40

File System Implementation

Bu bölümde, vsfs (**Veri Simple File System(Çok Basit Dosya Sistemi)**) olarak bilinen basit bir dosya sistemi uygulamasını tanıtıyoruz. Bu dosya sistemi, tipik bir UNIX dosya sisteminin basitleştirilmiş bir sürümüdür ve bu nedenle, bugün birçok dosya sisteminde bulacağınız bazı temel disk yapılarını, erişim yöntemlerini ve çeşitli ilkeleri tanıtmaya hizmet eder.

Dosya sistemi saf bir yazılımdır; CPU ve bellek sanallaştırma geliştirmemizin aksine, dosya sisteminin bazı yönlerinin daha iyi çalışmasını sağlamak için donanım özellikleri eklemeyeceğiz (yine de dosya sisteminin iyi çalıştığından emin olmak için cihaz özelliklerine dikkat etmek isteyeceğiz). Bir dosya sistemi oluştururken sahip olduğumuz büyük esneklik nedeniyle, kelimenin tam anlamıyla AFS'den (Andrew Dosya Sistemi) [H+88] ZFS'ye (Sun's Zettabyte Dosya Sistemi) [B07] kadar pek çok farklı sistem oluşturulmuştur. Bu dosya sistemlerinin tümü farklı veri yapılarına sahiptir ve bazı şeyleri benzerlerinden daha iyi veya daha kötü yapar. Bu nedenle, dosya sistemlerini öğreneceğimiz yol örnek olay incelemeleridir: ilk olarak, bu bölümde çoğu kavramı tanıtmak için basit bir dosya sistemi (vsfs) ve ardından bunların nasıl farklılaşabileceklerini anlamak için gerçek dosya sistemleri üzerine bir dizi çalışma, uygulama.

ÖNEMLİ NOKTA: BASİT BİR DOSYA SİSTEMİ NASIL UYGULANIR
Basit bir dosya sistemini nasıl oluşturabiliriz? Diskte hangi yapılara ihtiyaç var? Neyi takip etmeleri gerekiyor? Bunlara nasıl erişilir?

40.1 Düşünmenin Yolu

Dosya sistemlerini düşünmek için genellikle onların iki farklı yönünü düşünmenizi öneririz; Bu yönlerin her ikisini de anlarsanız, muhtemelen dosya sisteminin temel olarak nasıl çalıştığını anlarsınız.

Birincisi, dosya sisteminin **veri yapılarıdır(data structures)**. Başka bir deyişle, ne Dosya sistemi tarafından verilerini ve meta verilerini düzenlemek için disk üzerindeki yapı türleri kullanılıyor mu? Göreceğimiz ilk dosya sistemleri (aşağıdaki vsf'ler dahil), blok dizileri veya diğer nesneler gibi basit yapılar.

DİPNOT:DOSYA SİSTEMLERİNİN ZİHİNSEL MODELLERİ

Daha önce tartıştığımız gibi, gerçekten denediğiniz şey zihinsel modellerdir. Sistemleri öğrenirken geliştirmek. Dosya sistemleri için, zihinsel model sonunda şu soruların yanıtlarını içermelidir: diskte ne var? Yapılar dosya sisteminin verilerini ve meta verilerini depolar mı? ne zaman olur bir işlem bir dosyayı açar mı? Bir işlem sırasında hangi disk üzerindeki yapılara erişilir? Okumak mı yazmak mı? Zihinsel modeliniz üzerinde çalışarak ve geliştirerek, değil, neler olup bittiğine dair soyut bir anlayış geliştirin. Bazı dosya sistemi kodlarının özelliklerini anlamaya çalışmak (yine de elbette faydalıdır!).

SGI'nin XFS'si gibi daha karmaşık dosya sistemleri daha karmaşık ağaç tabanlı yapılar kullanır [S+96].

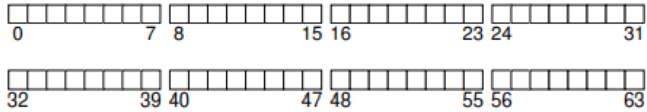
Bir dosya sisteminin ikinci yönü, **erişim yöntemleridir(access methods)**. Open(), read(), write(), vb. gibi bir işlem tarafından yapılan çağrıları yapılarına nasıl eşler? Belirli bir sistem çağrısının yürütülmesi sırasında hangi yapılar okunur? hangileri yazılır? Tüm bu adımlar ne kadar verimli bir şekilde gerçekleştirilir?

Bir dosya sisteminin veri yapılarını ve erişim yöntemlerini anlarsanız, sistemin zihniyetinin önemli bir parçası olan, sistemin gerçekten nasıl çalıştığına dair iyi bir zihinsel model geliştirmiş olursunuz. İlk uygulamamızı incelerken zihinsel modelinizi geliştirmeye çalışın.

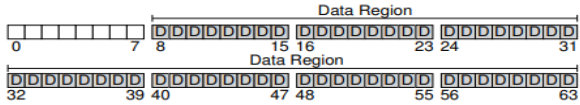
40.2 Genel Organizasyon

Şimdi vsfs dosya sisteminin veri yapılarının genel disk organizasyonunu geliştiriyoruz. Yapmamız gereken ilk şey, diski **bloklara** bölmek; basit dosya sistemleri yalnızca bir blok boyutu kullanır ve biz de burada tam olarak bunu yapacağız. Yaygın olarak kullanılan 4 KB'lik bir boyut seçelim.

Bu nedenle, dosya sistemimizi oluşturduğumuz disk bölümüne ilişkin görüşümüz basittir: her biri 4 KB boyutunda bir dizi blok. Bloklar, N 4-KB blok boyutunda bir bölmede 0'dan N - 1'e kadar adreslenir. Sadece 64 bloktan oluşan gerçekten küçük bir diskimiz olduğunu varsayalım.

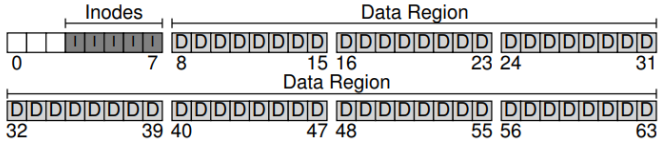


Şimdi bir dosya sistemi oluşturmak için bu bloklarda neleri depolamamız gerektiğini düşünelim. Tabii ki, akla gelen ilk şey kullanıcı verileridir. Aslında, herhangi bir dosya sistemindeki alanın çoğu kullanıcı verileridir (ve olmalıdır). Diskin kullanıcı verileri için kullandığımız bölgesine **veri bölgesi(data region)** diyelim ve yine basit olması için diskin sabit bir bölümünü bu bloklar için ayıralım, örneğin diskteki 64 bloğun son 56'sı:



Geçen bölümde (biraz) öğrendiğimiz gibi, dosya sistemi her dosya hakkındaki bilgileri izlemek zorundadır. Bu bilgi, **meta verinin(metadata)** önemli bir parçasıdır ve hangi veri bloklarının (veri bölgesinde) bir dosya içerdiğini, dosyanın boyutunu, sahibini ve erişim haklarını, erişim ve değişiklik zamanlarını ve diğer benzer türden bilgileri izler. Bu bilgiyi depolamak için, dosya sistemleri genellikle **inode** adı verilen bir yapıya sahiptir (aşağıda inode'lar hakkında daha fazla bilgi edineceğiz).

