40

Dosya Sistemi Uygulaması

Bu bölümde, vsfs **Çok Basit Dosya Sistemi(the very simple file system)** 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 bulabileceğiniz 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 (ancak dosya sisteminin iyi çalıştığından emin olmak için cihaz özelliklerine bağlılık ödemek isteyeceğiz). Bir dosya sistemi oluşturmada sahip olduğumuz büyük esneklik nedeniyle, AFS'den (Andrew Dosya Sistemi) [H + 88] ZFS'ye (Sun'ın Zettabyte Dosya Sistemi) [B07] kadar birçok farklı sistem inşa edilmiştir. Tüm bu dosya sistemleri farklı veri yapılarına sahiptir ve bazı şeyleri akranlarından daha iyi veya daha kötü yapar. Bu nedenle, dosya sistemleri hakkında öğreneceğimiz yol vaka çalışmalarından geçer: ilk olarak, bu bölümde çoğu kavramı tanıtmak için basit bir dosya sistemi (vsfs) ve daha sonra pratikte nasıl farklılık gösterebileceklerini anlamak için gerçek dosya sistemleri üzerine bir dizi çalışma.

İŞIN PÜF NOKTASI: BASIT BIR DOSYA SISTEMI NASIL UYGULANIR?
Basit bir dosya sistemini nasıl kurabiliriz? Diskte hangi yapılara ihtiyaç vardır? İzlemeleri için neye ihtiyaçları var? Bunlara nasıl erişilir?

40.1 Düşünmenin Yolu

Dosya sistemleri hakkında düşünmek için, genellikle iki farklı yönünü düşünmenizi öneririz; Bu yönlerin her ikisini de anlarsanız, muhtemelen dosya sisteminin temelde nasıl çalıştığını anlamışsınızdır.

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

Kenarda Dursun:DOSYA SISTEMLERININ ZIHINSEL MODELLERI
Daha önce de tartıştığımız gibi, zihinsel modeller, sistemler hakkında bilgi edinirken gerçekten geliştirmeye çalıştığınız şeydir. Dosya sistemleri için, zihinsel modeliniz sonunda aşağıdaki gibi soruların cevaplarını içermelidir: hangi disk üzerindeki yapılar dosya sisteminin verilerini ve meta verilerini depolar? Bir işlem bir dosyayı açtığında ne olur? Okuma veya yazma sırasında hangi disk üzerindeki yapılara erişilir? Zihinsel modeliniz üzerinde çalışarak ve geliştirerek, sadece bazı dosya sistemi kodlarının özelliklerini anlamaya çalışmak yerine, neler olup bittiğine dair soyut bir anlayış geliştirirsiniz (tabii ki bu da yararlıdır!).

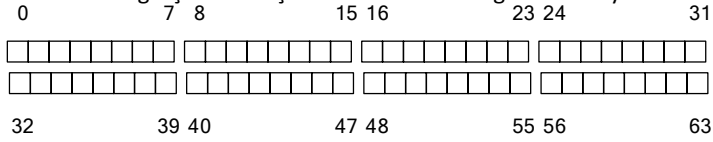
SGI'nın XFS'si gibi daha sofistike dosya sistemleri, daha karmaşık ağaç tabanlı yapılar kullanır [S + 96].

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

Bir dosya sisteminin veri yapılarını ve erişim yöntemlerini anlarsanız, sistem zihniyetinin önemli bir parçası olan gerçekten nasıl çalıştığına dair iyi bir zihinsel model geliştirmiş olursunuz. İlk uygulamamıza girerken zihinsel modelinizi geliştirmek için çalışmaya çalışın.

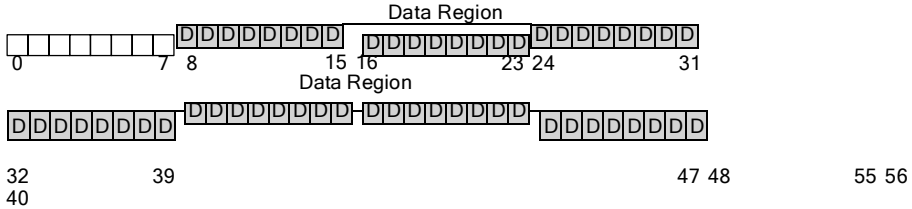
40.2 Genel Organizasyon

Şimdi vsfs dosya sisteminin veri yapılarının genel disk içi organizasyonunu geliştiriyoruz. Yapmamız gereken ilk şey, diski bloklara bölmektir; Basit dosya sistemleri sadece bir **blok(blocks)** boyutu kullanır ve burada tam olarak yapacağımız şey budur. Yaygın olarak kullanılan 4 KB'lık bir boyut seçelim. Bu nedenle, dosya sistemimizi oluşturduğumuz disk bölümü hakkındaki görüşümüz basittir: her biri 4 KB boyutunda bir dizi blok. Bloklar, N 4 KB'lık blokların boyutundaki bir bölümde, 0'dan N − 1'e kadar ad-dressed edilir. Sadece 64 bloklu gerçekten küçük bir diskimiz olduğunu varsayalım:



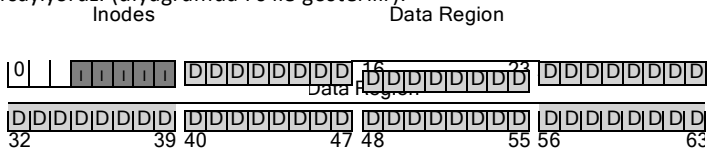
Şimdi bir dosya sistemi oluşturmak için bu bloklarda ne saklamamız gerektiğini düşünelim. Tabii ki akla gelen ilk şey kullanıcı verileridir. Aslında, herhangi bir dosya sistemindeki alanın çoğu kullanıcı verisidir (ve olmalıdır). Kullanıcı verileri için kullandığımız diskin bölgesine **veri bölgesi(data region)** diyelim ve,

yine basitlik için, diskin sabit bir bölümünü bu bloklar için ayırın, diskteki 64 bloğun son 56'sını söyleyin:



Son bölüm hakkında öğrendiğimiz gibi, dosya sistemi her dosya hakkındaki bilgileri izlemek zorundadır. Bu bilgiler **meta verilerin(metadata)** önemli bir parçasıdır ve hangi veri bloklarının bir dosyayı oluşturduğu, dosyanın boyutu, sahibi ve erişim hakları, erişim ve değiştirme zamanları ve diğer benzer bilgi türleri gibi şeyleri izler. Bu bilgileri saklamak için, dosya sistemleri genellikle **kayıt sistemi(inode)** adı verilen bir yapıya sahiptir (aşağıda kayıt sistemi hakkında daha fazla bilgi edineceğiz).

