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: birimin ne kadar büyük olduğu, kaç tane düğüm olduğu, özgür bir blok listesinin başına bir işaretçi vb. Diskin düğüm bölgesi, sistem dosyası için tüm düğüm'leri 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: dosyalar ve dizin hiyerarşisi. Bu kullanımı kolay sistem, geçmişin beceriksiz, kayıt tabanlı depolama sistemlerinden ileriye doğru atılmış gerçek bir adımdı ve dizin hiyerarşisi, önceki sistemler tarafından sağlanan daha basit, tek seviyeli hiyerarşilere göre gerçek bir ilerlemeydi.

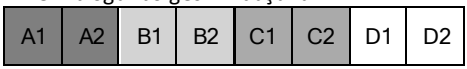
41.1 Problem : Kötü Performans

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

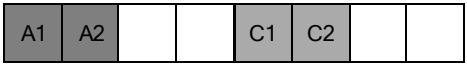
Ana sorun, eski UNIX dosya sisteminin diske rastgele erişimli bir bellekmiş gibi davranmasıydı; veriyi tutan ortamın disk olmasına aldırış etmeden her yere dağılan veriler, gerçek ve pahalı konumlandırma maliyetlerine sahipti. Örneğin, bir dosyanın veri blokları genellikle düğümünden çok uzaktaydı, bu nedenle kişi önce düğümü , ardından bir dosyanın veri bloklarını okuduğunda pahalı bir aramaya neden oluyordu (oldukça yaygın bir işlem).

Daha da kötüsü, boş alan dikkatli bir şekilde yönetilmediğinden, dosya sistemi oldukça parçalanmış hale gelirdi. Boş liste sonunda diske dağılmış bir grup bloğa işaret edecek ve dosyalar tahsis edildiğinde bir sonraki boş bloğu alacaklardı. Sonuç, mantıksal olarak bitişik bir dosyaya diskte ileri geri gidilerek erişilmesi ve böylece performansın önemli ölçüde düşmesiydi.

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 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 alamazsınız. Bunun yerine, önce E1 ve E2'yi okursunuz, sonra ararsınız, sonra E3 ve E4'ü okursunuz. Bu parçalanma sorunu, eski UNIX dosya sisteminde her zaman oluyordu ve performansa zarar veriyordu. Bir yan not: Bu sorun, disk birleştirme araçlarının tam olarak yardımcı olduğu şeydir; dosyaları bitişik olarak yerleştirmek ve bir veya birkaç bitişik bölge için boş alan oluşturmak için disk üzerindeki verileri yeniden düzenlerler, verileri hareket ettirirler ve ardından düğümleri ve değişiklikleri yansıtacak şekilde yeniden yazarlar.

Başka bir sorun: orijinal blok boyutu çok küçüktü (512 bayt). Bu nedenle, diskten veri aktarımı doğası gereği verimsizdi. Daha küçük bloklar iyiydi, çünkü dahili parçalanmayı (blok içindeki atık) en aza indirdiler, ancak transfer için kötüydü, çünkü her blok ona ulaşmak için bir konumlandırma ek yükü gerektirebilir. Böylece sorun:

CRUX:
PERFORMANSI ARTIRMAK İÇİN DİSK ÜZERİNDEKİ VERİLER NASIL DÜZENLENİR ?

