41

Yerellik ve Hızlı Dosya Sistemi

UNIX işletim sistemi ilk tanıtıldığında, UNIX sihirbazının kendisi Ken Thompson ilk dosya sistemini yazdı. Buna "eski UNIX dosya sistemi" diyelim ve gerçekten basitti. Temel olarak, veri yapıları diskte şöyle görünüyordu:

Süper blok (S) tüm dosya sistemi hakkında bilgi içeriyordu: hacmin ne kadar büyük olduğu, kaç tane inode var, ücretsiz bir blok listesinin başına bir işaretçi, ve benzerleri. Diskin inode bölgesi, dosya sistemi için tüm inode'ları içeriyordu. Son olarak, diskin çoğu veri blokları tarafından alındı.

Eski dosya sistemiyle ilgili iyi olan şey, basit olması ve dosya sisteminin sunmaya çalıştığı temel soyutlamaları desteklemesiydi: dosyaları ve dizin hiyerarşisi. Bu kullanımı kolay sistem, sakarlıktan ileriye doğru atılmış gerçek bir adımdı, geçmişin kayıt tabanlı depolama sistemleri, ve dizin hiyerarşisi daha basit olana göre gerçek bir ilerlemeydi, önceki sistemler tarafından sağlanan tek düzeyli hiyerarşiler.

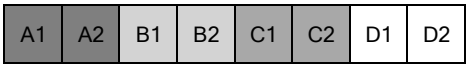
41.1 Sorun: Düşük Performans

Sorun: performans korkunçtu. Kirk McKu- sick ve Berkeley'deki meslektaşları [MJLF84] tarafından ölçüldüğü gibi, performans kötü başladı ve zamanla daha da kötüleşti, dosya sisteminin toplam disk bant genişliğinin sadece % 2'sini sağladığı noktaya kadar!

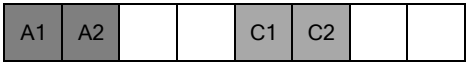
Ana sorun, eski UNIX dosya sisteminin diske rastgele erişimli bir bellekmiş gibi davranmasıydı; veriler, verileri tutan ortamın bir disk olduğu ve dolayısıyla gerçek ve pahalı konumlandırma maliyetlerine sahip olduğu gerçeğine bakılmaksızın her yere yayıldı. Örneğin, bir dosyanın veri blokları genellikle inode'undan çok uzaktaydı, böylece inode'u ve ardından bir dosyanın veri bloklarını ilk okuduğunda pahalı bir aramaya neden olur (oldukça yaygın bir işlem).

Daha da kötüsü, dosya sistemi oldukça **parçalanmıştı** (**fragmented**), boş alan dikkatli bir şekilde yönetilmediği için. Ücretsiz liste, diske yayılmış bir grup bloğa işaret edecektir, ve dosyalar tahsis edildiğinde, sadece bir sonraki serbest bloğu alacaklardı. Sonuç olarak, mantıksal olarak bitişik bir dosyaya disk boyunca ileri geri giderek erişilebiliyordu, böylece performansı önemli ölçüde azaltır.

Örneğin, her biri 2 boyutlu bloklardan oluşan dört dosya (A, B, C ve D) içeren aşağıdaki veri bloğu bölgesini düşünün:



B ve D silinirse, ortaya çıkan düzen:



Gördüğünüz gibi, boş alan iki bloğun iki parçasına bölünmüştür, dört güzel bitişik parça yerine. Şimdi dört blok boyutunda bir E dosyası ayırmak istediğinizi varsayalım:

Neler olduğunu görebilirsiniz: E, diske yayılır ve sonuç olarak, E'ye erişirken diskten en yüksek (sıralı) performansı elde edemezsiniz. Aksine, önce E1 ve E2'yi okursunuz, sonra ararsınız, sonra E3 ve E4'ü okursunuz. Bu parçalanma sorunu eski UNIX dosya sisteminde her zaman oldu ve performansa zarar verdi. Bir yan not: Bu sorun tam olarak disk **birleştirme (defragmentation)** araçlarının yardımcı olduğu şeydir; dosyaları bitişik olarak yerleştirmek ve bir veya birkaç bitişik bölge için boş alan açmak için disk üzerindeki verileri yeniden düzenlerler, verileri hareket ettirmek ve daha sonra değişiklikleri yansıtmak için inode'ları ve benzerlerini yeniden yazmak.

Başka bir sorun: orijinal blok boyutu çok küçüktü (512 bayt). Bu nedenle, diskten veri aktarımı doğal olarak verimsizdi. Daha küçük bloklar iyiydi çünkü **iç parçalanmayı** (**internal fragmentation**) en aza indirdiler (blok içindeki atıklar), ancak her bloğun ona ulaşması için bir pozisyon yükü gerektirebileceğinden transfer için kötü. Böylece, sorun:

İŞİN PÜF NOKTASI:
PERFORMANSI ARTIRMAK IÇIN DISK ÜZERINDEKI VERILER NASIL DÜZENLENIR

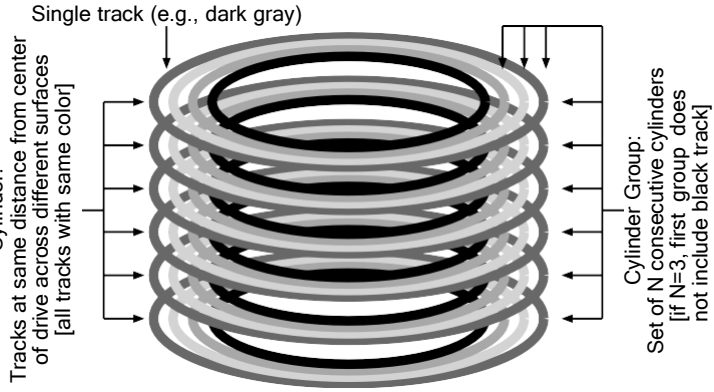
Performansı artırmak için dosya sistemi veri yapılarını nasıl düzenleyebiliriz? Bu veri yapılarının üstünde ne tür tahsis politikalarına ihtiyacımız var? Dosya sistemini "diske duyarlı" hale nasıl getirebiliriz?