İnode'ları barındırmak için, diskte onlar için de biraz yer ayırmamız gerekecek. Diskin bu bölümünü, yalnızca bir dizi disk içi inode tutan **kayıt sistemi tablosu(inode table)** olarak adlandıralım. Böylece, diskteki görüntümüz şimdi bu resme benziyor, inode'lar için 5 bloğumuzdan 64'ünü kullandığımızı varsayıyoruz. (diyagramda I's ile gösterilir):

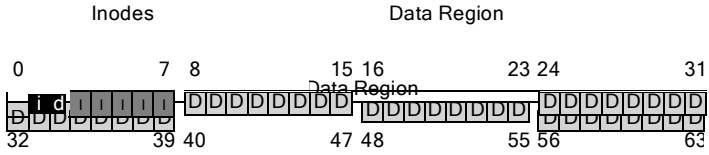


Burada inode'ların tipik olarak o kadar büyük olmadığını, örneğin 128 veya 256 bayt olduğunu not etmeliyiz. İnoda başına 256 bayt olduğunu varsayarsak, 4 KB'lık bir blok 16 inode tutabilir ve yukarıdaki dosya sistemimiz toplam 80 inode içerir. 64 blokluk küçük bir bölüm üzerine kurulu basit dosya sistemimizde,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

Dosya sistemimiz şu ana kadar veri bloklarına (D) ve inode'lara (I) sahiptir, ancak birkaç şey hala eksiktir. Tahmin edebileceğiniz gibi, hala ihtiyaç duyulan birincil bileşenlerden biri, inode'ların veya veri bloklarının ücretsiz mi yoksa tahsis edilmiş mi olduğunu izlemenin bir yoludur. Bu tür **ayırma yapıları(allocation structures)** bu nedenle herhangi bir dosya sisteminde gerekli bir unsurdur.

Elbette birçok tahsisat izleme yöntemi mümkündür. İnceleme için, ilk serbest bloğa işaret eden, daha sonra bir sonraki **ücretsiz bloğa(free list)** işaret eden ücretsiz bir liste kullanabiliriz. Bunun yerine, **biri veri bölgesi(data bitmap)** ve diğeri **kayıt sistemi tablosu (inode table)** için

bitmap olarak bilinen basit ve popüler bir yapı seçiyoruz. Bir bitmap basit bir yapıdır: her bit, karşılık gelen nesnenin/bloğun serbest (0) veya kullanımda (1) olup olmadığını belirtmek için kullanılır. Ve böylece bir inode bitmap (i) ve bir veri bitmap (d) ile yeni disk üzerindeki mizanpajımız.



Bu bitmap'ler için 4 KB'lık bir bloğun tamamını kullanmanın biraz fazla olduğunu fark edebilirsiniz; Böyle bir bitmap, 32K nesnelerin tahsis edilip edilmediğini izleyebilir ve yine de yalnızca 80 inode ve 56 veri bloğumuz vardır. Ancak, basitlik için bu bitmap'lerin 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 kaldığını fark etmemiş olabilir. Bunu, aşağıdaki diyagramda bir S ile gösterilen **süper blok(süperblocks)** için ayırıyoruz. Üst blok, örneğin, dosya sisteminde kaç tane inode ve veri bloğu olduğu (bu örnekte sırasıyla 80 ve 56), inode tablosunun nerede başladığı (blok 3) ve benzeri dahil olmak üzere bu belirli dosya sistemi hakkında bilgi içerir. Muhtemelen, dosya sistemi türünü tanımlamak için bir tür sihirli sayı da içerecektir (bu durumda, vsfs).

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

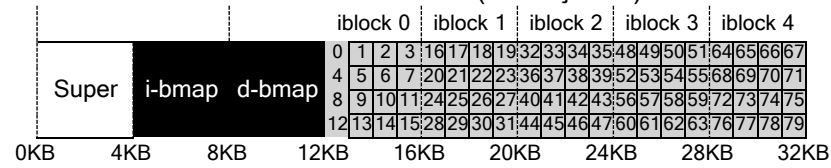
40.3 Dosya Organizasyonu: Kayıt sistemi

Bir dosya sisteminin en önemli disk yapılarından biri kayıt sistemi; hemen hemen tüm dosya sistemleri buna benzer bir yapıya sahiptir. inode adı, UNIX [RT74] ve muhtemelen daha önceki sistemlerde kendisine verilen tarihsel ad olan **dizin düğümünün(index node)** kısaltmasıdır, çünkü bu düğümler bir dizide orig-inally düzenlenmiştir ve dizi belirli bir inode'a erişirken dizine eklenmiştir.

Kenarda Dursun: Veri Yapıları — Dosya Kayıt Sistemi

Dosya kayıt sistemi**(inode)** uzunluğu ve bileşenleri bloklarının konumu gibi belirli bir dosyanın meta verilerini tutan yapıyı tanımlamak için birçok dosya sisteminde kullanılan genel addır. İsim en azından UNIX'e kadar uzanır (ve muhtemelen daha önceki sistemler olmasa da Multics'e kadar uzanır); **indeks düğümü(index node)** için kısadır, çünkü inode numarası, bu sayının inode'unu bulmak için bir dizi disk üzerindeki inode'a endekslemek için kullanılır. Göreceğimiz gibi, inode 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.)

