41

Yerellik ve Hızlı Dosya Sistemi (Locality and The Fast File System)

Unix işletim sistemi ilk kez tanıtıldığında, Unix büyücüsü Ken Thompson ilk dosya sistemini yazdı. Buna "eski Unix dosya sistemi" diyelim. Bu dosya sistemi oldukça basitti. Temelde veri yapıları diskte şöyle görünüyordu:

Süper blok (S) tüm dosya sistemi hakkında bilgi içeriyordu: bölümün (volume) ne kadar büyük olduğu, kaç tane dizin düğümü (inode-index node) olduğu, boş blok listesinin başına bir işaretçi (pointer) vb. Diskin dizin düğümü bölgesi diskteki bütün dizin düğümlerini içeriyordu. Son olarak diskin büyük bir bölümü veri bloklarından oluşuyordu.

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

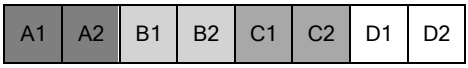
41.1 Sorun: Düşük Performans

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

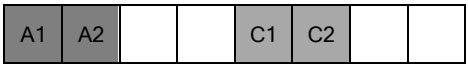
Asıl sorun, eski UNIX dosya sisteminin diske rastgele erişimli bir bellekmiş gibi davranmasıydı; veriler, verileri tutan ortamın bir disk olduğu ve dolayısıyla gerçek ve pahalı konumlandırma maliyetlerine sahip olduğu gerçeğine bakılmaksızın her yere yayıldı. Örneğin, bir dosyanın veri blokları genellikle dizin düğümünden çok uzaktı, bu nedenle dizin düğümünü ve ardından bir dosyanın veri bloklarını ilk okuduğunda pahalı bir aramaya neden oldu (oldukça yaygın bir işlem).

Daha da kötüsü, boş alan dikkatli bir biçimde yönetilmediği için dosya sistemi oldukça **parçalanacaktı (Fragmented)**. Boşluk listesi diske yayılmış bir grup bloğu işaret eder ve dosyalar tahsis edildiğinde bir sonraki boş bloğu alırlar. Sonuç olarak mantıkça sürekli bir dosyaya diskte ileri geri giderek erişilecek ve performans önemli ölçüde düşecektir.

Örneğin her biri 2 blok boyutunda olan dört dosya (A, B, C ve D) içeren aşağıdaki veri bloğu bölgesini düşünün:



B ve D silinirse, ortaya çıkan düzen şöyle olur:



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

Ne 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 meydana geldi 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 düzenler, verileri hareket ettirir ve daha sonra değişiklikleri yansıtmak için dizin düğümleri ve benzerlerini yeniden yazarlar.

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)** (blok içindeki israfı) en aza indirdiler, ancak her bloğun ona ulaşması için bir duruş yükü gerektirebileceğinden transfer için kötüydü. Böylece, sorun:

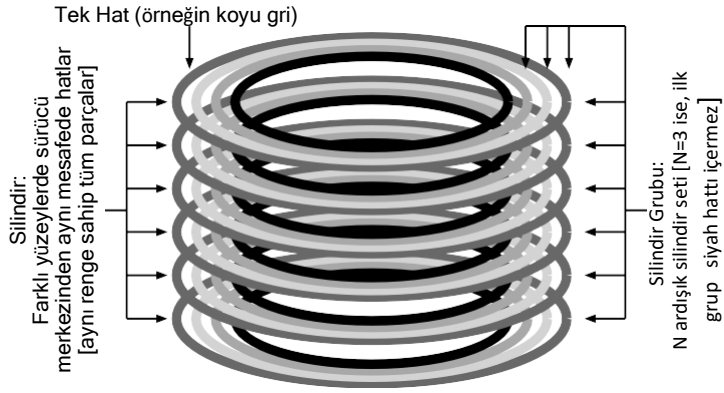
DÖNÜM NOKTASI:
PERFORMANSI ARTIRMAK IÇIN DISK ÜZERINDEKI VERILER NASIL DÜZENLENIR
Dosya sistemi veri yapılarını, uygunluğu artırmak için nasıl düzenleyebiliriz? Bu veri yapılarının üstünde ne tür tahsis politikalarına ihtiyacımız var? Dosya sistemini "disk farkında " hale nasıl getirebiliriz?

41.2 FFS: Çözüm Disk Farkındalığı

Berkeley'deki bir grup, akıllıca **Hızlı Dosya Sistemi (FFS) (Fast File System)** 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 da tam olarak yaptıkları şeydi. FFS böylece dosya sistemi araştırmalarında yeni bir çağ getirdi; dosya sistemi için aynı *arabirimi* koruyarak (open (), read (), write (), close () ve diğer dosya sistemi çağrıları dahil olmak üzere aynı API'ler) ancak dahili uygulamayı değiştirerek, yazarlar bugün hala devam eden yeni dosya sistemi inşasının yolunu açtılar. Hemen hemen tüm modern dosya sistemleri, performans, güvenilirlik veya başka nedenlerle iç kısımlarını değiştirirken mevcut arayüzüne bağlı kalır (ve böylece uygulamalarla uyumluluğu korur).

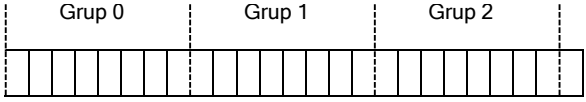
41.3 Organizasyon Yapısı: Silindir Grubu

İlk adım, disk üzerindeki yapıları değiştirmekti. FFS, diski bir dizi **silindir grubuna (cylinder groups)** böler. **Tek silindir (cylinder),** bir sabit sürücünün farklı yüzeylerinde, sürücünün merkezinden aynı mesafede bulunan bir dizi hattır; Geometrik şekle olan açık benzerliği nedeniyle silindir olarak adlandırılır. FFS, *N* ardışık silindirleri bir grupta toplar ve böylece tüm disk silindir gruplarının bir koleksiyonu olarak görülebilir. İşte altı plakalı bir sürücünün en dış dört hattı ve üç silindirden oluşan bir silindir grubunu gösteren basit bir örnek:



Modern sürücülerin, dosya sisteminin belirli bir silindirin kullanımda olup olmadığını gerçekten anlaması için yeterli bilgiyi dışa aktarmadığını unutmayın; Daha önce tartışıldığı gibi [AD14a], diskler blokların mantıksal 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 ve bunların her biri diskin adres alanının yalnızca ardışık bir parçasıdır. Aşağıdaki resim, her 8 bloğun farklı bir blok grubunda düzenlendiği bir örneği göstermektedir (gerçek grupların 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, aynı gruba iki dosya yerleştirerek, birbiri ardına erişmenin diskte uzun aramalarla sonuçlanmamasını sağlayabilir. Dosyaları ve dizinleri depolamak üzere bu grupları kullanmak için, FFS'nin dosyaları ve dizinleri bir gruba yerleştirme ve bunlarla ilgili gerekli tüm bilgileri izleme yeteneğine sahip olması gerekir. Bunu yapmak için FFS, bir dosya sisteminin her grupta sahip olmasını bekleyebileceğiniz tüm yapıları içerir, örneğin, dizin düğümleri için alan, veri blokları ve bazı yapılar bunların her birinin tahsis edilip edilmediğini veya ücretsiz olup olmadığını izleyin. İşte FFS'nin tek bir içinde tuttuğu şeyin bir tasviri

Silindir grubu:

Şimdi bu tek silindirli grubun bileşenlerini daha ayrıntılı olarak inceleyelim. FFS, güvenilirlik nedenleriyle her grupta **süper bloğun (super block)** (S) bir kopyasını tutar. Dosya sistemini yüklemek için süper blok gereklidir; birden çok kopya tutarak, bir kopya bozulursa, çalışan bir çoğaltma kullanarak dosya sistemini bağlayabilir ve dosya sistemine erişebilirsiniz.

Her grupta, FFS'nin grubun dizin düğümlerinin ve veri bloklarının tahsis edilip edilmediğini izlemesi gerekir. Grup başına **dizin düğüm bit eşlem (inode bitmap) (ib)** ve **veri bit eşlem (data bitmap)** (db), her gruptaki dizin düğümler ve veri blokları için bu rolü üstlenir. Bit eşlemler, bir dosya sistemindeki boş alanı yönetmenin mükemmel bir yoludur, çünkü bir dosyaya tahsis etmek için büyük bir boş alan yığını bulmak kolaydır.

Son olarak, **dizin düğüm** ve **veri bloğu** bölgeleri, önceki çok basit dosya sistemindeki (VSFS) bölgeler gibidir. Her silindir grubunun çoğu, her zamanki gibi, veri bloklarından oluşur.

FFS DOSYA OLUŞTURMA

Örnek olarak, bir dosya oluşturulduğunda hangi veri yapılarının güncelleştirilmesi gerektiğini düşünün; bu örnekte, kullanıcının yeni bir dosya oluşturduğunu
/foo/bar.txt ve dosyanın bir blok uzunluğunda (4KB) olduğunu varsayalım. Dosya yenidir ve bu nedenle yeni bir dizin düğümüne ihtiyaç duyar; Böylece hem dizin düğüm bit eşlemi hem de yeni tahsis edilen dizin düğümü diske yazılacaktır. Dosya ayrıca içinde veri içerir ve bu nedenle de tahsis edilmelidir; veri bit eşlemi ve bir veri bloğu böylece (nihayetinde) diske yazılacaktır. Bu nedenle, mevcut silindir grubuna en az dört yazma işlemi gerçekleşecektir (bu yazmaların gerçekleşmeden önce bir süre bellekte arabelleğe alınabileceğini unutmayın). Ama hepsi bu kadar değil! Özellikle, yeni bir dosya oluştururken, dosyayı dosya sistemi hiyerarşisine de yerleştirmeniz gerekir, yani dizin güncellenmelidir. Özellikle, üst dizin foo'nun bar.txt girişini eklemek için güncelleştirilmesi gerekir; bu güncelleştirme var olan bir foo veri bloğuna sığabilir veya yeni bir bloğun tahsis edilmesini gerektirebilir (ilişkili veri bit eşlemi ile). Foo'nun dizin düğümü hem dizinin yeni uzunluğunu yansıtacak hem de zaman alanlarını (son değiştirilen zaman gibi) güncellemek için güncellenmelidir. Genel olarak, sadece yeni bir dosya oluşturmak için çok fazla iş! Belki de bir dahaki sefere bunu yaptığınızda, daha müteşekkir olmalısınız ya da en azından her şeyin bu kadar iyi çalıştığına şaşırmalısınız.

41.4 İlkeler: Dosya ve Dizinler Nasıl Ayrılır

Bu grup yapısı uygulandığında, FFS artık performansı artırmak için dosyaların, dizinlerin ve ilişkili meta verilerin diske nasıl yerleştirileceğine karar vermelidir. Temel mantra basittir: ilgili şeyleri bir arada tutun (ve bunun sonucu, ilgisiz şeyleri birbirinden uzak tutun).

Bu nedenle, mantraya uymak için, FFS neyin "ilişkili" olduğuna karar vermeli ve aynı blok grubuna yerleştirmelidir; tersine, ilgisiz öğeler farklı blok gruplarına yerleştirilmelidir. Bu amaca ulaşmak için, FFS birkaç basit yerleşim buluşsal yönteminden yararlanır.

Birincisi, dizinlerin yerleştirilmesidir. FFS basit bir yaklaşım kullanır: düşük sayıda tahsis edilmiş dizin (gruplar arasında dizinleri dengelemek için) ve çok sayıda boş dizin düğümü (daha sonra bir grup dosya ayırabilmek için) ile silindir grubunu bulun ve dizin verilerini ve dizin düğümü bu gruba koyun. Tabii ki, burada başka sezgisel yöntemler de kullanılabilir (örneğin, boş veri bloklarının sayısını dikkate alarak).

Dosyalar için FFS iki şey yapar. İlk olarak, (genel durumda) bir dosyanın veri bloklarını dizin düğümü ile aynı grupta tahsis ettiğinden emin olur, böylece dizin düğümü ve veri arasında uzun arayışları önler (eski dosya sisteminde olduğu gibi). İkincisi, aynı dizinde bulunan tüm dosyaları, içinde bulundukları dizinin silindir grubuna yerleştirir. Böylece, bir kullanıcı dört dosya oluştururs a, /a/b,
/a/c, /a/d ve b/f, FFS ilk üçünü birbirine yakın (aynı grupta) ve dördüncüsünü uzak (başka bir grupta) yerleştirmeye çalışır.