Düğümleri barındırmak için diskte onlar için de biraz yer ayırmamız gerekecek. Diskin bu bölümüne, yalnızca bir dizi disk içi düğüm içeren **inode tablosu(inode table)** diyelim. Böylece, inode'lar için 64 bloğumuzun 5'ini kullandığımızı varsayarsak (şemada I'lerle gösterilmiştir):

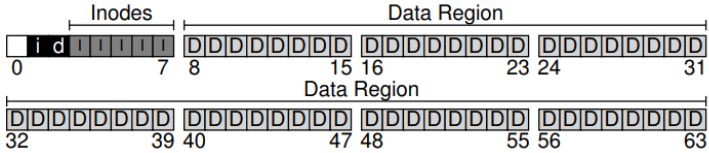


Burada düğümlerin tipik olarak o kadar büyük olmadığını, örneğin 128 veya 256 bayt olduğunu not etmeliyiz. İnode başına 256 bayt varsayarsak, 4 KB'lik bir blok 16 inode

tutabilir ve yukarıdaki dosya sistemimiz toplam 80 inode 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 kurulan aynı dosya sisteminin daha büyük bir inode tablosu tahsis edebileceğini ve böylece daha fazla dosya barındırabileceğini unutmayın.

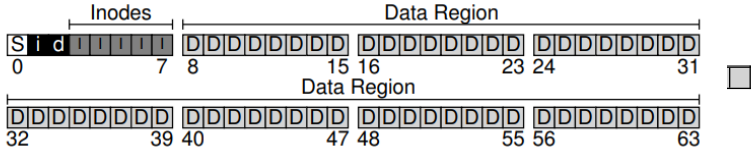
Dosya sistemimiz şimdiye kadar veri bloklarına (D) ve düğümlere (I) sahiptir, ancak birkaç şey hala eksiktir. Tahmin etmiş olabileceğiniz gibi, hala ihtiyaç duyulan bir birincil bileşen, inode'ların veya veri bloklarının boş veya tahsis edilmiş olup olmadığını izlemenin bir yoludur. Bu tür **ayırma yapıları(allocation structures)** bu nedenle herhangi bir dosya sisteminde gerekli bir öğedir.

Pek çok tahsis izleme yöntemi elbette mümkündür. Örneğin, ilk boş bloğa işaret eden ve ardından bir sonraki boş bloğa işaret eden bir **serbest liste(free list)** kullanabiliriz ve bu böyle devam eder. Bunun yerine, biri veri bölgesi (**(data)veri bitmap**) ve diğeri inode tablosu (**inode bitmap**) için bitmap olarak bilinen basit ve popüler bir yapı seçiyoruz. Bir bit eşlem bir basit yapı: her bit, karşılık gelen nesnenin/bloğun boş (0) veya kullanımda (1) olduğunu belirtmek için kullanılır. Ve böylece, bir inode bitmap (i) ve bir veri bitmap (d) içeren yeni disk üstü düzenimiz:



Bu bit eşlemler için 4 KB'lik bir bloğun tamamını kullanmanın biraz abartı olduğunu fark edebilirsiniz; böyle bir bitmap, 32K nesnelerin tahsis edilip edilmediğini izleyebilir ve yine de sadece 80 inode'umuz ve 56 veri bloğumuz var. Ancak, basitlik için bu bitmaplerin her biri için 4 KB'lik bir bloğun tamamını kullanıyoruz.

Dikkatli okuyucu (yani hala uyanık olan okuyucu), çok basit dosya sistemimizin disk üzerindeki yapısının tasarımında bir blok kaldığını fark etmiş olabilir. Bunu, aşağıdaki şemada S ile gösterilen süper blok için ayırdık. **Süper blok(super block)**, dosya sisteminde kaç tane inode ve veri bloğu olduğu (bu örnekte sırasıyla 80 ve 56), inode tablosunun nerede başladığı (blok 3) vb. dahil olmak üzere bu belirli dosya sistemi hakkında bilgiler içerir. Dosya sistemi türünü (bu durumda vsfs) tanımlamak için büyük olasılıkla sihirli bir sayı da içerecektir.



Bu nedenle, bir dosya sistemini kurarken, işletim sistemi çeşitli parametreleri başlatmak için önce süper bloğu okuyacak ve ardından birimi dosya sistemi ağacına ekleyecektir. Birim içindeki dosyalara erişildiğinde, sistem böylece gerekli disk üstü yapıları tam olarak nerede arayacağını bilecektir.

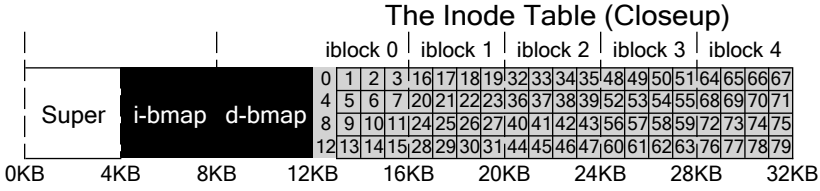
40.3 Dosya Organizasyonu: Inode

Bir dosya sisteminin en önemli disk üstü yapılarından biri **inode**'dur; hemen hemen tüm dosya sistemleri buna benzer bir yapıya sahiptir. inode adı **(index node)indeks düğümünün** kısaltmasıdır, UNIX'te [RT74] ve muhtemelen daha önceki sistemlerde kendisine verilen tarihsel addır, çünkü bu düğümler orijinal olarak bir dizide düzenlenmiştir ve dizi belirli bir düğüme erişirken dizine eklenir.

KENARA: VERİ YAPISI — INODE

İnode, birçok dosya sisteminde, belirli bir dosyanın uzunluğu, izinleri ve onu oluşturan blokların konumu gibi meta verileri tutan yapıyı tanımlamak için kullanılan genel addır. İsim en azından UNIX'e kadar gider (ve daha eski sistemler değilse muhtemelen Multics'e kadar uzanır); inode numarası, o sayının inode'unu bulmak için bir disk üzerindeki inode dizisini indekslemek için kullanıldığından, **index node**'un kısaltmasıdır. Göreceğimiz gibi, inode tasarımı, dosya sistemi tasarımının önemli bir parçasıdır. Modern sistemlerin çoğu, izledikleri her dosya için buna benzer bir yapıya sahiptir, ancak bunlara farklı adlar verebilir (dnodes, fnodes, vb.).