Her inode, daha önce dosyanın düşük seviyeli adı olarak adlandırdığımız bir sayı (i-numarası olarak adlandırılır) ile dolaylı olarak adlandırılır. VSFS'de (ve diğer **basit dosya sistemlerinde(low level name)**), bir **i-numarası(I number)** verildiğinde, ilgili inode'un diskte nerede bulunduğunu doğrudan hesaplayabilmeniz gerekir. Ex- bol için, yukarıdaki gibi vsfs'nin inode tablosunu alın: 20KB boyutunda (beş 4KB blok) ve böylece 80 inode'dan oluşur (her inode 256 bayt olduğu varsayılarak); Daha sonra, inode bölgesinin 12KB'den başladığını varsayalım (yani, üst blok 0KB'de başlar, inode bitmap'i 4KB adresinde, veri bitmap'i 8KB'de ve böylece inode tablosu hemen sonra gelir). vsfs'de, bu nedenle aşağıdakilere sahibiz: The Inode Tablo (Yakınçekim)



32 numaralı inode'u okumak için, dosya sistemi önce inode bölgesine (32 · boyut (inode) veya 8192) ofset kümesini hesaplar, diskteki inode tablosunun başlangıç adresine ekler (inodeStartAddr = **12KB(kilobayt))** ve böylece istenen inode bloğunun doğru bayt adresine ulaşır: **20KB(kilobayt).** Disklerin bayt adreslenebilir olmadığını, bunun yerine genellikle 512 bayt olmak üzere çok sayıda adreslenebilir sektörden oluştuğunu hatırlayın. Bu nedenle, inode 32 içeren inode bloğunu getirmek için, dosya sistemi istenen inode bloğunu getirmek için sec- tor 20×1024 veya 40'a bir okuma yayınlayacaktır. Daha genel olarak, inode bloğun 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 (örneğin, normal dosya, dizin, vb.), Boyutu, kendisine tahsis edilen blok sayısı, koruma bilgileri (dosyanın kime ait olduğu gibi)

Boyut	Ad	Bu kayıt sistemi alanı ne için ?
2	mod	bu dosya okunabilir/yazılabilir/yürütülebilir mi?
2	uid	bu dosyanın sahibi kim?
4	boyut	bu dosyada kaç bayt var?
4	zaman	bu dosyaya en son saat kaçta erişildi?
4	c zamanı	bu dosya saat kaçta oluşturuldu?
4	m zamanı	bu dosya en son ne zaman düzenlendi ?
4	d zamanı	bu inode en son ne zaman silindi ?
2	gid	bu dosya hangi gruba ait?
2	bağlantı sayısı	bu dosyaya kaç tane sabit bağlantı var?
4	bloklar	bu dosyaya kaç blok tahsis edildi?
4	bayraklar	ext2 bu inode'u nasıl kullanmalı?
4	osd1	işletim sistemine bağımlı bir alan
60	blok	bir dizi disk işaretçisi (toplam 15)
4	kuşak	dosya sürümü (NFS tarafından kullanılır)
4	dosya acl	mod bitlerinin ötesinde yeni bir izin modeli
4	dir_acl	erişim kontrol listeleri olarak adlandırılır

Figure 40.1: **Basitleştirilmiş Ext2 Inode**

dosyaya kimlerin erişebileceği gibi), dosyanın ne zaman oluşturulduğu, değiştirildiği veya en son ne zaman erişildiği de dahil olmak üzere bazı zaman bilgilerinin yanı sıra veri bloklarının diskte nerede bulunduğuna ilişkin bilgiler (ör. bir tür işaretçi). Bir dosyayla ilgili tüm bu bilgileri **meta veri(meta data)** olarak adlandırırız; Aslında, dosya sistemi içinde saf kullanıcı verisi olmayan herhangi bir bilgi genellikle bu şekilde adlandırılır. Bir

ext2 [P09] 'dan örnek inode Şekil 40.11'de gösterilmiştir. İnode'un tasarımındaki en önemli kararlardan biri, veri bloklarının nerede olduğunu nasıl ifade ettiğidir. Basit bir yaklaşım, inode içinde bir veya daha fazla **doğrudan işaretçiye(diretct pointer)** (disk adresi) sahip olmak olacaktır; her işaretçi, dosyaya ait bir disk bloğuna başvurur. Böyle bir yaklaşım sınırlıdır: örneğin, gerçekten büyük bir dosyaya sahip olmak istiyorsanız (örneğin, inode'daki doğrudan işaretçilerin sayısıyla çarpılan blok boyutundan daha büyük), şansınız kalmaz.

Çok Seviyeli Endeks

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 işaretçi (örneğin, 12) ve tek bir dolaylı işaretçi olabilir. Bir dosya yeterince büyürse, dolaylı bir blok ayrılır (diskin veri bloğu bölgesinden) ve dolaylı bir işaretçi için inode yuvası onu işaret edecek şekilde ayarlanır. 4 KB'lık bloklar ve 4 baytlık disk adresleri varsayarsak, bu da başka bir 1024 işaretçi ekler; dosya büyüyebilir (12 + 1024) · 4K veya 4144KB.

1Type bilgisi dizin girişinde tutulur ve bu nedenle inode'un kendisinde bulunmaz.

İPUCU: KAPSAM TABANLI YAKLAŞIMLARI GÖZ ÖNÜNDE BULUNDURUN

Farklı bir yaklaşım, işaretçiler yerine kapsamları kullanmaktır. Bir **kapsam(extents)** basitçe bir disk işaretçisi artı bir uzunluktur (bloklar halinde); bu nedenle, bir dosyanın her bloğu için bir işaretçi gerektirmek yerine, tek gereksinim duyulan bir işaretçi ve dosyanın diskteki konumunu belirtmek için bir uzunluktur. Sadece tek bir kapsam sınırlayıcıdır, çünkü bir dosya ayırırken bitişik bir disk içi boş alan yığını bulmakta zorlanabilirsiniz. Bu nedenle, kapsam tabanlı dosya sistemleri genellikle birden fazla kapsama izin verir, böylece dosya ayırma sırasında dosya sistemine daha fazla özgürlük verir.