41.2 HDS: Disk Farkındalığı Çözümdür

Berkeley'deki bir grup, akıllıca **Hızlı Dosya Sistemi (HDS)** (**Fast File System (FFS)**) olarak adlandırdıkları daha iyi, daha hızlı bir dosya sistemi kurmaya karar verdi. Buradaki fikir, dosya sistemi yapılarını ve ayırma politikalarını "disk farkında" olacak şekilde tasarlamak ve böylece performansı artırmaktı, ki bu tam olarak yaptıkları şeydi. HDS böylece dosya sistemi araştırmalarında yeni bir çağa girdi; dosya sistemi için aynı *arayüzü (interface)* koruyarak (`open()`, `read()`, `write()`, `close()`, ve diğer dosya sistemi çağrıları dahil olmak üzere aynı API'ler) ancak iç *uygulamayı* (*implementation)* değiştirmek, yazarlar yeni dosya sistemi inşasının yolunu açtılar, bugün devam eden çalışmalar. Hemen hemen tüm modern dosya sistemleri mevcut arayüze bağlı kalır (ve böylece uygulamalarla uyumluluğu korur) performans, güvenilirlik veya diğer nedenlerle iç kısımlarını değiştirirken.

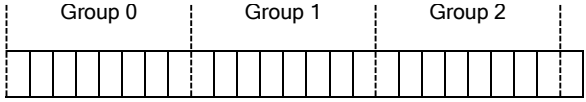
41.3 Organizasyon Yapısı: Silindir Grubu

İlk adım, disk üzerindeki yapıları değiştirmekti. HDS diski birkaç **silindir grubuna (cylinder groups)** böler. Tek bir **silindir (cylinder)** bir sabit sürücünün farklı yüzeylerinde, sürücünün merkezinden aynı uzaklıkta bulunan bir dizi izdir; sözde geometrik şekle açık benzerliği nedeniyle silindir olarak adlandırılır. HDS, N ardışık silindiri bir grupta toplar ve böylece tüm disk, silindir gruplarının bir koleksiyonu olarak görülebilir. Aşağıda, altı plakalı bir sürücünün en dıştaki dört izini ve üç silindirden oluşan bir silindir grubunu gösteren basit bir örnek verilmiştir:



Modern sürücülerin, dosya sisteminin belirli bir silindirin kullanımda olup olmadığını gerçekten anlaması için yeterli bilgiyi dışa aktarmadığına dikkat edin; daha önce tartışıldığı gibi [AD14a], diskler blokların mantıksal bir adres alanını dışa aktarır ve geometrilerinin ayrıntılarını istemcilerden gizler. Böylece modern dosya

sistemleri (Linux ext2, ext3 ve ext4 gibi) bunun yerine sürücüyü **blok grupları (block groups)** halinde düzenler, her biri, diskin adres alanının ardışık bir bölümüdür.. Aşağıdaki resim, her 8 bloğun farklı bir blok grubu halinde organize edildiği bir örneği göstermektedir (gerçek grupların çok daha fazla bloktan oluşacağını unutmayın):



İster silindir grupları ister blok grupları olarak adlandırın, bu gruplar FFS'nin performansı artırmak için kullandığı merkezi mekanizmadır. Kritik olarak, FFS iki dosyayı aynı grup içine yerleştirerek, birbiri ardına erişmenin diskte uzun aramalarla sonuçlanmamasını sağlayabilir. Dosyaları ve dizinleri depolamak için bu grupları kullanmak için, FFS'nin dosyaları ve dizinleri bir gruba yerleştirme ve onlar hakkında gerekli tüm bilgileri orada izleme yeteneğine sahip olması gerekir. Bunu yapmak için, FFS, bir dosya sisteminin her grup içinde sahip olmasını bekleyebileceğiniz tüm yapıları içerir; örneğin, düğümler için alan, veri blokları ve bunların her birinin tahsis edilip edilmediğini veya boş olup olmadığını takip eden bazı yapılar. Burada, FFS'nin tek bir silindir grubu içinde tuttuğu şeyin bir tasviri bulunmaktadır:

Şimdi bu tek silindir grubunun bileşenlerini daha detaylı inceleyelim. FFS, güvenilirlik nedenleriyle her gruptaki **süper blok (super block)** (S) bir kopyasını tutar. Dosya sistemini bağlamak için süper blok gereklidir; birden çok kopyayı saklayarak, bir kopya bozulursa, çalışan bir kopya kullanarak dosya sistemini bağlamaya ve dosya sistemine erişmeye devam edebilirsiniz.

Her grup içinde, FFS'nin grubun inode'larının ve veri bloklarının tahsis edilip edilmediğini izlemesi gerekir. Grup başına bir **düğüm bit işlemi (inode bitmap)** (ib) ve **veri bit işlemi (data bitmap)** (db) her gruptaki düğümler ve veri blokları için bu role hizmet eder. Bit eşlemler, bir dosya sistemindeki boş alanı yönetmenin mükemmel bir yoludur, çünkü büyük bir boş alan parçası bulmak ve onu bir dosyaya tahsis etmek kolaydır, belki de eski dosya sistemindeki boş listenin bazı parçalanma sorunlarından kaçınır.