Böyle bir tahsisat örneğine bakalım. Örnekte, her grupta yalnızca 10

dizin düğümü ve 10 veri bloğu olduğunu varsayın (her ikisi de gerçekçi olmayan şekilde küçük sayılar) ve FFS ilkelerine göre üç dizinin (kök dizin /, /a ve /b) ve dört dosyanın (/a/c, /a/d, /a/e, /b/f) içlerine yerleştirildiğini. 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 kullanırız (örneğin;
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
```

FFS ilkesinin iki olumlu şey yaptığını unutmayın: her dosyanın veri blokları her dosyanın dizin düğümünün yakınındadır ve aynı dizindeki dosyalar birbirine yakındır (yani, /a/c, /a/d ve /a/e'nin tümü Grup 1'dedir ve dizin /b ve onun dosyası /b/f Grup 2'de birbirine yakın).

Buna karşılık, şimdi dizin düğümleri gruplar arasında yayan ve hiçbir grubun dizin düğümü tablosunun hızlı bir şekilde dolmamasını sağlamaya çalışan bir dizin düğüm tahsis politikasına bakalım. Son tahsisat bu nedenle şöyle bir şeye benzeyebilir:

```
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 dizin düğümün yakınında tutarken, bir dizin içindeki dosyalar diskin etrafına keyfi olarak yayılır ve bu nedenle ada dayalı yerellik korunmaz. /a/c, /a /d ve /a/e dosyalarına erişim artık FFS yaklaşımına göre bir yerine üç gruba yayılıyor.

FFS politikası buluşsal yöntemleri, dosya sistemi trafiğinin kapsamlı çalışmalarına veya özellikle nüanslı herhangi bir şeye dayanmamaktadır; daha ziyade, eski moda **sağduyuya (common sense)** dayanıyorlar (sonuçta sağduyunun temsil ettiği şey 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]Some people refer to common sense as **horse sense**, especially people who work regularly with horses. However, we have a feeling that this idiom may be lost as the "mechanized horse", a.k.a. the car, gains in popularity. What will they invent next? A flying machine??!!

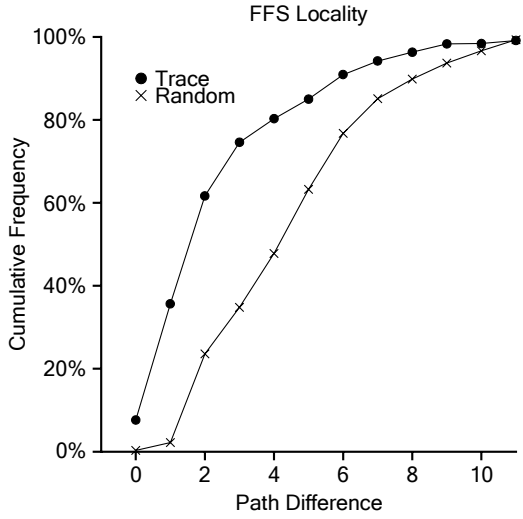


Figure 41.1: **FFS Locality for SEER Traces**

Bu tür ad alanı tabanlı yerellik bulunduğundan, FFS genellikle performansı artırarak ilgili dosyalar arasındaki aramaların güzel ve kısa olmasını sağlar.

41.5 Dosya Yerelliğini Ölçme

Bu sezgisel yöntemlerin mantıklı olup olmadığını daha iyi anlamak için, dosya sistemi erişiminin bazı izlerini analiz edelim ve gerçekten ad alanı yerelliği olup olmadığını görelim. Bazı nedenlerden dolayı, literatürde bu konuyla ilgili iyi bir çalışma yok gibi görünmektedir.

Özellikle, SEER izlerini [K94] kullanacağız ve dizin ağacında dosya erişimlerinin birbirinden ne kadar "uzak" olduğunu analiz edeceğiz. Örneğin, f dosyası açılır ve bir sonraki izlemede dosya yeniden açılırsa (diğer dosyalar açılmadan önce), dizin ağacında açılan bu ikisi arasındaki mesafe sıfırdır (aynı dosya oldukları için). Dir dizinindeki bir f dosyası (yani, dir/f) açılır ve ardından aynı dizinde (yani, dir/g) g dosyasının açılması durumunda, aynı dizini paylaştıkları ancak aynı dosya olmadıkları için iki dosya erişimi arasındaki mesafe birdir. Başka bir deyişle, mesafe metriğimiz, iki dosyanın ortak atasını bulmak için dizin ağacının ne kadar yukarısına gitmeniz gerektiğini ölçer; ağaçta ne kadar yakın olurlarsa, metrik o kadar düşük olur.

Şekil 41.1, SEER kümesindeki tüm iş istasyonları üzerindeki SEER izlerinde gözlemlenen lokaliteyi, tüm izlerin tamamı boyunca göstermektedir. Grafik, fark metriğini x ekseni boyunca çizer ve y ekseni boyunca bu farka ait olan dosya açılışlarının kümülatif yüzdesini gösterir. Özellikle, SEER izlemeleri için (grafikte "İzleme" olarak işaretlenmiştir), dosya erişimlerinin yaklaşık %7'sinin daha önce açılmış olan dosyaya ve dosya erişimlerinin yaklaşık% 40'ının aynı dosyaya veya aynı dizindeki bir dosyaya (yani, sıfır veya bir fark) olduğunu görebilirsiniz. Bu nedenle, FFS yerellik varsayımı mantıklı görünmektedir (en azından bu izler için).

İlginçtir ki, dosya erişimlerinin %25'i veya daha fazlası, iki mesafeye sahip dosyalara aitti. Bu tür bir yerellik, kullanıcı çok düzeyli bir şekilde bir dizi ilgili dizini yapılandırdığında ve sürekli olarak aralarında atladığında oluşur. Örneğin, bir kullanıcının src dizini varsa ve bir obj dizinine nesne dosyaları (.o dosyaları) oluşturuyorsa ve bu dizinlerin her ikisi de ana proj dizininin alt dizinleriyse, ortak bir erişim deseni proj/src/foo.c ve ardından proj /obj/foo.o olacaktır. Bu iki erişim arasındaki mesafe ikidir, çünkü proj ortak atadır. FFS, politikalarında bu tür bir yerelliği yakalamaz ve bu nedenle bu tür erişimler arasında daha fazla arama gerçekleşir.

Karşılaştırma için, grafik ayrıca "Rastgele" bir izlemenin yerelliğini de gösterir. Rasgele izleme, varolan bir SEER izlemesi içindeki dosyaları rasgele sırada seçerek ve bu rasgele sıralanmış erişimler arasındaki mesafe metriği hesaplanarak oluşturulmuştur. Gördüğünüz gibi, beklendiği gibi rasgele izlemelerde daha az ad alanı yerelliği vardır. Bununla birlikte, sonunda her dosya ortak bir atayı (örneğin, kök) paylaştığından, bazı yerellikler vardır ve bu nedenle rasgele bir karşılaştırma noktası olarak yararlıdır.