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

Şaşırtıcı olmayan bir şekilde, böyle bir yaklaşımda, daha büyük dosyaları desteklemek isteyebilirsiniz. Bunu yapmak için, inode'a başka bir işaretçi ekleyin: **iki katına çıkarılabilir dolaylı işaretçi(double indiriect pointer**). Bu işaretçi, her biri veri bloklarına işaretçiler içeren dolaylı bloklara işaretçiler içeren bir bloğu ifade eder. İki katına çıkarılabilir bir dolaylı blok, böylece ek bir 1024 · ile dosyaları büyütme imkanı ekler · 1024 veya 1 milyon 4 KB'lık bloklar, başka bir deyişle boyutu 4 GB'ın üzerindeki dosyaları destekler. Yine de daha fazlasını isteyebilirsiniz ve bahse gireriz ki bunun nereye gittiğini biliyorsunuzdur: **üçlü dolaylı işaretçi.(triple indirect pointer)**

Genel olarak, bu dengesiz ağaç, dosya bloklarına işaret etmek için **çok düzeyli dizin(multi-level index)** ap- proach olarak adlandırılır. On iki doğrudan işaretçinin yanı sıra hem tek hem de çift dolaylı blok içeren bir örneği inceleyelim. 4 KB'lık bir blok boyutu ve 4 baytlık işaretçiler varsayarak, bu yapı 4 GB'ın biraz üzerinde bir dosya barındırabilir (yani, (12 + 1024 + 1024) × 4 **KB(Kilo byte)**. Üçlü dolaylı bir bloğun eklenmesiyle bir dosyanın ne kadar büyük işlenebileceğini anlayabiliyor musunuz? (ipucu: oldukça büyük)

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

Merak ediyor olabilirsiniz: Neden böyle dengesiz bir ağaç kullanıyorsunuz? Neden farklı bir yaklaşım olmasın? Görünüşe göre, birçok 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 böyle bir gerçeği yansıtıyor; çoğu dosya gerçekten küçükse, bu durum için optimize etmek mantıklıdır. Böylece, az sayıda doğrudan işaretçi ile (12 tipik bir sayıdır), bir inode

Çoğu dosya küçüktür	~2K en yaygın boyuttur **Ortalama dosya boyutu büyüyor** **Çoğu bayt büyük dosyalarda depolanır** **Neredeyse 200K alanın ortalamasıdır Dosya sismleri çok sayıda dosya içerirOrtalama olarak neredeyse 100 bin**
Dosya sistemleri kabaca yarısı dolu **dosya sistemleri %50 dolu kalır**	**Diskler büyüdükçe bile,**
Dizinler genellikle küçüktür	Birçoğunun az sayıda girişi vardır; en Fazla 20 veya daha az olması gerekir.

Figür 40.2: **Dosya Sistemi Ölçüm Özeti**

doğrudan 48 KB veriye işaret edebilir ve daha büyük dosyalar için bir (veya daha fazla) dolaylı bloğa ihtiyaç duyar. Bkz. Agrawal et. al [A+07] yeni bir çalışma için; Şekil bu sonuçları özetler.

Tabii ki, inode tasarımı alanında, diğer birçok olasılık eski; 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 değiştiğinde farklı tasarımları patlatmaya istekli olmalısınız.

40.4 Dizin Organizasyonu

vsfs'de (birçok dosya sisteminde olduğu gibi), dizinlerin basit bir düzenlemesi vardır; Bir dizin temel olarak sadece (giriş adı, inode num- ber) çiftlerinin bir listesini içerir. Belirli bir dizindeki her dosya veya dizin için, dizinin veri bloklarında bir dize ve bir sayı vardır. Her dize için bir uzunluk da olabilir (değişken boyutlu adlar varsayarak).

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

```
inum | reclen | strlen | ad
  5      12        2     .
  2      12        3     ..
 12      12        4     foo

 13      12        4     çubuk
 24      36       28     Foobar oldukça uzun bir
                         isim
```

Bu örnekte, her girdinin bir inode numarası, kayıt uzunluğu (adın toplam baytı artı boşlukta kalan bayt), dize uzunluğu (adın gerçek uzunluğu) ve son olarak girdinin adı vardır. Her di- rektörlüğün iki ekstra 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 (örneğin, unlink()) öğesini çağırmak, dizinin ortasında boş bir alan bırakabilir ve bu nedenle bunu da işaretlemenin bir yolu olmalıdır (örneğin, sıfır gibi ayrılmış bir inode numarasıyla). Böyle bir silme, kayıt uzunluğunun kullanılmasının bir nedenidir: yeni bir girdi eski, daha büyük bir girişi yeniden kullanabilir ve böylece içinde fazladan alan olabilir.

Not: Bağlantılı Temelli Yaklaşımlar

İnode'ların tasarımında bir başka basit yaklaşım, **bağlantılı bir liste(linked list)** kullanmaktır. Bu nedenle, bir inode içinde, birden fazla işaretçiye sahip olmak yerine, dosyanın ilk bloğuna işaret etmek için sadece bir tanesine ihtiyacınız vardır. Daha büyük dosyaları işlemek için, bu veri bloğunun sonuna başka bir işaretçi ekleyin ve böylece büyük dosyaları destekleyebilirsiniz.

Tahmin edebileceğiniz gibi, bağlantılı dosya ayırma bazı iş yükleri için düşük performans gösterir; Örneğin, bir dosyanın son bloğunu okumayı veya yalnızca rastgele erişim yapmayı düşünün. Bu nedenle, bağlantılı ayırmanın daha iyi çalışmasını sağlamak için, bazı sistemler bir sonraki işaretçileri veri bloklarının kendileriyle depolamak yerine, bellek içi bir bağlantı bilgileri tablosu tutacaktır. Tablo, bir veri bloğu D'nin adresine göre dizine eklenir; bir girişin içeriği basitçe D'nin bir sonraki işaretçisidir, yani D'yi takip eden bir dosyadaki bir sonraki bloğun adresidir. Boş bir değer de orada olabilir (dosya sonunu gösterir) veya belirli bir bloğun serbest olduğunu belirtmek için başka bir işaretçi olabilir. Sonraki işaretçilerin böyle bir tablosuna sahip olmak, bağlantılı bir ayırma şemasının, istenen bloğu bulmak için önce (bellekteki) tabloyu tarayarak ve ardından doğrudan (diskte) erişerek, rastgele dosya erişimlerini etkili bir şekilde yapabilmesini sağlar.