Son olarak, **düğüm (inode)** ve **veri bloğu (data block)** Finally, the düğüm (inode) and veri bloğu (data block) regions are just like those in the pre- vious very-simple file system (VSFS). Most of each cylinder group, as usual, is comprised of data blocks.

BİR YANA: HDS DOSYA OLUŞTURMA

Örnek olarak, bir dosya oluşturulduğunda hangi veri yapılarının güncellenmesi gerektiğini düşünün; bu örnek için kullanıcının yeni bir dosya oluşturduğunu varsayalım `/foo/bar.txt` ve dosyanın bir blok uzunluğunda olduğunu (4KB). Dosya yenidir ve bu nedenle yeni bir düğüme ihtiyaç duyar; bu nedenle, hem düğüm bit işlemi hem de yeni tahsis edilen düğüm diske yazılacaktır. Dosyanın içinde ayrıca veriler vardır ve bu nedenle onun da tahsis edilmesi gerekir; veri bit işlemi ve bir veri bloğu bu nedenle (sonunda) diske yazılacaktır. Bu nedenle, mevcut silindir grubuna en az dört yazma işlemi gerçekleşecektir (bu yazma işlemlerinin gerçekleşmeden önce bir süre bellekte ara belleğe alınabileceğini hatırlayın). Ama bu hepsi değil! Özellikle, yeni bir dosya oluştururken, dosyayı dosya sistemi hiyerarşisine de yerleştirmelisiniz, yani dizin güncellenmelidir. Özellikle, `bar.txt` girişini eklemek için `foo` ana dizini güncellenmelidir; bu güncelleme mevcut bir `foo` veri bloğuna sığabilir veya yeni bir bloğun tahsis edilmesini gerektirebilir (ilişkili veri bit işlemi ile).Hem dizinin yeni uzunluğunu yansıtmak hem de zaman alanlarını (son değiştirilme zamanı gibi) güncellemek için `foo`'nun düğümü de güncellenmelidir. Genel olarak, sadece yeni bir dosya oluşturmak için çok iş var! Belki bir dahaki sefere bunu yaptığında daha minnettar olmalısın ya da en azından her şeyin bu kadar iyi çalıştığına şaşırmalısın.

41.4 İlkeler: Dosyalar ve Dizinler Nasıl Tahsis Edilir

Bu grup yapısı yürürlükteyken, FFS'nin artık performansı artırmak için dosyaları, dizinleri ve ilişkili meta verileri diske nasıl yerleştireceğine karar vermesi gerekiyor. Temel mantra basittir: *ilgili şeyleri bir arada tutun* (ve bunun sonucu olarak, *ilgisiz şeyleri birbirinden uzak tutun*).

Bu nedenle, mantraya uymak için HDS'nin neyin "ilgili" olduğuna karar vermesi ve onu aynı blok grubu içine yerleştirmesi gerekir; tersine, ilgisiz öğeler farklı blok gruplarına yerleştirilmelidir. Bu amaca ulaşmak için HDS, birkaç basit yerleştirme sezgiselinden yararlanır.

İlki, dizinlerin yerleştirilmesidir. HDS basit bir yaklaşım kullanır: az sayıda tahsis edilmiş dizine (gruplar arasında dizinleri dengelemek için) ve çok sayıda boş düğüme (daha sonra bir grup dosya tahsis edebilmek için) sahip silindir grubunu bulun ve dizin verilerini ve o gruptaki inode. Tabii ki, burada başka buluşsal yöntemler kullanılabilir (örneğin, serbest veri bloklarının sayısı hesaba katılarak).

Dosyalar için HDS iki şey yapar. Birincisi, (genel durumda) bir dosyanın veri bloklarının kendi inode'u ile aynı grupta tahsis edilmesini sağlar, böylece inode ve data arasında (eski dosya sisteminde olduğu gibi) uzun aramaları önler. İkincisi, aynı dizinde bulunan tüm dosyaları bulundukları dizinin silindir grubuna yerleştirir. Böylece, bir kullanıcı `/a/b` olmak üzere dört dosya oluşturursa, `/a/c`, `/a/d` ve `b/f`, HDS ilk üçünü birbirine yakın (aynı grup) ve dördüncüyü uzağa (başka bir grupta) yerleştirmeye çalışırdı.

Böyle bir tahsis örneğine bakalım. Örnekte, her grupta yalnızca 10 düğüm ve 10 veri bloğu olduğunu varsayalım (her ikisi de

gerçekçi olmayan küçük sayılar) ve üç dizin (kök dizin `/`, `/a` ve `/b`) ve dört dosya (`/a/c`, `/a/d`, `/a/e`, `/b/f`) HDS politikalarına göre içlerine yerleştirilir. Normal dosyaların her birinin iki blok boyutunda olduğunu ve dizinlerin yalnızca tek bir veri bloğuna sahip olduğunu varsayalım. Bu şekil için, her dosya veya dizin için bariz sembolleri kullanıyoruz (yani, kök dizin için `/`, `/a` için `a`, `/b/f` için `f` vb.).

```
group inodes      data
    0 /           /
    1 acde------ accddee---
    2 bf          bff
    3             
    4             
    5             
    6             
    7             
```

HDS ilkesinin iki olumlu şey yaptığını unutmayın: her dosyanın veri blokları, her dosyanın inode'una yakındır ve aynı dizindeki dosyalar birbirine yakındır (yani, `/a/c`, `/a/d` ve `/a/e` tümü Grup 1'dedir ve /b dizini ve `/b/f` dosyası Grup 2'de birbirine yakındır). Buna karşılık, şimdi, hiçbir grubun inode tablosunun hızla dolmamasını sağlamaya çalışarak, inode'ları gruplara basitçe yayan bir inode ayırma politikasına bakalım. Nihai tahsis, bu nedenle şöyle görünebilir:

```
group inodes      data
    0 /           /
    1 a           a
    2 b           b
    3 c           cc
    4 d           dd
    5 e           ee
    6 f           ff
    7             
```

Şekilden de görebileceğiniz gibi, bu politika gerçekten de dosya (ve dizin) verilerini ilgili inode'unun yakınında tutarken, bir dizindeki dosyalar keyfi olarak diskin etrafına yayılır ve bu nedenle ada dayalı konum korunmaz. HDS yaklaşımına göre `/a/c`, `/a/d` ve `/a/e` dosyalarına erişim artık bir yerine üç grubu kapsıyor.