Her inode, daha önce dosyanın **alt düzey adı(low-level name)** olarak adlandırdığımız bir sayıyla (**i-numarası(i number)** olarak adlandırılır) dolaylı olarak anılır. vsfs'de (ve diğer basit dosya sistemlerinde), bir i-numarası verildiğinde, karşılık gelen inode'un diskte nerede olduğunu doğrudan hesaplayabilmeniz gerekir. Örneğin, yukarıdaki gibi vsfs'nin inode tablosunu alın: 20 KB boyutunda (beş 4 KB blok) ve böylece 80 inode'dan oluşur (her inode'un 256 bayt olduğunu varsayarsak); ayrıca, inode bölgesinin 12 KB'de başladığını varsayalım (yani, süper blok 0 KB'de başlar, inode bitmap 4KB adresindedir, veri bitmap 8 KB'dedir ve dolayısıyla inode tablosu hemen sonra gelir). Böylece vsfs'de, dosya sistemi bölümünün başlangıcı için aşağıdaki düzene sahibiz (yakından görünümde):



32 numaralı inode'u okumak için, dosya sistemi önce inode bölgesine (32 · sizeof (inode) veya 8192) ofseti hesaplar, bunu diskteki inode tablosunun başlangıç adresine ekler (inodeStartAddr = 12KB) ve böylece istenen inode bloğunun doğru bayt adresine ulaşır: 20KB. Disklerin bayt adreslenebilir olmadığını, bunun yerine genellikle 512 bayt olan çok sayıda adreslenebilir sektörden oluştuğunu hatırlayın. Böylece, inode 32'yi içeren inode bloğunu getirmek için, dosya sistemi istenen inode bloğunu getirmek için 20x1024 veya 40 sektörüne bir okuma yayınlayacaktır. Daha genel olarak, inode bloğunun sektör adres sektörü aşağıdaki gibi hesaplanabilir.

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

Her inode'un içinde bir dosya hakkında ihtiyacınız olan bilgilerin neredeyse tamamı bulunur: türü (ör. normal dosya, dizin vb.), boyutu, kendisine tahsis edilen blok sayısı, koruma bilgisi (dosyanın kime ait olduğu gibi) ilave olarak

Boyut	İsim	Bu inode alanı ne için?
2	mode	bu dosya okunabilir/yazılabilir/çalıştırılabilir 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 dosya ne zaman oluşturuldu?
4	mtime	bu dosya en son ne zaman değiştirildi?
4	dtime	bu inode ne zaman silindi?
2	gid	Bu dosya hangi gruba ait?
2	links count	bu dosyaya giden kaç sabit bağlantı var?
4	blocks	Bu dosyaya kaç blok tahsis edilmiştir?
4	flags	ext2 bu inode'u nasıl kullanmalıdır?
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 denir

Figure 40.1: **Simplified Ext2 Inode**

dosyanın ne zaman oluşturulduğu, değiştirildiği veya en son erişildiği gibi bazı zaman bilgileri ve ayrıca veri bloklarının diskte nerede bulunduğuna ilişkin bilgiler (örneğin, bir tür işaretçiler). Bir dosya hakkındaki tüm bu bilgilere **meta veri(metadata)** diyoruz; aslında, dosya sistemi içindeki saf kullanıcı verileri olmayan herhangi bir bilgiye genellikle bu şekilde atıfta bulunulur. Bir ext2'den örnek inode [P09] Şekil 40.11'de gösterilmiştir.

İnode tasarımındaki en önemli kararlardan biri, veri bloklarının nerede olduğunu nasıl ifade ettiğidir. Basit bir yaklaşım, inode içinde bir veya daha fazla **doğrudan işaretçiye(direct pointers)** (disk adresleri) sahip olmak olabilir; her işaretçi, dosyaya ait bir disk bloğuna atıfta bulunur. Böyle bir yaklaşım sınırlıdır: örneğin, gerçekten büyük bir dosyaya sahip olmak istiyorsanız (örneğin, blok boyutunun inode'daki doğrudan işaretçi sayısıyla çarpımından daha büyük), şansınız kalmaz.

Çok Düzeyl Dizin

Daha büyük dosyaları desteklemek için, dosya sistemi tasarımcıları inode'lar içinde farklı yapılar oluşturmak zorunda kalmıştır. Yaygın fikir, dolaylı işaretçi 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 yuvası onu gösterecek şekilde ayarlanır. 4 KB bloklar ve 4 bayt disk adresleri varsayarsak, bu da 1024 işaretçi daha ekler; dosya (12 + 1024)·4K veya 4144KB olacak şekilde büyüyebilir.

Tür bilgisi dizin girişinde tutulur ve bu nedenle inode'un kendisinde bulunmaz.

İPUCU: KAPSAM TABANLI YAKLAŞIMLARI DİKKATE ALIN
Farklı bir yaklaşım, işaretçiler yerine kapsamları kullanmaktır. Kapsam, yalnızca bir disk işaretçisi artı bir uzunluktur (bloklar halinde); bu nedenle, bir dosyanın her bloğu için bir işaretçi gerektirmek yerine, tek ihtiyaç duyulan bir işaretçi ve bir dosyanın disk üzerindeki konumunu belirtmek için bir uzunluktur. Bir dosya tahsis edilirken diskte bitişik bir boş alan bulmakta sorun yaşanabileceğinden, yalnızca tek bir kapsam sınırlayıcıdır. Bu nedenle, kapsam tabanlı dosya sistemleri genellikle birden fazla uzantıya izin verir ve böylece dosya tahsisi sırasında dosya sistemine daha fazla özgürlük verir.

iki yaklaşımı karşılaştırırken, işaretçi tabanlı yaklaşımlar en esnek olanlardır ancak dosya başına büyük miktarda meta veri kullanırlar (özellikle büyük dosyalar için). Kapsam tabanlı yaklaşımlar daha az esnektir ancak daha derli topludur; özellikle, diskte yeterli boş alan olduğunda ve dosyalar bitişik olarak düzenlenebildiğinde iyi çalışırlar (bu zaten neredeyse tüm dosya ayırma ilkelerinin hedefidir).

Şaşırtıcı olmayan bir şekilde, böyle bir yaklaşımda daha da büyük dosyaları desteklemek isteyebilirsiniz. Bunu yapmak için inode'a başka bir işaretçi ekleyin: **çift dolaylı işaretçi(double indirect pointer)**. Bu işaretçi, her biri veri bloklarına işaretçiler içeren dolaylı bloklara işaretçiler içeren bir bloğa atıfta bulunur. Dolaylı bir çift blok böylece ek 1024 · 1024 veya 1 milyon 4KB blok ile dosyaları büyütme, başka bir deyişle boyutu 4GB'ın üzerinde olan dosyaları destekleme olanağı sağlar. Yine de daha fazlasını isteyebilirsiniz ve bunun nereye gittiğini bildiğinize bahse gireriz: **üçlü dolaylı işaretçi(triple indirect pointer)**.