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

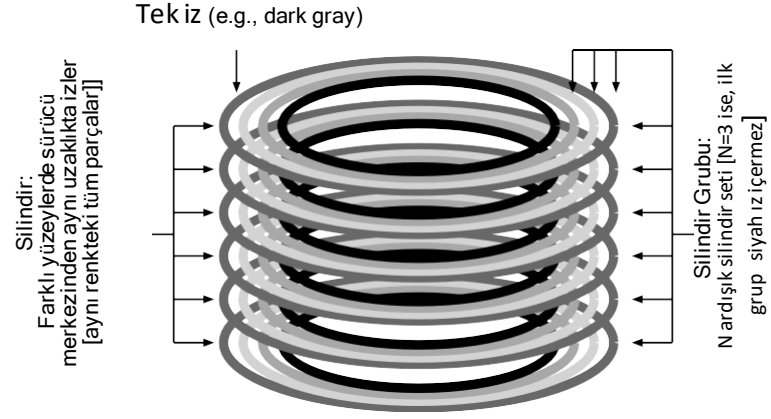
41.2 FFS: Çözüm Disk Farkındalığı

Berkeley'deki bir grup, zekice Hızlı Dosya Sistemi (FFS) adını verdikleri daha iyi, daha hızlı bir dosya sistemi oluşturmaya karar verdi. Buradaki fikir, dosya sistemi yapılarını ve ayırma ilkelerini "diskten haberdar" olacak şekilde tasarlamak ve böylece performansı artırmaktı, ki yaptıkları tam olarak buydu. Böylece FFS, dosya sistemi araştırmasında yeni bir çağ başlattı; dosya sistemiyle aynı arabirimi koruyarak (open(), read(), write(), close() ve diğer dosya sistemi çağrıları dahil aynı API'ler), ancak dahili uygulamayı değiştirerek, yazarlar yeni dosya sistemi yapımı için bugün devam eden çalışmanın yolunu açtılar. Neredeyse tüm modern dosya sistemleri, performans, güvenilirlik veya diğer nedenlerle dahili sistemlerini değiştirirken mevcut arayüze bağlı kalır (ve böylece uygulamalarla uyumluluğu korur).

41.3 Organizasyon Yapısı : Silindir Grubu

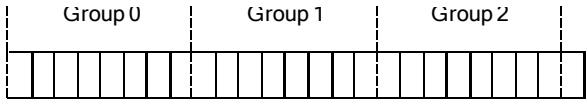
41.4

İlk adım, disk üzerindeki yapıları değiştirmekti. FFS, diski birkaç silindir grubuna böler. Tek bir silindir, bir sabit sürücünün farklı yüzeylerinde, sürücünün merkezinden aynı uzaklıkta bulunan bir dizi izdir; sözde geometrik şekle açık benzerliği nedeniyle silindir olarak adlandırılır. FFS, 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, her biri diskin adres alanının ardışık bir bölümü olan blok grupları halinde düzenler. Aşağıdaki resim, her 8 bloğun farklı bir blok grubu halinde düzenlendiği 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 kullanması , 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 vardır. İşte FFS'nin tek bir dosyada ne tuttuğunun bir örneği silindir grubu:

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

Her grup içinde, FFS'nin grubun düğümlerinin ve veri bloklarının tahsis edilip edilmediğini izlemesi gerekir. Grup başına bir düğüm bitmap (ib) ve veri bitmap (db), her gruptaki düğümler ve veri blokları için bu rolü yerine getirir. 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 ve veri bloğu bölgeleri önceki çok basit dosya sistemindeki (VSFS) gibidir. Her silindir grubunun çoğu, her zaman olduğu gibi, veri bloklarından oluşur.

FFS DOSYASI 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 (4 KB) olduğunu. Dosya yenidir ve bu nedenle yeni bir düğüme ihtiyaç duyar; bu nedenle, hem düğüm bitmap hem de yeni tahsis edilen düğüm diske yazılacaktır. Dosyanın içinde ayrıca veriler vardır ve bu nedenle onun da tahsis edilmesi gerekir; veri bit eşlemi ve bir veri bloğu böylece (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 bitmap ile). Hem dizinin yeni uzunluğunu yansıtmak hem de zaman alanlarını (son değiştirilme zamanı gibi) güncellemek için foo'nun düğümüde 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.

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

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

Dosyalar için FFS iki şey yapar. Birincisi, (genel durumda) bir dosyanın veri bloklarının kendi düğümü 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, FFS ilk üçü bire yakın yerleştirmeye çalışırdı başka bir (aynı grup) ve dördüncü uzakta (başka bir grupta).

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) FFS 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 (örn. /, /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 yaptığına dikkat edin: her dosyanın veri blokları, her dosyanın düğümüne 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).

