Bellek) geçmeden önce, Bölüm 7'nin sonunda yer alan çok kritik bir **EK (Appendix 7A)** var. Özellikle bir Bilgisayar Mühendisliği öğrencisi ve .NET geliştiricisi olarak **"Linker"**, **"Loader"** ve **"DLL"** kavramlarını bilmen şarttır. Mülakatlarda "Bir EXE dosyasına tıkladığında arkada ne olur?" sorusunun cevabı buradadır.

Senin için bu teknik süreci özetledim.

---

## APPENDIX 7A: Yükleme ve Bağlama (Loading and Linking)

Bir kod yazdığında (C# veya C++), o kodun çalıştırılabilir bir programa (EXE) dönüşmesi ve RAM'e yüklenmesi 3 aşamadan geçer:

1. **Derleme (Compilation):** Kaynak kod -> Obje kodu (.obj).
2. **Bağlama (Linking):** Obje kodları + Kütüphaneler -> Yükleme Modülü (.exe).
3. **Yükleme (Loading):** Yükleme Modülü -> RAM'deki Process.

---

## 1. Yükleme (Loading)

Diskteki programı RAM'e kopyalayıp çalışmaya hazır hale getirmektir. Ama **hangi adrese** koyacağız?

- **Mutlak Yükleme (Absolute Loading):**
  - Derleyici der ki: "Bu program kesinlikle RAM'in 1024. adresinde çalışacak."
  - **Sorun:** O adres doluysa program çalışmaz. Esneklik sıfırdır. Sadece çok basit gömülü sistemlerde kullanılır.
- **Yeniden Yerleştirilebilir Yükleme (Relocatable Loading):**
  - Derleyici der ki: "Adresleri bilmiyorum, her şeyi **başlangıca göre (Relative)** yazdım. Başlangıçtan 100 adım git, 500 adım git..."
  - Yükleyici (Loader) programı RAM'de boş bir yere (örn: 5000) koyar ve tüm adreslere 5000 ekler.
  - **Kısıt:** Program bir kere yüklendikten sonra RAM'de yeri değiştirilemez.
- **Dinamik Çalışma Zamanı Yüklemesi (Dynamic Run-Time Loading):**
  - Adres hesaplaması yükleme anında değil, **komut çalışırken** işlemci tarafından yapılır.
  - **Avantajı:** Program çalışırken durdurulup diske atılabilir, sonra RAM'in bambaşka bir yerine geri yüklenebilir. (Sanal belleğin temelidir).

---

## 2. Bağlama (Linking)

Senin yazdığın kod (`main.c`), başkasının yazdığı matematik kütüphanesini (`math.lib`) kullanıyorsa, bu ikisinin birleşmesi gerekir.

- **Statik Bağlama (Static Linking):**
  - Kullanılan tüm kütüphaneler (printf, sqrt vb.) senin `.exe` dosyanın içine kopyalanıp yapıştırılır.
  - **Sonuç:** EXE dosyası çok şişer.
- **Dinamik Bağlama (Dynamic Linking) - DLL:**
  - Kütüphaneler EXE'ye konmaz. Diskte ayrı bir dosya (`.dll` veya Linux'ta `.so`) olarak durur.
  - Program çalışırken işletim sistemine "Bana şu DLL lazım" der.
  - **Avantajı:**
    1. **Disk Tasarrufu:** 100 program aynı DLL'i kullanabilir, tek kopya yeter.
    2. **RAM Tasarrufu:** İşletim sistemi DLL'i RAM'e bir kere yükler, 100 program ortak kullanır (Sharing).
    3. **Kolay Güncelleme:** DLL dosyasını güncellersen, onu kullanan tüm programlar güncellenmiş olur. (Programı tekrar derlemeye gerek kalmaz).

**DLL Cehennemi (DLL Hell)**

Windows dünyasının meşhur sorunudur.

- A Programı `Shared.dll`'in **v1.0** versiyonunu istiyor.
- B Programı `Shared.dll`'in **v2.0** versiyonunu istiyor.
- İkisi de aynı klasördeki veya sistemdeki tek bir DLL'e bakıyorsa, biri mutlaka patlar.
- *.NET bu sorunu "Assembly Versioning" ve "GAC" ile büyük ölçüde çözmüştür.*

Hoş geldin **Bölüm 8'e: Sanal Bellek (Virtual Memory)**.

Burası işletim sistemlerinin en büyük "sihirbazlık" numarasıdır. Mülakatlarda, sınavlarda ve gerçek hayatta en çok karşına çıkacak konudur.

Temel soru şudur: **"Bilgisayarımda 8 GB RAM var ama nasıl 100 GB'lık GTA V oyununu oynayabiliyorum?"**

Cevap: **Sanal Bellek.**

Senin için bu giriş bölümünü teknik terimleri koruyarak, metaforlarla ve ders notu formatında özetledim.

---

## 1. Temel Kavramlar (Sözlük)

Önce şu 3 terimi netleştirelim, çünkü bölüm boyunca sürekli bunları kullanacağız:

| **Terim** | **Anlamı** | **Analoji** |
|---|---|---|
| **Sanal Bellek (Virtual Memory)** | Diskin (Harddisk/SSD) bir kısmının sanki RAM'miş gibi kullanılmasıdır. | Cebinde 100 TL var (RAM) ama kredi kartınla (Disk) 10.000 TL harcayabiliyorsun. |
| **Sanal Adres (Virtual Address)** | Programın *sandığı* adrestir. Programcı "Benim 1 Milyar bayt yerim var" sanır. | Matrix dünyasında gördüğün adres. (Hayali). |

| | | |
|---|---|---|
| **Gerçek Adres (Real Address)** | Verinin fiziksel RAM'de *gerçekten* durduğu yerdir. | Gerçek dünyadaki adres. (Fiziksel). |

---

## 2. Büyük Devrim: "Hepsini Yüklemeye Gerek Yok!"

Bölüm 7'de gördüğümüz **"Simple Paging" (Basit Sayfalama)** ile Bölüm 8'deki **"Virtual Memory Paging"** arasındaki o ince çizgiyi anlamak çok önemlidir:

- **Eskiden (Simple Paging):** Bir programın çalışması için **TÜM sayfalarının** RAM'de olması zorunluydu. 10 sayfalık programın varsa ve RAM'de 9 sayfalık yer varsa, o program çalışmazdı.
- **Şimdi (Virtual Memory):** Programın çalışması için sadece **o an lazım olan parçanın (Resident Set)** RAM'de olması yeterlidir.
    - *Analoji:* Netflix'te 2 saatlik filmi izlemek için filmin tamamının inmesini beklemezsin. Sadece önündeki 2 dakikayı (Buffer) indirirsin, gerisi sunucuda (Diskte) durur.

---

## 3. Mekanizma Nasıl Çalışır? (Page Fault Senaryosu)

Sanal belleğin kalbi **"Hata Yönetimi"** üzerine kuruludur.

1. **Başlangıç:** Programın sadece ilk sayfasını RAM'e yükle ve çalıştır.
2. **İşleyiş:** İşlemci çalışırken sürekli Sanal Adreslere erişir.
3. **Hata (The Fault):** İşlemci, RAM'de olmayan bir sayfayı (örneğin Sayfa 50) isterse ne olur?
    - Donanım **"Memory Access Fault"** (Bellek Erişim Hatası) verir.
    - (Buna teknik dilde **Page Fault** denir).
4. **Müdahale:** İşletim Sistemi (OS) devreye girer:
    - Process'i bloklar (Uyutur).
    - Diske gider, o sayfayı bulur.
    - RAM'e getirir (Gerekirse RAM'den başka birini kovar).
    - Tabloyu günceller: "Artık Sayfa 50 RAM'de."
    - Process'i uyandırır: "Kaldığın yerden devam et."