Genel olarak, bu dengesiz ağaca, dosya bloklarını işaret etmeye yönelik çok düzeyli dizin yaklaşımı adı verilir. On iki doğrudan işaretçinin yanı sıra hem tek hem de çift dolaylı blok içeren bir örneği inceleyelim. 4 KB'lik bir blok boyutu ve 4 baytlık işaretçileri varsayarsak, bu yapı, boyutu 4 GB'ın biraz üzerinde olan bir dosyayı barındırabilir (yani, (12 + 1024 + 1024) × 4 KB). Üçlü dolaylı bloğun eklenmesiyle ne kadar büyük bir dosyanın işlenebileceğini anlayabilir misiniz? (ipucu: oldukça büyük)

Birçok dosya sistemi, Linux ext2 [P09] ve ext3 gibi yaygın olarak kullanılan dosya sistemleri, NetApp'in WAFL'si ve orijinal UNIX dosya sistemi dahil olmak üzere çok düzeyli bir dizin kullanır. SGI XFS ve Linux ext4 gibi diğer dosya sistemleri, basit işaretçiler yerine **kapsamları(extents)** kullanır; kapsam tabanlı şemaların nasıl çalıştığına ilişkin ayrıntılar için öncekine bakın (sanal bellek tartışmasındaki bölümlere benzerler).

Merak ediyor olabilirsiniz: neden böyle dengesiz bir ağaç kullanıyorsunuz? Neden farklı bir yaklaşım olmasın? Görünüşe göre, pek çok araştırmacı dosya sistemlerini ve bunların nasıl kullanıldığını inceledi ve neredeyse her seferinde on yıllar boyunca geçerli olan belirli "gerçekleri" buldular. Böyle bir bulgu, çoğu dosyanın küçük olmasıdır. Bu dengesiz tasarım, gerçeklik; çoğu dosya gerçekten küçükse, bu durum için optimize etmek mantıklıdır. Böylece, az sayıda doğrudan işaretçiyle (12 tipik bir sayıdır), bir inode.

Most files are small	~2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer!!br0ken!!

Figure 40.2: **Dosya Sistemi Ölçüm Özeti**

daha büyük dosyalar için bir (veya daha fazla) dolaylı bloğa ihtiyaç duyarak doğrudan 48 KB veriye işaret edebilir. Bkz. Agrawal ve diğerleri. yakın tarihli bir çalışma için al [A+07]; Figür 40.2 bu sonuçları özetlemektedir.

Elbette, inode tasarımı alanında başka birçok olasılık mevcuttur; ne de olsa inode sadece bir veri yapısıdır ve ilgili bilgiyi depolayan ve etkili bir şekilde sorgulayabilen herhangi bir veri yapısı yeterlidir. Dosya sistemi yazılımı kolayca değiştirilebildiğinden, iş yükleri veya teknolojiler değişirse farklı tasarımları keşfetmeye istekli olmalısınız.

40.4 Dizin Organizasyonu

vsfs'de (birçok dosya sisteminde olduğu gibi), dizinlerin basit bir organizasyonu vardır; bir dizin temelde sadece (giriş adı, inode numarası) çiftlerinin bir listesini içerir. Belirli bir dizindeki her dosya veya dizin için, dizinin veri blok(lar)ında bir dizi ve bir sayı vardır. Her dize için bir uzunluk da olabilir (değişken boyutlu adlar varsayılarak).

