

## Dosya Sistem Uygulaması

Bu bölümde sizi **vsfs (Çok basit dosya sistemlemesi(the Very Simple File System))** olarak bilinen basit sistem uygulaması ile tanıştıracacağız.Bu dosya sistemi basitleştirilmiş UNIX in tipik bir versiyonudur ve bu sunucular bize bazı temel disk yapılarını tanıtır ,erişim metodları ve bugün kü dosya sistemlerinde bulabnileceğimiz bunun varyasyonlarıdır.

Bu dosyalama sistemi sade bir yazılımdır ,bizim CPU ve hafıza sanallaştırmamıza rağmen dosya sisteminin daha iyi çalışmasını sağlamak için donanım özellikleri eklemiyoruz (Ancak dikkatinizi çekmek isteriz ki cihazın karakteristik özelliklerinin dosya sistemini iyi çalıştırdığından emin olunmalıdır).Dosya sistemi oluştururken ki büyük esnekliğimizden dolayı çok farklı cihazlar üretildi ,AFS`den (The Andrew file System)[H+88] ZFS`ye kadar (Sun`s Zettabyte File System)[B07].Bütün bu dosyalama sistemleri farklı veri yapılarına sahiptir ve kendi içlerinde bazı şeyleri daha iyi bazı şeyleri daha kötü yaparlar.bizim dosyalama sistemini öğrenmemiz anlık öğrenme ile olur: ilk olarak ,bu bölümde çoğu konsepti tanıtacağımız basit dosyalama sistemi(vsfs) ,ikinci olarak pratikteki farklılıklarını anlamak için yaptığımız gerçek dosyalama sistemi serisidir.

İşin Özünde:BASİT BİR DOSYA SİSTEMİ NASIL UYGULANIR

Basit bir dosya sistemi nasıl oluşturabiliriz? Disk üzerinde hangi yapılara ihtiyacımız vardır?Takip etmek için neye ihtiyacımız var? Nasıl erişilir?

### 40.1 Düşünmenin Yolu

Dosyalama sistemleri hakkında düşündüğümüzde ,genellikle iki farklı parça olarak düşünmeyi öneririz;eğer bu iki parçayıda anlarsanız temel olarak dosyalama sisteminin nasıl çalıştığını muhtemelen anlarsınız.

İlk parça dosyalama sisteminin **veri yapısıdır(data structures)**.Başka bir deyişle ,dosyalama sisteminin kendi verisini ve metadatasını ne tip bir temel disk yapısında çalıştırır?İlk bakacağımız dosyalama sistemi (aşağıdaki vsfs`ler de dahil olmak üzere) çalışan basit yapılar ,dizi bloğu veya diğer nesneler gibi ,oysa daha karmaşık dosyalama sistemleri ,SGI ve XFS gibi ,daha karmaşık ağaç temelli yapı kullanır..

1

### Dosya Sistemlerinin Zihinsel Modelleri

Daha öncede tartıştığımız gibi ,zihinsel modeller bir sistem öğrenmeye çalışırken onu geliştirmeye çalışmaktır.Dosya sistemleri için , sizin zihinsel modelleriniz eninde sonunda şunun gibi cevapları içermelidir:temel disk yapısı dosyalama sisteminin verisini ve metadatasını nasıl depolar? Bir işlem bir dosya açtığında ne olur? Zihinsel modelinizi geliştirmeye çalışırken ,neler olup bittiğini anlamak için sadece dosyalama sisteminin özelliklerini öğrenmek yerine soyut bir anlama geliştirirsiniz.

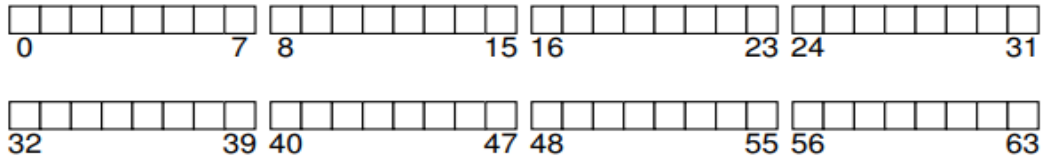
Dosyalama sisteminin ikinci parçası **erişim metodlarıdır(access methods)**. Yapısında open(),read(),write()vb işlemlerin nasıl çağırıldığını nerden biliyor? Spesifik bir sistem çalıştığında hangi yapı okunur?Hangileri yazılır? Bu adımlar hangi verimlilikle çalışır?

Bir dosyalama sisteminin veri yapılarını ve erişim metodlarını anlamak istiyorsak ,gerçekte nasıl çalıştığını anlayabileceğiniz bir zihinsel model geliştirmelisiniz ki bu da sistem zihniyetinin önemli bir parçasıdır.Biz ilk araştırma uygulamamızı yaparken sizde kendi zihinsel modelinizi geliştirmek üzerinde çalışın.

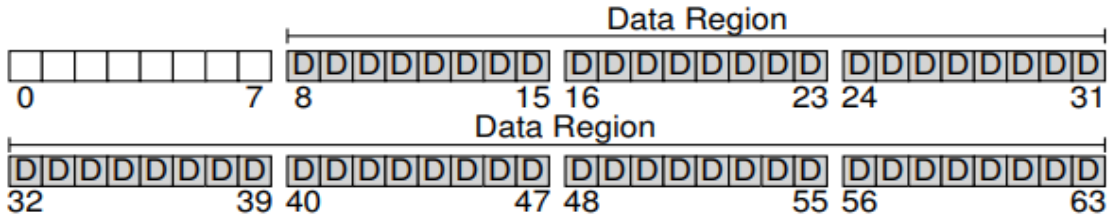
### 40.2 Genel Organizasyon

Şu anda vsfs dosya sisteminin veri yapısının genel disk organizasyonunu geliştireyoruz.yapmamız gereken ilk şey diskimizi **parçalara(blocks)** bölmek;basit dosyalama sistemi sadece bir parça kullanır ve bu da tam olarak burada yapacağımız şeydir.Hadi genellikle kullanılan bir ölçü olan 4KB seçelim.

Aşağıda gördüğünüz disk bölümleri dosya sistemimizi oluşturacağımız yerlerdir: bunlar bir dizi bölümlerdir ve her birinin boyutu 4KB dir.Bu bölümler 0`dan N-1`e kadar adreslenmiştir ,bu bölümlerin büyüklüğü  $N*4KB$ `dir.Elimizde sadece 64 bölüme sahip küçük bir disk olduğunu var sayalım:

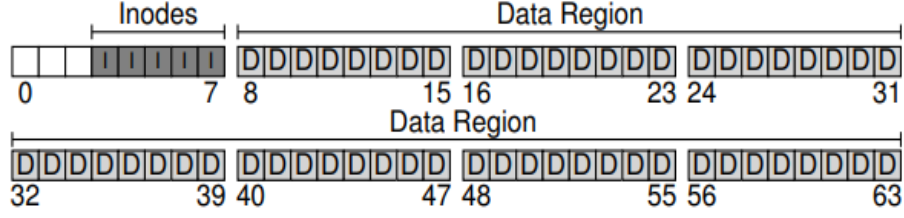


Şimdi bir dosyalama sistemi oluşturmak için bu parçalarda ne depolamaya ihtiyacımız olduğunu düşünelim. Tabiki de ,ilk akla gelen şey kullanıcı bilgisidir. Hatta ,dosyalama sisteminin büyük bir bölümü kullanıcı bilgilerine ayrılır. Şimdi diskin kullanıcı bilgilerini tuttuğumuz bölgesine **bilgi bölgesi(data region)** diyelim ve diskin bir bölümünü bu basitleştirme işlemi için ayıralım diyelim ki bu bölüm 64`ün 56`sı olsun:



Bir önceki bölümde öğrendiğimiz gibi ,dosyalama sistemi her bir dosyada ki her bir bilginin yerini tutmalıdır. Bu bilgi **metadatanın(metadata)** anahtar parçasıdır ve hangi bilgi parçasının dosyayı içerdiğinin ,dosyanın büyüklüğünün ,sahibinin ve erişim haklarının ,erişim ve düzenleme zamanının ve bunun gibi çeşitli bilgilerin bilgisini tutmalıdır. Bu bilgileri depolamak için dosyalama sisteminin genellikle **inode(inode)** adı verilen bir yapıya sahiptir(aşağıda inodlar hakkında daha fazla şey okuyacağız).

