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 bu system gerçekten basitti. Temel olarak, veri yapıları diskte şöyle görünüyordu:

Süper blok (S), tüm dosya sistemi hakkında birimin ne kadar büyük olduğu, kaç düğüm olduğu, boş blok listesinin başındaki bir işaretçi bilgisi ve benzeri bilgileri içerirdi. Diskin düğüm bölgesi, dosya sistemi için tüm düğümleri içeriyordu. Son olarak, diskin çoğu veri blokları tarafından alınırdı.

Eski dosya sisteminin iyi yanı, 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.

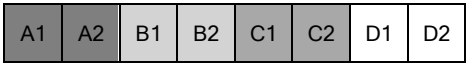
41.1 Sorun: Düşük Performans

Sorun: performansın korkunç olmasıydı. 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!

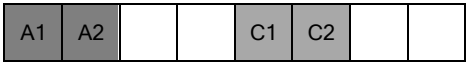
Asıl sorun, eski UNIX dosya sisteminin diske rastgele erişimli bir bellekmiş gibi davranmasıydı; veriler, verileri tutan ortamın bir disk olduğu ve dolayısıyla ciddi ve pahalı konumlandırma maliyetlerine sahip olduğu gerçeğine bakılmaksızın her yere dağılırdı. Örneğin, bir dosyanın veri blokları genellikle düğümden çok uzaktı, bu nedenle düğümün ve ardından bir dosyanın ver bloklarının okunması malyetli bir aramaya neden olurdu. (Bu oldukça yaygın bir işlemdi).

Daha da kötüsü, boş alan dikkatli bir şekilde yönetilmediğinden, dosya sistemi oldukça **parçalanmış (fragmented**) hale gelirdi. Boş liste, diske yayılmış bir grup bloğa işaret ederdi ve dosyalar tahsis edildiğinde, bir sonraki boş bloğu alırlardı. Sonuç olarak, mantıksal olarak bitişik bir dosyaya disk boyunca ileri geri giderek erişilecek ve böylece performans önemli ölçüde düşecekti.

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 dörtlü güzel bir bitişik parça yerine iki bloktan oluşan iki parçaya bölünmüştür. Diyelim ki şimdi dört blok boyutunda bir E dosyası tahsis etmek istiyorsunuz:

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 (defragmentation)** araçlarının tam olarak yardımcı olduğu şeydir; dosyaları bitişik olarak yerleştirmek ve bir veya birkaç bitişik bölge için boş alan oluşturmak için disk verilerini 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 doğal olarak verimsizdi. Daha küçük bloklar iyiydi çünkü **iç parçalanmayı (internal fragmentation)** (blok içindeki israfı) en aza indirdiler, ancak her bloğun ona ulaşmak için bir konumlandırma yükü gerektirebileceğinden transfer için kötüydü. Böylece, sorun:

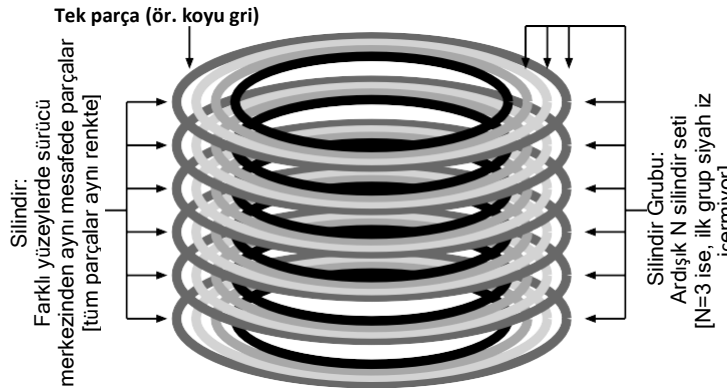
ÖNEMLİ NOKTA:
PERFORMANSI ARTIRMAK İÇİN DİSK ÜZERİNDEKİ VERİLER NASIL DÜZENLENİR
Dosya sistemi veri yapılarını performansı artırmak için nasıl düzenleyebiliriz? Bu veri yapılarının yanı sıra ne tür tahsisat politikalarına ihtiyacımız var? Dosya sistemini nasıl "diske duyarlı" hale getirebiliriz?