---

## 4. Neden Bunu Yapıyoruz? (Faydaları)

Bu karmaşık "Git-Gel" işinin bize iki devasa faydası vardır:

1. **Daha Çok Process:** Her programın sadece ufak bir kısmını RAM'de tuttuğumuz için, RAM'e aynı anda 10 yerine 100 program sığdırabiliriz. (Multiprogramming coşar).
2. **RAM'den Büyük Programlar:** Programcı artık "Kullanıcının RAM'i yeter mi?" diye korkmaz. 4 GB RAM'i olan bilgisayarda 50 GB'lık veri işleyebilirsin. Sınır artık RAM değil, Disktir.

---

## 5. Sistem Yavaşlamaz mı? (Locality Prensibi)

*"Sürekli diske gidip veri getirmek bilgisayarı kağnı gibi yapmaz mı?"* (Disk, RAM'den binlerce kat yavaştır).

Cevap: **Hayır, yapmaz.** Çünkü programlar **"Locality of Reference" (Yerellik)** ilkesine göre çalışır.

- Bir program çalışırken zamanının %90'ını kodun %10'luk bir kısmında (döngülerde, belirli fonksiyonlarda) geçirir.
- Yani bir kere gerekli sayfaları yükleyince (Resident Set), uzun süre diske gitmeden RAM'den çalışmaya devam ederiz. Nadiren yeni sayfa gerekir.

Sanal belleğin (Virtual Memory) kalbine, donanımın bu işi nasıl hallettiğine iniyoruz.

Burada iki ana oyuncu var: **Sayfalama (Paging)** ve **Sayfa Tabloları (Page Tables)**.

Senin için metni teknik terimleri koruyarak, "salağa anlatır gibi" basitleştirip özetledim.

---

## 1. Thrashing (Çırpınma/Debelenme)

Sanal bellek harikadır, RAM'den tasarruf sağlar. Ama işletim sistemi açgözlü davranıp, "Şunu atayım, bunu alayım" derken ipin ucunu kaçırırsa ne olur?

- **Senaryo:** İşletim sistemi RAM dolu diye Sayfa A'yı diske atar.
- **Hemen Sonra:** İşlemci "Bana Sayfa A lazım" der.
- **Tekrar:** İşletim sistemi Sayfa A'yı geri getirir, yer açmak için Sayfa B'yi atar.
- **Hemen Sonra:** İşlemci "Bana Sayfa B lazım" der.

Bu kısır döngüye **Thrashing** denir. Bilgisayar sürekli diskle uğraşmaktan (Swapping), gerçek iş yapmaya (Executing) vakit bulamaz. Sonuç: Sistem kilitlenir, disk ışığı yanar durur.

**Çözüm: Locality (Yerellik) İlkesi**
Bölüm 1'de bahsettiğimiz ilke burada hayat kurtarır. Programlar rastgele değil, kümeler halinde (Cluster) çalışır. OS, "Bu sayfa az önce kullanıldı, yine kullanılacak, onu atma!" diyerek Thrashing'i önler.

---

## 2. Sayfalama (Paging) ve Sayfa Tablosu Girişi (PTE)

Sanal bellekli sistemlerde de RAM ve Programlar yine eşit parçalara (Page/Frame) bölünür. Ama bu sefer Sayfa Tablosu (Page Table) biraz daha karmaşıktır.

PTE (Page Table Entry) İçeriği (Şekil 8.1):

Her satırda sadece "Hangi Frame'de olduğu" yazmaz. Ekstra bilgiler vardır:

1. **Present (P) Bit:** "Bu sayfa şu an RAM'de mi?" (1: Evet, 0: Hayır, diskte).
    - Eğer 0 ise ve erişilmeye çalışılırsa -> **Page Fault!**
2. **Modify (M) Bit / Dirty Bit:** "Bu sayfa RAM'e geldikten sonra değişti mi?"
    - Eğer 1 ise: Sayfa RAM'den atılırken diske yazılmalı (Save edilmeli).
    - Eğer 0 ise: Sayfa üzerinde değişiklik yapılmadı, direkt silinebilir (Diskteki kopyasıyla aynı).
3. **Frame Number:** Eğer P=1 ise, sayfanın RAM'deki adresi.

---

## 3. Çok Seviyeli Sayfa Tabloları (Multi-Level Page Tables)

Sorun:

32-bit bir sistemde her process 4 GB sanal belleğe sahip olabilir.

- Sayfa boyutu 4KB olsun.
- Sayfa sayısı: $2^{32} / 2^{12} = 1$ Milyon Sayfa.
- Her satır (PTE) 4 Byte olsa -> **4 MB**'lık bir tablo eder.
- 100 tane process çalışsa -> Sadece tablolar **400 MB RAM** yer! Bu kabul edilemez.

Çözüm: Tabloları da Sayfalamak (Two-Level Paging)

Sayfa tablosunun kendisini de parçalara bölüp, sadece lazım olan kısmını RAM'de tutarız.

Mekanizma (Şekil 8.3):

Sanal adresi 3 parçaya böleriz (Örn: 10 bit, 10 bit, 12 bit).

1. **Root Page Table (Kök Tablo):** RAM'de daima durur. İlk 10 bit burayı işaret eder. Buradan "User Page Table"ın nerede olduğunu öğreniriz.
2. **User Page Table (Kullanıcı Tablosu):** Bu tablo çok büyük olduğu için parçalıdır ve diskte olabilir. İkinci 10 bit, bu tablonun içindeki satırı gösterir.
3. **Offset:** Sayfa içindeki satır.

**Analoji:**

- **Tek Seviyeli:** Şehrin tüm telefon rehberini (4000 sayfa) cebinde taşıyorsun.
- **İki Seviyeli:** Cebinde sadece "İlçe Listesi" (Kök Tablo) var. Kadıköy'e gidince Kadıköy rehberini (User Table) kütüphaneden alıp bakıyorsun.

Sanal belleğin derinliklerine iniyoruz. Önceki adımda "Her process için devasa bir sayfa tablosu (Page Table) tutmak RAM'i tüketir" demiştik.

Şimdi mühendislerin bu sorunu çözmek için geliştirdiği **"Ters Sayfa Tablosu"** ve sistemi hızlandırmak için kullandıkları donanım hilesi **"TLB"**yi inceleyeceğiz.

---

## 1. Ters Sayfa Tablosu (Inverted Page Table)

Normalde her Process'in kendine ait bir tablosu vardır ve "Benim 1. sayfam nerede, 2. sayfam nerede?" diye tutar. Sanal bellek büyüdükçe bu tablo devasa boyutlara ulaşır.

Mühendisler demiş ki: **"Tabloyu sanal sayfalara göre değil, fiziksel RAM'e göre yapalım!"**

- **Normal Tablo:** "Process A'nın 5. sayfası -> Frame 100'de."
- **Ters Tablo:** "Frame 100 -> Process A'nın 5. sayfasına ait."

Mantık:

RAM'de kaç tane çerçeve (Frame) varsa, tablonun boyutu o kadardır. Process sayısı veya sanal bellek boyutu ne olursa olsun tablo boyutu sabittir. RAM'den büyük tasarruf sağlar.

- **Nasıl Bulunur?** Sanal adresi bir **Hash Fonksiyonuna** sokarız, o bize tablonun neresine bakmamız gerektiğini söyler. (PowerPC, UltraSPARC ve IA-64 mimarileri bunu kullanır).

---

## 2. TLB (Translation Lookaside Buffer) - Donanım Hilesi

Sanal bellek harikadır ama bir bedeli vardır: Yavaşlık.

İşlemci bir veriyi okumak istediğinde iki kez RAM'e gitmek zorundadır:

1. **Adres Çevirisi:** Sayfa tablosuna git, verinin nerede olduğunu öğren.
2. **Veri Erişimi:** Öğrendiğin adrese git, veriyi al.

Bu, bilgisayarı %50 yavaşlatır. Çözüm **TLB** adında, işlemcinin içine gömülü, süper hızlı bir önbellektir.

TLB Nedir?

İşlemcinin "Kopyayı" çektiği yerdir. Son yapılan adres çevirilerini (Örn: "Sayfa 5 -> Frame 12'de") hafızasında tutar.