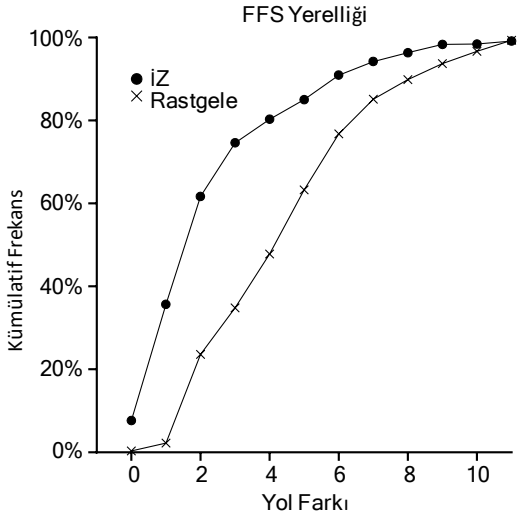
Buna karşılık, şimdi, hiçbir grubun düğüm tablosunun hızla dolmamasını sağlamaya çalışarak, düğümleri gruplara basitçe yayan bir düğüm ayırma politikasına bakalım. Nihai tahsis, bu nedenle şöyle görünebilir:

```
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 yakınında tutarken, bir dizindeki dosyalar keyfi olarak diske yayılır ve bu nedenle ada dayalı konum korunmaz. /a/c, /a/d ve /a/e dosyalarına erişim artık FFS yaklaşımına göre bir yerine üç gruba yayılıyor.

FFS ilke buluşsal yöntemleri, dosya sistemi trafiğine veya özellikle nüanslı herhangi bir şeye ilişkin kapsamlı araştırmalara dayanmaz; daha ziyade, eski moda sağduyuya 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.. Be-

1 Bazı insanlar, özellikle düzenli olarak atlarla çalışan insanlar, sağduyuya at duyusu derler. Ancak, "mekanize at", yani araba popülerlik kazandıkça bu deyimin kaybolabileceğine dair bir his var. Bundan sonra ne icat edecekler? Uçan bir makine mi?



Şekil 41.1: **SEER İzleri için FFS Konumu**

Böyle bir ad alanı tabanlı yerellik mevcut olduğundan, FFS genellikle performansı iyileştirerek ilgili dosyalar arasındaki aramaların güzel ve kısa olmasını sağlar.

41.6 Dosya Yerelliği Ölçme

Bu buluşsal yöntemlerin anlamlı olup olmadığını daha iyi anlamak için, dosya sistemi erişiminin bazı izlerini inceleyelim ve gerçekten de ad-uzayı konumu olup olmadığına bakalım. Nedense 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 sonraki sırada yeniden açılırsa (diğer dosyalar açılmadan önce), dizin ağacında 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 bir açılışı gelirse, iki dosya erişimi arasındaki mesafe, paylaştıkları için birdir. aynı dizin ama aynı dosya değil. Başka bir deyişle, mesafe ölçütümüz, iki dosyanın ortak atasını 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 izleri için (grafikte "İz" olarak işaretlenmiştir), dosya erişimlerinin yaklaşık %7'sinin daha önce açılan dosyaya yapıldığını görebilirsiniz.

ve dosya erişimlerinin yaklaşık %40'ının ya aynı dosyaya ya da aynı dizindeki bir dosyaya yapıldığını (yani, sıfır veya bir farkla). Bu nedenle, FFS yerellik varsayımı mantıklı görünmektedir (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 proj ortak atadır. FFS, ilkelerinde 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 sırayla 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, bir karşılaştırma noktası olarak rasgele yararlıdır.

41.7 Büyük Dosya Özel Durumu

FFS'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 grubunun bu şekilde doldurulması, sonraki "ilgili" dosyaların bu blok grubuna yerleştirilmesini engellediğinden ve dolayısıyla dosya erişim konumuna zarar verebileceğinden istenmez.

Böylece, büyük dosyalar için FFS 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 FFS'nin tasviri şöyledir grup düğüm data

```
0   /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
1   ---------- ---------- ---------- ---------- ----------
2   ---------- ---------- ---------- ---------- ----------
```

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

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

```
grup düğümleri   veri
  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.