41.6 Büyük Dosya Özel Durumu

FFS'de, dosya yerleştirme genel politikasının önemli bir istisnası vardır ve büyük dosyalar için ortaya çıkar. Farklı bir kural olmadan, büyük bir dosya ilk yerleştirildiği blok grubunu (ve belki de diğerlerini) tamamen doldurur. Bir blok grubunun bu şekilde doldurulması istenmeyen bir durumdur, çünkü sonraki "ilişkili" dosyaların bu blok grubuna yerleştirilmesini önler ve böylece dosya erişim yerelliğine zarar verebilir.

Bu nedenle, büyük dosyalar için FFS aşağıdakileri yapar. İlk blok grubuna bir miktar blok tahsis edildikten sonra (örneğin, 12 blok veya bir dizin düğümü bulunan doğrudan işaretçilerin sayısı), FFS, dosyanın bir sonraki "büyük" parçasını (örneğin, ilk dolaylı blok tarafından işaret edilenler) başka bir blok grubuna (belki de düşük kullanımı için seçilmiş) yerleştirir. Ardından, dosyanın bir sonraki öbeği başka bir farklı blok grubuna yerleştirilir ve bu şekilde devam eder.

Bu politikayı daha iyi anlamak için bazı diyagramlara bakalım. Büyük dosya özel durumu olmadan, tek bir büyük dosya tüm bloklarını diskin bir bölümüne yerleştirir. Grup başına 10 dizin düğümü ve 40 veri bloğu ile yapılandırılmış bir FFS'de 30 bloklu bir dosyanın (/a) küçük bir örneğini araştırıyoruz. İşte FFS'nin büyük dosya istisnası olmadan tasviri:

```
group inodes     data
  0   /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
  1   ---------- ---------- ---------- ---------- ----------
  2   ---------- ---------- ---------- ---------- ----------
```

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

Büyük dosya özel durumu (burada her bir öbekte beş bloğa ayarlanmıştır), FFS bunun yerine dosyayı gruplar arasında yayar ve herhangi bir grupta ortaya çıkan kullanım çok yüksek değildir:

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

Zeki okuyucu (yani siz), bir dosyanın bloklarını disk boyunca yaymanın, özellikle sıralı dosya erişiminin nispeten yaygın olduğu durumlarda (örneğin, bir kullanıcı veya uygulama sırayla 0 ile 29 arasındaki parçaları okur). Ve haklısın, ah zeki okuyucumuz! Ancak bu sorunu, yığın boyutunu dikkatlice seçerek çözebilirsiniz. Özellikle, parça boyutu yeterince büyükse, dosya sistemi zamanının çoğunu diskten veri aktararak harcayacak ve bloğun parçaları arasında arama yapmak için (göreceli olarak) çok az zaman harcayacaktır. Ödenen genel gider başına daha fazla iş yaparak genel gideri azaltma işlemine **amortisman (amortization)** denir ve bilgisayar sistemlerinde yaygın bir tekniktir.

Bir örnek verelim: Bir disk için ortalama konumlandırma süresinin (yani, arama ve döndürme) 10 ms olduğunu varsayalım. Amacınız zamanımızın yarısını parçalar arasında arama yaparak ve zamanımızın yarısını veri aktararak geçirmekse (ve böylece en yüksek disk performansının %50'sini elde etmekse), bu nedenle her 10 ms konumlandırma için 10 ms veri aktarımı yapmanız gerekir. Öyleyse soru şu hale geliyor: Transferde 10 ms harcamak için bir parçanın ne kadar büyük olması gerekiyor? Kolay, sadece eski arkadaşımız matematiği, özellikle diskler bölümünde bahsedilen boyutsal analizi kullanın [AD14a]:

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

Temel olarak, bu denklemin söylediği şey şudur: 40 MB / s ile veri aktarırsanız , zamanınızın yarısını aramak ve zamanınızın yarısını aktarmak için her aradığınızda yalnızca 409.6KB aktarmanız gerekir. Benzer şekilde, en yüksek bant genişliğinin %90'ını (yaklaşık 3.6MB olduğu ortaya çıktı) veya hatta en yüksek bant genişliğinin %99'unu (39.6MB!) Elde etmek için ihtiyaç duyacağınız yığının boyutunu hesaplayabilirsiniz. Gördüğünüz gibi, zirveye ne kadar yaklaşmak isterseniz, bu parçalar o kadar büyür (bu değerlerin bir grafiği için Şekil 41.2'ye bakın).

Ancak FFS, büyük dosyaları gruplar arasında yaymak için bu tür bir hesaplama kullanmadı. Bunun yerine, dizin düğümünün kendisinin yapısına dayanan basit bir yaklaşım benimsedi. İlk on iki doğrudan blok, dizin düğümü ile aynı gruba yerleştirildi; sonraki her dolaylı blok ve tüm işaret ettiği bloklar farklı bir gruba yerleştirildi. 4 KB blok boyutu ve 32 bit disk adresleri ile bu strateji, dosyanın her 1024 bloğunun (4 MB) ayrı gruplara yerleştirildiğini, tek özel durumun doğrudan işaretçiler tarafından işaret edildiği gibi dosyanın ilk 48 KB'si olduğunu ima eder. Disk sürücülerindeki eğilimin, disk üreticilerinin aynı yüzeye daha fazla bit sıkıştırmada iyi oldukları için aktarım hızının oldukça hızlı bir şekilde geliştiğini unutmayın, ancak sürücülerin aramalarla ilgili mekanik yönleri (disk kolu hızı ve dönme hızı) oldukça yavaş gelişir [P98]. Bunun anlamı, zamanla, mekanik maliyetlerin nispeten daha pahalı hale gelmesi ve bu nedenle, söz konusu maliyetleri amortismana tabi tutmak için daha fazla transfer yapmanız gerektiğidir.