Örneğin, bir dizin dizininin (inode numarası 5) içinde sırasıyla 12, 13 ve 24 inode numaraları olan üç dosya (foo, bar ve foobar oldukça uzun bir isimdir) olduğunu varsayalım. dir için diskteki veriler şöyle görünebilir:

```
inum | reclen | 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 girişin bir inode numarası, kayıt uzunluğu (ad için toplam bayt artı kalan boşluk), dize uzunluğu (ismin gerçek uzunluğu) ve son olarak girişin adı vardır. Her dizinin fazladan iki girişi olduğunu unutmayın, . "nokta" ve .. "nokta-nokta"; nokta dizini yalnızca geçerli dizindir (bu örnekte dir), nokta-nokta ise üst dizindir (bu durumda kök).

Bir dosyayı silmek (ör. unlink()'i çağırmak) dizinin ortasında boş bir alan bırakabilir ve dolayısıyla bunu da işaretlemenin bir yolu olmalıdır (ör. sıfır gibi ayrılmış bir inode numarası ile). Böyle bir silme, kayıt uzunluğunun kullanılmasının bir nedenidir: yeni bir giriş eski, daha büyük bir girişi yeniden kullanabilir ve bu nedenle içinde fazladan boşluk olabilir.

DİPNOT: BAĞLANTI TABANLI YAKLAŞIMLAR

Düğüm tasarımında daha basit bir yaklaşım, **bağlantılı bir liste(linked list)** kullanmaktır. Bu nedenle, bir inode içinde, birden çok işaretçiye sahip olmak yerine, dosyanın ilk bloğunu işaret etmek için yalnızca birine ihtiyacınız vardır. Daha büyük dosyaları işlemek için, o veri bloğunun sonuna başka bir işaretçi ekleyin ve bu şekilde devam ederek büyük dosyaları destekleyebilirsiniz.

Tahmin edebileceğiniz gibi, bağlantılı dosya ayırma bazı iş yükleri için yetersiz performans gösterir; örneğin bir dosyanın son bloğunu okumayı veya sadece rasgele erişim yapmayı düşünün. Bu nedenle, bağlantılı tahsisin daha iyi çalışmasını sağlamak için, bazı sistemler sonraki işaretçileri veri bloklarının kendileriyle birlikte depolamak yerine bellek içi bir bağlantı bilgileri tablosu tutacaktır. Tablo, bir veri bloğu D'nin adresi tarafından indekslenir; bir girdinin içeriği basitçe D'nin sonraki işaretçisidir, yani D'yi izleyen bir dosyadaki sonraki bloğun adresidir. Orada da bir boş değer (dosya sonunu gösteren) veya belirtmek için başka bir işaretleyici olabilir. belirli bir bloğun ücretsiz olması. Böyle bir sonraki işaretçiler tablosuna sahip olmak, bağlantılı bir ayırma şemasının, yalnızca istenen bloğu bulmak için (bellekteki) tabloyu tarayarak ve ardından doğrudan (diskte) ona erişerek, rastgele dosya erişimlerini etkili bir şekilde yapabilmesini sağlar.

Böyle bir tablo tanıdık geliyor mu? Açıkladığımız, **dosya ayırma tablosu(file allocation table)** veya **FAT** dosya sistemi olarak bilinen şeyin temel yapısıdır. Evet, NTFS'den [C94] önceki bu klasik eski Windows dosya sistemi, basit bir bağlantılı tabanlı ayırma şemasına dayalıdır. Standart bir UNIX dosya sisteminden başka farklılıklar da vardır; örneğin, kendi başına düğümler yoktur, bunun yerine bir dosya hakkında meta verileri depolayan ve doğrudan söz konusu dosyanın ilk bloğuna atıfta bulunan dizin girişleri vardır, bu da sabit bağlantıların oluşturulmasını imkansız hale getirir. Daha fazla zarif olmayan ayrıntı için Brouwer'a [B02] bakın.

Dizinlerin tam olarak nerede saklandığını merak ediyor olabilirsiniz. Genellikle, dosya sistemleri dizinleri özel bir dosya türü olarak ele alır. Bu nedenle, bir dizinin inode tablosunda bir yerde bir inode'u vardır ("normal dosya" yerine "dizin" olarak işaretlenmiş inode'un tür alanı ile). Dizin, inode tarafından işaret edilen veri bloklarına (ve belki de dolaylı bloklara) sahiptir; bu veri blokları, basit dosya sistemimizin veri bloğu bölgesinde yaşar. Disk üzerindeki yapımız bu nedenle değişmeden kalır.

Ayrıca, bu tür bilgileri depolamanın tek yolunun bu basit doğrusal dizin girişleri listesi olmadığını da not etmeliyiz. Daha önce olduğu gibi, herhangi bir veri yapısı mümkündür. Örneğin, XFS [S+96], dizinleri B-ağacı biçiminde saklayarak, dosya oluşturma işlemlerini (oluşturulmadan önce bir dosya adının kullanılmadığından emin olmak zorunda olan) kendi içinde taranması gereken basit listelere sahip sistemlerden daha hızlı hale getirir. bütünlük

DİPNOT: BOŞ ALAN YÖNETİMİ

Boş alanı yönetmenin birçok yolu vardır; bit eşlemler yalnızca bir yoldur. Bazı eski dosya sistemleri, süper bloktaki tek bir işaretçinin ilk boş bloğa işaret etmek için tutulduğu **ücretsiz listeler(free lists)** kullandı; bu bloğun içinde bir sonraki boş işaretçi tutuldu, böylece sistemin boş blokları arasında bir liste oluşturuldu. Bir bloğa ihtiyaç duyulduğunda, ana blok kullanıldı ve liste buna göre güncellendi.

Modern dosya sistemleri daha karmaşık veri yapıları kullanır. Örneğin, SGI'nin XFS'si [S+96], hangi disk parçalarının boş olduğunu kompakt bir şekilde temsil etmek için bir tür **B-ağacı(B-tree)** kullanır. Herhangi bir veri yapısında olduğu gibi, farklı zaman-uzay değiş tokuşları mümkündür.

40.5 Boş Alan Yönetimi

Bir dosya sistemi, yeni bir dosya veya dizin tahsis edildiğinde ona yer bulabilmek için hangi düğümlerin ve veri bloklarının boş olduğunu ve hangilerinin olmadığını izlemelidir. 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 gerekecek. Böylece dosya sistemi, bitmap üzerinden ücretsiz bir inode arayacak ve onu dosyaya tahsis edecektir; dosya sisteminin inode'u kullanıldığı gibi (1 ile) işaretlemesi ve sonunda disk üzerindeki bit eşlemi doğru bilgilerle güncellemesi gerekir. Bir veri bloğu tahsis edildiğinde benzer bir dizi faaliyet gerçekleşir.

Yeni bir dosya için veri blokları tahsis edilirken başka hususlar da devreye girebilir. Örneğin, ext2 ve ext3 gibi bazı Linux dosya sistemleri, yeni bir dosya oluşturulduğunda ücretsiz olan ve veri bloklarına ihtiyaç duyan bir dizi blok (mesela 8) arayacaktır; böyle bir boş blok dizisi bularak ve sonra bunları yeni oluşturulan dosyaya tahsis ederek, dosya sistemi dosyanın bir kısmının diskte bitişik olmasını garanti ederek performansı artırır. Bu tür bir **ön tahsis(pre-allocation)** politikası, bu nedenle, veri blokları için alan tahsis edilirken yaygın olarak kullanılan bir buluşsal yöntemdir.

40.6 Erişim Yolları: Okuma ve Yazma

Artık dosyaların ve dizinlerin diskte nasıl saklandığına dair bir fikrimiz olduğuna göre, bir dosyayı okuma veya yazma etkinliği sırasında işlem akışını takip edebilmeliyiz. Bu **erişim yolunda(access point)** ne olduğunu anlamak, bir dosya sisteminin nasıl çalıştığını anlamanın ikinci anahtarıdır; dikkat etmek!.
Aşağıdaki örnekler için, dosya sisteminin değiştirildiğini varsayalım. monte edilir ve böylece süper blok zaten bellekte olur. Her şey başka (yani düğümler, dizinler) hala diskte.

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

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

Diskten Dosya Okumak

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 yaptığınızda, dosya sisteminin dosya hakkında bazı temel bilgileri (izin bilgileri, dosya boyutu) elde etmek için önce dosya çubuğu için inode'u bulması gerekir. , vb.). Bunu yapmak için, dosya sistemi inode'u bulabilmelidir, ancak şu anda sahip olduğu tek şey tam yol adıdır. Dosya sistemi yol adını **geçmeli(traverse)** ve böylece istenen inode'u bulmalıdır.

Tüm geçişler dosya sisteminin kökünde, kısaca / olarak adlandırılan **kök dizinde(root directory)** başlar. Böylece, FS'nin diskten okuyacağı ilk şey, kök dizinin inode'udur. Ama bu inode nerede? Bir inode bulmak için i sayısını bilmeliyiz. Genellikle, bir dosyanın veya dizinin i numarasını üst dizininde buluruz; kökün ebeveyni yoktur (tanım gereği). Bu nedenle, kök inode sayısı "iyi bilinmelidir"; FS, dosya sistemi monte edildiğinde bunun ne olduğunu bilmelidir. Çoğu UNIX dosya sisteminde, kök inode numarası 2'dir. Bu nedenle, işleme başlamak için FS, inode numarası 2'yi (ilk inode bloğu) içeren bloğu okur.