**Akış Şeması (Şekil 8.7):**

1. **İşlemci:** "Sayfa 5 lazım."
2. **TLB Kontrolü:** "Sayfa 5'in adresi bende kayıtlı mı?"
    - **TLB Hit (Buldum!):** Süper! RAM'deki tabloya gitmeye gerek yok. Direkt veriyi al. (Milisaniyeler sürer).
    - **TLB Miss (Yok):** Mecburen RAM'deki Sayfa Tablosuna git, adresi öğren. Sonra bunu TLB'ye kaydet ki bir dahakine hızlı bulalım.

---

## 3. Sayfa Boyutu (Page Size) Ne Olmalı?

Mühendislerin en çok tartıştığı konu: Sayfalar küçük mü olsun (örn: 512 Byte), büyük mü (örn: 4 MB)?

| Boyut | Avantaj | Dezavantaj |
|---|---|---|
| **Küçük Sayfa** | **Internal Fragmentation Azalır:** Son sayfadaki boşluk çok küçük olur. | **Tablo Büyür:** Belleği çok fazla küçük parçaya böldüğümüz için sayfa tablosu devasa olur. |
| **Büyük Sayfa** | **Tablo Küçülür:** Daha az sayfa, daha küçük tablo. Diskten okumak (I/O) daha verimlidir. | **Internal Fragmentation Artar:** Son sayfanın yarısı boş kalırsa büyük israf olur. |

Modern Trend:

RAM kapasiteleri arttığı için (GB'lar seviyesinde), Internal Fragmentation (birkaç KB israf) artık o kadar dert edilmiyor. Bu yüzden performans için Büyük Sayfa (4 KB, hatta bazen 2 MB veya 1 GB "Huge Pages") tercih ediliyor.

> **Grafik Yorumu (Şekil 8.10a):** Sayfa boyutu büyüdükçe "Page Fault" (Sayfa Hatası) önce artar (çünkü alakasız verileri de yüklersin), sonra azalır (çünkü sayfa o kadar büyür ki programın tamamını içine alır).

Şimdiye kadar **Paging (Sayfalama)** ve **Segmentation (Bölütleme)** tekniklerini ayrı ayrı gördük.

- **Paging:** Donanım için harikadır (Verimli, boşluk bırakmaz) ama programcı için kördür (Kodun neresi fonksiyon, neresi veri bilmez).
- **Segmentation:** Programcı için harikadır (Mantıksaldır) ama donanım için zordur (Parça parça boşluklar kalır).

Mühendisler demiş ki: **"Neden ikisinin de en iyi özelliklerini alıp birleştirmiyoruz?"**

İşte modern işlemcilerin (Intel x86 dahil) kullandığı o hibrit yapı. Senin için özetledim.

---

## 1. Çoklu Sayfa Boyutları (Multiple Page Sizes)

TLB (Hızlı adres çeviren önbellek) sınırlı bir kaynağa sahiptir. Eğer programın çok büyükse (Örn: 10 GB), 4KB'lık sayfalarla TLB'yi hemen doldurursun ve sürekli "TLB Miss" yersin.

Modern işlemciler (MIPS, Alpha, UltraSPARC, x86) buna çözüm olarak **Çoklu Sayfa Boyutu** destekler.

- **Küçük Sayfalar (4KB):** Hassas işler ve küçük veri yapıları için.
- **Büyük Sayfalar (2MB veya 4MB):** İşletim sistemi çekirdeği veya oyunun büyük doku dosyaları (Textures) için.
- **Sonuç:** TLB'deki tek bir girişle devasa bir alanı yönetebilirsin. Performans artar.

---

## 2. Sanal Bellekte Bölütleme (Segmentation)

Sanal bellek sadece sayfalama ile yapılmaz, bölütleme ile de yapılabilir.

**Programcı Gözüyle Avantajları:**

1. **Büyüyen Veri Yapıları:** Örneğin "Stack" veya "Heap" dolarsa, işletim sistemi sadece o segmenti büyütür veya başka yere taşır. Paging'de bunu yapmak zordur.
2. **Bağımsız Derleme:** Kodu değiştirdin mi? Sadece o kodun olduğu segmenti yeniden derle (Recompile). Diğer segmentlere dokunma.
3. **Paylaşım (Sharing):** "Math.dll" kütüphanesini iki program kullanacaksa, işletim sistemi o segmenti iki process'in de tablosuna ekler.
4. **Koruma (Protection):** "Bu segmentte kod var, buraya yazma (Read-Only)" demek, "5. sayfaya yazma" demekten çok daha mantıklıdır.

---

## 3. HİBRİT YAPI: Paging ve Segmentation Bir Arada

İşte en güçlü teknik budur. Modern sistemler genellikle kullanıcının **Segment** gördüğü ama arka planda donanımın **Page** yönettiği bir yapı kullanır.

**Nasıl Çalışır?**

1. **En Üstte:** Kullanıcının programı **Segmentlere** bölünür. (Örn: Segment 1: Kod, Segment 2: Veri).
2. **Ortada:** Her Segment, kendi içinde **Sayfalara (Pages)** bölünür.
3. **En Altta:** Bu sayfalar fiziksel RAM'deki **Çerçevelere (Frames)** dağıtılır.

**Adres Yapısı:** Sanal Adres 3 parçadan oluşur: `Segment Numarası` + `Sayfa Numarası` + `Offset`

**Adres Çevirisi (Adım Adım):**

1. **Process:** "Segment 1, Sayfa 5, Satır 20'yi ver."
2. **Segment Tablosu:** Segment 1'in "Sayfa Tablosu" nerede? -> Adresini bulur.
3. **Sayfa Tablosu:** O tablonun 5. Sayfasına bak. -> Hangi Frame'de? (Örn: Frame 100).
4. **Fiziksel Adres:** Frame 100 + Satır 20.

**Faydası Nedir?**

- **Dış Parçalanma Yok:** Çünkü en altta Paging kullanıyoruz, her şey eşit boyutta.
- **Mantıksal Yönetim:** Üstte Segmentation var, paylaşım ve koruma çok kolay.

---

## 4. Koruma ve Paylaşım (Protection & Sharing)

Segmentasyonun en büyük artısı buradadır. Şekil 8.13'te görüleceği gibi, koruma bitleri (Read/Write/Execute) **Segment Tablosunda** tutulur.

- Eğer bir sayfayı "Sadece Okunabilir" yapmak istersen Paging'de bunu her sayfa için tek tek işaretlemen gerekir.
- Segmentation'da ise "Bu segment KOD segmentidir" dersin, içindeki 1000 sayfa otomatik olarak "Sadece Okunabilir" olur. Yönetimi çok daha kolaydır.

İşletim Sistemlerinin en kritik karar mekanizmalarına, yani **"Sanal Bellek Politikalarına"** (Virtual Memory Policies) geçiyoruz. Donanım bize sayfalama (Paging) ve TLB gibi araçları verdi ama bunları *nasıl* kullanacağımıza İşletim Sistemi karar verir.

Özellikle **"Sayfa Değiştirme Algoritmaları" (Page Replacement Algorithms)** mülakatların ve sınavların en popüler konusudur.

Senin için bu politikaları ve algoritmaları teknik terimleri koruyarak özetledim.

---

## 1. Getirme Politikası (Fetch Policy)

Sayfayı RAM'e *ne zaman* yükleyelim?

1. **Talep Üzerine Sayfalama (Demand Paging):**
    - **Mantık:** Sadece istendiğinde yükle. Başlangıçta program boş başlar, ilk komutta "Page Fault" yer, sayfa yüklenir. Sonra ikinci komutta yine hata, yine yükle...
    - **Avantaj:** Gereksiz hiçbir şey yüklenmez.
    - **Dezavantaj:** Başlangıçta sistem çok yavaşlar (Page Fault fırtınası).
2. **Önceden Sayfalama (Prepaging):**
    - **Mantık:** Diskten bir sayfa okurken, yanındaki sayfaları da okuyayım (nasılsa disk kafası oraya gitmişken).
    - **Avantaj:** Gelecekteki page fault'ları önler. Disk I/O verimlidir.
    - **Dezavantaj:** Ya o yan sayfalar hiç kullanılmazsa? Boşuna RAM işgal eder.