41.2 FFS: Disk Farkındalığı Çözümdür

Berkeley'deki bir grup, zekice **Hızlı Dosya Sistemi (Fast File System) (FFS)** adını verdikleri daha iyi, daha hızlı bir dosya sistemi oluşturmaya karar verdi. Buradaki fikir, dosya sistemi yapılarını ve ayırma ilkelerini "diske duyarlı" olacak şekilde tasarlamak ve böylece performansı artırmaktı, ki yaptıkları tam olarak buydu. Böylece FFS, dosya sistemi araştırmasında yeni bir çağ başlattı; yazılımcılar, dosya sistemiyle aynı arabirimi koruyarak (open(), read(), write(), close() ve diğer dosya sistemi çağrıları dahil olmak üzere aynı API'ler), ancak dahili uygulamayı değiştirerek, bugün devam eden yeni dosya sistemi yapımının yolunu açtı. Neredeyse tüm modern dosya sistemleri, performans, güvenilirlik veya başka nedenlerle iç kısımlarını değiştirirken mevcut arayüze bağlı kalır (ve böylece uygulamalarla uyumluluğu korur).

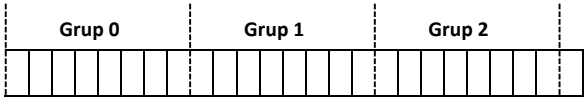
41.3 Organizasyon Yapısı: Silindir Grubu

İlk adım, disk üzerindeki yapıları değiştirmekti. FFS, diski bir dizi **silindir grubuna (cylinder groups)** böler. Tek bir **silindir (cylinder)**, bir sabit sürücünün farklı yüzeylerinde, sürücünün merkezinden aynı uzaklıkta bulunan bir dizi izdir; sözde geometrik şekle açık benzerliği nedeniyle silindir olarak adlandırılır. FFS, N ardışık silindiri bir grupta toplar ve bu nedenle tüm disk, silindir gruplarının bir koleksiyonu olarak görülebilir. Aşağıda, altı plakalı bir sürücünün en dıştaki dört parçası 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 aktarmadığını unutmayın; daha önce tartışıldığı gibi [AD14a], diskler blokların mantıksal bir adres alanını dışa aktarır ve geometrilerinin ayrıntılarını istemcilerden gizler. Böylece modern dosya

sistemler (Linux ext2, ext3 ve ext4 gibi) bunun yerine sürücüyü, her biri diskin adres alanının ardışık bir bölümü olan **blok grupları (block groups)** halinde düzenler. Aşağıdaki resim, her 8 bloğun farklı bir blok grubunda düzenlendiği bir örneği göstermektedir (gerçek grupların daha birçok 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, örneğin inode'lar için alan, veri blokları ve bunların her birinin tahsis edilip edilmediğini izlemek için bazı yapıları içerir. İşte FFS'nin tek bir silindir grubu içinde tuttuğu şeyin bir tasviri:

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 monte etmek için süper blok gereklidir; birden çok kopya tutarak, bir kopya bozulursa, çalışan bir çoğaltma kullanarak dosya sistemini bağlayabilir ve dosya sistemine erişebilirsiniz.

Her grup içinde, FFS'nin grubun düğümlerinin ve veri bloklarının tahsis edilip edilmediğini izlemesi gerekir. Grup başına bir **düğüm bitmap** (**inode bitmap)** (ib) ve **veri bitmap (data 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 yığını bulmak ve bir dosyaya tahsis etmek kolaydır, eski dosya sistemindeki özgür listenin parçalanma sorunlarından bazılarını önleyebilir.

Son olarak, **düğüm** (**inode)** ve **veri bloğu (data block)** bölgeleri önceki çok basit dosya sistemindeki (very-simple file system (VSFS)) gibidir. Her silindir grubunun çoğu, her zaman olduğu gibi, veri bloklarından oluşur.

KENAR: FFS DOSYASI OLUŞTURMA

Örnek olarak, bir dosya oluşturulduğunda hangi veri yapılarının güncellenmesi gerektiğini düşünün; bu örnek için kullanıcının yeni bir `/foo/bar.txt` dosyası oluşturduğunu ve dosyanın bir blok uzunluğunda (4KB) olduğunu varsayalım. Dosya yenidir ve bu nedenle yeni bir düğüme ihtiyaç duyar; bu nedenle, hem düğüm bitmap'e hem de yeni ayrılan inode 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 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, `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 düğümü da güncellenmelidir. Genel olarak, sadece yeni bir dosya oluşturmak için çok iş var! 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 Tahsis Edilir

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

Bu nedenle, düşünceye 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 yönteminden yararlanır.