İnode okunduktan sonra FS, kök dizinin içeriğini içeren veri bloklarına işaretçiler bulmak için onun içine bakabilir. Böylece FS, dizini okumak için bu disk üzerindeki işaretçileri kullanacak ve bu durumda foo için bir giriş arayacak. Bir veya daha fazla dizin veri bloğunu okuyarak, foo için girişi bulacaktır; FS bir kez bulunduktan sonra, daha sonra ihtiyaç duyacağı foo inode sayısını da bulmuş olacaktır (varsayalım ki 44).

Bir sonraki adım, istenen inode bulunana kadar yol adını tekrar tekrar geçmektir. Bu örnekte, FS, şunu içeren bloğu okur:

DİPNOT: OKUMALAR TAHSİS YAPILARINA ERİŞMEZ

Bit eşlemler gibi ayırma yapılarının birçok öğrencinin kafasını karıştırdığını gördük. Özellikle çoğu kişi, bir dosyayı sadece okurken ve herhangi bir yeni blok tahsis etmediğinizde, bitmap'e hala başvurulacağını düşünür. Bu doğru değil! Bit eşlemler gibi ayırma yapılarına yalnızca ayırma gerektiğinde erişilir. Düğümler, dizinler ve dolaylı bloklar, bir okuma talebini tamamlamak için ihtiyaç duydukları tüm bilgilere sahiptir; inode zaten ona işaret ettiğinde bir bloğun tahsis edildiğinden emin olmaya gerek yoktu

foo'nun inode'u ve ardından dizin verileri, son olarak bar'ın inode sayısını bulmak. open() işleminin son adımı, çubuğun inode'unu belleğe okumaktır; FS daha sonra son bir izin kontrolü yapar, işlem başına açık dosya tablosunda bu işlem için bir dosya tanıtıcısı tahsis eder ve bunu kullanıcıya döndürür.

Açıldıktan sonra, program dosyadan okumak için bir read() sistem çağrısı yapabilir. İlk okuma (lseek() çağrılmadığı sürece 0 uzaklığında), böyle bir bloğun konumunu bulmak için inode'a başvurarak dosyanın ilk bloğunda okuyacaktır; ayrıca inode'u yeni bir son erişim zamanı ile güncelleyebilir. Okuma, bu dosya tanıtıcı için bellek içi açık dosya tablosunu daha da günceller, dosya ofsetini bir sonraki okuma ikinci dosya bloğunu okuyacak şekilde günceller, vb.

Bir noktada, dosya kapatılacak. Burada yapılacak çok daha az iş var; Açıkçası, dosya tanıtıcının yeniden konumlandırılması gerekir, ancak şimdilik FS'nin gerçekten yapması gereken tek şey bu. Disk I/O'leri gerçekleşmez.

Tüm bu sürecin bir tasviri Şekil 40.3'te (sayfa 11) bulunur; şekilde aşağı doğru zaman artar. Şekilde açık, son olarak dosyanın inode'unu bulmak için çok sayıda okuma yapılmasına 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 yazma ile güncellemesini gerektirir. Biraz zaman ayırın ve neler olduğunu anlayın.

Açık tarafından üretilen I/O miktarının yol adının uzunluğuyla orantılı olduğuna da dikkat edin. Yoldaki her ek dizin için, verilerini olduğu kadar inode'unu 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 istenen girişi bulmak için birçok veri bloğunu okumamız gerekebilir. Evet, bir dosyayı okurken hayat oldukça kötüye gidebilir; birazdan öğreneceğiniz gibi, bir dosyayı yazmak (ve özellikle yeni bir tane oluşturmak) daha da kötüdür.

Diske Dosya Yazma

Bir dosyaya yazmak da benzer bir işlemdir. İlk olarak, dosya açılmalıdır (yukarıdaki gibi). Ardından uygulama, dosyayı yeni içeriklerle güncellemek için write() çağrıları yapabilir. Son olarak dosya kapatılır.

Okumanın aksine, dosyaya yazmak da bir blok **tahsis edebilir(allocate)** (örneğin, bloğun üzerine yazılmadığı sürece). Yeni bir dosya yazarken, her yazma işlemi yalnızca verileri diske yazmakla kalmaz, aynı zamanda önce karar vermek zorundadır.

	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 write**	**read**	**read**	**write**		
write()	**read write**				**read write**				**write**	
write()	**read write**				**read write**					**write**
write()	**read write**				**read write**					**write**

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

dosyaya tahsis etmek ve böylece diskin diğer yapılarını buna göre güncellemek için hangi blok (örneğin, veri bitmap ve inode). Böylece, bir dosyaya her yazma işlemi mantıksal olarak beş I/O üretir: biri veri bit eşlemini okumak için (bu daha sonra yeni tahsis edilen bloğu kullanılmış olarak işaretlemek için güncellenir), biri bit eşlemi yazmak için (yeni durumunu diske yansıtmak için) , inode'u okumak ve sonra yazmak için iki tane daha (yeni bloğun konumu ile güncellenir) ve son olarak gerçek bloğun kendisini yazmak için bir tane daha.

Dosya oluşturma gibi basit ve yaygın bir işlem düşünüldüğünde, yazma trafiğinin miktarı daha da kötüdür. Bir dosya oluşturmak için, dosya sistemi yalnızca bir inode tahsis etmekle kalmamalı, aynı zamanda yeni dosyayı içeren dizinde yer tahsis etmelidir. Bunu yapmak için toplam I/O trafiği miktarı oldukça yüksektir: biri inode bitmap'e okur (boş bir inode bulmak için), biri inode bitmap'e yazar (tahsis edilmiş olarak işaretlemek için), biri yeni inode'a yazar (başlatmak için), biri dizinin verilerine (dosyanın üst düzey adını inode numarasına bağlamak için) ve biri güncellemek için dizin inode'una okuma ve yazma. Dizinin yeni girişi karşılamak için büyümesi gerekiyorsa, ek I/O'lere (yani, veri bitmap'ine ve yeni dizin bloğuna) da ihtiyaç duyulacaktır. Bütün bunlar sadece bir dosya oluşturmak için!

ÖNEMLİ NOKTA: DOSYA SİSTEMİ G/Ç MALİYETLERİ NASIL AZALTILIR

Bir dosyayı açma, okuma veya yazma gibi en basit işlemler bile diske dağılmış çok sayıda G/Ç işlemi gerektirir. Bir dosya sistemi, bu kadar çok işlemin yüksek maliyetlerini azaltmak için ne yapabilir I/Os?

/foo/bar dosyasının oluşturulduğu ve ona üç bloğun yazıldığı belirli bir örneğe bakalım. Şekil 40.4 (sayfa 13), open() (dosyayı oluşturan) sırasında ve üç 4KB yazmanın her biri sırasında neler olduğunu gösterir. Şekilde, diske yapılan okumalar ve yazmalar hangi sistem çağrısının neden olduğu altında gruplandırılmış ve bunların gerçekleşebileceği kabaca sıralama şeklin yukarıdan aşağıya doğru devam etmektedir. Dosyayı oluşturmanın ne kadar zahmetli olduğunu görebilirsiniz: Bu durumda 10 G/Ç, yol adını yürümek ve son olarak dosyayı oluşturmak. Ayrıca her ayırma yazma maliyetinin 5 I/O olduğunu da görebilirsiniz: inode'u okumak ve güncellemek için bir çift, veri bitmap'i okumak ve güncellemek için başka bir çift ve son olarak verilerin kendisinin yazılması. Bir dosya sistemi bunların herhangi birini makul bir verimlilikle nasıl başarabilir

40.7 Önbelleğe Alma ve Arabelleğe Alma

Yukarıdaki örneklerin gösterdiği gibi, dosyaları okumak ve yazmak pahalı olabilir ve (yavaş) diskte birçok I/O'ye neden olabilir. Açıkça büyük bir performans sorunu olabilecek bir sorunu çözmek için çoğu dosya sistemi, önemli blokları önbelleğe almak için agresif bir şekilde sistem belleğini (DRAM) kullanır.

Yukarıdaki açık örneği hayal edin: önbelleğe alma olmadan, açık olan her dosya, dizin hiyerarşisindeki her seviye için en az iki okuma gerektirir (biri söz konusu dizinin inode'unu okumak için ve en az biri verilerini okumak için). Uzun bir yol adıyla (örneğin, /1/2/3/ ... /100/file.txt), dosya sistemi dosyayı açmak için kelimenin tam anlamıyla yüzlerce okuma gerçekleştirir!

Erken dosya sistemleri bu nedenle popüler blokları tutmak için **sabit boyutlu bir önbellek(fixed-size cache)** tanıttı. Sanal bellek tartışmamızda olduğu gibi, **LRU** ve farklı değişkenler gibi stratejiler hangi blokların önbellekte tutulacağına karar verir. Bu sabit boyutlu önbellek, genellikle önyükleme sırasında toplam belleğin kabaca %10'u kadar tahsis edilir.

Bununla birlikte, belleğin bu **statik bölümlenmesi(static partitioning)** israfa neden olabilir; ya dosya sistemi belirli bir zamanda belleğin %10'una ihtiyaç duymazsa? Yukarıda açıklanan sabit boyutlu yaklaşımla, dosya önbelleğindeki kullanılmayan sayfalar başka bir kullanım için yeniden tasarlanamaz ve bu nedenle boşa gider.