---

## 2. Yerleştirme Politikası (Placement Policy)

Sayfayı RAM'in *neresine* koyalım?

- Paging kullanıyorsak bu soru **önemsizdir**. Çünkü tüm çerçeveler (Frame) eşittir. Herhangi bir boş çerçeveye koyabilirsin.
- Ancak **NUMA** (Çok işlemcili, dağınık bellekli) sistemlerde, işlemciye en yakın RAM modülüne koymak performansı artırır.

---

## 3. Değiştirme Politikası (Replacement Policy) - KRİTİK KONU

RAM tamamen doldu. Yeni bir sayfa (Sayfa X) yüklememiz lazım. RAM'deki mevcut sayfalardan hangisini kurban edip atalım?

### A. Çerçeve Kilitleme (Frame Locking)

Bazı sayfalar asla atılamaz (Locked/Pinned).

- İşletim sistemi çekirdeği (Kernel).
- I/O tamponları (O sırada diske yazılan veri atılamaz).

### B. Temel Algoritmalar (Sınav Sorusu)

Şekil 8.14'teki örnek senaryoya göre algoritmaları inceleyelim: **Erişim Sırası:** 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2 **RAM Kapasitesi:** 3 Çerçeve.

#### 1. Optimal Politika (OPT):

- **Kural:** Gelecekte *en uzun süre* kullanılmayacak olanı at.
- **Analiz:** 5 geldiğinde RAM'de {2, 3, 1} var. Geleceğe bak: 2 hemen kullanılacak, 5 birazdan, 3 daha sonra... 1 ise hiç kullanılmayacak veya çok geç. O zaman 1'i at.

- **Sorun:** Geleceği görmek imkansızdır. Bu algoritma sadece diğerlerini kıyaslamak için "Teorik Üst Sınır" olarak kullanılır.

**2. En Az Son Kullanılan (LRU - Least Recently Used):**

- **Kural:** Geçmişte *en uzun süredir* kullanılmayan (en yaşlı) sayfayı at.
- **Mantık:** "Bir sayfa uzun süredir kullanılmadıysa, muhtemelen bir daha da kullanılmaz" (Locality Prensibi).
- **Analiz:** 5 geldiğinde RAM'de {2, 3, 1} var. Geçmişe bak: 1 en son geldi (yeni), 2 az önce kullanıldı. 3 ise en eskiden kullanıldı. O zaman 3'ü at. (Dikkat: LRU "ilk geleni" değil, "en son erişileni" takip eder).
- **Performans:** Optimal'e çok yakındır, harikadır.
- **Sorun:** Uygulaması çok pahalıdır. Her erişimde sayfaya "zaman damgası" basmak veya bir listeyi güncellemek donanımı yorar.

**3. İlk Giren İlk Çıkar (FIFO - First-In First-Out):**

- **Kural:** RAM'e ilk kim girdiyse onu at. (En eski sayfayı at).
- **Mantık:** Basit kuyruk mantığı.
- **Sorun:** Belki ilk giren sayfa (Örn: Ana Fonksiyon) en çok kullanılan sayfadır? FIFO bunu umursamaz, atar. Bu da performansı düşürür. Ayrıca **Belady Anomalisi** denen garip bir duruma yol açar (RAM'i artırsan bile hata sayısı artabilir).

**4. Saat Algoritması (Clock Policy):**

- LRU'nun "fakir adam" versiyonudur. LRU kadar iyi ama FIFO kadar basittir.
- Bir sonraki adımda bunu detaylı inceleyeceğiz çünkü modern işletim sistemleri (Linux, Windows) genelde bunu kullanır.

Harika, şimdi işletim sistemlerinin "sihirli değneği" olan ve modern sistemlerde (Windows, Linux, macOS) sıklıkla kullanılan **Saat Algoritması (Clock Policy)** konusuna geliyoruz.

Bu algoritma, "LRU kadar akıllı olsun ama FIFO kadar basit çalışsın" fikrinin ürünüdür.

Senin için metni teknik terimleri koruyarak ve görselleştirerek özetledim.

---

## 1. Saat Algoritması (Clock Policy)

LRU harikadır ama her sayfaya "erişim zamanı" damgası basmak donanımı çok yorar. Saat algoritması bunun yerine her sayfaya sadece **1 bitlik** bir işaret koyar: **Kullanım Biti (Use Bit).**

**Mekanizma:** RAM'deki sayfaları bir **saat kadranı** gibi dairesel dizdiğimizi düşün. Bir de saatin yelkovanı gibi dönen bir **işaretçi (pointer)** var.

1. **Sayfa Yüklendiğinde:** `Use Bit = 1` yapılır.
2. **Sayfaya Erişildiğinde:** `Use Bit = 1` yapılır.
3. **Yer Lazım Olduğunda (Replacement):**
    - İbre dönmeye başlar.
    - Önüne gelen sayfanın `Use Bit`i **1** ise: "Bu yakın zamanda kullanılmış, buna dokunma" der. Biti **0** yapar (ikinci şans verir) ve bir sonraki sayfaya geçer.
    - Önüne gelen sayfanın `Use Bit`i **0** ise: "Hah! Buna uzun süredir kimse dokunmamış (veya ikinci şansını da kullanmamış). Kurban sensin!" der ve onu atar.

**Neden Akıllıca?** Eğer bir sayfa sık kullanılıyorsa, ibre ona gelene kadar birileri mutlaka tekrar erişip bitini 1 yapacaktır. Böylece o sayfa sürekli "dokunulmazlık" kazanır ve RAM'de kalır. Kullanılmayanlar ise 0'a düşer ve ilk turda atılır.

---

## 2. Gelişmiş Saat Algoritması (Use + Modify Bit)

Sadece "Kullanıldı mı?" (`u`) diye bakmak yetmez. Bir de "Değiştirildi mi?" (`m`) diye bakmak performansı artırır.

- **Değiştirilmemiş (Clean) Sayfa:** Diskteki kopyasıyla aynıdır. Atarken diske yazmaya gerek yoktur, direkt üzerine yazabiliriz. (Hızlı).
- **Değiştirilmiş (Dirty) Sayfa:** Atmadan önce diske kaydetmek gerekir. (Yavaş - Disk I/O).

**Strateji:** İşletim sistemi kurban seçerken şu öncelik sırasına göre arar (En iyiden en kötüye):

1. **(u=0, m=0):** Kullanılmamış ve Değişmemiş. (En iyi kurban! Hemen atılabilir).
2. **(u=0, m=1):** Kullanılmamış ama Değişmiş. (Mecburen diske yazacağız ama en azından yakın zamanda kullanılmamış).
3. **(u=1, m=0):** Kullanılmış ama Değişmemiş. (Atması kolay ama yakında yine lazım olabilir).
4. **(u=1, m=1):** Kullanılmış ve Değişmiş. (En kötü aday. Hem lazım olacak hem de diske yazmak lazım).

Bu yöntemle işletim sistemi hem en az kullanılanı bulur hem de disk trafiğini (I/O) azaltır. Macintosh sistemleri bunu kullanmıştır.

---

## 3. Sayfa Tamponlama (Page Buffering)

Bu, yukarıdaki algoritmaların performansını katlayan bir "Yedek Kulübesi" taktiğidir (VAX/VMS sisteminde kullanılmıştır).

**Mantık:** Bir sayfayı (Kurban) RAM'den atmaya karar verdiğimizde, onu **hemen silmeyiz.** Onu "Değiştirilenler Listesi" veya "Boşlar Listesi" adında bir bekleme kuyruğuna (Tampon) alırız. Sayfa fiziksel olarak hala RAM'dedir ama "Silindi" etiketi yemiştir.