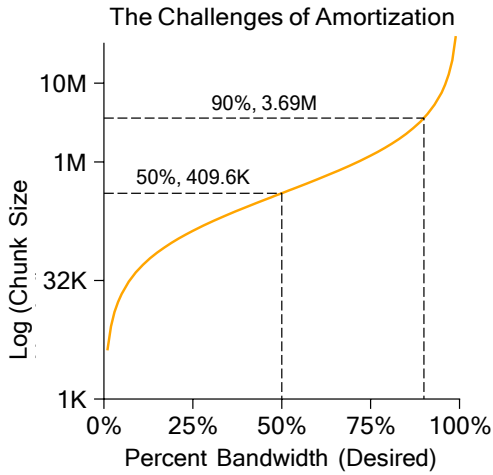


Figure 41.2: **Amortization: How Big Do Chunks Have To Be?**

41.7 FFS Hakkında Diğer Birkaç Şey

FFS birkaç yenilik daha getirdi. Özellikle, tasarımcılar küçük dosyaları barındırma konusunda son derece endişeliydi; Anlaşıldığı gibi, birçok dosya o zamanlar 2KB veya daha büyüktü ve 4KB blokları kullanmak, veri aktarımı için iyi olsa da alan verimliliği için o kadar iyi değildi. Bu **iç parçalanma (internal fragmentation)**, diskin yaklaşık yarısının tipik bir dosya sistemi için boşa harcanmasına neden olabilir.

FFS tasarımcılarının karşılaştığı çözüm basitti ve sorunu çözdü. Dosya sisteminin dosyalara ayırabileceği 512 baytlık küçük bloklar olan **alt blokları (sub-blocks)** tanıtmaya karar verdiler. Bu nedenle, küçük bir dosya oluşturduysanız (örneğin 1KB boyutunda), iki alt bloğu kaplar ve böylece 4KB'lık bir bloğun tamamını boşa harcamaz. Dosya büyüdükçe, dosya sistemi tam 4 KB veri edinene kadar dosyaya 512 baytlık bloklar ayırmaya devam edecektir. Bu noktada, FFS 4KB'lık bir blok bulacak, alt blokları içine kopyalayacak ve alt blokları gelecekte kullanmak üzere serbest bırakacaktır.

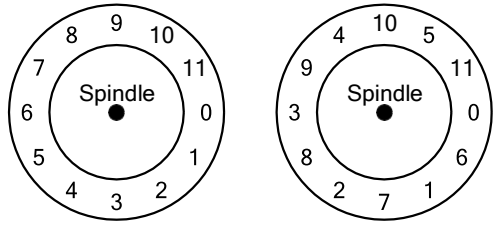


Figure 41.3: **FFS: Standard Versus Parameterized Placement**

Bu işlemin verimsiz olduğunu, dosya sistemi için çok fazla ekstra çalışma gerektirdiğini (özellikle kopyayı gerçekleştirmek için çok fazla ekstra G/Ç) gözlemleyebilirsiniz. Ve yine haklı olurdun! Bu nedenle, FFS genellikle libc kütüphanesini değiştirerek bu verimsiz davranıştan kaçındı; kütüphane yazmaları arabelleğe alır ve daha sonra bunları dosya sistemine 4 KB'lık parçalar halinde verir, böylece çoğu durumda alt blok uzmanlığını tamamen önler.

FFS'nin tanıttığı ikinci bir 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 bilgisayar CPU'sunun işlemlerini daha uygulamalı bir şekilde kontrol etmesini gerektiriyordu. FFS'de, Şekil 41.3'te solda olduğu gibi, diskin ardışık kesimlerine bir dosya yerleştirildiğinde bir sorun ortaya çıktı.

Özellikle, sorun sıralı okumalar sırasında ortaya çıktı. FFS önce 0'ı okumak için bir okuma yayınlar; Okuma tamamlandığında ve FFS blok 1'e bir okuma yayınladığında, çok geçti: blok 1 kafanın altında dönmüştü ve şimdi blok 1'e atanan okuma tam bir rotasyona neden olacaktı.

FFS, Şekil 41.3'te sağda görebileceğiniz gibi bu sorunu farklı bir düzen ile çözdü. FFS, diğer tüm blokları atlayarak (örnekte), disk kafasını geçmeden önce bir sonraki bloğu istemek için yeterli zamana sahiptir. Aslında, FFS, belirli bir disk için, ekstra rotasyonlardan kaçınmak için düzen yaparken kaç

blok atlaması gerektiğini anlayacak kadar akıllıydı; *FFS*, belirli performansı çözeceği için bu tekniğe **parametreleştirme (parameterization)** adı verildi. Diskin parametreleri ve tam kademeli düzen şemasına karar vermek için bunları kullanın.

Düşünüyor olabilirsiniz: Bu şema sonuçta o kadar da iyi değil. Aslında, bu tür bir düzende en yüksek bant genişliğinin yalnızca%50'sini elde edersiniz, çünkü her bloğu bir kez okumak için her parçayı iki kez dolaşmanız gerekir. Neyse ki, modern diskler çok daha akıllıdır: tüm parçayı dahili olarak okurlar ve dahili bir disk önbelleğinde arabelleğe alırlar (genellikle bu nedenle **parça arabelleği (track buffer)** olarak adlandırılırlar). Ardından, parçaya sonraki okumalarda, disk sadece istenen verileri önbelleğinden döndürür. Böylece dosya sistemleri artık bu inanılmaz derecede düşük seviyeli detaylar hakkında endişelenmek zorunda değil. Soyutlama ve daha üst düzey arayüzler, doğru tasarlandığında iyi bir şey olabilir.

Diğer bazı kullanılabilirlik iyileştirmeleri de eklendi. FFS, **uzun dosya adlarına (long file names)** izin veren ilk dosya sistemlerinden biriydi, böylece geleneksel sabit boyutlu yaklaşım (örneğin, 8 karakter) yerine dosya sisteminde daha anlamlı adlar sağladı. Ayrıca, **sembolik bağlantı (symbolic link)** adı verilen yeni bir kavram tanıtıldı. Önceki bir bölümde [AD14b] tartışıldığı gibi, sabit bağlantılar sınırlıdır, çünkü her ikisi de dizinlere işaret edememektedir (dosya sistemi hiyerarşisinde döngüler getirme korkusuyla) ve yalnızca aynı birimdeki dosyalara işaret edebilmektedirler (yani, dizin düğüm numarası hala anlamlı olmalıdır). Sembolik bağlantılar, kullanıcının bir sistemdeki başka bir dosya veya dizin için bir "diğer ad" oluşturmasına izin verir ve bu nedenle çok daha esnektir. FFS ayrıca dosyaları yeniden adlandırmak için atomik bir rename () işlemi başlattı. Temel teknolojinin ötesindeki kullanılabilirlik iyileştirmeleri, FFS'ye daha güçlü bir kullanıcı tabanı kazandırdı.