HDS politika buluşsal yöntemleri, dosya sistemi trafiğine veya özellikle nüanslı herhangi bir şeye ilişkin kapsamlı araştırmalara dayanmaz; daha ziyade, eski moda **sağduyuya (common sense)** dayalıdırlar (sonuçta CS'nin anlamı bu değil mi?)[1]. Bir dizindeki dosyalara genellikle birlikte erişilir: bir grup dosyayı derlediğinizi ve ardından bunları tek bir yürütülebilir dosyaya bağladığınızı hayal edin.

[1]Bazı insanlar, özellikle düzenli olarak atlarla çalışan insanlar, sağduyuya **at duygusu** (horse sense) derler. Ancak, "mekanize at", yani araba popülerlik kazandıkça bu deyimin kaybolabileceğine dair bir his var. Bundan sonra ne icat edecekler? Uçan bir makine mi??!!

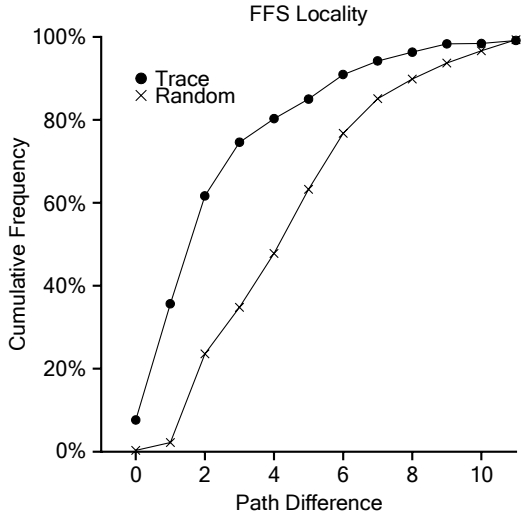


Figure 41.1: **FFS Locality For SEER Traces**

Bu tür ad alanı tabanlı konum bulunduğundan, HDS genellikle performansı iyileştirerek ilgili dosyalar arasındaki aramaların güzel ve kısa olmasını sağlar.

41.5 Dosya Konumunu Ölçme

Bu buluşsal yöntemlerin anlamlı olup olmadığını daha iyi anlamak için, dosya sistemi erişiminin bazı izlerini inceleyelim ve gerçekten ad-uzayı konumu olup olmadığına bakalım. Bazı nedenlerden dolayı, literatürde bu konuyla ilgili iyi bir çalışma yok gibi görünüyor.

Spesifik olarak, SEER izlemelerini [K94] kullanacağız ve dizin ağacında dosya erişimlerinin birbirinden ne kadar "uzak" olduğunu analiz edeceğiz. Örneğin, `f` dosyası açılırsa ve ardından izlemede bir sonraki dosya açılırsa (başka herhangi bir dosya açılmadan önce), dizin ağacındaki bu iki açılış arasındaki mesafe sıfırdır (çünkü bunlar aynı dosyadır). Dizin dizinindeki (yani, `dir/f`) bir `f` dosyası açılırsa ve ardından aynı dizindeki (yani, `dir/g`) g dosyasının açılması gelirse, iki dosya erişimi arasındaki mesafe, paylaştıkları için birdir aynı dizin ama aynı dosya değil. Başka bir deyişle, mesafe ölçütümüz, iki dosyanın *ortak atasını(common ancestor)* bulmak için dizin ağacında ne kadar yukarı gitmeniz gerektiğini ölçer; ağaca ne kadar yakınsa, metrik o kadar düşük olur.

Şekil 41.1, tüm izlerin tamamı boyunca SEER kümesindeki tüm iş istasyonlarında SEER izlerinde gözlemlenen konumu göstermektedir. Grafik, fark metriğini x ekseni boyunca çizer ve y ekseni boyunca bu farktan olan kümülatif dosya açma yüzdesini gösterir. Spesifik olarak, SEER izlemeleri için (grafikte "İzleme" olarak işaretlenmiştir), dosya erişimlerinin yaklaşık %7'sinin daha önce açılan dosyaya ve dosya erişimlerinin yaklaşık %40'ının ya

aynı dosyaya ya da dosyaya yapıldığını görebilirsiniz. aynı dizinde bire (yani, sıfır veya birlik bir fark). Bu nedenle, FFS yerellik varsayımı mantıklı görünüyor (en azından bu izler için).

İlginç bir şekilde, dosya erişimlerinin yaklaşık %25'i, iki mesafeye sahip dosyalara yapıldı. Bu tür bir konum, kullanıcı bir dizi ilgili dizini çok düzeyli bir şekilde yapılandırdığında ve sürekli olarak bunlar arasında atladığında ortaya çıkar. Örneğin, bir kullanıcının bir `src` dizini varsa ve nesne dosyalarını (`.o` dosyaları) bir `obj` dizinine oluşturuyorsa ve bu dizinlerin her ikisi de bir ana proje dizininin alt dizinleriyse, ortak erişim modeli `proj/src/foo` olacaktır. `.c` ve ardından `proj/obj/foo.o`. Bu iki erişim arasındaki mesafe ikidir, çünkü proje ortak atadır. HDS, politikalarında bu tür bir konumu yakalamaz ve bu nedenle bu tür erişimler arasında daha fazla arama gerçekleşir.