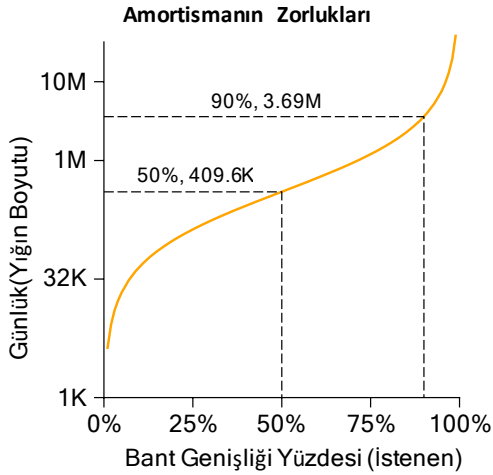
Spesifik olarak, parça boyutu yeterince büyükse, dosya sistemi zamanının çoğunu diskten veri aktarmakla ve (nispeten) küçük bir süre bloğun parçaları arasında arama yapmakla geçirecektir. Ödenen genel gider başına daha fazla iş yaparak bir genel gideri azaltma işlemine amortisman denir ve bilgisayar 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 harcamanız gerekirdi.Böylece soru aktarımı için 10 ms harcamanız gerekirdi.Böylece soru şöyle olur : Transferde 10 ms harcamak için bir parçanın ne kadar büyük olması gerekir ? Kolay , sadece eski dostumuz matematiği , özelliklede disklerle ilgili bölümde [AD14a] bahsedilen boyutsal analizi kullanın :

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

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 bkz. Şekil 41.2).

Ancak FFS, büyük dosyaları gruplara dağıtmak için bu tür hesaplama kullanmadı.Bunun yerine , basit bir yaklaşım benimsedi.



Şekil 41.2: **Amortisman: Parçaların Ne Kadar Büyük Olması Gerekir?**

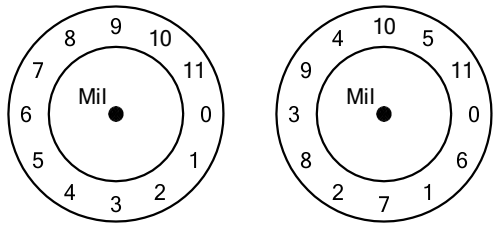
düğümün kendisinin yapısı. İlk on iki doğrudan blok, düğüm ile aynı gruba yerleştirildi; sonraki her dolaylı blok ve tüm işaret ettiği bloklar farklı bir gruba yerleştirildi. 4KB blok boyutu ve 32 bit disk adresleriyle bu strateji, dosyanın (4MB) her 1024 bloğunun ayrı gruplara yerleştirildiğini ima 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 arasındaki verilerin daha fazlasını aktarmanız gerekir .

41.8 FFS Hakkında Birkaç Diğer Şey

FFS 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, tipik bir dosya sistemi için diskin kabaca yarısının boşa harcanmasına neden olabilir.

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



Şekil 41.3: **FFS: Standart ve Parametreli Yerleştirme**

4 KB 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, FFS 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 kaçınır.

FFS'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, çok geçti: 1. blok başın altında dönmüştü ve şimdi 1. bloğa yapılan okuma tam bir dönüşe 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 sonraki bloğu istemek için yeterli zamana sahip olur. Aslında, FFS, fazladan dönüşlerden kaçınmak için belirli bir diskin mizanpajı yaparken kaç blok atlaması gerektiğini anlayacak kadar akıllıydı; FFS, diskin belirli performans parametrelerini anlayacağından ve bunları tam kademeli düzen şemasına karar vermek için kullanacağından, bu tekniğe parametrelendirme 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 olarak adlandırılır) ara belleğe alırlar. Ardından, izin sonraki okumalarında, disk

İPUCU: SİSTEMDE KULLANIŞLI OLUN

Muhtemelen FFS'den çıkarılacak en temel ders, yalnızca kavramsal olarak iyi olan diske duyarlı düzen fikrini ortaya koymakla kalmayıp, aynı zamanda sistemi daha kullanışlı hale getiren bir dizi özellik de eklemesidir. 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 FFS'yi daha kullanışlı hale getirdi ve bu nedenle muhtemelen benimsenme şansını artırdı. . Bir sistemi kullanılabilir hale getirmek, genellikle derin teknik yenilikleri kadar veya onlardan daha önemlidir.