Birincisi, dizinlerin yerleştirilmesidir. FFS basit bir yaklaşım kullanır: az sayıda ayrılmış dizin (gruplar arasında dizinleri dengelemek için) ve çok sayıda boş düğüm (daha sonra bir grup dosya ayırabilmek için) ile silindir grubunu bulmak ve dizin verilerini ve düğümü bu gruba koymak. Tabii ki, burada başka sezgisel yöntemler de 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 düğüm ve veri arasında (eski dosya sisteminde olduğu gibi) uzun aramaları önler. İkincisi, aynı dizinde bulunan tüm dosyaları, içinde bulundukları dizinin silindir grubuna yerleştirir. Bu nedenle, bir kullanıcı /a/b, /a/c, /a/d ve b/f olmak üzere dört dosya oluşturursa, FFS ilk üçünü birbirine yakın (aynı grupta) ve dördüncüsünü uzak (başka bir grupta) yerleştirmeye çalışır.

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

gerçekçi olmayan küçük sayılar) ve üç dizinin (kök dizin `/`, `/a`, ve `/b`) ve dört dosyanın (`/a/c`, `/a/d`, `/a/e`, `/b/f`) FFS politikaları uyarınca içlerinde yerleştirildiğini varsayalım. Normal dosyaların her birinin iki blok boyutunda olduğunu ve dizinlerin yalnızca tek bir veri bloğuna sahip olduğunu farz edelim. Bu şekil için, her dosya veya dizin için bariz sembolleri kullanırız (yani, kök dizin için / , /a için, /b / f için f vb.).

```
grup düğümleri   veri
    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 düğümünün yakınındadır ve aynı dizindeki dosyalar birbirine yakındır (yani, /a/c, /a/d ve /a/e'nin tümü Grup 1'dedir ve /b/f dosyası Grup 2'de birbirine yakındır).

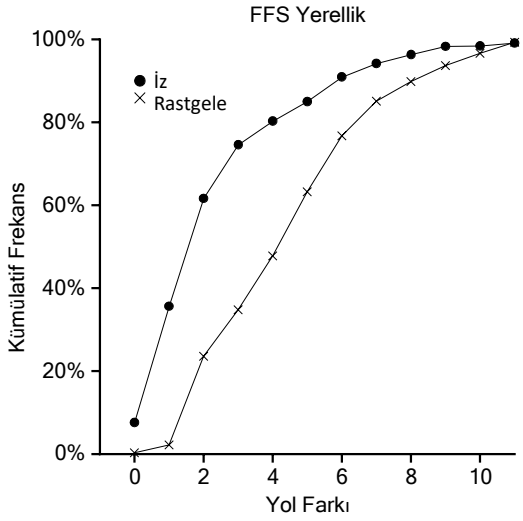
Buna karşılık, şimdi, hiçbir grubun düğüm tablosunun hızla dolmamasını sağlamaya çalışarak, düğümleri gruplar arasında basitçe yayan bir düğüm tahsis etme (ayırma) politikasına bakalım. Son tahsisat bu nedenle şöyle bir şeye benzeyebilir:

```
grup düğümleri   veri
    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'un yakınında tutarken, bir dizin içindeki dosyalar diskin etrafına keyfi olarak yayılır ve bu nedenle ada dayalı yerellik korunmaz. FFS yaklaşımına göre /a/c, /a/d ve /a/e dosyalarına erişim artık bir yerine üç grubu kapsıyor.

FFS politika buluşsal yöntemleri, dosya sistemi trafiğine ilişkin kapsamlı araştırmalara veya özellikle incelikli herhangi bir şeye dayanmaz; daha ziyade, eski moda **sağduyuya (common sense)** dayalıdırlar (sonuçta sağduyunun (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, sağduyuyu **at duygusu (horse sense)** olarak adlandırır. 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 Yerelliği**

Bu tür ad alanı tabanlı yerellik mevcut olduğundan, FFS genellikle ilgili dosyalar arasındaki aramaların güzel ve kısa olmasını sağlayarak uygunluğu artırır.

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 açılırsa ve ardından izlemede bir sonraki dosya açılırsa (başka herhangi bir dosya açılmadan önce), dizin ağacında bu iki açılış arasındaki mesafe sıfırdır (çünkü bunlar aynı dosyadır). Dizin (yani, dir/f) içindeki bir f dosyası açılırsa ve ardından aynı dizinde (yani, dir/g) g dosyası açılırsa, iki dosya erişimi arasındaki mesafe, paylaştıkları için birdir, aynı dizin ama aynı dosya değillerdir. Başka bir deyişle, mesafe ölçütümüz, iki dosyanın *ortak atasını* bulmak için dizin ağacında ne kadar yukarı gitmeniz gerektiğini ölçer; ağaca ne kadar yakınsa, ölçüm 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 farka ait olan dosya açılışlarının kümülatif yüzdesini gösterir. Özellikle, SEER izlemeleri için (grafikte "İz" 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).