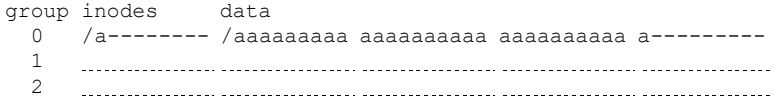
Karşılaştırma için, grafik ayrıca "Rastgele" bir iz için konumu da gösterir. Rastgele iz, mevcut bir SEER izinden dosyaların rasgele sırada seçilmesi ve bu rasgele sıralanmış erişimler arasındaki mesafe metriğinin hesaplanmasıyla oluşturulmuştur. Gördüğünüz gibi, beklendiği gibi rastgele izlemelerde daha az ad alanı konumu var. Ancak, sonunda her dosya ortak bir atayı (örneğin kök) paylaştığından, bir miktar yerellik vardır ve bu nedenle rastgele bir karşılaştırma noktası olarak kullanışlıdır.

41.6 Büyük Dosya İstisnası

HDS'de, genel dosya yerleştirme politikasının önemli bir istisnası vardır ve bu, büyük dosyalar için ortaya çıkar. Farklı bir kural olmadan, büyük bir dosya ilk yerleştirildiği blok grubunu (ve belki diğerlerini) tamamen dolduracaktır. Bir blok grubunu bu şekilde doldurmak istenmez, çünkü sonraki "ilgili" dosyaların bu blok grubuna yerleştirilmesini engeller ve bu nedenle dosya erişim konumuna zarar verebilir.

Böylece, büyük dosyalar için HDS aşağıdakileri yapar. İlk blok grubuna belirli sayıda blok tahsis edildikten sonra (örneğin, 12 blok veya bir inode içinde mevcut olan doğrudan işaretçilerin sayısı), FFS dosyanın bir sonraki "büyük" yığınını yerleştirir (örneğin, tarafından işaret edilenler) ilk dolaylı blok) başka bir blok grubunda (belki de düşük kullanımı için seçilmiştir). Ardından, dosyanın bir sonraki öbeği başka bir farklı blok grubuna yerleştirilir ve bu böyle devam eder.

Bu politikayı daha iyi anlamak için bazı şemalara bakalım. Büyük dosya istisnası olmadan, tek bir büyük dosya tüm bloklarını diskin bir bölümüne yerleştirir. Grup başına 10 düğüm ve 40 veri bloğu ile yapılandırılmış bir FFS'de 30 blok içeren küçük bir dosya (`/a`) örneğini araştırıyoruz. Büyük dosya istisnası olmadan HDS'nin tasviri aşağıdadır:



Resimde görebileceğiniz gibi `/a`, Grup 0'daki veri bloklarının çoğunu doldururken, diğer gruplar boş kalır. Kök dizinde (`/`) şimdi başka dosyalar oluşturulmuşsa, gruptaki verileri için fazla yer yoktur.

Büyük dosya istisnasıyla (burada her yığında beş bloğa ayarlanmıştır), HDS bunun yerine dosyayı gruplara yayar ve bunun sonucunda herhangi bir grup içinde kullanım çok yüksek olmaz:

```
group inodes      data
  0   /a          /aaaaa
  1               aaaaa
  2               aaaaa
  3               aaaaa
  4               aaaaa
  5               aaaaa
  6
```

Akıllı okuyucu (yani sizsiniz), bir dosyanın bloklarını diske yaymanın, özellikle görece yaygın sıralı dosya erişimi durumunda (örneğin, bir kullanıcı veya uygulama 0'dan 29'a kadar olan parçaları sırayla okuduğunda) performansa zarar vereceğini fark edecektir. . Ve haklısın, ey zeki okuyucumuz! Ancak yığın boyutunu dikkatli bir şekilde seçerek bu sorunu çözebilirsiniz.

Spesifik olarak, parça boyutu yeterince büyükse, dosya sistemi zamanının çoğunu diskten veri aktarmakla ve (nispeten) küçük bir süre bloğun parçaları arasında arama yapmakla geçirecektir. Ödenen genel gider başına daha fazla iş yaparak bir genel gideri azaltma işlemine **amortisman(amortization)** denir ve bilgisayar sistemlerinde yaygın bir tekniktir.

Bir örnek yapalım: bir disk için ortalama konumlandırma süresinin (yani arama ve döndürme) 10 ms olduğunu varsayalım. Ayrıca diskin 40 MB/s hızında veri aktardığını varsayalım. Amacınız, zamanımızın yarısını parçalar arasında arama yapmak ve zamanımızın yarısını veri aktarmak (ve böylece en yüksek disk performansının %50'sini elde etmek) olsaydı, bu nedenle, her 10 ms konumlandırma için veri aktarımı için 10 ms harcamanız gerekirdi. Böylece soru şu olur: Transferde 10 ms harcamak için bir parçanın ne kadar büyük olması gerekir? Kolay, sadece eski dostumuz matematiği, özellikle de diskler [AD14a] bölümünde bahsedilen boyutsal analizi kullanın:

$$\frac{40 \cancel{MB}}{\cancel{sec}} \cdot \frac{1024\ KB}{1 \cancel{MB}} \cdot \frac{1\ \cancel{sec}}{1000\ \cancel{ms}} \cdot 10\ \cancel{ms} = 409.6\ KB \tag{41.1}$$

Temel olarak, bu denklemin söylediği şey şudur: 40'ta veri aktarırsanız MB/sn, zamanınızın yarısını aramaya ve yarısını aktarmaya harcamak için her aradığınızda yalnızca 409.6KB aktarmanız gerekir. Benzer şekilde, en yüksek bant genişliğinin %90'ına (yaklaşık 3,6 MB olduğu ortaya çıkıyor) veya hatta en yüksek bant genişliğinin %99'una (39,6 MB!) ulaşmak için ihtiyaç duyacağınız yığın boyutunu hesaplayabilirsiniz. Gördüğünüz gibi, zirveye ne kadar yaklaşırsanız, bu parçalar o kadar büyür (bu değerlerin grafiği için Şekil 41.2'ye bakın).