Modern sistemler ise aksine, **dinamik bir bölümleme(dynamic partitioning)** yaklaşımı kullanır. Spesifik olarak, birçok modern işletim sistemi, sanal bellek sayfalarını ve dosya sistemi sayfalarını **birleşik bir sayfa önbelleğine(unified page cache)** [S00] entegre eder. Bu şekilde, belirli bir zamanda hangisinin daha fazla belleğe ihtiyaç duyduğuna bağlı olarak bellek, sanal bellek ve dosya sistemi arasında daha esnek bir şekilde tahsis edilebilir.

Şimdi önbelleğe alma ile dosya açma örneğini hayal edin. İlk açma, dizin inode'unda ve verilerinde okunacak çok fazla I/O trafiği oluşturabilir, ancak alt-

İPUCU: STATİK VS. DİNAMİK BÖLÜMLEME

Bir kaynağı farklı istemciler/kullanıcılar arasında bölüştürürken, **statik bölümleme(static partitioning)** veya **dinamik bölümleme(dynamic partitioning)** kullanabilirsiniz. Statik yaklaşım, kaynağı basitçe bir kez sabit oranlara böler; örneğin, iki olası bellek kullanıcısı varsa, belleğin sabit bir kısmını bir kullanıcıya, geri kalanını diğerine verebilirsiniz. Dinamik yaklaşım, zaman içinde farklı miktarlarda kaynak vererek daha esnektir; 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 bölümleme, her kullanıcının kaynaktan bir miktar pay almasını sağlar, genellikle daha öngörülebilir bir performans sunar ve uygulanması genellikle daha kolaydır. Dinamik bölümleme daha iyi bir kullanım sağlayabilir (kaynağa aç kullanıcıların boşta kalan kaynakları tüketmesine izin vererek), ancak uygulanması daha karmaşık olabilir ve boştaki kaynakları başkaları tarafından tüketilen ve daha sonra uzun zaman alan kullanıcılar için daha kötü performansa yol açabilir. gerektiğinde geri almak için. Çoğu zaman olduğu gibi, en iyi yöntem yoktur; bunun yerine, eldeki sorunu düşünmeli ve hangi yaklaşımın en uygun olduğuna karar vermelisiniz.

aynı dosyanın (veya aynı dizindeki dosyaların) sıralı dosya açılışları çoğunlukla önbelleğe çarpar ve bu nedenle I/O gerekmez.

Önbelleğe almanın yazma işlemleri üzerindeki etkisini de ele alalım. Yeterince büyük bir önbellekle okuma I/O'den tamamen kaçınılabilirken, yazma trafiğinin kalıcı olması için diske gitmesi gerekir. Bu nedenle, bir önbellek yazma trafiğinde okumalar için yaptığıyla aynı türden bir filtre işlevi görmez. Bununla birlikte, **(write buffering)yazma arabelleğe** almanın (bazen adlandırıldığı şekliyle) kesinlikle bir dizi performans avantajı vardır. İlk olarak, dosya sistemi, yazma işlemlerini geciktirerek bazı güncellemeleri daha küçük bir I/O **kümesi(batch)** halinde toplayabilir; örneğin, bir inode bit eşlemi bir dosya oluşturulduğunda güncellenirse ve birkaç dakika sonra başka bir dosya oluşturulduğunda güncellenirse, dosya sistemi ilk güncellemeden sonra yazmayı geciktirerek bir I/O kaydeder. İkincisi, birkaç yazma işlemini bellekte arabelleğe alarak, sistem daha sonra sonraki I/O'leri programlayabilir ve böylece performansı artırabilir. Son olarak, bazı yazma işlemleri geciktirilerek tamamen önlenir; örneğin, bir uygulama bir dosya oluşturur ve ardından onu silerse, dosya oluşturma işlemini diske yansıtmak için yazma işlemlerini geciktirmek, bunları tamamen **önler(avoids)**. Bu durumda tembellik (blokları diske yazarken) bir erdemdir.

Yukarıdaki nedenlerden dolayı, çoğu modern dosya sistemi arabelleği, beş ila otuz saniye arasında herhangi bir süre boyunca belleğe yazar ve bu da başka bir ödünleşimi temsil eder: güncellemeler diske yayılmadan önce sistem çökerse, güncellemeler kayıp; ancak, yazma işlemlerini bellekte daha uzun süre tutarak, gruplama, zamanlama ve hatta yazma işlemlerinden kaçınarak performans iyileştirilebilir.

İPUCU: DAYANIKLILIK/PERFORMANS ÖDÜLÜNÜ ANLAYIN
Depolama sistemleri genellikle kullanıcılara bir dayanıklılık/performans dengesi sunar. Kullanıcı, yazılan verilerin hemen dayanıklı olmasını istiyorsa, sistem yeni yazılan verileri diske işlemek için tüm çabayı göstermelidir ve bu nedenle yazma yavaştır (ancak güvenlidir). Bununla birlikte, kullanıcı küçük bir veri kaybını tolere edebilirse, sistem yazmaları bir süre bellekte ara belleğe alabilir ve daha sonra diske (arka planda) yazabilir. Bunu yapmak, yazma işlemlerinin hızlı bir şekilde tamamlanmış gibi görünmesini sağlayarak algılanan performansı artırır; ancak, bir kilitlenme meydana gelirse, henüz diske kaydedilmemiş yazmalar kaybedilecek ve dolayısıyla değiş tokuş yapılacaktır. Bu değiş tokuşu doğru bir şekilde nasıl yapacağınızı anlamak için, depolama sistemini kullanan uygulamanın ne gerektirdiğini anlamak en iyisidir; örneğin, web tarayıcınız tarafından indirilen son birkaç görüntünün kaybedilmesi tolere edilebilirken, banka hesabınıza para ekleyen bir veritabanı işleminin bir kısmının kaybedilmesi daha az tolere edilebilir olabilir. Tabii zengin değilseniz; bu durumda, son kuruşuna kadar istiflemeyi neden bu kadar önemsiyorsunuz?

Bazı uygulamalar (veritabanları gibi) bu takastan hoşlanmaz. Bu nedenle, yazma arabelleğinden kaynaklanan beklenmeyen veri kaybını önlemek için, fsync()'i çağırarak, önbellek çevresinde çalışan **doğrudan(direct)** I/O arabirimlerini kullanarak veya **ham disk(raw disk)** arabirimini kullanarak ve dosya sisteminden kaçınarak, basitçe diske yazmaya zorlarlar. tamamen 2. Çoğu uygulama, dosya sistemi tarafından yapılan değiş tokuşlarla yaşarken, varsayılan ayar tatmin edici olmazsa, sistemin istediğinizi yapmasını sağlamak için yeterli kontrol vardır.

40.8 Özet

Bir dosya sistemi oluşturmak için gereken temel makineleri gördük. Genellikle inode adı verilen bir yapıda depolanan her dosya (meta veri) hakkında bazı bilgilerin olması gerekir. Dizinler, ad → inode numarası eşlemelerini depolayan belirli bir dosya türüdür. Ve başka yapılara da ihtiyaç var; örneğin dosya sistemleri, hangi inode'ların veya veri bloklarının serbest veya tahsis edilmiş olduğunu izlemek için genellikle bitmap gibi bir yapı kullanır.

Dosya sistemi tasarımının müthiş yönü, özgürlüğüdür; Önümüzdeki bölümlerde keşfedeceğimiz dosya sistemlerinin her biri, dosya sisteminin bazı yönlerini en iyi duruma getirmek için bu özgürlükten yararlanır. Keşfedilmemiş bıraktığımız birçok politika kararı da var. Örneğin yeni bir dosya oluşturulduğunda diskte nereye yerleştirilmelidir? Bu politika ve diğerleri de gelecek bölümlerin konusu olacaktır. yoksa onlar mı?[3]

[2]Take a database class to learn more about old-school databases and their former insistence on avoiding the OS and controlling everything themselves. But watch out! Those database types are always trying to bad mouth the OS. Shame on you, database people. Shame.

[3]Cue mysterious music that gets you even more intrigued about the topic of file systems.

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`. *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/ãeb/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. FREENIX, 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)

Çeşitli işlemler gerçekleştikçe dosya sistemi durumunun nasıl değiştiğini incelemek için bu vsfs.py aracını kullanın. Dosya sistemi, yalnızca bir kök dizinle boş bir durumda başlar. Simülasyon gerçekleşirken, çeşitli işlemler gerçekleştirilir, böylece dosya sisteminin diskteki durumu yavaş yavaş değişir. Ayrıntılar için README'ye bakın.

Sorular

1. Simülatörü bazı farklı rasgele çekirdeklerle çalıştırın (örneğin 17, 18, 19, 20) ve her durum değişikliği arasında hangi işlemlerin gerçekleşmiş olması gerektiğini çözüp çözemeyeceğinize bakın.

   Yöntem seed(), random sayı üretecini başlatmak için kullanılır. Random sayı üretici, random bir sayı üretebilmek için başlamak üzere bir sayıya (bir tohum değeri) ihtiyaç duyar. Varsayılan olarak rastgele sayı üreteci geçerli sistem saatini kullanır. Random sayı üretecinin başlangıç numarasını özelleştirmek için seed() yöntemi kullanılır.