İPUCU: SİSTEMİ KULLANILABİLİR HALE GETİRİN

Muhtemelen FFS'den alınan en temel ders, yalnızca kavramsal olarak iyi bir fikir olan diske duyarlı düzeni tanıtmakla kalmayıp, aynı zamanda sistemi daha kullanışlı hale getiren bir dizi özellik eklediğidir. Uzun dosya adları, sembolik bağlantılar ve atomik olarak çalışan bir yeniden adlandırma işlemi, bir sistemin kullanışlılığını geliştirdi; hakkında bir araştırma makalesi yazmak zor olsa da ("The Symbolic Link: Hard Link's Long Lost Cousin" hakkında 14 sayfalık bir kitap okumaya çalıştığınızı hayal edin), bu tür küçük özellikler FFS'yi daha kullanışlı hale getirdi ve böylece muhtemelen kabul görme şansını artırdı. Bir sistemi kullanılabilir hale getirmek, genellikle derin teknik yenilikleri kadar veya daha önemlidir.

41.8 Özet

FFS'nin tanıtılması, dosya yönetimi sorununun bir işletim sistemindeki en ilginç sorunlardan biri olduğunu açıkça ortaya koyduğu ve cihazların en önemlisi olan sabit diskle nasıl başa çıkmaya başlayabileceğini gösterdiği için dosya sistemi tarihinde bir dönüm noktasıydı. O zamandan beri, yüzlerce yeni dosya sistemi gelişti, ancak bugün hala birçok dosya sistemi FFS'den ipuçları alıyor (örneğin, Linux ext2 ve ext3 bariz entelektüel torunlardır). Elbette tüm modern sistemler FFS'nin ana dersini açıklar: diske bir disk gibi davranın.

Kaynaklar

[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 filesystem 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 bölümde, FFS tabanlı dosya ve dizin ayırmanın nasıl çalıştığını daha iyi anlamak için kullanabileceğiniz basit bir FFS simülatörü olan ffs.py tanıtılmaktadır. Simülatörün nasıl çalıştırılacağı hakkında ayrıntılar için README'ye bakın.

Sorular

1. in.largefile dosyasını inceleyin ve simülatörü -f in.largefile ve -L 4 işaretleriyle çalıştırın. İkincisi, büyük dosya istisnasını 4 bloğa ayarlar. Ortaya çıkan tahsisat nasıl görünecek? Kontrol etmek için -c ile çalıştırın.

İn.largefile dosyası simülatöre 40 blokluk /a dosyasını oluşturmasını söylüyor ve büyük dosya istisnasını 4 bloğa ayarladığımız için simülatör 1 /a dizin düğümü(inode) oluşturur ve veri kısmını 4erli gruplar halinde sıra ile 10 gruba yerleştirir.

Grup	Dizin Düğüm(Inode)	Veri(Data)
0	/a	/aaaaa
1	-----	aaaa
2	-----	aaaa
3	-----	aaaa
4	-----	aaaa
5	-----	aaaa
6	-----	aaaa
7	-----	aaaa
8	-----	aaaa
9	-----	aaaa

Simülatörü -c komutuyla çalıştırdığımızda bize verdiği çıktı ise şu şekildedir.

```
num_groups:       10
inodes_per_group: 10
blocks_per_group: 30

free data blocks: 259 (of 300)
free inodes:      98 (of 100)

spread inodes?    False
spread data?      False
contig alloc:     1

      0000000000000000000 1111111111 2222222222
      0123456789012345678 9012345678 9012345678

group inodes    data
    0 /a------- /aaaa----- ---------- ----------
    1 ---------- aaaa------ ---------- ----------
    2 ---------- aaaa------ ---------- ----------
    3 ---------- aaaa------ ---------- ----------
    4 ---------- aaaa------ ---------- ----------
    5 ---------- aaaa------ ---------- ----------
    6 ---------- aaaa------ ---------- ----------
    7 ---------- aaaa------ ---------- ----------
    8 ---------- aaaa------ ---------- ----------
    9 ---------- aaaa------ ---------- ----------
```

2. Şimdi -L 30 ile çalıştırın. Ne görmeyi bekliyorsunuz? Bir kez daha, haklı olup olmadığınızı görmek için -c komutunu çalıştırın. /a dosyasına tam olarak hangi

blokların ayrıldığını görmek için -S komutunu da kullanabilirsiniz.

Simülatörü -L 4 yerine -L 30 ile çalıştırırsak veriyi 30 blokluk parçalara bölmeye çalışacağı için ve inode da ilk grupta saklanacağı için ilk gruba 29 blok ikinci gruba kalan 11 blok olacak şekilde ayırır.

```
Grup            Dizin Düğüm(Inode)          Veri(Data)
0               /a                          /aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
1               -----                       aaaaaaaaaaa
```

Simülatörü -c komutuyla çalıştırdığımızda bize verdiği çıktı ise şu şekildedir.

```
group inodes    data
    0 /a-------- /aaaaaaaaaa aaaaaaaaaa aaaaaaaaaa
    1 ---------- aaaaaaaaaa a--------- ----------
    2 ---------- ---------- ---------- ----------
    3 ---------- ---------- ---------- ----------
    4 ---------- ---------- ---------- ----------
    5 ---------- ---------- ---------- ----------
    6 ---------- ---------- ---------- ----------
    7 ---------- ---------- ---------- ----------
    8 ---------- ---------- ---------- ----------
    9 ---------- ---------- ---------- ----------
```

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 bir şeydir. /a dosya aralığını hesaplayın. Ne olduğunu görmek için ffs.py -f in.largefile -L 4 -T -c komutunu çalıştırın. Aynı şeyi -L 100 ile yapın. Büyük dosya özel durum parametresi düşük değerlerden yüksek değerlere değiştikçe dosya genişliğinde ne gibi bir fark bekliyorsunuz?