**Faydası:**

1. **Hızlı Geri Dönüş:** Eğer işlemci "Pardon ya, o attığın sayfa bana lazımdı" derse, diskten okumak yerine bu listeden **anında** geri getirilir. (Geri Dönüşüm Kutusundan dosya kurtarmak gibi).
2. **Toplu Yazma:** "Değiştirilenler Listesi" dolunca, hepsi tek seferde topluca diske yazılır. Diski sürekli meşgul etmek yerine tek seferde iş halledilir.

Sanal belleğin en stratejik kararına geldik: **"Bir programa RAM'de kaç koltuk (sayfa) ayıralım?"**

Eğer çok az verirsek, program sürekli "Page Fault" yer ve sürünür. Çok fazla verirsek, RAM dolar ve diğer programlara yer kalmaz.

İşte bu dengeyi sağlamak için kullanılan stratejiler ve efsanevi **"Working Set" (Çalışma Kümesi)** teorisi.

---

## 1. Yerleşik Küme Boyutu (Resident Set Size)

Bir process'e kaç sayfa vereceğiz? İki yaklaşım var:

- **Sabit Tahsis (Fixed Allocation):** Her programa baştan sabit bir sayı verilir (Örn: Herkese 50 sayfa).
    - *Sorun:* Bazı program 10 sayfayla yetinir, diğeri 500 sayfaya ihtiyaç duyar. Adaletli değildir.
- **Değişken Tahsis (Variable Allocation):** Programın ihtiyacına göre zamanla artırılır veya azaltılır.
    - *Sorun:* İşletim sisteminin sürekli programı izlemesi (Monitor) gerekir.

---

## 2. Working Set (Çalışma Kümesi) Teorisi

Denning tarafından ortaya atılan bu teori şunu der: **"Bir programın verimli çalışması için, o an aktif olarak kullandığı tüm sayfaların (Locality) RAM'de olması gerekir."**

Bu aktif sayfa grubuna **Working Set** denir.

- **W(t, Δ):** $t$ anında, son $\Delta$ (Delta) zaman diliminde kullanılan sayfalar kümesi.

- **Kural:** Eğer RAM'de bir programın Working Set'ini sığdıracak kadar yer yoksa, o programı **hiç çalıştırma** (Swap out yap). Çünkü çalıştırırsan Thrashing (sistemi kilitleme) yapar.

  **Grafik Yorumu (Şekil 8.18):** Program çalışırken bazen çok az sayfaya (sakin dönem), bazen çok sayfaya (yeni fonksiyona geçiş dönemi) ihtiyaç duyar. Working Set boyutu sürekli dalgalanır.

---

## 3. PFF (Page Fault Frequency) Algoritması

"Working Set"i tam olarak hesaplamak çok zordur (her saniye hangi sayfalara erişildiğini takip etmek gerekir). Bunun yerine daha pratik bir gösterge kullanılır: **Page Fault Sıklığı.**

İşletim sistemi process'in nabzını (Hata Oranını) ölçer:

1. **Nabız Çok Yüksek (Çok Hata):** "Bu program boğuluyor! Demek ki Working Set'i RAM'e sığmamış."
    - **Müdahale:** Ona RAM'den **yeni yer ver** (Resident Set'i büyüt).
2. **Nabız Çok Düşük (Hiç Hata Yok):** "Bu program çok rahat, yayılmış."
    - **Müdahale:** Elindeki **fazla sayfaları al** (Resident Set'i küçült) ve ihtiyacı olana ver.

Bu yöntemle sistem kendi kendini dengeler (Self-tuning).

---

## 4. Temizleme Politikası (Cleaning Policy)

Değiştirilmiş (Dirty) sayfaları diske ne zaman yazalım?

1. **Talep Üzerine (Demand Cleaning):** Sayfa RAM'den atılacağı zaman yaz.
    - *Sorun:* Sayfa atılırken diske yazmayı beklemek süreci yavaşlatır.
2. **Önceden Temizleme (Precleaning):** İşletim sistemi boş kaldıkça (Idle), değiştirilmiş sayfaları diske yazar ve bayraklarını temizler.
    - *Avantaj:* Sayfa atılacağı zaman zaten yedeği diskte olduğu için anında silinebilir. Hızlıdır.

Sanal belleğin en stratejik kararına geldik: **"Bir programa RAM'de kaç koltuk (sayfa) ayıralım?"**

Eğer çok az verirsek, program sürekli "Page Fault" yer ve sürünür. Çok fazla verirsek, RAM dolar ve diğer programlara yer kalmaz.

İşte bu dengeyi sağlamak için kullanılan stratejiler ve efsanevi **"Working Set" (Çalışma Kümesi)** teorisi.

---

## 1. Yerleşik Küme Boyutu (Resident Set Size)

Bir process'e kaç sayfa vereceğiz? İki yaklaşım var:

- **Sabit Tahsis (Fixed Allocation):** Her programa baştan sabit bir sayı verilir (Örn: Herkese 50 sayfa).
    - *Sorun:* Bazı program 10 sayfayla yetinir, diğeri 500 sayfaya ihtiyaç duyar. Adaletli değildir.
- **Değişken Tahsis (Variable Allocation):** Programın ihtiyacına göre zamanla artırılır veya azaltılır.
    - *Sorun:* İşletim sisteminin sürekli programı izlemesi (Monitor) gerekir.

---

## 2. Working Set (Çalışma Kümesi) Teorisi

Denning tarafından ortaya atılan bu teori şunu der: **"Bir programın verimli çalışması için, o an aktif olarak kullandığı tüm sayfaların (Locality) RAM'de olması gerekir."**

Bu aktif sayfa grubuna **Working Set** denir.

- **W(t, Δ):** $t$ anında, son $\Delta$ (Delta) zaman diliminde kullanılan sayfalar kümesi.
- **Kural:** Eğer RAM'de bir programın Working Set'ini sığdıracak kadar yer yoksa, o programı **hiç çalıştırma** (Swap out yap). Çünkü çalıştırırsan Thrashing (sistemi kilitleme) yapar.

**Grafik Yorumu (Şekil 8.18):** Program çalışırken bazen çok az sayfaya (sakin dönem), bazen çok sayfaya (yeni fonksiyona geçiş dönemi) ihtiyaç duyar. Working Set boyutu sürekli dalgalanır.

---

## 3. PFF (Page Fault Frequency) Algoritması

"Working Set"i tam olarak hesaplamak çok zordur (her saniye hangi sayfalara erişildiğini takip etmek gerekir). Bunun yerine daha pratik bir gösterge kullanılır: **Page Fault Sıklığı.**

İşletim sistemi process'in nabzını (Hata Oranını) ölçer:

1. **Nabız Çok Yüksek (Çok Hata):** "Bu program boğuluyor! Demek ki Working Set'i RAM'e sığmamış."
    - **Müdahale:** Ona RAM'den **yeni yer ver** (Resident Set'i büyüt).
2. **Nabız Çok Düşük (Hiç Hata Yok):** "Bu program çok rahat, yayılmış."
    - **Müdahale:** Elindeki **fazla sayfaları al** (Resident Set'i küçült) ve ihtiyacı olana ver.

Bu yöntemle sistem kendi kendini dengeler (Self-tuning).

---

## 4. Temizleme Politikası (Cleaning Policy)

Değiştirilmiş (Dirty) sayfaları diske ne zaman yazalım?

1. **Talep Üzerine (Demand Cleaning):** Sayfa RAM'den atılacağı zaman yaz.
    - *Sorun:* Sayfa atılırken diske yazmayı beklemek süreci yavaşlatır.
2. **Önceden Temizleme (Precleaning):** İşletim sistemi boş kaldıkça (Idle), değiştirilmiş sayfaları diske yazar ve bayraklarını temizler.
    - *Avantaj:* Sayfa atılacağı zaman zaten yedeği diskte olduğu için anında silinebilir. Hızlıdır.

Şimdi **PFF (Page Fault Frequency)** algoritmasının eksikliklerini kapatan daha gelişmiş bir yöntemden ve sistemin genel yükünü nasıl dengeleyeceğimizden (Load Control) bahsedeceğim. Ayrıca Android'in neden farklı bir yol izlediğine değineceğiz.