Inodları barındırmak için ,diskte boş alanlara ihtiyacımız olacaktır. Diskin bu parçasına **inode table(inode table)** diyelim ki basit olarak diskin üstünde inodları tutsun. Şu anda hayalimizdeki disk aşağıdaki resim gibi olmalıdır ,inodlar için 64 bloğumuzdan 5 ini kullandığımızı düşünelim (diyagramda I ile belirtilmişlerdir

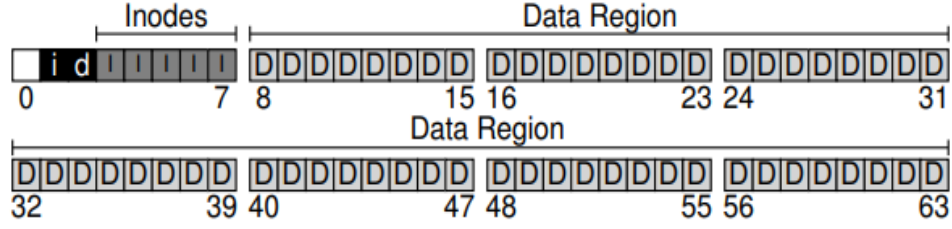


Burada, inode'ların tipik olarak o kadar büyük olmadığını, örneğin 128 veya 256 bayt olduğunu not etmeliyiz. İnode başına 256 bayt varsayarsak, 4 KB'lık bir blok 16 inod tutabilir ve yukarıdaki dosya sistemimiz toplam 80 inod içerir. 64 blokluk küçük bir bölüm üzerine kurulu basit dosya sistemimizde bu sayı, dosya sistemimizde sahip olabileceğimiz maksimum dosya sayısını temsil eder; Ancak, daha büyük bir disk üzerine kurulu aynı dosya sisteminin basitçe ayırabileceğini unutmayın daha büyük bir inode tablosu ve böylece daha fazla dosya barındırır.

Dosya sistemimizde şu ana kadar veri blokları (D) ve inode'lar (i) var, ancak hala birkaç şey eksik. Tahmin edebileceğiniz gibi hala gerekli olan birincil bileşenlerden biri, inode'ların veya veri bloklarının boş veya ayrılmış olup olmadığını izlemenin bir yoludur. Bu nedenle, bu tür **tahsis yapıları(allocation structures)** bir gerekliliktir herhangi bir dosya sistemindeki öge.

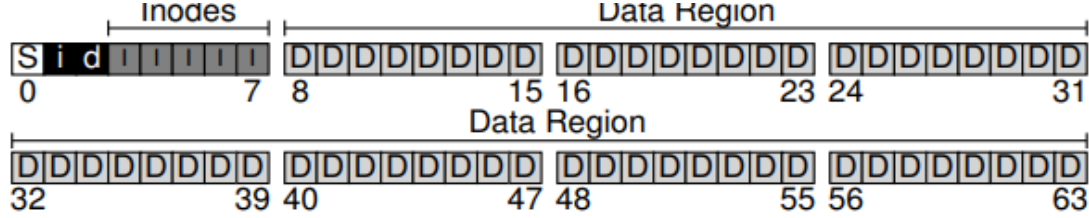
Elbette birçok tahsis izleme yöntemi mümkündür. Örneğin, **ilk boş(free list)** bloğa işaret eden, ardından bir sonraki boş bloğa işaret eden vb. Boş bir liste kullanabiliriz. Bunun yerine, biri veri bölgesi (**veri bitmap(the data bitmap)**) ve biri inode tablosu (**inode bitmap(the inode bitmap)**) için **eşlem(bitmap)** olarak bilinen basit ve popüler bir yapı seçiyoruz.

Bir bitmap basit bir yapıdır: her bit, karşılık gelen nesnenin / bloğun boş (0) veya kullanımda (1) olup olmadığını belirtmek için kullanılır. Ve böylece bir inode bitmap (i) ve bir veri bitmap (d) ile yeni disk içi düzenimiz:



Bu bitmapler için 4 KB'lık bir bloğun tamamını kullanmanın biraz fazla olduğunu fark edebilirsiniz; Böyle bir bitmap, 32 K nesnenin tahsis edilip edilmediğini izleyebilir ve yine de yalnızca 80 inode'umuz ve 56 veri bloğumuz var. Ancak, basitlik için bu bitmaplerin her biri için 4 KB'lık bir bloğun tamamını kullanıyoruz.

Dikkatli okuyucu (yani hala uyanık olan okuyucu), çok basit dosya sistemimizin disk üzerindeki yapısının tasarımında bir blok kalmış olabilir. Bunu, aşağıdaki diyagramda bir S ile gösterilen **süper blok(superblock)** için ayırıyoruz. Süper blok hakkında bilgi içerir bu özel dosya sistemi,örneğin kaç tane inode ve veri blokları, dosya sisteminde (bu örnekte sırasıyla 80 ve 56), burada inode tablosu (blok 3) vb. Başlar. Muhtemelen dosya sistemi türünü tanımlamak için bir tür sihirli sayı da içerecektir (bu durumda, vsfs).



Bu nedenle, bir dosya sistemi kurarken, işletim sistemi önce çeşitli parametreleri başlatmak için süper bloğu okuyacak ve ardından birimi dosya sistemi ağacına ekleyecektir. Birim içindeki dosyalara erişildiğinde, sistem böylece gerekli diskte tam olarak nereye bakacağını bilecektir yapılar.

#### 40.3 Dosya Organizasyonu: Inode

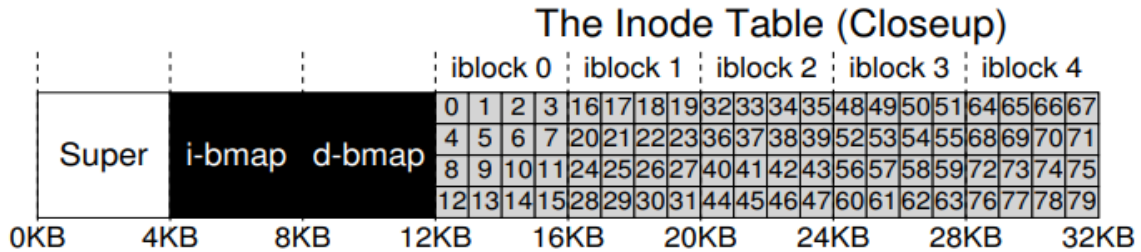
Bir dosya sisteminin en önemli disk üzerindeki yapılarından biri **inode(inode)**; hemen hemen tüm dosya sistemleri buna benzer bir yapıya sahiptir. Inode adı ,unix'te [RT74] ve muhtemelen daha önceki sistemlerde kendisine verilen geçmiş ad olan **indeks düğümünün(index node)** kısaltmasıdır, çünkü bu düğümler bir dizide orijinal olarak düzenlenmiştir ve dizi belirli bir inode'a

erişirken dizine eklenmiştir.

### VERİ YAPISI -INODE

**Kod(the inode)**, uzunluğu, izinleri ve kurucu bloklarının konumu gibi belirli bir dosyanın meta verilerini tutan yapıyı yazmak için birçok dosya sisteminde kullanılan genel addır. Ad, en azından UNIX'E kadar uzanır (ve muhtemelen daha önceki sistemlerde değilse Multics'e daha da geri döner); kısaltması **dizin düğümü(index node)**, çünkü inode numarası, bu sayının inode'unu bulmak için bir dizi disk içi inoda indekslemek için kullanılır. Göreceğimiz gibi, inode'un tasarımı dosya sistemi tasarımının önemli bir parçasıdır. Çoğu modern sistem, izledikleri her dosya için böyle bir yapıya sahiptir, ancak belki de onlara farklı şeyler (dnode'lar, fnode'lar vb.) Derler.).

Her inode örtülü olarak bir sayı ile anılır (**i-numarası(i-number)**) olarak adlandırılır),daha önce dosyanın **alt düzey adını(low-level name)** vermiştik. Vsfs'de (ve diğer basit dosya sistemleri), bir i numarası verildiğinde, diskte karşılık gelen inodun nerede olduğunu doğrudan hesaplayabilmelisiniz. Örneğin, vsf'lerin inode tablosunu yukarıdaki gibi alın: 20KB boyutunda (beş 4KB blok) ve böylece 80 inoddan oluşur (her inodun 256 bayt olduğu varsayılarak); inode bölgesinin 12KB'DE başladığını varsayarsak (yani, süper blok 0kb'de başlar, inode bitmap 4KB adresindedir, veri bit eşlemi 8kb'dir ve bu nedenle inode tablosu hemen sonra gelir). Vsfs'de, dosya sistemi bölümünün başlangıcı için aşağıdaki düzene sahibiz (yakın çekim görünümünde):