Böyle bir masa tanıdık geliyor mu? Açıkladığımız şey, **dosya ayırma tablosu(file allocation table)** veya **FAT(dosya ayırma tablosu)** dosya sistemi olarak bilinen şeyin temel yapısıdır. Evet, NTFS'den [C94] önceki bu klasik eski Windows dosya sistemi, basit bir bağlantılı tabanlı ayırma şemasını temel alır. Standart bir UNIX dosya sisteminden başka farklılıklar da vardır; örneğin, kendi başına inode yoktur, bunun yerine bir dosya hakkında meta verileri depolayan ve doğrudan söz konusu dosyanın ilk bloğuna başvuran dizin girişleri vardır, bu da sabit bağlantılar oluşturmayı imkansız kılar. Zarif olmayan detaylar hakkında daha fazla bilgi için Brouwer [B02] bölümüne bakın.

Dizinlerin tam olarak nerede depolandığını merak ediyor olabilirsiniz. Genellikle, dosya sistemleri dizinleri özel bir dosya türü olarak değerlendirir. Bu nedenle, bir dizinin inode tablosunda bir yerde bir inode vardır (inode'un tür alanı "normal dosya" yerine "dizin" olarak işaretlenmiştir). Dizin, inode tarafından işaret edilen 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, dizin denemelerinin bu basit doğrusal listesinin bu tür bilgileri depolamanın tek yolu olmadığını tekrar belirtmeliyiz. Daha önce olduğu gibi, herhangi bir veri yapısı mümkündür. Örneğin, XFS [S+96] dizinleri B ağacı biçiminde depolayarak, dosya oluşturma işlemlerini (oluşturmadan önce bir dosya adının kullanılmadığından emin olmak gerekir) tamamen taranması gereken basit listelere sahip sistemlerden daha hızlı hale getirir.

Kenarda dursun: Boş Alan Yönetimi

Boş alanı yönetmenin birçok yolu vardır; bitmap'ler sadece bir yoldur. Bazı erken dosya sistemleri, süper bloktaki tek bir işaretçinin ilk serbest bloğa işaret etmek için tutulduğu **serbest listeleri(free list)** kullandı; Bu bloğun içinde bir sonraki serbest işaretçi tutuldu, böylece sistemin serbest blokları aracılığıyla 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'nın XFS [S+96], diskin hangi parçalarının boş olduğunu kompakt bir şekilde temsil etmek için bir tür **B ağacı(B tree)** kullanır. Herhangi bir veri yapısında olduğu gibi, farklı zaman-mekan ödünleşimleri mümkündür.

40.5 Boş Alan Yönetimi

Bir dosya sistemi, hangi inode'ların ve veri bloklarının boş olduğunu ve hangilerinin olmadığını izlemelidir, böylece yeni bir dosya veya dizin tahsis edildiğinde, bunun için yer bulabilir. 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, bitmap üzerinden ücretsiz bir in-ode arayacak ve dosyaya tahsis edecektir; dosya sisteminin inode'u kullanıldığı gibi işaretlemesi (1 ile) ve sonunda diskteki bitmap'i doğru bilgilerle güncellemesi gerekecektir. Benzer bir etkinlik kümesi, bir veri bloğu tahsis edildiğinde gerçekleşir.

Yeni bir dosya için veri blokları ayrılırken başka bazı hususlar da devreye girebilir. Örneğin, ext2 ve ext3 gibi bazı Linux dosya sistemleri, yeni bir dosya oluşturulduğunda ve veri bloklarına ihtiyaç duyduğunda ücretsiz olan bir dizi blok (örneğin 8) arayacaktır; dosya sistemi, serbest blokların böyle bir benzerliğini bularak ve daha sonra bunları yeni oluşturulan dosyaya ayırarak, dosyanın bir kısmının diskte bitişik olacağını **garanti eder(pre-allocation)** ve böylece performansı artırır. Bu nedenle, böyle bir ön tahsis politikası, veri blokları için alan ayırırken yaygın olarak kullanılan bir sezgisel yöntemdir.

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**(acces path)** neler olduğunu anlamak, bir dosya sisteminin nasıl çalıştığına dair bir anlayış geliştirmenin ikinci anahtarıdır; Dikkat!

Aşağıdaki örnekler için, dosya sisteminin bağlandığını ve böylece üst bloğun zaten bellekte olduğunu varsayalım. Diğer her şey (yani, inode'lar, dizinler) hala disktedir.

	veri kayıt sistemi bitharitası	kök foo bar kayıt sistemi	kök foo bar bar bar veri veri veri veri [0] [1] [2]
open(bar)		**oku** **oku** **oku**	**oku** **oku**
oku()		**oku** **yaz**	**oku**
oku()		**oku** **yaz**	**oku**
oku()		**yaz** **yaz**	**oku**

Figür 40.3: **Dosya Okuma Zaman Çizelgesi (Aşağı Doğru Artan Süre)**

Diskten Dosya Okuma

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

Bir open("/foo/bar", O RDONLY) çağrısı yaptığınızda, dosya sistemi öncelikle dosya çubuğunun inode'unu bulmalı, dosya hakkında bazı temel bilgiler edinmelidir (izin bilgileri, dosya boyutu, vb.). Bunu yapmak için, dosya sistemi inode'u bulabilmelidir, ancak şu anda sahip olduğu tek şey tam **yol adıdır(traverse).** Dosya sistemi yol adından geçmeli ve böylece istenen inode'u bulmalıdır.

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

İnode okunduktan sonra, FS, kök dizinin içeriğini içeren veri bloklarına işaretçiler bulmak için içine bakabilir. FS bu nedenle dizini okumak için bu disk üstü işaretçileri kullanır, bu durumda foo için bir giriş arar. Bir veya daha fazla dizin veri bloğunda okuyarak, foo girişini bulacaktır; Bir kez bulunduğunda, FS daha sonra ihtiyaç duyacağı foo'nun inode sayısını da bulmuş olacaktır (44 olduğunu varsayalım).