Elimizdeki 40 blokluk veriyi L 4 ile 4 blokluk parçalar halinde diske yerleştirirsek ffs blokları diskimizdeki 10 gruba da yayacaktır bu yüzden filespan diskin toplam boyutuna yakın bir değer olacaktır.

```
num_groups:        10
inodes_per_group: 10
blocks_per_group: 30

free data blocks: 259 (of 300)
free inodes:      98 (of 100)

spread inodes?    False
spread data?      False
contig alloc:     1

     0000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

group inodes    data
    0 /a--------- /aaaa----- ---------- ----------
    1 ---------- aaaa------ ---------- ----------
    2 ---------- aaaa------ ---------- ----------
    3 ---------- aaaa------ ---------- ----------
    4 ---------- aaaa------ ---------- ----------
    5 ---------- aaaa------ ---------- ----------
    6 ---------- aaaa------ ---------- ----------
    7 ---------- aaaa------ ---------- ----------
    8 ---------- aaaa------ ---------- ----------
    9 ---------- aaaa------ ---------- ----------

span: files
  file:         /a  filespan: 372
               avg  filespan: 372.00

span: directories
  dir:           /  dirspan: 373
               avg  dirspan: 373.00
```

Veriyi L 100 ile yerleştirirsek ffs dosyayı veri blok uzunluğu olan 30 blokluk parçalara ayıracaktır ve diskte 2 gruba yazacaktır bu yüzden ikinci durumda filespan çok daha küçük olacaktır.

```
num_groups:        10
inodes_per_group: 10
blocks_per_group: 30

free data blocks: 259 (of 300)
free inodes:      98 (of 100)

spread inodes?    False
spread data?      False
contig alloc:     1

     0000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

group inodes    data
    0 /a--------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa
    1 ---------- aaaaaaaaaa a--------- ----------
    2 ---------- ---------- ---------- ----------
    3 ---------- ---------- ---------- ----------
    4 ---------- ---------- ---------- ----------
    5 ---------- ---------- ---------- ----------
    6 ---------- ---------- ---------- ----------
    7 ---------- ---------- ---------- ----------
    8 ---------- ---------- ---------- ----------
    9 ---------- ---------- ---------- ----------

span: files
  file:         /a  filespan:  59
               avg  filespan:  59.00

span: directories
  dir:           /  dirspan:  60
               avg  dirspan:  60.00
```

4. Şimdi yeni bir giriş dosyasına bakalım, in.manyfiles. FFS politikasının bu dosyaları gruplar arasında nasıl düzenleyeceğini düşünüyorsunuz? (Hangi dosyaların ve dizinlerin oluşturulduğunu görmek için -v ile çalıştırabilir veya yalnızca cat in.manyfiles komutunu kullanabilirsiniz). Haklı olup olmadığınızı görmek için simülatörü -c ile çalıştırın.

Kök dizinde a, b, c, d, e, f, g, h ve i adlı 2 blokluk dosyaları ilk gruba yerleştirir. j klasörünü ve l, m, n, o, p, r ve C adlı dosyaları ikinci gruba yerleştirir. t klasörünü ve u, v, w, x, y, z, A ve B dosyalarını üçüncü gruba yerleştirir.

```
num_groups:       10
inodes_per_group: 10
blocks_per_group: 30

free data blocks: 245 (of 300)
free inodes:      72 (of 100)

spread inodes?    False
spread data?      False
contig alloc:     1

     0000000000000000000 1111111111 2222222222
     0123456789012345678 9123456789 0123456789

group inodes    data
    0 /abcdefghi /aabbccdde effgghhii- ----------
    1 jlmnopqrC- jlmnopqrCC C--------- ----------
    2 tuvwxyzAB- tuuuvvvwww xxxyyyzzzA AABBB-----
    3 ---------- ---------- ---------- ----------
    4 ---------- ---------- ---------- ----------
    5 ---------- ---------- ---------- ----------
    6 ---------- ---------- ---------- ----------
    7 ---------- ---------- ---------- ----------
    8 ---------- ---------- ---------- ----------
    9 ---------- ---------- ---------- ----------
```

5. FFS'yi değerlendirmek için kullanılan bir metriğe dirspan denir. Bu metrik belirli bir dizindeki dosyaların, özellikle dizindeki tüm dosyaların inode'ları ve veri blokları ile dizinin kendisinin inode ve veri bloğu arasındaki maksimum mesafeyi hesaplar. in.manyfiles ve -T bayrağı ile çalıştırın ve üç dizinin dirspan'ını hesaplayın. Kontrol etmek için -c ile çalıştırın. FFS, dirspan'i en aza indirmede ne kadar iyi bir iş çıkarıyor?

Diskteki gruplarımız 10 blok inode 30 blok veri olacak şekilde ayrılmıştır, ffs dosyaları ilgili bölümün ilk bloğundan başlayacak şekilde doldurduğu için inode ve veri arasında en az 10 blokluk bir dirspan olacaktır ancak ffs aynı klasördeki dosyaları bir gruba yerleştireceği için dosyalar arasındaki mesafe fazla olmayacaktır.

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'ı nasıl etkiler?

Inode tablosunun boyutunu 5 yaptığımız için ffs aynı klasördeki dosyaları farklı gruplara dağıtacaktır bu durum dosyanın inode bloku ile veri bloku arasındaki mesafeyi kısaltacağı için filespan ortalamasını düşürecektir ancak aynı klasördeki dosyalar farklı gruplara dağılacağı için aynı klasördeki dosyalar arası uzaklığı oldukça arttıracaktır.

```
group inodes    data
    0 /jy------- /jyyy----- ---------- ----------
    1 atp------- aatp------ ---------- ----------
    2 buz------- bbuuuzzz-- ---------- ----------
    3 clq------- cclq------ ---------- ----------
    4 dvA------- ddvvvAAA-- ---------- ----------
    5 emr------- eemr------ ---------- ----------
    6 fwB------- ffwwwBBB-- ---------- ----------
    7 gnC------- ggnCCC---- ---------- ----------
    8 hx-------- hhxxx----- ---------- ----------
    9 io-------- iio------- ---------- ----------

span: files
  file:         /a  filespan:  11
  file:         /b  filespan:  11
  file:         /c  filespan:  11
  file:         /d  filespan:  11
  file:         /e  filespan:  11
  file:         /f  filespan:  11
  file:         /g  filespan:  11
  file:         /h  filespan:  11
  file:         /i  filespan:  11
  file:       /t/u  filespan:  13
  file:       /j/l  filespan:  11
  file:       /t/v  filespan:  13
  file:       /j/m  filespan:  11
  file:       /t/w  filespan:  13
  file:       /j/n  filespan:  11
  file:       /t/x  filespan:  13
  file:       /j/o  filespan:  11
  file:       /t/y  filespan:  12
  file:       /j/p  filespan:  11
  file:       /t/z  filespan:  15
  file:       /j/q  filespan:  11
  file:       /t/A  filespan:  15
  file:       /j/r  filespan:  11
  file:       /t/B  filespan:  15
  file:       /j/C  filespan:  13
               avg  filespan:  11.92

span: directories
  dir:           /  dirspan: 371
  dir:          /j  dirspan: 371
  dir:          /t  dirspan: 332
               avg  dirspan: 358.00
```