32 Numaralı inodu okumak için, dosya sistemi önce off-set'i inode bölgesine (32 · sizeof (inode) veya 8192) hesaplar, diskteki inode tablosunun başlangıç adresine ekler (inodeStartAddr = 12KB) ve böylece istenen bayt adresinin doğru bayt adresine ulaşır düğüm

bloğu: 20KB. Disklerin bayt adreslenebilir olmadığını, bunun yerine genellikle 512 bayt olmak üzere çok sayıda adreslenebilir sektörden oluştuğunu hatırlayın. Bu nedenle, inode 32'yi içeren inode bloğunu almak için dosya sistemi, istenen inode bloğunu almak için  $20 \times 1024/512$  veya 40 saniyeye okuma yapar. Daha genel olarak, inode bloğunun sektör adres sektörü aşağıdaki gibi hesaplanabilir:

```
blk    = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Her bir inodun içinde, bir dosya hakkında ihtiyacınız olan hemen hemen tüm bilgiler bulunur: türü (örneğin, normal dosya, dizin vb.), boyutu, kendisine tahsis edilen blok sayısı, koruma bilgileri (dosyanın kime ait olduğu gibi

Boyut	İsim	Bu inod alanı ne için?
2	mode	bu dosya okunabilir / yazılabilir / yürütülebilir mi?
2	uid	bu dosyanın sahibi kim?
4	size	bu dosyada kaç bayt var?
4	time	bu dosyaya en son ne zaman erişildi?
4	ctime	bu dosyaya en son ne zaman erişildi?
4	mtime	bu dosya en son ne zaman değiştirildi?
4	dtime	bu inode saat kaçta silindi?
2	gid	bu dosya hangi gruba ait?
2	links count	bu dosyaya kaç tane sabit bağlantı var?
4	blocks	bu dosyaya kaç tane sabit bağlantı var?
4	flags	ext2 bu inodu nasıl kullanmalı?
4	osd1	işletim sistemine bağlı bir alan
60	block	bir dizi disk işaretçisi (toplam 15)
4	generation	dosya sürümü (NFS tarafından kullanılır)
4	file acl	mod bitlerinin ötesinde yeni bir izin modeli
4	dir acl	erişim kontrol listeleri olarak adlandırılan

Şekil 40.1: Basitleştirilmiş Ext2 İnode

kimlerin erişebileceği gibi), dosyanın ne zaman oluşturulduğu, değiştirildiği veya en son erişildiği dahil olmak üzere bazı zaman bilgilerinin yanı sıra veri bloklarının diskte nerede bulunduğuna dair bilgiler (örneğin, bir tür işaretçi). Bir dosyayla ilgili tüm bu bilgilere **meta veri(metadata)** olarak atıfta bulunuruz; Aslında, dosya sistemi içinde saf kullanıcı verisi olmayan herhangi bir bilgiye genellikle bu şekilde atıfta bulunulur. Ext2 [P09] 'dan bir örnek inode Şekil 40.11'de gösterilmiştir.

İnode'un tasarımındaki en önemli kararlardan biri, nasıl veri bloklarının nerede olduğunu ifade eder. Basit bir yaklaşımdır inode'un içinde bir veya daha fazla **doğrudan işaretçi(direct pointer)**

(disk adresi) bulundurun; Her işaretçi, dosyaya ait bir disk bloğunu ifade eder. Böyle bir yaklaşım sınırlıdır: örneğin, gerçekten büyük bir dosyaya sahip olmak istiyorsanız (örneğin, inode'daki doğrudan işaretçi sayısı ile çarpılan blok boyutundan daha büyük), şansınız yaver gitmez.

### Çok Düzeyli Dizin

Daha büyük dosyaları desteklemek için, dosya sistemi tasarımcıları inode'lar içinde farklı yapılar tanıtmak zorunda kaldılar. Yaygın bir fikir, **dolaylı işaretçi(indirect pointer)** olarak bilinen özel bir işaretçiye sahip olmaktır. Kullanıcı verilerini içeren bir bloğa işaret etmek yerine, her biri kullanıcı verilerine işaret eden daha fazla işaretçi içeren bir bloğa işaret eder. Bu nedenle, bir inode sabit sayıda doğrudan işaretçiye (örneğin, 12) ve tek bir dolaylı işaretçiye sahip olabilir. Bir dosya yeterince büyürse, dolaylı bir blok tahsis edilir (diskin veri bloğu bölgesinden) ve dolaylı bir işaretçi için inode'un yuvası onu gösterecek şekilde ayarlanır. 4 KB'lık bloklar ve 4 baytlık disk adresleri varsayarsak, bu başka bir 1024 işaretçi ekler; Dosya büyüyebilir  $(12 + 1024) \cdot 4K$  veya 4144KB.

### İPUCU: KAPSAM TABANLI YAKLAŞIMLARI DÜŞÜNÜN

İşaretçiler yerine **kapsamları(extents)** kullanmak farklı bir yaklaşımdır. Kapsam, yalnızca bir disk işaretçisi artı bir uzunluktur (bloklar halinde); Bu nedenle, bir dosyanın her bloğu için bir işaretçi gerektirmek yerine, tek gereken bir dosyanın disk üzerindeki konumunu belirtmek için bir işaretçi ve uzunluktur. Bir dosya tahsis edilirken bitişik bir disk alanı bulmakta sorun yaşayabileceğinden, yalnızca tek bir kapsam sınırlayıcıdır. Bu nedenle, kapsam tabanlı dosya sistemleri genellikle birden fazla kapsam sağlar, böylece dosya ayırma sırasında dosya sistemine daha fazla özgürlük verir.

İki yaklaşımı karşılaştırırken, işaretçi tabanlı yaklaşımlar en esnektir, ancak dosya başına büyük miktarda meta veri kullanır (özellikle büyük dosyalar için). Kapsam tabanlı yaklaşımlar daha az esnektir ancak daha kompakttır; Kısmen, diskte yeterli boş alan olduğunda ve dosyalar bitişik olarak yerleştirilebildiğinde iyi çalışırlar (bu, neredeyse her dosya ayırma ilkesinin amacıdır).

Şaşırtıcı olmayan bir şekilde, böyle bir yaklaşımda, daha büyük dosyaları bile desteklemek isteyebilirsiniz. Bunu yapmak için, inode'a başka bir işaretçi eklemeniz yeterlidir: **çift dolaylı işaretçi(double**



**indirect pointer**). Bu işaretçi, işaretçiler içeren bir bloğu ifade eder her biri veri bloklarına işaretçi içeren dolaylı bloklara. Bu nedenle, bir dou ble dolaylı blok, ek 1024 · 1024 veya 1 milyon 4KB blok içeren, yani boyutu 4 gb'ın üzerinde olan dosyaları destekleyen dosyalar büyütme olanağı ekler. Yine de daha fazlasını isteyebilirsiniz ve bahse gireriz bunun nereye gittiğini biliyorsunuzdur: **üçlü dolaylı işaretçi(triple indirect pointer)**.

2

Genel olarak, bu dengesiz ağaç, dosya bloklarına işaret etmek için **çok düzeyli dizin(multi-level index)** erişim noktası olarak adlandırılır. Hem tek hem de çift dolaylı bloğun yanı sıra on iki doğrudan işaretçi içeren bir örneği inceleyelim. 4 KB'lık bir blok boyutu ve 4 baytlık işaretçiler toplandığında, bu yapı 4 gb'ın biraz üzerinde bir dosyayı (ör.,  $(12 + 1024 + 10242) \times 4$  KB).Üçlü dolaylı bir bloğun eklenmesiyle ne kadar büyük bir dosyanın işlenebileceğini anlayabiliyor musunuz? (ipucu: oldukça büyük)

Birçok dosya sistemi, yaygın olarak kullanılanlar da dahil olmak üzere çok düzeyli bir dizin kullanır.Linux ext2 [P09] ve ext3, Netapp'ın WAFL gibi dosya sistemlerinin yanı sıra orijinal UNIX dosya sistemi. SGI XFS dahil olmak üzere diğer dosya sistemleri ve Linux ext4, basit işaretçiler yerine **kapsamları(extents)** kullanın; için önceki kenara bakın kapsam tabanlı şemaların nasıl çalıştığına ilişkin ayrıntılar (bunlar aşağıdaki bölümlerdeki bölümlere benzer):sanal bellek tartışması).

Merak ediyor olabilirsiniz: neden böyle dengesiz bir ağaç kullanıyorsunuz? Nedenfarklı bir yaklaşım değil mi? Anlaşıldığı üzere, birçok araştırmacı çalışılan dosya sistemleri ve bunların nasıl kullanıldığı ve neredeyse her seferinde on yıllar boyunca geçerli olan belirli "gerçekleri" bulun. Böyle bir bulgu şudur çoğu dosyanın küçük olduğunu. Bu dengesiz tasarım böyle bir gerçeği yansıtır; eğer çoğu dosya gerçekten küçüktür, bu durum için optimize etmek mantıklıdır. Böylelikle, az sayıda doğrudan işaretçiyle (12 tipik bir sayıdır), bir inodex, bir (veya daha fazla) dolaylı bloğa ihtiyaç duyan 48 KB veriyi doğrudan gösterebilir daha büyük dosyalar için. Bkz. Agrawal et. son zamanlarda yapılan bir çalışma için al [A + 07]; Şekil 40.2 bu sonuçları özetlemektedir.

Çoğu dosya küçüktür	2K en yaygın boyuttur
Ortalama dosya boyutu büyüyor	Neredeyse 200 Bin ortalama
Çoğu bayt büyük dosyalarda saklanır	Birkaç büyük dosya alanın çoğunu kullanır
Dosya sistemleri çok sayıda dosya içerir	Ortalama olarak neredeyse 100 bin
Dosya sistemleri kabaca yarısı dolu	Diskler büyüdükçe bile dosya sistemleri % 50 dolu kalın
Dizinler genellikle küçüktür	Birçoğunun çok az girişi vardır; çoğu 20 veya daha azına sahip olmak

#### Şekil 40.2: Dosya Sistemi Ölçüm Özeti

Tabii ki, inode tasarımı alanında, diğer birçok olasılık vardır; Sonuçta, inode sadece bir veri yapısıdır ve ilgili bilgileri depolayan ve etkili bir şekilde sorgulayabilen herhangi bir veri yapısı yeterlidir. Dosya sistemi yazılımı kolayca değiştirildiğinden, iş yükleri veya teknolojiler değişirse farklı tasarımlardan yararlanmaya istekli olmalısınız.

### 40.4 Dizin Organizasyonu

Vsfs'de (birçok dosya sisteminde olduğu gibi) dizinlerin basit bir organizasyonu vardır; Bir dizin temel olarak yalnızca (giriş adı, kod numarası) çiftlerinin bir listesini içerir. Belirli bir dizindeki her dosya veya dizin için bir dize vardır ve dizinin veri blok (lar) ında bir sayı. Her dize için, orada bir uzunluk da olabilir (değişken boyutlu isimler varsayılarak).

Örneğin, dizin dizininin (inode numarası 5) içinde üç dosya olduğunu varsayalım (foo, bar ve foobar oldukça uzun bir addır), sırasıyla inode sayıları 12, 13 ve 24'tür. Dir için disk üzerindeki veriler şöyle görünebilir:

```
inum | reflen | strlen | name
5     12     2       .
2     12     3       ..
12    12     4       foo
13    12     4       bar
24    36    28     foobar_is_a_pretty_longname
```

Bu örnekte, her girdinin bir kod numarası, kayıt uzunluğu (adın toplam baytı artı kalan boşluk), dize uzunluğu (adın gerçek uzunluğu) ve son olarak girdinin adı vardır. Her papaz evinin iki ek girişi olduğunu unutmayın., . "nokta" ve .. "nokta-nokta"; nokta dizini yalnızca geçerli dizindir (bu örnekte dır), nokta-nokta ise ana dizindir (bu durumda kök).

Bir dosyayı silmek (örneğin, bağlantıyı kaldır () ögesini çağırmak) dizinin ortasında boş bir alan bırakabilir ve bu nedenle bunu işaretlemenin bir yolu da olmalıdır (örneğin, sıfır gibi ayrılmış bir kod numarasıyla). Böyle bir silme, kayıt uzunluğunun

kullanılmasının bir nedenidir: yeni bir giriş, eski, daha büyük bir girişi yeniden kullanabilir ve böylece içinde fazladan boşluk olabilir.

### Bağlantılı TABANLI Yaklaşımlar

Düğüm tasarlamada daha basit bir başka yaklaşım, **bağlantılı bir liste(linked list)** kullanmaktır. Bu nedenle, bir inode'un içinde, birden çok işaretçiye sahip olmak yerine, dosyanın ilk bloğuna işaret etmek için yalnızca bir tanesine ihtiyacınız vardır. Daha büyük dosyaları işlemek için, bu veri bloğunun sonuna başka bir işaretçi vb. Ekleyin ve böylece büyük dosyaları destekleyebilirsiniz.

Tahmin edebileceğiniz gibi, bağlantılı dosya tahsisi aşağıdakiler için kötü performans gösterir bazı iş yükleri; Bir dosyanın son bloğunu okumayı düşünün, örneğin, ya da sadece rastgele erişim yapmak. Böylece, bağlantılı tahsisin daha iyi çalışmasını sağlamak için, bazı sistemler bunun yerine bir bellek içi bağlantı bilgisi tablosu tutacaktır bir sonraki işaretçileri veri bloklarının kendileriyle saklamak. Tablo bir veri bloğunun adresi ile dizine eklenir D; Bir girdinin içeriği basitçe D'nin bir sonraki işaretçisi, yani bir dosyadaki bir sonraki bloğun adresi D'yi takip eder. Boş bir değer de orada olabilir (bir dosya sonunu gösterir) veya belirli bir bloğun boş olduğunu gösteren başka bir işaretleyici. Böyle olması bir sonraki işaretçiler tablosu, bağlantılı bir ayırma şemasının bunu yapabilmesini sağlar rastgele dosya erişimlerini etkili bir şekilde yapın, sadece ilk önce tarayarak (bellekte) istenen bloğu bulmak için tablo ve ardından (diskte) erişme doğrudan.

Böyle bir masa tanıdık geliyor mu? Tanımladığımız şey, **dosya ayırma tablosu(file allocation table)** veya **FAT(FAT)** dosya sistemi olarak bilinen şeyin temel yapısıdır. Evet, NTFS [C94] 'ten önceki bu klasik eski Windows dosya sistemi, basit bir bağlantılı tabanlı ayırma şemasına dayanmaktadır. Standart bir UNIX dosya sisteminden başka farklılıklar da vardır; Örneğin, kendi başına kodlar yoktur, bunun yerine bir dosya hakkında meta verileri depolayan ve doğrudan söz konusu dosyanın ilk bloğuna atıfta bulunan dizin girişleri vardır, bu da sabit bağlantılar oluşturmayı imkansız kılar. Daha fazla ayrıntı için Brouwer'a [B02] bakın.

Dizinlerin tam olarak nerede saklandığını merak ediyor olabilirsiniz. Çoğu zaman, dosya sistemleri dizinleri özel bir dosya türü olarak ele alır. Bu nedenle, bir dizinin inode tablosunda bir yerde bir inode vardır (inode'un tür alanı “normal dosya” yerine “dizin” olarak işaretlenmiştir). Dizin, inode'un işaret ettiği veri bloklarına (ve belki de dolaylı bloklara) sahiptir; Bu veri blokları, basit dosya sistemimizin veri bloğu bölgesinde yaşar. Böylece disk üzerindeki yapımız değişmeden kalır.

Ayrıca, bu basit doğrusal dizin deneme listesinin bu tür bilgileri depolamanın tek yolu olmadığını bir kez daha belirtmeliyiz. Daha önce olduğu gibi, herhangi bir veri yapısı mümkündür. Örneğin, XFS [S + 96] dizinleri B ağacı biçiminde depolar ve dosya oluşturma işlemlerini (bir dosya adının oluşturulmadan önce kullanılmadığından emin olmak zorunda olan), tamamı taranması gereken basit listelere sahip sistemlerden daha hızlı hale getirir.

## BOŞ ALAN YÖNETİMİ

Boş alanı yönetmenin birçok yolu vardır; bitmapler sadece bir yoldur. Bazı ilk dosya sistemleri, süper bloktaki tek bir işaretçinin ilk serbest bloğu işaret edecek şekilde tutulduğu **serbest listeler(free list)** kullandı; Bu bloğun içinde bir sonraki serbest işaretçi tutuldu, böylece sistemin serbest blokları arasında bir liste oluşturuldu. Bir bloğa ihtiyaç duyulduğunda, kafa bloğu kullanıldı ve liste buna göre güncellendi.

Modern dosya sistemleri daha karmaşık veri yapıları kullanır. Örneğin, sgi'nin XFS [S + 96], diskin hangi parçalarının boş olduğunu kompakt bir şekilde temsil etmek için bir tür **B ağacı(B-tree)** kullanır. Her veri yapısında olduğu gibi, farklı zaman-mekan dengeleri mümkündür.

### 40.5 Boş Alan Yönetimi

Bir dosya sistemi, hangi inode'ların ve veri bloklarının boş olduğunu izlemelidir ve hangileri değildir, böylece yeni bir dosya veya dizin tahsis edildiğinde şunları bulabilir bunun için yer. Bu nedenle **boş alan yönetimi(free space management)** tüm dosya sistemleri için önemlidir. Vsfs'de bu görev için iki basit bitmap'imiz var.

Örneğin, bir dosya oluşturduğumuzda, o dosya için bir inode ayırmamız gerekecektir. Böylece dosya sistemi, bit eşlemede boş olan bir in-ode'yi arayacak ve dosyaya ayıracaktır; Dosya sisteminin inodu kullanıldığı gibi işaretlemesi (1 ile) ve sonunda disk üzerindeki bit eşlemine doğru bilgilerle güncellemesi gerekecektir. Bir veri bloğu ayrıldığında benzer bir dizi etkinlik gerçekleşir.

Tahsis ederken başka bazı hususlar da devreye girebilir

yeni bir dosya için veri blokları. Örneğin, bazı Linux dosya sistemleri, örneğin ext2 ve ext3 olarak, ücretsiz olan bir dizi blok (örneğin 8) arayacaktır yeni bir dosya oluşturulduğunda ve veri bloklarına ihtiyaç duyduğunda; Böyle bir sayıda boş blok bularak ve ardından bunları yeni oluşturulan dosyaya ayırarak, dosya sistemi, dosyanın bir bölümünün bitişik olacağını garanti eder. disk, böylece performansı artırır. Böyle bir **ön tahsis(pre-allocation)** politikası bu nedenle, veri blokları için alan ayırırken yaygın olarak kullanılan bir buluşsal yöntem..

#### 40.6 Erişim Yolları: Okuma ve Yazma

Artık dosyaların ve dizinlerin diskte nasıl depolandığına dair bir fikrimiz olduğuna göre, bir dosyayı okuma veya yazma etkinliği sırasında işlem akışını takip edebilmeliyiz. Bu nedenle, bu **erişim yolunda(access path)** neler olduğunu anlamak, bir dosya sisteminin nasıl çalıştığına dair bir anlayış geliştirmenin ikinci anahtarıdır; dikkat et!

Aşağıdaki örnekler için, dosya sisteminin monte edildiğini ve böylece süper bloğun zaten bellekte olduğunu varsayalım. Diğer her şey (yani, inode'lar, dizinler) hala diskte.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data	bar data	bar data