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

Diğer bazı kullanılabilirlik iyileştirmeleri de eklendi. FFS, uzun dosya adlarına izin veren ilk dosya sistemlerinden biriydi, böylece dosya sisteminde geleneksel sabit boyutlu yaklaşım (ör. 8 karakter) yerine daha anlamlı adlar sağladı. Ayrıca, sembolik bağlantı 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, düğüm 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. FFS ayrıca dosyaları yeniden adlandırmak için atomik bir yeniden adlandırma() işlemi başlattı. Temel teknolojinin ötesindeki kullanılabilirlik iyileştirmeleri de muhtemelen FFS'ye daha güçlü bir kullanıcı tabanı kazandırdı.

41.9 Özet

FFS'nin tanıtılması, dosya sistemi tarihinde bir dönüm noktasıydı, 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 diskler. O zamandan beri yüzlerce yeni dosya sistemi geliştirildi, ancak bugün hala birçok dosya sistemi FFS'den ipuçları alıyor (örneğin, Linux ext2 ve ext3 bariz entelektüel torunlardır). Kesinlikle tüm modern sistemler, FFS'nin ana dersini açıklar: diske bir diskmiş gibi davranın.

Referanslar

[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.*

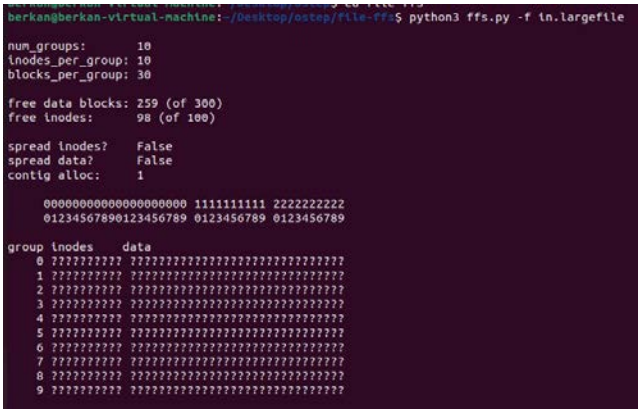
[P98] "Hardware Technology Trends and Database Opportunities" by David A. Patterson. Keynote Lecture at SIGMOD '98, June 1998. *A great and simple overview of disk technology trends and how they change over time.*

Homework (Simulation)

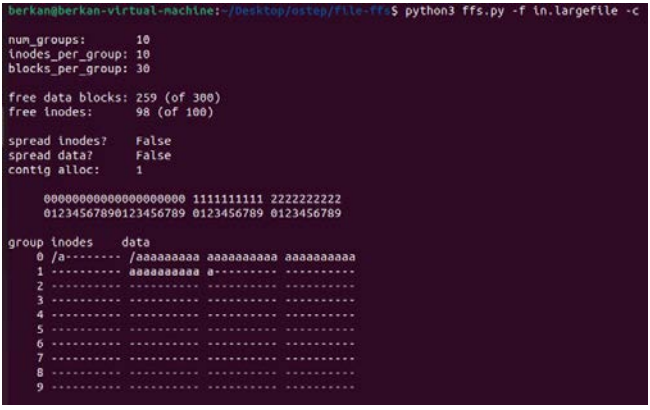
Bu bölümde, FFS tabanlı dosya ve dizin tahsisinin nasıl çalıştığını daha iyi anlamak için kullanabileceğiniz basit bir FFS simülatörü olan ffs.py tanıtılmaktadır. Simülatörün nasıl çalıştırılacağına ilişkin ayrıntılar için BENİOKU'ya bakın.