7. FFS yeni bir dizinin inode'unu hangi gruba yerleştirmelidir? Varsayılan (simülatör) ilkesi, en fazla boş inode'lara sahip grubu arar. Farklı bir politika, en fazla inode'lara sahip bir dizi grup arar. Örneğin, -A 2 ile çalıştırırsanız, yeni bir dizin ayırırken, simülatör gruplar halinde gruplara bakar ve ayırma için en iyi çifti seçer.
   Bu stratejiyle tahsisin nasıl değiştiğini görmek için ./ffs.py
   -f in.manyfiles -I 5 -A 2 -c komutunu çalıştır. Dirspan'ı nasıl etkiler? Bu politika neden iyi olabilir?
   Bu komutu çalıştırdığımızda ffs yi dosyaları 10 gruba dağıtmak yerine 5 gruba dağıtmaya zorladığımız için aynı klasördeki dosyaların aynı gruba gelme olasılığı da artacaktır böylelikle dirspan de azalacaktır. Bu yöntem filespani çok yükseltmeden dirspani azalttığı için dosya veri okuma hızının önemli olduğu ancak aynı klasördeki dosyalar arasında gezinmenin de önemli olduğu durumlarda daha iyi performans verecektir.

```
group inodes    data
    0 /ejmyr---- /eejmyyyr- ---------- ----------
    1 ---------- ---------- ---------- ----------
    2 aftwpB---- aafftwwwpB BB-------- ----------
    3 ---------- ---------- ---------- ----------
    4 bgunzC---- bbgguuunzz zCCC------ ----------
    5 ---------- ---------- ---------- ----------
    6 chlxq----- cchhlxxxq- ---------- ----------
    7 ---------- ---------- ---------- ----------
    8 divoA----- ddiivvvoAA A--------- ----------
    9 ---------- ---------- ---------- ----------

span: files
  file:         /a  filespan:  11
  file:         /b  filespan:  11
  file:         /c  filespan:  11
  file:         /d  filespan:  11
  file:         /e  filespan:  11
  file:         /f  filespan:  12
  file:         /g  filespan:  12
  file:         /h  filespan:  12
  file:         /i  filespan:  12
  file:       /t/u  filespan:  14
  file:       /j/l  filespan:  12
  file:       /t/v  filespan:  14
  file:       /j/m  filespan:  11
  file:       /t/w  filespan:  14
  file:       /j/n  filespan:  14
  file:       /t/x  filespan:  14
  file:       /j/o  filespan:  14
  file:       /t/y  filespan:  13
  file:       /j/p  filespan:  14
  file:       /t/z  filespan:  16
  file:       /j/q  filespan:  14
  file:       /t/A  filespan:  16
  file:       /j/r  filespan:  13
  file:       /t/B  filespan:  16
  file:       /j/C  filespan:  18
               avg  filespan:  13.20

span: directories
  dir:           /  dirspan: 333
  dir:          /j  dirspan: 335
  dir:          /t  dirspan: 336
               avg  dirspan: 334.67
```

8. İnceleyeceğimiz son bir politika değişikliği, dosya parçalanmasıyla ilgilidir. ffs.py -f in.fragmented -v komutunu çalıştırın ve kalan dosyaların nasıl ayrılacağını tahmin edip edemeyeceğinize bakın. Yanıtınızı onaylamak için -c ile çalıştırın. /i dosyasının veri düzeni hakkında ilginç olan nedir? Neden sorunlu?

Bu komut önce 1 blokluk a, b, c, d, e, f, g ve h dosyalarını oluşturacak ardından a, c, e ve g dosyalarını silecek sonra 8 bloklu i dosyasını oluşturacaktır. İ dosyasının ilk dört veri bloğu a, c, e ve g dosyalarının verileri için açılan boşluklara yerleşecek kalan 4 blokluk veri ise h dosyasının veri bloğundan sonra eklenecektir. Bu durum i dosyasının parçalanmasına (fragmentation) ve veri okuma hızının düşmesine neden olacaktır.

9. Bitişik tahsis (-C) adını verdiğimiz yeni bir politika, her dosya bitişik olarak ayrılır. Özellikle, -C n ile, dosya sistemi bir bloğu ayırmadan önce n bitişik bloğun bir grup içinde serbest kalmasını sağlamaya çalışır. Farkı görmek için ./ffs.py -f in.fragmented -v -C 2 -c komutunu çalıştırın. -C'ye geçirilen parametre arttıkça düzen nasıl değişir? Son olarak, -C filespan ve dirspan'ı nasıl etkiler?

Bu komutu çalıştırdığımızda ffs i nin veri bloklarını en az 2 blokluk parçalara ayırmaya çalışacağı için a, c, e ve g nin silinmesiyle oluşan 1 blokluk boşluklara i nin veri

blokları eklenmeyecek ve h den sonraki kısma bir bütün olarak eklenecektir. -C nin arttırılması büyük dosyaların küçük parçalara ayrılıp okuma hızının düşmesini engelleyecektir ancak gruplarda boş kalan blokları arttıracağı için diskin verimsiz kullanımına neden olacaktır ayrıca -C ye girilen parametrenin büyümesi filespani azaltırken dirspani arttıracaktır.

```
op create /a [size:1] ->success
op create /b [size:1] ->success
op create /c [size:1] ->success
op create /d [size:1] ->success
op create /e [size:1] ->success
op create /f [size:1] ->success
op create /g [size:1] ->success
op create /h [size:1] ->success
op delete /a ->success
op delete /c ->success
op delete /e ->success
op delete /g ->success
op create /i [size:8] ->success

num_groups:       10
inodes_per_group: 10
blocks_per_group: 30

free data blocks: 287 (of 300)
free inodes:      94 (of 100)

spread inodes?    False
spread data?      False
contig alloc:     2

     0000000000000000000 1111111111 2222222222
     0123456789012345678 9012345678 9012345678

group inodes    data
    0 /ib-d-f-h- /-b-d-f-hi iiiiiii--- ----------
    1 ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ----------
    3 ---------- ---------- ---------- ----------
    4 ---------- ---------- ---------- ----------
    5 ---------- ---------- ---------- ----------
    6 ---------- ---------- ---------- ----------
    7 ---------- ---------- ---------- ----------
    8 ---------- ---------- ---------- ----------
    9 ---------- ---------- ---------- ----------
```