								[0]	[1]	[2]
open(bar)			read		read	read				
					read		read			
read()					read			read		
					write					
read()					read				read	
					write					
read()					read					read
					write					

Şekil 40.3: Dosya Okuma Zaman Çizelgesi (Zaman Aşağı Doğru Artıyor)

## Diskten Dosya Okuma

Bu basit örnekte, önce bir dosyayı (ör. /foo/bar) açmak, okumak ve ardından kapatmak istediğinizi varsayalım. Bu basit örnek için, dosyanın yalnızca 12 KB boyutunda (yani 3 blok) olduğunu varsayalım.

Bir open ("/foo / bar", O\_RDONLY) çağrısı verdiğinizde, dosya sistemi, dosya hakkında bazı temel bilgileri (izin bilgileri, dosya boyutu vb.) Elde etmek için önce dosya çubuğunun kodunu bulmalıdır.). Bunu yapmak için, dosya sisteminin inodu bulabilmesi gerekir, ancak şu anda sahip olduğu tek şey tam yol adıdır. Dosya sistemi yol adını **geçmeli(travers)** ve böylece istenen inodu bulmalıdır.

Tüm geçişler, dosya sisteminin kökünden, basitçe / olarak adlandırılan **kök dizinde(root directory)** başlar. Bu nedenle, fs'nin diskten okuyacağı ilk şey kök dizinin inodudur. Ama bu inode nerede? Bir inode bulmak için onun ı numarasını bilmeliyiz. Genellikle, bir dosyanın veya dizinin ı numarasını ana dizinde buluruz; kökün ebeveyni yoktur (tanım gereği). Bu nedenle, kök kod numarası "iyi bilinmeli" olmalıdır; FS, dosya sistemi monte

edildiğinde ne olduğunu bilmelidir. Çoğu UNIX dosya sisteminde kök kod numarası 2'dir. Böylece, işleme başlamak için FS, 2 numaralı inodu (ilk inod bloğu) içeren blokta okur.

Inode okunduktan sonra FS, kök dizinin içeriğini içeren veri bloklarının işaretçilerini bulmak için içine bakabilir. Bu nedenle FS, dizini okumak için bu disk üzerindeki işaretçileri kullanır, bu durumda foo için bir giriş arar. Bir veya daha fazla izin veri bloğunda okuyarak, foo girişini bulacaktır; Bir kez bulunduğunda FS, daha sonra ihtiyaç duyacağı foo'nun (44 olduğunu söyleyin) inode numarasını da bulmuş olacaktır.

### Okumalar Ayırma Yapılarına Erişmiyor

Birçok öğrencinin bitmapler gibi tahsis yapılarıyla kafasının karıştığını gördük. Özellikle, çoğu zaman bir dosyayı okuduğunuzda ve yeni bloklar ayırmadığınızda, bitmap'e hala danışılacağını düşünür. Bu doğru değil! Bit eşlemler gibi ayırma yapılarına yalnızca ayırma gerektiğinde erişilir. Kodlar, izinler ve dolaylı bloklar, bir okuma yeniden görevini tamamlamak için ihtiyaç duydukları tüm bilgilere sahiptir; Kod zaten ona işaret ettiğinde bir bloğun ayrıldığından emin olmaya gerek yoktur.

Bir sonraki adım, istenene kadar yol adını yinelemeli olarak geçmektir inode bulundu. Bu örnekte FS, foo'nun inode'unu ve ardından izin verilerini içeren bloğu okur ve sonunda inode numarasını bulur bardan. Open() ögesinin son adımı, çubuğun inode'unu belleğe okumaktır; FS daha sonra son bir izin kontrolü yapar, bunun için bir dosya tanımlayıcısı ayırır işlem başına açık dosya tablosundaki işlem ve kullanıcıya döndürür.

Açıldıktan sonra, program okumak için bir read () sistem çağrısı verebilir dosyadan. İlk okuma (lseek() çağrılmadıkça ofset 0'da) böylece dosyanın ilk bloğunda okuyacak, bulmak için inode'a danışacak böyle bir bloğun konumu; Ayrıca inodu yeni bir son erişilen zamanla güncelleyebilir. Okuma, bellek içi açık dosya tablosunu daha da güncelleyecektir bu dosya tanımlayıcısı için, dosya ofsetini bir sonraki okumanın yapacağı şekilde güncellemek ikinci dosya bloğunu vb. okuyun.

Bir noktada dosya kapatılacak. Burada yapılacak çok daha az iş var; Açıkçası, dosya tanımlayıcısının ayrılması gerekiyor, ancak şimdilik

fs'nin gerçekten yapması gereken tek şey bu. Disk G / Ç'si yok.

Tüm bu sürecin bir tasviri Şekil 40.3'te (sayfa 11) bulunur; Şekilde zaman aşağı doğru artar. Şekilde, open, dosyanın inode'unu bulmak için çok sayıda okumanın gerçekleşmesine neden olur. Daha sonra, her bloğun okunması, dosya sisteminin önce inode'a başvurmasını, ardından bloğu okumasını ve ardından inode'un son erişilen zaman alanını bir yazıyla güncellemesini gerektirir. Biraz zaman geçirin ve neler olduğunu anlayın.

Ayrıca, açık tarafından oluşturulan G / Ç miktarının yol adının uzunluğuna uygun olduğunu unutmayın. Yoldaki her ek dizin için, verilerinin yanı sıra kendi kodunu da okumamız gerekir. Bunu daha da kötüleştirmek, büyük dizinlerin varlığı olacaktır; Burada, bir dizinin içeriğini almak için yalnızca bir blok okumamız gerekirken, büyük bir dizinde,

### **Diske Dosya Yazma**

Bir dosyaya yazmak da benzer bir işlemdir. İlk olarak, dosya açılmalıdır (yukarıdaki gibi). Ardından, uygulama dosyayı yeni içeriklerle güncellemek için write() çağrılarını verebilir. Son olarak, dosya kapatıldı.Okumanın aksine, dosyaya yazmak da bir blok **ayırabilir(allocate)** (örneğin bloğun üzerine yazılmadığı sürece). Yeni bir dosya yazarken, her yazının yalnızca diske veri yazması değil, önce karar vermesi gerekir



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read		read	read	write		
write()	read write			read				write		
write()	read write			write read				write		
write()	read write			write read						write
write()	read write			write						write

Şekil 40.4: Dosya Oluşturma Zaman Çizelgesi (Aşağı Doğru Artan Süre)

dosyaya hangi bloğun tahsis edileceği ve böylece dosyanın diğer yapılarının güncelleneceği buna göre disk (örneğin, veri bitmap ve inode). Böylece, her biri bir dosya mantıksal olarak beş G / Ç oluşturur: biri veri bitmapini okumak için (yani daha sonra yeni tahsis edilen bloğu kullanıldığı gibi işaretlemek için güncellendi), birini yazmak için bitmap (yeni durumunu diske yansıtmak için), okumak ve yazmak için iki tane daha inode (yeni bloğun konumu ile güncellenir) ve son olarak gerçek bloğun kendisini yazmak için bir tane.

Dosya oluşturma gibi basit ve yaygın bir işlem düşünüldüğünde yazma trafiği miktarı daha da kötüdür. Bir dosya oluşturmak için, dosya sistem yalnızca bir inode ayırmakla kalmamalı, aynı zamanda içinde yer ayırmalıdır yeni dosyayı içeren dizin. Toplam G / Ç trafiği miktarı bunu yapmak oldukça yüksektir: inode bitmapine bir okuma (ücretsiz bir inode bulmak için), inode bitmapine bir yazma (tahsis edildiğini işaretlemek için), yenisine bir yazma inode'un kendisi (başlatmak için), biri dizinin verilerine

(bağlamak için dosyanın inode numarasına üst düzey adı) ve bir okuma ve yazma güncellemek için inode dizinine. Dizinin yeni girdiyi ac'ye dönüştürmek için büyümesi gerekiyorsa, ek G / Ç'ler (yani, veri bit eşlemine ve yeni izin bloğu) da gerekli olacaktır. Bütün bunlar sadece bir dosya oluşturmak için!

/Foo/bar dosyasının oluşturulduğu belirli bir örneğe bakalım,ve ona üç blok yazılmıştır. Şekil 40.4 (sayfa 13), hap'ın open() (dosyayı oluşturan) sırasında ve üçünün her biri sırasında ne yazdığını göstermektedir 4KB yazıyor.