İşte metnin özeti:

## 1. VSWS (Variable-Interval Sampled Working Set) Politikası

PFF algoritması "Sayfa hatası sıklığına bak, ona göre yer ayır" diyordu ama bir kusuru vardı: Program bir modülden diğerine geçerken (Transition) ani hata artışlarına tepki vermekte yavaş kalıyordu.

**VSWS**, PFF'nin daha hassas ve kontrollü versiyonudur. Üç parametre kullanır:

1. **M (Minimum Süre):** "Çok sık kontrol edip sistemi yorma."
2. **L (Maksimum Süre):** "Çok da boşlama, en geç L süresinde bir kontrol et."
3. **Q (Hata Limiti):** "Eğer Q kadar hata olursa, süreyi bekleme, hemen kontrol et."

**Mantık:**

- Eğer hata sayısı azsa, L süresi dolana kadar bekle (Sistemi yorma).
- Eğer hata sayısı aniden artarsa (Q'yu geçerse), hemen müdahale et ve gereksiz sayfaları temizle.
- Bu yöntem, ani geçişleri (bir fonksiyon bitti, diğeri başladı) çok daha iyi yakalar.

---

## 2. Temizleme Politikası (Cleaning Policy)

Değiştirilmiş (Dirty) sayfaları diske ne zaman yazalım?

- **Talep Üzerine (Demand Cleaning):** Sayfa RAM'den atılacağı zaman yaz.
    - *Sorun:* Sayfa atılırken diske yazmayı beklemek süreci yavaşlatır.
- **Önceden Temizleme (Precleaning):** Fırsat buldukça yaz.

    - *Sorun:* Ya sayfayı şimdi yazdım ama program 1 saniye sonra tekrar değiştirdiyse? Boşuna yazmış oldum (I/O israfı).

**En İyi Çözüm: Sayfa Tamponlama (Page Buffering)** Sayfaları hemen silmeyip "Değiştirilenler Listesi"nde bekletiriz. Liste dolunca **toplu halde** yazarız. Hem hızlıdır hem de gereksiz yazmaları önler.

---

## 3. Yük Kontrolü (Load Control) - Thrashing'i Önlemek

İşletim sisteminin en kritik kararı: **"Aynı anda kaç program (Multiprogramming Level) çalışsın?"**

- **Az Program:** İşlemci boş kalır (Idle). Verimsiz.
- **Çok Program:** Herkese az yer düşer. Sürekli sayfa hatası (Page Fault) oluşur. Sistem kilitlenir (**Thrashing**).

**Denge Noktası Neresi? (L=S Kriteri)** Denning'in teorisine göre ideal nokta şudur: **Sayfa Hatası Arasındaki Süre (L) = Bir Hatayı Çözme Süresi (S).** Yani işlemci, iş yapmaya harcadığı vakit kadar, hata düzeltmeye vakit harcıyorsa sınırdayız demektir. Daha fazla program açma!

---

## VAKA ANALİZİ: Android Bellek Yönetimi (Neden Swap Yok?)

Windows ve Linux, RAM dolunca verileri diske (Swap/Page File) yazar. **Android ise bunu yapmaz.** (En azından klasik anlamda).

**Neden?** Mobil cihazlarda disk (Flash Memory) sınırlı ömre sahiptir ve yazma işlemi pili tüketir.

**Android Ne Yapar? (Low Memory Killer)** RAM dolduğunda Android diske yazmakla uğraşmaz. Arka planda duran en önemsiz uygulamayı **direkt öldürür (Kill).**

- *Uygulama:* "Hey, beni kapatma, durumumu kaydet!" der.
- *Android:* "Tamam, durumunu (küçük bir veri paketini) kaydettim. Şimdi git. Kullanıcı seni tekrar açarsa o durumdan yeniden başlatırım."

Bu yüzden Android'de uygulamalar "Dondu" demez, direkt kapanır ve açtığında kaldığı yerden devam eder gibi görünür (aslında yeniden yüklenmiştir).

**Bölüm 8**'in sonlarına yaklaşırken, **İşletim Sistemlerinin Gerçek Dünyadaki Uygulamaları**na, yani **UNIX** ve **Solaris**'in bu işleri nasıl hallettiğine odaklanacağız.

Bu sistemler, özellikle sunucu tarafında çok güçlü oldukları için bellek yönetimi konusunda oldukça yaratıcı ("Two-Handed Clock" gibi) çözümler geliştirmişlerdir.

Senin için metni teknik terimleri koruyarak ve görselleştirerek özetledim.

---

## 1. Process Askıya Alma (Process Suspension)

Eğer sistem **Thrashing**'e girdiyse (disk ışığı sürekli yanıyorsa), yükü azaltmak için bazı process'leri geçici olarak durdurup (Suspend) diskteki takas alanına (Swap) göndermemiz gerekir.

**Kimi Kurban Edeceğiz?** (6 Aday):

1. **En Düşük Öncelikli (Lowest-Priority):** En kolay karar. Önemsiz olanı at.
2. **Hata Yapan (Faulting Process):** Zaten şu an "Page Fault" yedi ve bizi yoruyor. Onu atarsak, disk I/O yükünden hemen kurtuluruz.
3. **En Son Başlayan (Last Activated):** "Çömez" process. Henüz RAM'de yerleşik bir düzeni (Working Set) yoktur, atması ve geri alması kolaydır.
4. **En Küçük Olan (Smallest Resident Set):** Geri yüklemesi hızlıdır ama "uslu duran küçük programları" cezalandırmış oluruz.
5. **En Büyük Olan (Largest Process):** "Fil" process. Bunu atarsak RAM'de devasa bir alan açılır, diğer herkes rahatlar.
6. **En Çok Süresi Kalan (Largest Remaining Window):** İşi daha çok uzun sürecek olanı at, kısa sürecek olanlar hemen bitip çıksın.

---

## 2. UNIX ve Solaris Bellek Yönetimi

Bu sistemler belleği ikiye ayırır:

1. **Paging System:** Kullanıcı programları (User Space) için.
2. **Kernel Memory Allocator:** İşletim sistemi çekirdeği (Kernel Space) için.

### A. Paging System: İki İbreli Saat (Two-Handed Clock) Algoritması

Standart "Saat Algoritması" (tek ibreli) büyük RAM'lerde yavaş kalabilir. UNIX buna çok zekice bir ekleme yapmıştır: **İkinci bir ibre.**

**Mekanizma (Şekil 8.21):** Bir saat kadranı düşünün, üzerinde iki ibre var ve aralarında belirli bir açı (Handspread) var.

1. **Ön İbre (Front Hand):** Temizlikçidir. Geçtiği her sayfanın "Kullanım Bitini" (Reference Bit) **sıfırlar (0 yapar).**

2. **Arka İbre (Back Hand):** Müfettiştir. Ön ibreden biraz sonra gelir ve kontrol eder.
    - Eğer bit hala **0** ise: "Demek ki iki ibre arasındaki sürede kimse bu sayfayı kullanmamış. Atın bunu!" der.
    - Eğer bit **1** olmuşsa: "Biri bu arada kullanmış, affettim" der (dokunmaz).

**Parametreler:**

- **Scanrate (Hız):** RAM doldukça ibreler daha hızlı dönmeye başlar (daha agresif temizlik).
- **Handspread (Aralık):** İki ibre arasındaki mesafe, sayfalara tanınan "ikinci şans" süresidir.

**B. Kernel Memory Allocator: Lazy Buddy (Tembel Ahbap) Sistemi**

Çekirdek (Kernel), kullanıcı programları gibi 4KB'lık sayfalara değil, çok daha küçük ve değişken boyutlu (örn: dosya tablosu için 100 byte) alanlara ihtiyaç duyar.

Normal "Buddy System" (Böl/Birleştir) burada biraz hantal kalabilir. Bir bloğu birleştirirsin, milisaniye sonra tekrar bölmen gerekir.