Bir sonraki adım, istenen inode bulunana kadar yol adını yinelemeli olarak geçmektir. Bu örnekte, FS

Kenarda Dursun: OKUMALAR TAHSISAT YAPILARINA ERIŞMIYOR

Birçok öğrencinin bitmap'ler gibi tahsis yapılarıyla kafasının karıştığını gördük. Özellikle, çoğu zaman sadece bir dosyayı okurken ve yeni bloklar ayırmadığınızda, bitmap'e hala danışılacağını düşünür. Bu doğru değil! Bit eşlemler gibi ayırma yapılarına yalnızca ayırma gerektiğinde erişilir. İnode'lar, dizinler ve dolaylı bloklar, bir okuma yeniden görevini tamamlamak için ihtiyaç duydukları tüm bilgilere sahiptir; inode zaten işaret ettiğinde bir bloğun tahsis edildiğinden emin olmaya gerek yoktur.

foo'nun inode'u ve ardından dizin verileri, sonunda çubuğun inode numarasını bulur. open()'in son adımı, bar'ın inode'unu hafızaya okumaktır; FS daha sonra son bir izin denetimi yapar, işlem başına açık dosya tablosunda bu işlem için bir dosya tanımlayıcısı ayırır ve kullanıcıya döndürür.

foo'nun inode'u ve ardından dizin verileri, sonunda çubuğun inode numarasını bulur. open()'in son adımı, bar'ın inode'unu hafızaya okumaktır; FS daha sonra son bir izin denetimi yapar, işlem başına açık dosya tablosunda bu işlem için bir dosya tanımlayıcısı ayırır ve kullanıcıya döndürür.

Bir noktada, dosya kapatılacaktır. Burada yapılacak çok daha az iş var; Açıkçası, dosya tanımlayıcısı serbest bırakılmalıdır, ancak şimdilik, FS'nin gerçekten yapması gereken tek şey budur. Disk G/Ç'leri gerçekleşmez.

Tüm bu sürecin bir tasviri Şekil 40.3'te (sayfa 11) bulunur; zaman şekilde aşağı doğru artar. Şekilde, açık, dosyanın inode'unu nihayet bulmak için çok sayıda okumanın gerçekleşmesine neden olur. Daha sonra, her bloğun okunması, dosya sisteminin önce inode'a danışmasını, ardından bloğu okumasını ve ardından inode'un son erişilen zaman alanını bir yazma ile güncellemesini gerektirir. Biraz zaman ayırın ve neler olup bittiğini anlayın.

Ayrıca, açık tarafından üretilen G/Ç miktarının, yol adının uzunluğuna uygun olduğunu unutmayın. Yoldaki her ek dizin için, inode'unu ve verilerini okumalıyız. Bunu daha da kötüleştirmek, büyük dizinlerin varlığı olacaktır; Burada, bir dizinin içeriğini almak için yalnızca bir bloğu okumamız gerekirken, büyük bir dizinde, istenen girişi bulmak için birçok veri bloğunu okumamız gerekebilir. Evet, bir dosyayı okurken hayat oldukça kötüye gidebilir; öğrenmek üzere olduğunuz gibi, bir dosya yazmak (ve özellikle de yeni bir dosya 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 güncelleştirmek için write() çağrıları yapabilir. Son olarak, dosya kapatılır.

Okumadan farklı olarak, dosyaya yazmak da bir blok **ayırabilir(allocate)** (örneğin, bloğun üzerine yazılmadığı sürece). Yeni bir dosya yazarken, her yazma yalnızca diske veri yazmakla kalmayıp, önce karar vermelidir.

	veri bitmap	inode bitmap	kök inode	foo inode	bar inode	kök veri	foo veri	bar veri [0]	bar veri [1]	bar veri [2]
oluştur (/foo/bar)		**oku** **yaz**	**oku**	**oku**	**oku** **yaz**	**oku**	**oku** **yaz**			
yaz()	**oku** **yaz**				**oku** **yaz**			**yaz**		
yaz()	**oku** **yaz**				**oku** **yaz**				**yaz**	
yaz()	**oku** **yaz**				**oku** **yaz**					**yaz**

Figür 40.4: **Dosya Oluşturma Zaman Çizelgesi (Zaman Aşağı Doğru Artıyor) File Creation Timeline (Time Increasing Downward)**

hangi bloğun dosyaya tahsis edileceği ve böylece diskin diğer yapılarının buna göre güncelleneceği (örneğin, veri bitmap'i ve inode). Böylece, bir dosyaya yazılan her yazma mantıksal olarak beş G / Ç oluşturur: biri veri bitmap'ini okumak için (daha sonra yeni tahsis edilen bloğu kullanıldığı gibi işaretlemek için güncellenir), biri bitmap'i yazmak (yeni durumunu diske yansıtmak için), iki tane daha inode'u okumak ve yazmak için (yeni bloğun konumu ile güncellenir), ve son olarak gerçek bloğun kendisini yazmak için bir tane.

Yazma trafiği miktarı, dosya oluşturma gibi basit ve yaygın bir işlem düşünüldüğünde daha da kötüdür. Bir dosya oluşturmak için, dosya sistemi sadece bir inode tahsis etmekle kalmamalı, aynı zamanda yeni dosyayı içeren dizin içinde de yer ayırmalıdır. Bunu yapmak için toplam G / Ç trafiği miktarı oldukça yüksektir: biri inode bitmap'e okunur (ücretsiz bir inode bulmak için), biri inode bitmap'ine yazar (tahsis edilmiş olarak işaretlemek için), biri yeni inode'un kendisine yazar (başlatmak için), biri dizinin verilerine (dosyanın üst düzey adını inode numarasına bağlamak için), ve bir okumak ve güncellemek için inode dizinine yazmak. Dizinin yeni girişi barındıracak şekilde büyümesi gerekiyorsa, ek G/Ç'lere (yani, veri bitmap'ine ve yeni dizin bloğuna) da ihtiyaç duyulacaktır. Bütün bunlar sadece bir dosya oluşturmak için!