Şekilde, diske okuma ve yazma, altında gruplandırılmıştır sistem çağrısı onların oluşmasına ve alabilecekleri kaba sıralamaya neden oldu yer, şeklin üstünden alta doğru gider. Ne kadar olduğunu görebilirsiniz dosyayı oluşturmak için çalışın: Bu durumda 10 G / Ç, yol adını yürümek için ve son olarak dosyayı oluşturun. Ayrıca, tahsis edilen her yazının maliyetler 5 G / Ç: inodu okumak ve güncellemek için bir çift, okumak için başka bir çift ve veri bit eşlemine güncelleyin ve son olarak verilerin kendisinin yazılması. Bir dosya sistemi bunlardan herhangi birini makul bir verimlilikle nasıl başarabilir?

#### DOSYA SİSTEMİ G / Ç MALİYETLERİ NASIL AZALTILIR

Dosya açma, okuma veya yazma gibi en basit işlemler bile diskin üzerine dağılmış çok sayıda G / Ç işlemine neden olur. Ne bir dosya sistemi, bu kadar çok G / Ç yapmanın yüksek maliyetlerini azaltmak için yapabilir mi?

#### Önbelleğe Alma ve Arabelleğe Alma

Yukarıdaki örneklerde, dosyaların okunması ve yazılması pahalı olabilir ve bu da (yavaş) diske birçok G / Ç'ye neden olabilir. Neyi düzeltmek için açıkçası, çoğu dosya sistemi agresif bir şekilde büyük bir performans sorunu olabilir önemli blokları önbelleğe almak için sistem belleğini (DRAM) kullanın. Yukarıdaki açık örneği hayal edin: önbelleğe almadan her dosya açılır izin hiyerarşisindeki her seviye için en az iki okuma gerekir (söz konusu dizinin kodunu okumak için bir tane ve okumak için en az bir tane verileri). Uzun bir yol adıyla (ör. /1/2/3/ ... /100/ dosya.txt), dosya sistem sadece dosyayı açmak için kelimenin tam anlamıyla yüzlerce okuma yapardı!

Böylece ilk dosya sistemleri, popüler blokları tutmak için **sabit boyutlu bir(fixed-size cache)** önbellek tanıttı. Sanal bellek tartışmamızda olduğu gibi, **LRU(LRU)** ve farklı varyantlar gibi stratejiler hangi blokların önbellekte tutulacağına karar verecektir. Bu sabit boyutlu önbellek genellikle önyüklemeye sırasında toplam belleğin yaklaşık% 10'u olacak şekilde tahsis edilir.

Bununla birlikte, belleğin bu **statik bölümlenmesi(static partitioning)** savurgan olabilir; Dosya sistemi belirli bir zamanda belleğin% 10'una ihtiyaç duymuyorsa

ne olur? Yukarıda açıklanan sabit boyutlu yaklaşımla, dosya önbellegeindeki kullanılmayan sayfalar başka bir kullanım için yeniden amaçlanamaz ve bu nedenle boşa gider.

Modern sistemler, aksine, **dinamik bir bölümlleme(dynamic partitioning)** yaklaşımı kullanır.

Özellikle, birçok modern işletim sistemi sanal belleği entegre eder sayfalar ve dosya sistemi sayfaları **birleştirilmiş sayfa önbellegeine(unified page cache)** [S00]. Bu şekilde, bellek, sanal bellek ve dosya arasında daha esnek bir şekilde tahsis edilebilir sistem, belirli bir zamanda hangisinin daha fazla belleğe ihtiyaç duyduğuna bağlı olarak.

Şimdi önbellege alma ile dosya açma örneğini hayal edin. İlk açık mayıs dizin kodunda ve verilerde okumak için çok fazla G / Ç trafiği oluşturun, ancak aynı dosyanın (veya aynı dizindeki dosyaların) sonraki dosya açılmaları çoğunlukla önbellege isabet eder ve bu nedenle G / Ç gerekmez.

#### İPUCU: STATİK VE DİNAMİK BÖLÜMLEMEYİ ANLAYIN

Bir kaynağı farklı istemciler / kullanıcılar arasında bölüştürürken, **statik bölümlleme(static partitioning)** veya **dinamik bölümlleme(dynamic partitioning)** kullanabilirsiniz. Statik yaklaşım, kaynağı yalnızca bir kez sabit oranlara böler; Örneğin, iki olası bellek kullanıcısı varsa, bir kullanıcıya belleğin sabit bir kısmını, geri kalanını diğerine verebilirsiniz. Dinamik yaklaşım daha esnektir ve zaman içinde kaynağın farklı miktarlarını verir; Örneğin, bir kullanıcı belirli bir süre için daha yüksek bir disk bant genişliği yüzdesi elde edebilir, ancak daha sonra sistem geçiş yapabilir ve farklı bir kullanıcıya kullanılabilir disk bant genişliğinin daha büyük bir kısmını vermeye karar verebilir.

Her yaklaşımın avantajları vardır. Statik bölümlleme, her kullanıcının kaynağın bir kısmını almasını sağlar, genellikle daha öngörülebilir performans sağlar ve uygulanması genellikle daha kolaydır. Dinamik bölümlleme, daha iyi kullanım sağlayabilir (kaynağa aç kullanıcıların boşta kalan kaynakları tüketmesine izin vererek), ancak uygulanması daha karmaşık olabilir ve boşta kalan kaynakları başkaları tarafından tüketilen ve gerektiğinde geri alınması uzun zaman alan kullanıcılar için daha kötü performansla yol açabilir. Onuncu durumda olduğu gibi, en iyi yöntem yoktur; Bunun yerine, eldeki sorunu düşünmeli ve hangi yaklaşımın en uygun olduğuna karar vermelisiniz. Gerçekten, bunu her zaman yapman gerekmiyor mu?

Önbelleğe almanın yazma üzerindeki etkisini de düşünelim. Oysa okuma G / Ç yeterince büyük bir önbellek ile tamamen önlenebilir, yazma trafiği kalıcı olmak için diske gitmek. Bu nedenle, bir önbellek hizmet vermez okuma için yaptığı yazma trafiğinde aynı tür bir filtre gibi. Bu dedi,**yazma arabelleğinin(write buffering)** (bazen çağrıldığı gibi) kesinlikle biçim başına bir takım avantajları vardır. İlk olarak, yazma işlemlerini geciktirerek dosya sistemi **toplu(batch)** iş yapabilir bazı güncellemeler daha küçük bir G / Ç kümesine; örneğin, bir inode bit eşlem bir dosya oluşturulduğunda güncellenir ve birkaç dakika sonra şu şekilde güncellenir: başka bir dosya oluşturulur, dosya sistemi yazmayı geciktirerek bir G / Ç kaydeder ilk güncellemeden sonra.İkincisi, bellekte bir dizi yazmayı arabelleğe alarak, sistem daha sonra sonraki G / Ç'leri **planlayabilir(Schedule)** ve böylece biçim başına artışı artırabilir. Son olarak, bazı yazmalar geciktirilerek tamamen önlenir; Örneğin, bir uygulama bir dosya oluşturur ve ardından silerse, yazmaların dosya oluşturmayı diske yansıtacak şekilde geciktirilmesi, bunları tamamen **önler(avoid)**s. Bu durumda tembellik (blokları diske yazarken) bir erdemdir.

Yukarıdaki nedenlerden dolayı, çoğu modern dosya sistemi arabelleği, başka bir değiş tokuşu temsil eden, beş ila otuz saniye arasında herhangi bir yerde mem oride yazar: güncellemeler diske sunulmadan önce sistem çökerse, güncellemeler kaybolur; ancak, yazmaları mem oride daha uzun süre tutarak performans düşer. grupta, zamanlama ve yazmaktan kaçınmak bile.