**Lazy Buddy Çözümü:**

- **Mantık:** Bir bellek bloğu serbest kaldığında, onu **hemen birleştirme (Coalesce).**
- **Bekle:** Belki hemen tekrar aynı boyutta istek gelir.
- **Kural:** Sadece elimizde çok fazla "boşta bekleyen küçük parça" (Locally Free) biriktiyse birleştirme yap. Bu sayede işlemci yükü (CPU Overhead) azalır.

Harika, şimdi gerçek dünyanın en büyük oyuncusuna, **Linux**'a geldik. Linux, her yerde (kol saatlerinden süper bilgisayarlara kadar) çalıştığı için bellek yönetimi biraz "İsviçre Çakısı" gibidir; her duruma uyum sağlamak zorundadır.

Linux bellek yönetimini üç ana başlıkta inceleyelim: Adresleme, Sayfa Değiştirme ve Çekirdek Belleği.

---

## 1. Linux Sanal Bellek Adreslemesi (3 Seviyeli Yapı)

Linux'un en büyük derdi **Platform Bağımsızlığıdır.**

- **Alpha İşlemciler (64-bit):** Adresler çok büyük olduğu için 3 seviyeli sayfa tablosuna ihtiyaç duyar.
- **Intel x86 (32-bit):** Donanım sadece 2 seviyeyi destekler.

**Çözüm: "Mış Gibi Yapmak"** Linux kodunda **3 Seviyeli Paging** yapısı vardır:

1. **Page Directory (Kök Dizin)**

2. **Page Middle Directory (Orta Dizin)**
3. **Page Table (Sayfa Tablosu)**

Eğer Linux'u Intel x86 üzerinde derlersen, derleyici "Orta Dizin"i otomatik olarak **1 elemanlı (etkisiz)** hale getirir ve yok sayar. Böylece aynı kod hem 2 seviyeli hem 3 seviyeli donanımda çalışır.

> **Hugepages:** Linux genelde 4KB sayfa kullanır ama performans için (örneğin Veritabanları veya Ağ paketleri için) **2MB'lık Hugepages** desteği de vardır. Bu sayede TLB (Önbellek) daha az şişer.

---

## 2. Sayfa Değiştirme (Replacement): "Split LRU" Devrimi

Linux 2.6.28 sürümüne kadar eski usül "Saat Algoritması" (Clock) kullanıyordu. Ama RAM miktarları GB'lara çıkınca o "ibrenin" dönüp bütün sayfaları taraması işlemciyi yemeye başladı.

Linux mühendisleri bunu çöpe atıp **"Split LRU" (Bölünmüş LRU)** sistemine geçti.

**Mantık: İki Liste (Active vs Inactive)** RAM'deki sayfaları iki listeye ayırdılar:

1. **Active List (VIP Alanı):** Sık kullanılanlar. Buradan kimse atılmaz.
2. **Inactive List (Bekleme Odası):** Az kullanılanlar. Kurbanlar buradan seçilir.

**Nasıl Çalışır? (Şekil 8.24)**

- Bir sayfa ilk yüklendiğinde **Inactive** listesine girer.
- Eğer o sayfaya **ikinci kez** erişilirse, **Active** listesine (VIP) terfi eder.
- Eğer Active listesindeki bir sayfaya uzun süre erişilmezse (Timeout), tekrar **Inactive** listesine düşer.
- RAM dolunca Linux sadece **Inactive** listesine bakar ve oradaki en eskiyi atar.

> **Fark:** Eski sistemde tüm RAM taranıyordu. Şimdi sadece "Düşme Hattı"ndaki (Inactive) sayfalar taranıyor. Bu çok daha hızlıdır.

---

## 3. Çekirdek Bellek Tahsisi: Slab Allocator

Kullanıcı programları için 4KB'lık sayfalar (Buddy System ile) harikadır. Ama İşletim Sistemi Çekirdeği (Kernel) çok küçük nesneler kullanır (Dosya tanımlayıcıları, Sempaforlar vb. bazen 32 byte, bazen 100 byte).

Eğer 32 byte'lık bir veri için 4096 byte'lık (4KB) koca bir sayfa verirsen, belleğin %99'u israf olur (**Internal Fragmentation**).

**Çözüm: SLAB Allocator** Buddy System'den koca bir sayfa (Pasta) alır ve onu küçük dilimlere (Slab) böler.

- **SLAB:** En performanslısı. Önbellek dostudur.
- **SLUB:** Daha basiti ve modern Linux'ta varsayılan olanı.
- **SLOB:** En az yer kaplayanı (Gömülü sistemler ve eski telefonlar için).

## 8.5 WINDOWS BELLEK YÖNETİMİ

Windows, donanıma göre esnek davranır (Intel'de 4KB sayfa, Itanium'da 8KB sayfa kullanır). Ama asıl olay yazılımdadır.

**1. Sanal Adres Haritası (Virtual Address Map)**

32-bit Windows'ta her process'in teorik olarak **4 GB** sanal belleği vardır.

- **Alt 2 GB (User Space):** Senin programın burayı kullanır. (0x00000000 - 0x7FFFFFFF).
- **Üst 2 GB (System Space):** İşletim sistemi çekirdeği (Kernel) buradadır. Tüm process'ler bu alanı ortak görür ama erişemez.

> **.NET Notu:** `OutOfMemoryException` hatası aldığında genelde RAM bitmemiştir, senin o 2 GB'lık User Space alanında yer kalmamıştır veya parçalanmıştır (Fragmentation).

**2. Sayfa Durumları (States)**

Windows'ta belleğin 3 hali vardır. Bunu bilmek önemlidir:

1. **Available (Müsait):** Boş arazi. Kullanılmıyor.
2. **Reserved (Rezerve):** "Burayı bana ayırın, inşaat yapacağım ama henüz malzemeyi getirmedim."
    - Diğer process'ler burayı alamaz ama fiziksel RAM harcanmaz. Sadece adres aralığı tutulur.
3. **Committed (Tahsisli):** "İnşaat başladı."
    - Artık fiziksel RAM'de veya diskte (Pagefile) karşılığı vardır.

**3. Yeni Oyuncu: `Swapfile.sys` vs `Pagefile.sys`**

Eskiden sadece `pagefile.sys` vardı. Windows 8 ve Metro (Store) uygulamalarıyla hayatımıza `swapfile.sys` girdi.

- **Pagefile.sys:** Klasik masaüstü programları (.NET, C++, Oyunlar) burayı kullanır. Sayfa sayfa (4KB) taşınır.
- **Swapfile.sys:** Modern (Store/UWP) uygulamalar kullanır.
    - Bu uygulamalar telefondaki gibi çalışır: Arka plana atılınca **TÜM PROCESS** paketlenip bu dosyaya atılır.

- Amaç: Hızlı açılıp kapanma ve RAM'i anında boşaltma.

---

## 8.6 ANDROID BELLEK YÖNETİMİ

Android, Linux çekirdeği üzerine kuruludur ama bellek yönetimi çok farklıdır. En büyük fark: **Swap Yoktur.**

**1. Neden Swap Yok?**

Bilgisayarda RAM dolunca veriyi diske yazarız. Ama telefonlarda disk **Flash Memory**'dir (SD Kart gibi).

- Flash belleğin ömrü yazma sayısına bağlıdır. Sürekli swap yaparsan telefonun hafızası 1 yılda ölür.
- Ayrıca Flash bellek RAM'e göre çok yavaştır ve pili sömürür.

**2. Çözüm: Low Memory Killer (Az Bellek Katili)**

RAM dolduğunda Android "Diske yazayım" demez. Elinde tırpanla bekleyen bir **"Low Memory Killer"** vardır.

- RAM kritik seviyeye inerse, arka plandaki en önemsiz uygulamayı bulur.
- **Kafasını Keser (Kill):** Uygulamayı tamamen kapatır.
- Uygulama kapanmadan önce son durumunu (küçük bir not kağıdı gibi) kaydeder. Kullanıcı tekrar açtığında o nottan devam eder.

**3. ION ve ASHMem**

- **ASHMem (Anonymous Shared Memory):** İki uygulamanın belleği güvenli şekilde paylaşmasını sağlar (Linux'taki Shared Memory'nin Androidcesi).
- **ION:** Kamera, Grafik Kartı (GPU) gibi donanımların kendilerine has bellek havuzlarını yönetir.