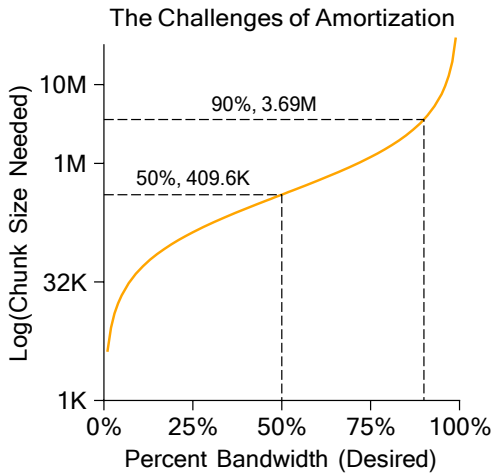


Figure 41.2: **Amortisman: Parçalar Ne Kadar Büyük Olmalı?**

Ancak HDS, büyük dosyaları gruplara dağıtmak için bu tür bir hesaplama kullanmadı. Bunun yerine, inode'un kendi yapısına dayanan basit bir yaklaşım benimsedi. İlk on iki doğrudan blok, inode ile aynı gruba yerleştirildi; sonraki her dolaylı blok ve işaret ettiği tüm bloklar farklı bir gruba yerleştirildi. 4KB blok boyutu ve 32 bit disk adresleriyle bu strateji, dosyanın (4MB) her 1024 bloğunun ayrı gruplara yerleştirildiğini ima eder; tek istisna, doğrudan işaretçiler tarafından işaret edilen dosyanın ilk 48KB'sidir. Disk sürücülerindeki eğilimin, disk üreticileri aynı yüzeye daha fazla bit sıkıştırmada iyi olduklarından, ancak sürücülerin aramalarla ilgili mekanik yönlerinin (disk kol hızı ve dönüş hızı) daha çok iyileşme gösterdiğinden, aktarım hızının oldukça hızlı bir şekilde arttığına dikkat edin. yavaşça [P98]. Bunun anlamı, zamanla mekanik maliyetlerin nispeten daha pahalı hale gelmesidir ve bu nedenle söz konusu maliyetleri amorti etmek için aramalar arasında daha fazla veri aktarmanız gerekir.

41.7 HDS Hakkında Diğer Birkaç Şey

HDS birkaç yenilik daha getirdi. Özellikle, tasarımcılar küçük dosyaları barındırma konusunda son derece endişeliydiler; Görünüşe göre, o zamanlar birçok dosyanın boyutu 2 KB kadardı ve 4 KB blokları kullanmak, veri aktarımı için iyi olsa da, alan verimliliği için o kadar iyi değildi. Bu **dahili parçalanma(internal fregmantation)**, diskin kabaca yarısının tipik bir dosya sistemi için boşa harcanmasına yol açabilir.

HDS tasarımcılarının bulduğu çözüm basitti ve sorunu çözdü. Dosya sisteminin dosyalara ayırabileceği 512 baytlık küçük bloklar olan **alt bloklar(sub-blocks)** sunmaya karar verdiler. Bu nedenle, küçük bir dosya oluşturduysanız (1 KB boyutunda),iki alt bloğu kaplar ve bu nedenle

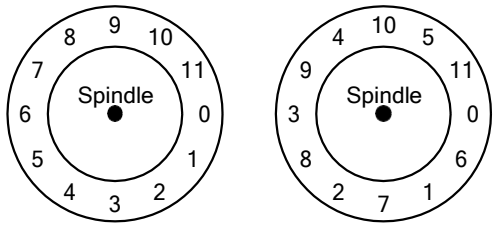


Figure 41.3: **HDS: Standart ve Parametreli Yerleştirme**

4 KB'lik bir bloğun tamamını boşa harcayın. Dosya büyüdükçe, dosya sistemi tam 4 KB veri elde edene kadar 512 baytlık blokları ayırmaya devam edecektir. Bu noktada, FFS 4 KB'lik bir blok bulacak, alt blokları buna kopyalayacak ve alt blokları ileride kullanmak üzere serbest bırakacaktır.

Bu işlemin verimsiz olduğunu, dosya sistemi için çok fazla ek iş gerektirdiğini (özellikle kopyalamayı gerçekleştirmek için çok fazla G/Ç) gerektirdiğini gözlemleyebilirsiniz. Ve yine haklısın! Bu nedenle, HDS genellikle `libc` kitaplığını değiştirerek bu kötümser davranıştan kaçındı; kitaplık, yazma işlemlerini arabelleğe alır ve ardından bunları 4 KB'lik parçalar halinde dosya sistemine verir, böylece çoğu durumda alt blok uzmanlaşmasından tamamen kaçınılır.

HDS'nin getirdiği ikinci güzel şey, performans için optimize edilmiş bir disk düzeniydi. O zamanlarda (SCSI ve diğer daha modern cihaz arayüzlerinden önce), diskler çok daha az karmaşıktı ve ana CPU'nun işlemlerini daha uygulamalı bir şekilde kontrol etmesini gerektiriyordu. Şekil 41.3'te solda olduğu gibi, diskin ardışık sektörlerine bir dosya yerleştirildiğinde FFS'de bir sorun ortaya çıktı.