/foo/bar dosyasının oluşturulduğu ve üç bloğun yazıldığı belirli bir örneğe bakalım. Şekil 40.4 (sayfa 13), open() (dosyayı oluşturan) ve üç 4KB yazma işleminin her biri sırasında hangi hap-kalemlerin olduğunu göstermektedir..

Şekilde, diske yapılan okuma ve yazmalar, hangi sistem çağrısının gerçekleşmesine neden olduğu altında gruplandırılmıştır ve bunların gerçekleşebileceği kaba sıralama, şeklin üstünden aşağısına doğru gider. Dosyayı oluşturmanın ne kadar iş olduğunu görebilirsiniz: Bu durumda, yol adını yürümek ve son olarak dosyayı oluşturmak için 10 G/Ç. Ayrıca, her tahsis yazma maliyetinin 5 G / Ç olduğunu da görebilirsiniz: inode'u okumak ve güncellemek için bir çift, veri bitmap'ini okumak ve güncellemek için başka bir çift ve son olarak verilerin kendisinin yazılması. Bir dosya sistemi bunlardan herhangi birini makul bir verimlilikle nasıl başarabilir?

İşin püf noktası: DOSYA SISTEMI G/Ç MALIYETLERI NASIL AZALTILIR?

Bir dosyayı açmak, okumak veya yazmak gibi en basit işlemler bile, diskin üzerine dağılmış çok sayıda G/Ç işlemine neden olur. Bir dosya sistemi bu kadar çok G/Ç yapmanın yüksek maliyetlerini azaltmak için ne yapabilir?

40.7 Önbelleğe Alma ve Arabelleğe Alma

Yukarıdaki örneklerin gösterdiği gibi, dosyaları okumak ve yazmak masraflı olabilir ve (yavaş) diske birçok G / Ç gerektirebilir. Açıkça büyük bir performans sorunu olacak şeyi çözmek için, çoğu dosya sistemi önemli blokları önbelleğe almak için agresif bir şekilde sistem belleği (DRAM) kullanır.

Yukarıdaki açık örneği düşünün: önbelleğe alınmadan, açılan her dosya, dizin hiyerarşisindeki her düzey 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 sadece dosyayı açmak için kelimenin tam anlamıyla yüzlerce okuma gerçekleştirir!

İlk dosya sistemleri böylece popüler blokları tutmak için sabit boyutlu bir önbellek tanıttı. Sanal bellek tartışmamızda olduğu gibi, LRU ve farklı varyantlar gibi stratejiler hangi blokların önbellekte tutulacağına karar verecektir. Bu **sabit boyutlu önbellek(fixed size cache)** genellikle önyükleme sırasında toplam belleğin kabaca %10'u olacak şekilde ayrılır.

Bununla birlikte, belleğin bu ***statik bölümlenmesi(static partitioning)*** israf edici olabilir; ya dosya sistemi belirli bir zamanda belleğin% 10'una ihtiyaç duymuyorsa?

Yukarıda açıklanan sabit boyutlu yaklaşımla, dosya önbelleğindeki

kullanılmayan sayfalar başka bir kullanım için yeniden kullanılamaz ve bu nedenle boşa gider. Modern sistemler, aksine, **dinamik bir bölümleme(dynamic partitioning)** yaklaşımı kullanır. Özellikle, birçok modern işletim sistemi sanal bellek sayfalarını ve dosya sistemi sayfalarını **birleşik bir sayfa önbelleğine(unified page cache)** [S00] entegre eder. Bu şekilde, bellek, belirli bir zamanda hangisinin daha fazla belleğe ihtiyaç duyduğuna bağlı olarak sanal bellek ve dosya sistemi arasında daha esnek bir şekilde ayrılabilir.

Şimdi dosyayı önbelleğe alma ile açık bir örnek olarak hayal edin. İlk açılış, dizin inode ve verilerde okumak için çok fazla G / Ç trafiği oluşturabilir, ancak alt-

İpucu: Statik ve Dinamik Bölümlemeyi Anlayın

Bir kaynağı farklı istemciler/kullanıcılar arasında bölüştürürken, **statik bölümleme(statical partitioning)** veya **dinamik bölümleme(dynamical partitioning)** kullanabilirsiniz. Statik yaklaşım, kaynağı bir kez sabit oranlara böler; örneğin, iki olası bellek kullanıcısı varsa, belleğin sabit bir kısmını bir kullanıcıya, geri kalanını ise diğerine verebilirsiniz. Dinamik yaklaşım daha esnektir ve zaman içinde farklı miktarlarda kaynak verir; Örneğin, bir kullanıcı belirli bir süre için daha yüksek bir disk bant genişliği yüzdesi alabilir, ancak daha sonra sistem geçiş yapabilir ve farklı bir kullanıcıya kullanılabilir disk bant genişliğinin daha büyük bir bölümünü vermeye karar verebilir.

Her yaklaşımın avantajları vardır. Statik bölümleme, her kullanıcının kaynaktan bir miktar pay almasını, genellikle daha öngörülebilir performans sunmasını ve genellikle daha kolay uygulanmasını sağlar. Dinamik bölümleme, 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 kazanılması uzun zaman alan kullanıcılar için daha kötü performansa yol açabilir. On tanesinde olduğu gibi, en iyi yöntem yoktur; bunun yerine, eldeki sorunu düşünmeli ve hangi yaklaşımın en uygun olduğuna karar vermelisiniz. Gerçekten, bunu her zaman yapıyor olmanız gerekmez mi?

Aynı dosyanın (veya aynı dizindeki dosyaların) sıralı dosya açılışları çoğunlukla önbellekte isabet eder ve bu nedenle G/Ç gerekmez.

Önbelleğe almanın yazmalar üzerindeki etkisini de göz önünde bulunduralım. Okuma G/Ç'si yeterince büyük bir önbellekle tamamen önlenebilirken, yazma trafiğinin kalıcı hale gelmesi için diske gitmesi gerekir. Bu nedenle, önbellek, yazma trafiğinde okumalar için yaptığı gibi aynı filtre türü olarak işlev görmez.