#### İpucu: DAYANIKLILIK/PERFORMANS DENGELİMESİNİ ANLAYIN

Depolama sistemleri genellikle aşağıdakilere dayanıklılık / performans takası sunar kullanıcı. Kullanıcı, yazılan verilerin derhal kalıcı olmasını isterse, sistem, yeni yazılan verileri diske işlemek için tüm çabayı göstermelidir ve bu nedenle yazma yavaştır (ancak güvenlidir). Ancak, eğer kullanıcı küçük bir veri kaybını tolere edebilir, sistem tampon yazabilir bir süre bellekte ve daha sonra diske (arka toprağa) yazın. Bunu yapmak, yazmaların hızlı bir şekilde tamamlandığını gösterir, böylece algılanan performansı kanıtlarım; Ancak, bir kilitlenme meydana gelirse, yazmaz yine de diske bağlı olanlar kaybolacak ve dolayısıyla takas olacak.Bu değiş tokuşun nasıl doğru bir şekilde yapılacağını anlamak için, depolama sistemini kullanan uygulamanın ne gerektirdiğini anlamak en iyisidir; Örneğin, web tarayıcınız tarafından indirilen son birkaç görüntüyü kaybetmek, bankanıza para ekleyen bir veritabanı işleminin bir kısmını kaybetmek tolere

edilebilir olsa da hesap daha az tolere edilebilir olabilir. Tabii zengin değilseniz; O halde neden her kuruşunu biriktirmeyi bu kadar önemsiyorsunuz?

Bazı uygulamalar (veritabanları gibi) bu değiş tokuştan hoşlanmaz. Böylelikle, yazma arabelleği nedeniyle beklenmeyen veri kaybını önlemek için, yalnızca zorlarlar **doğrudan G / Ç(direct I/O)** arabirimlerini kullanarak fsync() ögesini çağırarak diske yazar. önbellekte veya **ham disk(raw disk)** arabirimini kullanarak ve önbellekten kaçınarak çalışın. dosya sistemi altogether2 Çoğu uygulama, dosya sistemi tarafından yapılan değiş tokuşlarla yaşarken, aşağıdakileri elde etmek için yeterli kontrol vardır: sistem, varsayılan olarak tatmin edici değilse, istediğinizi yapmak için.

## 40.8 Özet

Bir dosya sistemi oluşturmak için gereken temel makineleri gördük. Genellikle her dosya (meta veriler) hakkında bazı bilgiler olması gerekir inode adı verilen bir yapıda saklanır. Dizinler yalnızca belirli bir türdür adı depolayan dosya → kod numarası eşlemeleri. Ve diğer yapılar örneğin, dosya sistemleri genellikle aşağıdaki gibi bir yapı kullanır: hangi inode'ların veya veri bloklarının boş veya ayrılmış olduğunu izlemek için bitmap.

Dosya sistemi tasarımının müthiş yönü özgürlüğüdür; Önümüzdeki bölümlerde incelediğimiz dosya sistemlerinin her biri, dosya sisteminin bir yönünü optimize etmek için bu özgürlükten yararlanıyor. Ayrıca açıkça keşfedilmemiş bıraktığımız birçok politika kararı var. Örneğin, yeni bir dosya oluşturulduğunda, diske nereye yerleştirilmelidir? Bu politika ve diğerleri ayrıca gelecekteki bölümlerin konusu olun. Yoksa yapacaklar mı?

eski okul veritabanları ve işletim sisteminden kaçınma ve her şeyi kendileri kontrol etme konusundaki eski eğilimleri hakkında daha fazla bilgi edinmek için bir veritabanı dersi alın. Ama dikkat et! Şunlar

veritabanı türleri her zaman işletim sistemini bozmaya çalışıyor. Yazıklar olsun veri tabanı çalışanlar. Utanç.

Dosya sistemleri konusuyla daha da ilginizi çeken gizemli müziğe işaret edin.

## References

[A+07] “A Five-Year Study of File-System Metadata” by Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch. FAST '07, San Jose, California, February 2007. *An excellent recent analysis of how file systems are actually used. Use the bibliography within to follow the trail of file-system analysis papers back to the early 1980s.*

[B07] “ZFS: The Last Word in File Systems” by Jeff Bonwick and Bill Moore. Available from: [http://www.ostep.org/Citations/zfs\\_last.pdf](http://www.ostep.org/Citations/zfs_last.pdf). *One of the most recent*

*important file systems, full of features and awesomeness. We should have a chapter on it, and perhaps soon will.*

[B02] "The FAT File System" by Andries Brouwer. September, 2002. Available online at: <http://www.win.tue.nl/~aeb/linux/fs/fat/fat.html>. *A nice clean description of FAT. The file system kind, not the bacon kind. Though you have to admit, bacon fat probably tastes better.*

[C94] "Inside the Windows NT File System" by Helen Custer. Microsoft Press, 1994. *A short book about NTFS; there are probably ones with more technical details elsewhere.*

[H+88] "Scale and Performance in a Distributed File System" by John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West.. ACM TOCS, Volume 6:1, February 1988. *A classic distributed file system; we'll be learning more about it later, don't worry.*

[P09] "The Second Extended File System: Internal Layout" by Dave Poirier. 2009. Available: <http://www.nongnu.org/ext2-doc/ext2.html>. *Some details on ext2, a very simple Linux file system based on FFS, the Berkeley Fast File System. We'll be reading about it in the next chapter.*

[RT74] "The UNIX Time-Sharing System" by M. Ritchie, K. Thompson. CACM Volume 17:7, 1974. *The original paper about UNIX. Read it to see the underpinnings of much of modern operating systems.*

[S00] "UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD" by Chuck Silvers. FREEIX, 2000. *A nice paper about NetBSD's integration of file-system buffer caching and the virtual-memory page cache. Many other systems do the same type of thing.*

[S+96] "Scalability in the XFS File System" by Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, Geoff Peck. USENIX '96, January 1996, San Diego, California. *The first attempt to make scalability of operations, including things like having millions of files in a directory, a central focus. A great example of pushing an idea to the extreme. The key idea behind this file system: everything is a tree. We should have a chapter on this file system too.*

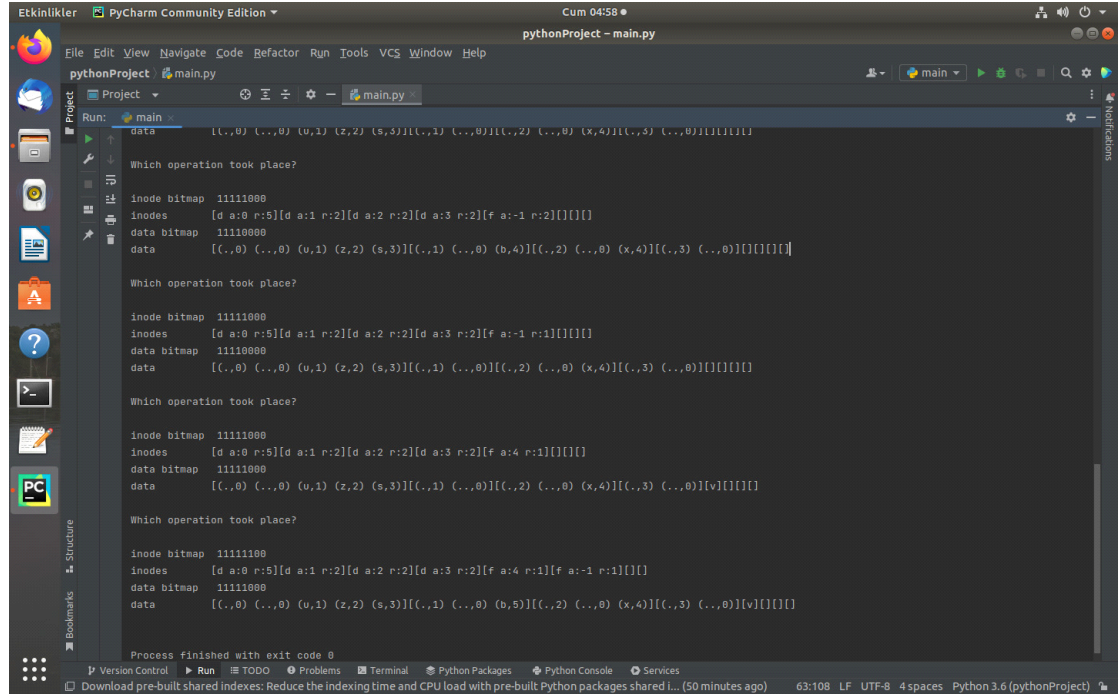
## Ödev (Simülasyon)

Bu aracı kullanın, vsfs.py , değişken işlemler gerçekleştikçe dosya sistemi durumunun nasıl değiştiğini incelemek. Dosya sistemi, yalnızca bir kök diziniyle boş bir durumda başlar. Simülasyon gerçekleştikçe, çeşitli işlemler gerçekleştirilir, böylece dosya sisteminin disk üzerindeki durumu yavaş yavaş değiştirilir. Ayrıntılar için README sayfasına bakın.

## Sorular

1-) Simülatörü bazı farklı rastgele değerlerle çalıştırın (örneğin 17, 18, 19, 20) ve her durum değişikliği arasında hangi işlemlerin yapılması gerektiğini anlayıp anlayamayacağınızı görün.





Yukarıda ki Ekran resimlerimizde seed değerimizi 17 konumuna getirerek denemelerimizi yaptık ve program değişiklikler karşısında ne yapacağını şaşırmadan sonuçlarımızı ekrana bastırdı.mkdir ,creat ,unlink ,link ,fd değerlerimiz ise aşağıda ki gibi olmuştur.

mkdir('/u')

creat('/a')

unlink('/a')

mkdir('/z')

mkdir('/s')

creat('/z/x')

link('/z/x', '/u/b')

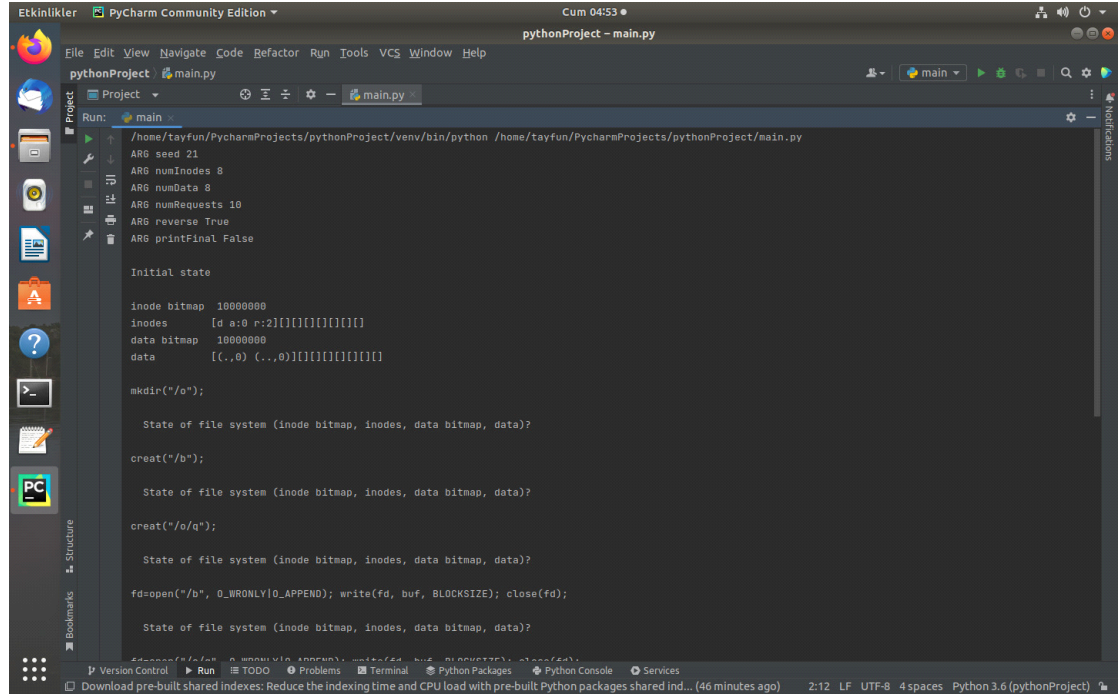
unlink('/u/b')

fd=open("/z/x", O\_WRONLY|O\_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

creat('/u/b')

2-) Şimdi -r bayrağıyla çalıştırmak dışında farklı rastgele değerle (örneğin 21, 22, 23, 24) kullanarak aynısını yapın, böylece işlem gösterilirken durum değişikliğini tahmin etmenizi sağlayın. Hangi blokları ayırmayı tercih ettikleri açısından inode ve veri bloğu ayırma algoritmaları hakkında ne sonuca varabilirsiniz?