Sorular

1. in.largefile dosyasını inceleyin ve ardından -f in.largefile ve -L bayrağıyla simülatörü çalıştırın. 4. İkincisi, büyük dosya istisnasını 4 bloğa ayarlar. Ortaya çıkan tahsis neye benzeyecek? Kontrol etmek için -c ile çalıştırın



HER GRUBUN DÖRT 'A' VERİ BLOĞU VARDIR.



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.

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f  in.largefile -L 30

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 ?????????? ??????????????????????????????
    1 ?????????? ??????????????????????????????
    2 ?????????? ??????????????????????????????
    3 ?????????? ??????????????????????????????
    4 ?????????? ??????????????????????????????
    5 ?????????? ??????????????????????????????
    6 ?????????? ??????????????????????????????
    7 ?????????? ??????????????????????????????
    8 ?????????? ??????????????????????????????
    9 ?????????? ??????????????????????????????
```

Bu çıktının anlamı, büyük dosya istisnası seçeneğini eklememekle aynı durumu gösterir.

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f  in.largefile -S

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 ?????????? ??????????????????????????????
    1 ?????????? ??????????????????????????????
    2 ?????????? ??????????????????????????????
    3 ?????????? ??????????????????????????????
    4 ?????????? ??????????????????????????????
    5 ?????????? ??????????????????????????????
    6 ?????????? ??????????????????????????????
    7 ?????????? ??????????????????????????????
    8 ?????????? ??????????????????????????????
    9 ?????????? ??????????????????????????????

symbol  inode#  filename     filetype   block_addresses
/           0   /            directory  0
a           1   /a           regular    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 32 33 34 35 36 37 38 39 40
```

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?

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f in.largefile -L 4 -T -
c

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
```

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f in.largefile -L 100 -T
 -c

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 aaaaaaaaa aaaaaaaaa
    1 ---------- aaaaaaaaa 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 girdi dosyasına bakalım, in.manyfiles. FFS 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.

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ cat in.manyfiles
file /a 2
file /b 2
file /c 2
file /d 2
file /e 2
file /f 2
file /g 2
file /h 2
file /i 2

dir /j
dir /t

file /t/u 3
file /j/l 1
file /t/v 3
file /j/m 1
file /t/w 3
file /j/n 1
file /t/x 3
file /j/o 1
file /t/y 3
file /j/p 1
file /t/z 3
file /j/q 1
file /t/A 3
file /j/r 1
file /t/B 3
file /j/C 3
```

Aynı klasördeki dosyaların düğümleri ve data blokları aynı grupta olmalıdır.

5. FFS'yi değerlendirmek için bir ölçüme dirspan denir. Bu metrik yayılmayı hesaplar belirli bir dizindeki dosyaların sayısı, özellikle dizindeki tüm dosyaların düğümleri ve veri blokları ile dizinin kendisinin inode'u ve data bloğu arasındaki maksimum mesafe. in.manyfiles ve -T bayrağıyla 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?

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f  in.manyfiles -T

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

     00000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

group inodes    data
    0 ?????????? ??????????????????????????????
    1 ?????????? ??????????????????????????????
    2 ?????????? ??????????????????????????????
    3 ?????????? ??????????????????????????????
    4 ?????????? ??????????????????????????????
    5 ?????????? ??????????????????????????????
    6 ?????????? ??????????????????????????????
    7 ?????????? ??????????????????????????????
    8 ?????????? ??????????????????????????????
    9 ?????????? ??????????????????????????????