Bununla birlikte, **yazma tamponlaması(write buffering)** (bazen adlandırıldığı gibi) kesinlikle bir dizi performans avantajına sahiptir. İlk olarak, dosya sistemi yazmaları geciktirerek bazı güncelleştirmeleri daha küçük bir G/Ç kümesine **toplu(batch)** olarak ekleyebilir; örneğin, bir dosya oluşturulduğunda bir inode bitmap güncellenirse ve birkaç dakika sonra başka bir dosya oluşturulduğunda güncellenirse, dosya sistemi ilk güncellemeden sonra yazmayı geciktirerek bir G/Ç kaydeder. İkincisi, bellekte bir dizi yazmayı arabelleğe alarak, sistem sonraki G/Ç'leri **zamanlayabilir(schedule)** ve böylece performansı artırabilir. Son olarak, bazı yazılar geciktirilerek tamamen **önlenir(avoids)**; örneğin, bir uygulama bir dosya oluşturur ve sonra onu silerse, yazmaları dosya oluşturmayı diske yansıtacak şekilde geciktirmek bunları tamamen önler. Bu durumda, tembellik (diske blok yazarken) bir erdemdir.

Yukarıdaki nedenlerden dolayı, çoğu modern dosya sistemi arabelleği, beş ila otuz saniye arasında herhangi bir yerde hatıra olarak yazar, bu da başka bir dengeyi temsil eder: güncellemeler diske yayılmadan önce sistem çökerse, güncellemeler kaybolur; ancak, yazmaları daha uzun süre hafızada tutarak, toplu işlem, zamanlama ve hatta yazmalardan kaçınarak performans iyileştirilebilir.

İPUCU: DAYANIKLILIK/PERFORMANS DENGESINI ANLAYIN
Depolama sistemleri genellikle kullanıcılara 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 yavaş (ancak güvenli) olur. Bununla birlikte, kullanıcı küçük bir veri kaybını tolere edebiliyorsa, sistem bellekteki yazmaları bir süre arabelleğe alabilir ve daha sonra diske (arka zeminde) yazabilir. Bunu yapmak, yazmaların hızlı bir şekilde tamamlandığını gösterir, böylece algılanan performansı geliştirir; ancak, bir çökme meydana gelirse, henüz diske bağlanmamış yazmalar kaybolur ve bu nedenle takas edilir. Bu ödünleşimin nasıl düzgün bir şekilde yapılacağını anlamak için, depolama sistemini kullanarak uygulamanın ne gerektirdiğini anlamak en iyisidir; örneğin, web tarayıcınız tarafından indirilen son birkaç görüntüyü kaybetmek tolere edilebilir olsa da, banka hesabınıza para ekleyen bir veritabanı işleminin bir kısmını kaybetmek daha az tolere edilebilir olabilir. Zengin olmadığınız sürece elbette; bu durumda, neden her son kuruşu biriktirmeyi bu kadar önemsiyorsunuz?

Bazı uygulamalar (veritabanları gibi) bu ödünleşimden hoşlanmaz. Bu nedenle, yazma arabelleğe alma nedeniyle beklenmeyen veri kaybını önlemek için, fsync() öğesini çağırarak, önbelleğin etrafında çalışan **doğrudan G/Ç(direct I/O)** arabirimlerini kullanarak veya **ham disk (raw disk)**arabirimini kullanarak ve dosya sistemi tamamen

Çoğu uygulama dosya sistemi tarafından yapılan ödünleşimlerle yaşarken, varsayılanın tatmin edici olmaması durumunda, sistemin istediğiniz şeyi yapmasını sağlamak için yeterli denetim vardır.

40.8 ÖZET

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

Dosya sistemi tasarımının müthiş yönü özgürlüğüdür; Gelecek bölümlerde keşfedeceğimiz dosya sistemlerinin her biri, dosya sisteminin bazı yönlerini optimize etmek için bu özgürlükten yararlanır. Ayrıca, keşfedilmemiş bıraktığımız birçok politika kararı da var. Örneğin, yeni bir dosya oluşturulduğunda, diskte nereye yerleştirilmelidir? Bu politika ve diğerleri gelecek bölümlerin de konusu olacak. Yoksa yapacaklar mı?

[2] Eski okul veritabanları ve işletim sisteminden kaçınma ve her şeyi kendileri kontrol etme konusundaki eski eğilimleri hakkında daha fazla bilgi edinmek için bir veritabanı sınıfına katılın. Ama dikkat et! Bu veritabanı türleri her zaman işletim sistemini kötülemeye çalışıyor. Ayıp. 3Cue dosya sistemleri konusu hakkında daha da ilgi çekici hale getiren gizemli müzik.

Referanslar(Referances)

[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/~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. 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 vsfs.py bu aracı kullanın. Dosya sistemi, yalnızca bir kök dizinle boş bir durumda başlar. Simülasyon gerçekleştikçe, çeşitli işlemler gerçekleştirilir, böylece dosya sisteminin disk üzerindeki durumu yavaşça değiştirilir. Ayrıntılar için README'ye bakın.

Sorular

1. Simülatörü bazı farklı rastgele tohumlarla çalıştırın (örneğin 17, 18, 19, 20) ve her durum değişikliği arasında hangi işlemlerin gerçekleşmesi gerektiğini bulup bulamayacağınızı görün.
2. Şimdi, -r bayrağıyla çalıştırmak dışında, farklı rastgele tohumlar (örneğin 21, 22, 23, 24) kullanarak aynısını yapın, böylece işlem gösterilirken durum değişikliğini tahmin etmenizi sağlar. Hangi blokları tahsis etmeyi tercih ettikleri açısından inode ve veri bloğu tahsis algoritmaları hakkında ne sonuca varabilirsiniz?
3. Şimdi dosya sistemindeki veri bloklarının sayısını çok düşük sayılara (örneğin iki) düşürün ve simülatörü yüz kadar istek için çalıştırın. Bu son derece kısıtlı düzende dosya sisteminde ne tür dosyalar ortaya çıkıyor? Ne tür işlemler başarısız olur?
4. Şimdi aynısını yapın, ancak inoder 'larla.Çok az sayıda inode ile , ne tür işlemler başarılı olabilir ? Hangisi genellikle başarısız olur? Dosya sisteminin son durumu ne olabilir ?