Özellikle, sıralı okumalar sırasında sorun ortaya çıktı. FFS önce 0'ı bloke etmek için bir okuma verir; okuma tamamlandığında ve FFS 1. blok için bir okuma yayınladığında, artık çok geçti: 1. blok kafanın altında dönmüştü ve şimdi 1. blok için okuma tam bir dönüşe neden olacaktı.

HDS, Şekil 41.3'te sağda görebileceğiniz gibi bu sorunu farklı bir düzen ile çözdü. Diğer tüm blokları atlayarak (örnekte), HDS'nin bir sonraki bloğu disk kafasını geçmeden önce istemek için yeterli zamanı olur. Aslında, HDS, ekstra dönüşlerden kaçınmak için belirli bir disk için kaç blok atlaması gerektiğini anlayacak kadar akıllıydı; HDS, diskin belirli performans parametrelerini anlayacağı ve bunları tam kademeli düzen şemasına karar vermek için kullanacağı için bu tekniğe **parametrelendirme(parameterization)** adı verildi.

Düşünüyor olabilirsiniz: bu şema o kadar da iyi değil. Aslında, bu tür bir düzende en yüksek bant genişliğinin yalnızca %50'sini elde edeceksiniz, çünkü her bloğu bir kez okumak için her yol boyunca iki kez dolaşmanız gerekir. Neyse ki, modern diskler çok daha akıllıdır: dahili olarak tüm izi okurlar ve onu dahili bir disk önbelleğinde (tam da bu nedenle genellikle **iz arabelleği(track buffer)** olarak adlandırılır) ara belleğe alırlar. Ardından, izin sonraki okumalarında, disk

İPUCU: SİSTEMİ KULLANIŞLI HALE GETİRİN

Muhtemelen HDS'den çıkarılacak en temel ders, yalnızca kavramsal olarak iyi olan disk farkındalı düzen fikrini ortaya koymakla kalmamış, aynı zamanda sistemi daha kullanışlı hale getiren bir dizi özellik de eklemiştir. Uzun dosya adları, sembolik bağlantılar ve atomik olarak çalışan bir yeniden adlandırma işlemi, tümü bir sistemin faydasını iyileştirdi; hakkında bir araştırma makalesi yazmak zor olsa da ("The Symbolic Link: Hard Link's Long Lost Cousin" hakkında 14 sayfalık bir makale okumaya çalıştığınızı hayal edin), bu tür küçük özellikler HDS'yi daha kullanışlı hale getirdi ve dolayısıyla benimsenme şansını muhtemelen artırdı. Bir sistemi kullanılabilir hale getirmek, genellikle derin teknik yenilikleri kadar veya ondan daha önemlidir.

sadece istenen verileri önbelleğinden döndürün. Böylece dosya sistemleri artık bu inanılmaz derecede düşük düzeyli ayrıntılar hakkında endişelenmek zorunda kalmıyor. Soyutlama ve daha üst düzey arayüzler, uygun şekilde tasarlandığında iyi bir şey olabilir.

Diğer bazı kullanılabilirlik iyileştirmeleri de eklendi. HDS, **uzun dosya adlarına(long file names)** izin veren ilk dosya sistemlerinden biriydi, böylece dosya sisteminde geleneksel sabit boyutlu yaklaşım (ör. 8 karakter) yerine daha anlamlı adlar sağladı.Ayrıca, **sembolik bağlantı(symbolic link)** adı verilen yeni bir kavram tanıtıldı. Önceki bir bölümde [AD14b] tartışıldığı gibi, sabit bağlantılar, her ikisinin de dizinlere işaret edememesi (dosya sistemi hiyerarşisinde döngüler oluşturma korkusuyla) ve yalnızca aynı birim içindeki dosyalara işaret edebilmeleri (yani, inode numarası hala anlamlı olmalıdır). Sembolik bağlantılar, kullanıcının bir sistemdeki başka herhangi bir dosya veya dizine bir "takma ad" oluşturmasına izin verir ve bu nedenle çok daha esnektir. HDS, dosyaları yeniden adlandırmak için atomik `rename()` işlemi de başlattı. Temel teknolojinin ötesindeki kullanılabilirlik iyileştirmeleri de muhtemelen HDS'ye daha güçlü bir kullanıcı tabanı kazandırdı.

41.8 Özet

HDS'nin tanıtılması dosya sistemi tarihinde bir dönüm noktası oldu, çünkü dosya yönetimi sorununun bir işletim sistemindeki en ilginç sorunlardan biri olduğunu açıkça ortaya koydu ve birinin bu en önemli sorunla nasıl başa çıkabileceğini gösterdi. aygıtlar, sabit disk. O zamandan beri yüzlerce yeni dosya sistemi geliştirildi, ancak bugün hala birçok dosya sistemi HDS'den ipuçları alıyor (örneğin, Linux ext2 ve ext3 bariz entelektüel torunlardır). Kesinlikle tüm modern sistemler, HDS'nin ana dersini açıklar: diske bir diskmiş gibi davranın.