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

span: directories
  dir:          /  dirspan: ?
  dir:         /j  dirspan: ?
  dir:         /t  dirspan: ?
              avg  dirspan: ?
```

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f  in.manyfiles -T -c

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

     00000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

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

span: files
  file:         /a  filespan:  11
  file:         /b  filespan:  12
  file:         /c  filespan:  13
  file:         /d  filespan:  14
  file:         /e  filespan:  15
  file:         /f  filespan:  16
  file:         /g  filespan:  17
  file:         /h  filespan:  18
  file:         /i  filespan:  19
  file:       /t/u  filespan:  12
  file:       /j/l  filespan:  10
  file:       /t/v  filespan:  14
  file:       /j/m  filespan:  10
  file:       /t/w  filespan:  16
  file:       /j/n  filespan:  10
  file:       /t/x  filespan:  18
  file:       /j/o  filespan:  10
  file:       /t/y  filespan:  20
  file:       /j/p  filespan:  10
  file:       /t/z  filespan:  22
  file:       /j/q  filespan:  10
  file:       /t/A  filespan:  24
  file:       /j/r  filespan:  10
  file:       /t/B  filespan:  26
  file:       /j/C  filespan:  12
               avg  filespan:  14.76

span: directories
  dir:          /  dirspan:  28
  dir:         /j  dirspan:  20
  dir:         /t  dirspan:  34
              avg  dirspan:  27.33
```

6. Şimdi grup başına düğüm 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?

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f in.manyfiles -I 5 -c

num_groups:       10
inodes_per_group: 10
blocks_per_group: 30

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

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

     0000000000000000000 1111111111 2222222222
     0123456789012345678 9123456789 0123456789

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------- ---------- ----------
```

Aynı klasördeki dosyalar farklı gruplara dağılmıştır. Dirspanı artıracaktır.

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f in.manyfiles -I 5

num_groups:       10
inodes_per_group: 10
blocks_per_group: 30

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

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

     0000000000000000000 1111111111 2222222222
     0123456789012345678 9123456789 0123456789

group inodes    data
    0 ?????????? ??????????????????????????????
    1 ?????????? ??????????????????????????????
    2 ?????????? ??????????????????????????????
    3 ?????????? ??????????????????????????????
    4 ?????????? ??????????????????????????????
    5 ?????????? ??????????????????????????????
    6 ?????????? ??????????????????????????????
    7 ?????????? ??????????????????????????????
    8 ?????????? ??????????????????????????????
    9 ?????????? ??????????????????????????????

berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$
```

7. FFS, yeni bir dizinin düğümünü 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?

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ./ffs.py -f in.manyfiles -I 5 -A
 2 -c

num_groups:       10
inodes_per_group: 10
blocks_per_group: 30

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

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

     0000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

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 ---------- ---------- ---------- ----------
```

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ffs.py -f in.manyfiles -A 2

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
     01234567890123456789 0123456789 0123456789

group inodes    data
    0 ?????????? ??????????????????????????????
    1 ?????????? ??????????????????????????????
    2 ?????????? ??????????????????????????????
    3 ?????????? ??????????????????????????????
    4 ?????????? ??????????????????????????????
    5 ?????????? ??????????????????????????????
    6 ?????????? ??????????????????????????????
    7 ?????????? ??????????????????????????????
    8 ?????????? ??????????????????????????????
    9 ?????????? ??????????????????????????????
```

Farklı klasörleri uzak tutar. Dirspanı azaltacaktır. Grup başına düğüm sayısı küçük olduğunda, aynı klasördeki dosyaları daha yakın hale getirir.

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?

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ./ffs.py -f in.fragmented -v
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:     1

     0000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

group inodes    data
    0 ?????????? ??????????????????????????????
    1 ?????????? ??????????????????????????????
    2 ?????????? ??????????????????????????????
    3 ?????????? ??????????????????????????????
    4 ?????????? ??????????????????????????????
    5 ?????????? ??????????????????????????????
    6 ?????????? ??????????????????????????????
    7 ?????????? ??????????????????????????????
    8 ?????????? ??????????????????????????????
    9 ?????????? ??????????????????????????????
```

Sürekli bir blok yerine bölünür. Dosyayı okumak veya yazmak yavaş olacaktır.

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ./ffs.py -f in.fragmented -c

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:     1

     0000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

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

9. Bitişik tahsis (-C) olarak adlandırdığımız yeni bir politika, her dosya bitişik olarak tahsis edilir. Spesifik olarak, -Cn 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?

```
berkan@berkan-virtual-machine:~/Desktop/ostep/file-ffs$ python3 ./ffs.py -f in.fragmented -v -C
2 -c
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 ---------- ---------- ---------- ----------
```

Filespan ve dirspan'i artıracaktır.