```
random_seed(18)
print(random.random())
print("Verilen değere göre karşılık gelen sayı değeri:")
```

   Verilen kodda 18 sayısına karşılık gelen değer şu şekildedir:

```
0.18126486333322134
Verilen değere göre karşılık gelen sayı değeri:
Initial state
```

   18 sayısına karşılık gelen değer 0.1812 olarak hesaplanmıştır.

```
random_seed(19)
print(random.random())
print("Verilen değere göre karşılık gelen sayı değeri:")
```

   19 değerine karşılık gelen değer ise aşağıdaki şekildedir.

```
0.6771258268002703
Verilen değere göre karşılık gelen sayı değeri:
Initial state
```

Şekilde de görüldüğü gibi random_seed()içindeki değer arttıkça ekran çıktısındaki değer de artış göstermektedir.

2. Şimdiyi aynı, farklı rastgele çekirdekler kullanarak (21, 22, 23, 24 diyelim) -r bayrağıyla çalıştırmayı dışarıda yapın, böylece işlem gösterilirken durumu değiştirmeni tahminde bulunmayı sağlar. Hangi blokları tahsis etmeyi tercih ettikleri açısından, inode ve data-block tahsisleri hakkında ne deneyebilirsin?

   Sağlanan kod, data blok tahsisini yönetmek için bitmap sınıfı ve dosya sistemindeki bir bloğun içeriğini temsil etmek için blok sınıfı gibi, dosya sistemini ve bileşenlerini simüle etmek için kullanılan birkaç sınıfı tanımlıyor.

   Simülatörü "ters" modda ("-r" bayrağıyla) çalıştırabilir, verilen işlemlerden durum değişikliklerini tahmin edip edemeyeceğinizi görmek için durumlar yerine işlemleri yazdırabilirsiniz.

3. Şimdi dosya sistemindeki veri bloğu sayısını çok düşük sayılara indirin (iki diyelim) ve simülatörü yüz kadar istek için çalıştırın. Bu oldukça kısıtlı düzende dosya sisteminde ne tür dosyalar bulunur? Ne tür işlemler başarısız olur?

   Ekran çıktıları azalıyor. Konsol ekranında taradığı dosya sayısı azalıyor.

```
ARG seed 0
ARG numInodes 8
ARG numData 8
ARG numRequests 10
ARG reverse False
ARG printFinal False

Initial state

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

Which operation took place?
```

4. Şimdi aynısını yapın, ancak düğümlerle. Çok az düğümle ne tür işlemler başarılı olabilir? Hangisi genellikle başarısız olur? Dosya sisteminin son durumu ne olabilir?

   Dosya sayısı az olan işlemlerde başarılı olur. Dosya düğüm sayısı azalınca daha az dosya tarıyor Bu işlem sonucunda ise taranan dosya sayısı daha az olacağından dolayı diğer dosyalar hakkında daha az bilgi sahibi oluruz. Dosya çalıştığında yalnızca başlangıç kısmı çalışmış olur.