References

[AD14a] "Operating Systems: Three Easy Pieces" (Chapter: Hard Disk Drives) by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Arpaci-Dusseau Books, 2014. *There is no way you should be reading about FFS without having first understood hard drives in some detail. If you try to do so, please instead go directly to jail; do not pass go, and, critically, do not collect 200 much-needed simoleons.*

[AD14b] "Operating Systems: Three Easy Pieces" (Chapter: File System Implementation) by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau . Arpaci-Dusseau Books, 2014. *As above, it makes little sense to read this chapter unless you have read (and understood) the chapter on file system implementation. Otherwise, we'll be throwing around terms like "inode" and "indirect block" and you'll be like "huh?" and that is no fun for either of us.*

[K94] "The Design of the SEER Predictive Caching System" by G. H. Kuenning. MOBICOMM '94, Santa Cruz, California, December 1994. *According to Kuenning, this is the best overview of the SEER project, which led to (among other things) the collection of these traces.*

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM TOCS, 2:3, August 1984. *McKusick was recently honored with the IEEE Reynold B. Johnson award for his contributions to file systems, much of which was based on his work building FFS. In his acceptance speech, he discussed the original FFS software: only 1200 lines of code! Modern versions are a little more complex, e.g., the BSD FFS descendant now is in the 50-thousand lines-of-code range.*

[P98] "Hardware Technology Trends and Database Opportunities" by David A. Patterson. Keynote Lecture at SIGMOD '98, June 1998. *A great and simple overview of disk technology trends and how they change over time.*

Ödev(Simülasyon)

Bu bölüm, FFS tabanlı dosya ve dizin ayırmanın nasıl çalıştığını daha iyi anlamak için kullanabileceğiniz basit bir HDS simülatörü olan `ffs.py`'yi tanıtmaktadır. Simülatörün nasıl çalıştırılacağına ilişkin ayrıntılar için README'ya bakın.

Sorular

1. `in.largefile` dosyasını inceleyin ve ardından `-f in.largefile` ve `-L 4` bayrağıyla simülatörü çalıştırın. İkincisi, büyük dosya istisnasını 4 bloğa ayarlar. Ortaya çıkan tahsis neye benzeyecek? Kontrol etmek için `-c` ile çalıştırın.
2. Şimdi `-L 30` ile çalıştırın. Ne görmeyi bekliyorsunuz? Bir kez daha, haklı olup olmadığınızı görmek için `-c`'yi açın. `/a` dosyasına tam olarak hangi blokların tahsis edildiğini görmek için `-S`'yi de kullanabilirsiniz.
3. Şimdi dosya hakkında bazı istatistikler hesaplayacağız. Birincisi, dosyanın herhangi iki veri bloğu arasındaki veya inode ile herhangi bir veri bloğu arasındaki maksimum mesafe olan filespan dediğimiz şeydir. `/a`'nın dosya genişliğini hesaplayın. Ne olduğunu görmek için `ffs.py -f in.largefile -L 4 -T -c` komutunu çalıştırın. `-L 100` ile aynısını yapın. Büyük dosya özel durum parametresi düşük değerlerden yüksek değerlere değişirken, filespan'de ne gibi bir fark beklersiniz?
4. Şimdi yeni bir girdi dosyasına bakalım, `in.manyfiles`. HDS politikasının bu dosyaları gruplara nasıl yerleştireceğini düşünüyorsunuz? (hangi dosyaların ve dizinlerin oluşturulduğunu görmek için `-v` ile çalıştırabilir veya yalnızca `cat in.manyfiles` ile çalıştırabilirsiniz). Haklı olup olmadığınızı görmek için simülatörü `-c` ile çalıştırın.
5. HDS'yi değerlendirmek için bir metriğe *dirspan* denir. Bu ölçüm, belirli bir dizindeki dosyaların yayılmasını, özellikle dizindeki tüm dosyaların inode'ları ve veri blokları ile dizinin kendisinin inode'u ve data bloğu arasındaki maksimum mesafeyi hesaplar. `in.manyfiles` ve `-T` bayrağıyla çalıştırın ve üç dizinin dirspan'ını hesaplayın. Kontrol etmek için `-c` ile çalıştırın. HDS, dirspan'i en aza indirmede ne kadar iyi bir iş çıkarıyor?
6. Şimdi grup başına inode tablosunun boyutunu 5 (`-I 5`) olarak değiştirin. Bunun dosyaların düzenini nasıl değiştireceğini düşünüyorsunuz? Haklı olup olmadığınızı görmek için `-c` ile çalıştırın. Dirspan'i nasıl etkiler?
7. HDS yeni bir dizinin inode'unu hangi gruba yerleştirmelidir? Varsayılan (simülatör) politikası, en fazla boş düğüme sahip grubu arar. Farklı bir ilke, en fazla serbest düğüme sahip bir grup grubu arar. Örneğin, `-A 2` ile çalıştırırsanız, yeni bir dizin tahsis ederken, simülatör gruplara çiftler halinde bakar ve tahsis için en iyi çifti seçer. Bu stratejiyle ayırmanın nasıl değiştiğini görmek için `./ffs.py -f in.manyfiles -I 5 -A 2 -c` komutunu çalıştırın. Dirspan'ı nasıl etkiler? Bu politika neden iyi olabilir?
8. Keşfedeceğimiz son bir politika değişikliği, dosya parçalanmasıyla ilgilidir. `./ffs.py -f in.fragmented -v` komutunu çalıştırın ve kalan dosyaların nasıl tahsis edildiğini tahmin edip edemeyeceğinize bakın. Cevabınızı onaylamak için `-c` ile çalıştırın `/i` dosyasının veri düzeninde ilginç olan nedir? Neden sorunlu?
9. Bitişik tahsis (`-C`) dediğimiz yeni bir politika, her dosyanın bitişik olarak tahsis edilmesini sağlamaya çalışır. Spesifik olarak, `-C n` ile dosya sistemi, bir bloğu tahsis etmeden önce bir grup içinde `n` bitişik bloğun serbest olmasını sağlamaya

çalışır. Farkı görmek için `./ffs.py -f in.fragmented -v -C 2 -c` komutunu çalıştırın. `-C`'ye geçirilen parametre arttıkça layout nasıl değişir? Son olarak, `-C` filespan ve dirspan'i nasıl etkiler?