```
Run: main
/home/tayfun/PycharmProjects/pythonProject/venv/bin/python /home/tayfun/PycharmProjects/pythonProject/main.py
ARG seed 21
ARG numInodes 8
ARG numData 8
ARG numRequests 10
ARG reverse True
ARG printFinal False

Initial state

inode bitmap 10000000
inodes [d a:0 r:2][][][]
data bitmap 10000000
data [(..0) (...0)][][][]

mkdir("/o");

State of file system (inode bitmap, inodes, data bitmap, data)?

creat("/b");

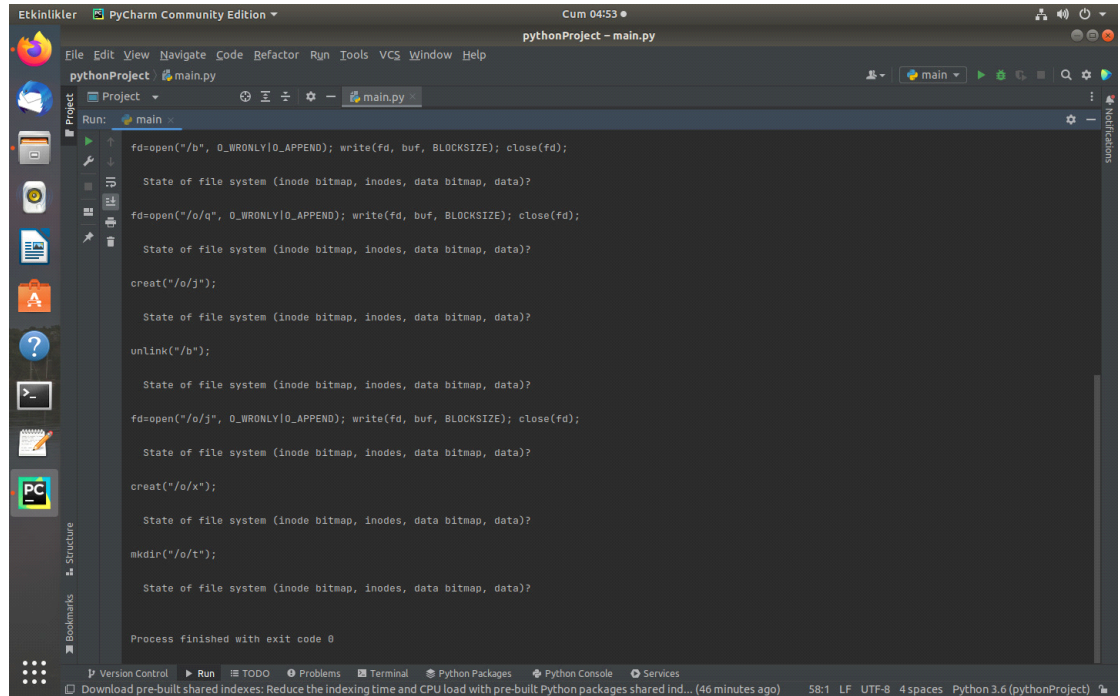
State of file system (inode bitmap, inodes, data bitmap, data)?

creat("/o/a");

State of file system (inode bitmap, inodes, data bitmap, data)?

fd=open("/b", 0_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

State of file system (inode bitmap, inodes, data bitmap, data)?
```



```
fd=open("/b", 0_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

State of file system (inode bitmap, inodes, data bitmap, data)?

fd=open("/o/a", 0_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

State of file system (inode bitmap, inodes, data bitmap, data)?

creat("/o/j");

State of file system (inode bitmap, inodes, data bitmap, data)?

unlink("/b");

State of file system (inode bitmap, inodes, data bitmap, data)?

fd=open("/o/j", 0_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

State of file system (inode bitmap, inodes, data bitmap, data)?

creat("/o/x");

State of file system (inode bitmap, inodes, data bitmap, data)?

mkdir("/o/t");

State of file system (inode bitmap, inodes, data bitmap, data)?

Process finished with exit code 0
```

Algoritma, veriler için ilk uygun bloğu tahsis etmeyi tercih eder. İlk önce Boşta olan ilk katmanı doldurduktan sonra diğer katmanlara geçiş sağlar.

3-) Şimdi dosya sistemindeki veri bloklarının sayısını çok düşük sayılara (iki diyelim) düşürün ve simülatörü yüz kadar istek için çalıştırın. Bu son derece kısıtlı düzende dosya sisteminde ne tür

dosyalar bulunur? Ne tür operasyonlar başarısız olur?

Sistem çoğunlukla izinler ve boş dosyalarla sonuçlanacaktır. Doğal olarak bundan sonra creat, mkdir, write işlemleri başarısız olacaktır.

4-) Şimdi aynısını yapın, ancak inode'larla. Çok az sayıda inode ile ne tür işlemler başarılı olabilir? Hangisi genellikle başarısız olur? Dosya sisteminin son durumu ne olabilir?

Daha az inode ile mkdir, creat işlemleri başarısız olmaya başlayacak.

Benzetilmiş dosya sisteminin bir yönü, yalnızca bir veri bloğu dosyasına izin vermesi ve bir işlem başarısız olur olmaz durmasıdır. Dolayısıyla, inode veya data bloğunun tek başına kullanılabilirliği pek bir fark yaratmaz.