---

## 8.7 BÖLÜM ÖZETİ (Sanal Bellek)

Tebrikler! İşletim sistemlerinin en karmaşık bölümü olan **Sanal Bellek** bitti.

**Neler Öğrendik?**

1. **Sanal Bellek:** Diski RAM gibi kullanma sanatıdır. Process'in tamamının yüklenmesine gerek yoktur.
2. **Donanım Desteği:**

    - **Paging:** Belleği eşit parçalara bölme.
    - **TLB:** Adres çevirisini hızlandıran süper önbellek.
3. **İşletim Sistemi Politikaları:**
    - **Fetch:** Ne zaman getireyim? (Demand Paging).
    - **Replacement:** Kimi atayım? (LRU, Clock Algoritması).
    - **Load Control:** Kaç kişi çalışsın? (Thrashing olmasın diye L=S kuralı).
4. **Gerçek Sistemler:**
    - Linux: Split-LRU ve Slab Allocator.
    - Windows: Working Set Manager.
    - Android: Swap yok, Kill var.

## GÖZDEN GEÇİRME SORULARI (Kısa Cevaplar)

**8.1. Simple Paging (Basit) ile Virtual Memory Paging farkı nedir?**

- **Simple Paging:** Bir programın çalışması için **TÜM** sayfalarının aynı anda RAM'de olması şarttır.
- **Virtual Memory Paging:** Programın sadece o an kullanılan sayfalarının (Resident Set) RAM'de olması yeterlidir.

8.2. Thrashing nedir?

RAM dolduğunda işletim sisteminin sürekli sayfa değiştirmekten (Swap in/out) iş yapmaya vakit bulamaması ve sistemin kilitlenmesidir.

8.3. "Locality" (Yerellik) ilkesi neden sanal bellek için hayatidir?

Çünkü bu ilke olmasaydı (yani programlar rastgele sayfalara erişseydi), sürekli Thrashing olurdu. Programlar belirli bir süre aynı kümede (Working Set) çalıştığı için sanal bellek verimli çalışır.

**8.4. Sayfa Tablosu Girişinde (PTE) neler bulunur?**

- **Frame Number:** Sayfanın RAM'deki adresi.
- **Present (P) Bit:** RAM'de mi yoksa diskte mi?
- **Modify (M) Bit:** Değişiklik yapıldı mı? (Diske yazılmalı mı?).
- **Reference (R) Bit:** Yakın zamanda kullanıldı mı? (LRU için).

8.5. TLB (Translation Lookaside Buffer) ne işe yarar?

RAM'deki sayfa tablosuna gidip gelmek yavaştır. TLB, son yapılan adres çevirilerini işlemci içinde tutan süper hızlı bir önbellektir.

---

## SEÇİLMİŞ KRİTİK PROBLEM ÇÖZÜMLERİ

**SORU 8.1: Adres Çevirimi (Tam Sınav Sorusu)**
**Verilenler:**

- Sayfa Boyutu: 1,024 byte ($2^{10}$).
- Tablo:
    - Sayfa 0 -> Frame 4
    - Sayfa 1 -> Frame 7
    - Sayfa 2 -> Frame 3 (Valid=0, RAM'de yok!)
    - Sayfa 3 -> Frame 4 (Valid=0, RAM'de yok!)
    - Sayfa 4 -> Frame 2
    - Sayfa 5 -> Frame 1

a. Çeviri Mantığı:

Sanal Adres 1024'e bölünür. Bölüm Sayfa No, Kalan Offset olur.

$$Fiziksel Adres = (Frame No \times 1024) + Offset$$

**b. Adresleri Çevir:**

1. **1,052:**
    - Sayfa No: $1052 / 1024 = 1$ (Kalan 28).
    - Tabloya bak: Sayfa 1 -> Frame 7.
    - Fiziksel: $(7 \times 1024) + 28 = \mathbf{7196}$.
2. **2,221:**
    - Sayfa No: $2221 / 1024 = 2$ (Kalan 173).
    - Tabloya bak: Sayfa 2 -> Valid Bit = 0.
    - Sonuç: **PAGE FAULT** (Sayfa Hatası). RAM'de yok.
3. **5,499:**
    - Sayfa No: $5499 / 1024 = 5$ (Kalan 379).
    - Tabloya bak: Sayfa 5 -> Frame 1.
    - Fiziksel: $(1 \times 1024) + 379 = \mathbf{1403}$.

---

**SORU 8.4: Sayfa Değiştirme Algoritmaları (EFSANE SORU)**
**Senaryo:** Sayfa sırası: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. RAM Kapasitesi: 3 Çerçeve.

**a. FIFO (İlk Giren Çıkar):**

- 7, 0, 1 dolar.
- 2 gelir -> 7 çıkar (En eski). Durum: 2, 0, 1.
- 0 gelir -> Zaten var (Hit).
- 3 gelir -> 0 çıkar (En eski). Durum: 2, 3, 1.
- ... (Bu şekilde devam eder).
- **Toplam Hata:** 10 (veya tabloya göre değişebilir, mantık budur).

**b. LRU (En Az Son Kullanılan):**

- 7, 0, 1 dolar.
- 2 gelir -> 7 çıkar (En az kullanılan). Durum: 2, 0, 1.
- 0 gelir -> Hit. (0 tazelendi).
- 3 gelir -> 1 çıkar (En eski kullanılan). Durum: 2, 0, 3.
- 0 gelir -> Hit.
- 4 gelir -> 2 çıkar (En eski kullanılan). Durum: 4, 0, 3.

**c. Clock (Saat Algoritması):**

- Her sayfanın bir Use Biti vardır.
- 2 geldiğinde ibre 7'ye bakar. Use=1 ise 0 yapar geçer. Use=0 olanı atar.

**d. Optimal (Geleceği Gören):**

- 7, 0, 1 dolar.
- 2 gelir -> Geleceğe bak. 0 hemen lazım, 1 sonra lazım, 7 hiç lazım değil. 7'yi at.

---

**SORU 8.11: EMAT (Effective Memory Access Time) Hesabı**
Bu soru işlemci performansını hesaplamak için sorulur.

**Verilenler:**

- RAM erişim süresi: 200 ns.
- TLB (Önbellek) erişim süresi: 20 ns.
- Hit Rate (TLB'de bulma oranı): %85.

**Mantık:**

- **Hit (TLB'de var):** TLB süresi + Veriyi alma süresi (RAM) = $20 + 200 = 220$ ns.
- **Miss (TLB'de yok):** TLB süresi + Tabloyu okuma (RAM) + Veriyi alma (RAM) = $20 + 200 + 200 = 420$ ns.

Hesap:

$$EMAT = (Hit Oranı \times Hit Süresi) + (Miss Oranı \times Miss Süresi)$$

$$EMAT = (0.85 \times 220) + (0.15 \times 420)$$

$$EMAT = 187 + 63 = \mathbf{250 \text{ ns}}$$

> **Yorum:** TLB olmasaydı her erişim $400$ ns sürerdi. TLB sayesinde ortalama $250$ ns'ye düştü.

---

**SORU 8.18: Adres Formatı**
**Verilenler:**

- Mantıksal Uzay: 32 sayfa, her biri 2 KB.
- Fiziksel Uzay: 1 MB.

**a. Mantıksal Adres Formatı:**

- Sayfa Boyutu = 2 KB = $2^{11}$ bayt. -> **Offset = 11 bit.**
- Sayfa Sayısı = 32 = $2^5$. -> **Sayfa No = 5 bit.**
- **Toplam Mantıksal Adres = 16 bit.**

**b. Sayfa Tablosu Boyutu:**

- 32 sayfa olduğu için tabloda **32 satır** vardır.
- Çerçeve Sayısı = 1 MB / 2 KB = $2^{20} / 2^{11} = 2^9 = 512$ Frame.
- Frame No için **9 bit** gerekir.