İlginç bir şekilde, 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 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.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 izin yerelliğini de gösterir. Rastgele iz, mevcut bir SEER izinden dosyaların rasgele sırada seçilmesi ve bu rasgele sıralanmış erişimler arasındaki mesafe metriğinin hesaplanmasıyla oluşturulmuştur. Gördüğünüz gibi, beklendiği gibi rasgele izlemelerde daha az ad alanı yerelliği vardır. Bununla birlikte, sonunda her dosya ortak bir atayı (örneğin, kök) paylaştığından, bir miktar yerellik vardır ve bu nedenle rasgele bir karşılaştırma noktası olarak kullanışlıdır.

41.6 Büyük Dosya İstinası

FFS'de, doya yerleştirme genel politikasının büyük dosyalar için ortaya çıkan önemli bir istisnası vardır. Farklı bir kural olmadan, büyük bir dosya ilk yerleştirildiği blok grubunu (ve belki de diğerlerini) tamamen doldurur. Bir blok grubunun bu şekilde doldurulması, sonraki "ilgili" dosyaların bu blok grubuna yerleştirilmesini engellediğinden ve dolayısıyla dosya erişim konumuna zarar verebileceğinden istenmeyen bir durumdur.

Bu nedenle, FFS büyük dosyalar için aşağıdakileri yapar. İlk blok grubuna bir miktar blok tahsis edildikten sonra (örneğin, 12 blok veya bir düğüm 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 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 düğümleri     veri
  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, 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
```

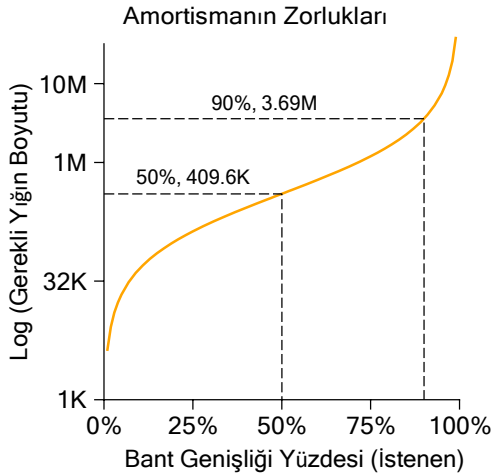
Zeki okuyucu (yani siz), bir dosyanın bloklarını diske yaymanın, özellikle nispeten yaygın sıralı dosya erişimi durumunda (örneğin, bir kullanıcı veya uygulama 0'dan 29'a kadar olan parçaları sırayla okuduğunda) performansa zarar vereceğini fark edecektir. Ve zeki okuyucumuz, haklısınız! Ancak bu sorunu, yığın boyutunu dikkatlice seçerek çözebilirsiniz. Spesifik olarak, parça boyutu yeterince büyükse, dosya sistemi zamanının çoğunu diskten veri aktarmakla ve (göreceli olarak) küçük bir süre bloğun parçaları arasında arama yapmakla geçirecektir. Ödenen genel gider başına daha fazla iş yaparak bir genel gideri azaltma işlemine **amortisman (amortization)** denir ve bilgisayar sistemlerinde yaygın bir tekniktir.

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

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

Temel olarak, bu denklemin söylediği şey şudur: 40 MB/s hızında veri aktarırsanız, zamanınızın yarısını aramaya, yarısını da aktarmaya harcamak için her aramanızda 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 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, düğümün kendi yapısına dayanan



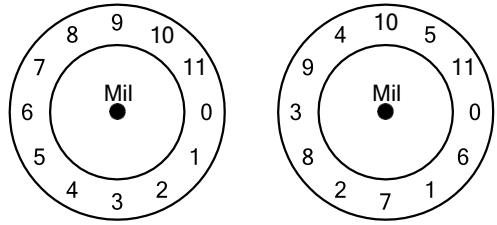
Şekil 41.2: **Amortisman : Parçalar Ne Kadar Büyük Olmalı?**

basit bir yaklaşım benimsemiştir İlk on iki doğrudan blok, inode ile aynı gruba yerleştirildi; sonraki her dolaylı blok ve işaret ettiği tüm 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 ima eder, tek istisna doğrudan işaretçiler tarafından işaret edilen dosyanın ilk 48KB'sidir. 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 [Sf98]. Bunun anlamı, zamanla mekanik maliyetlerin nispeten daha pahalı hale gelmesidir ve bu nedenle söz konusu maliyetleri amorti etmek için aramalar arasında daha fazla veri aktarmanız gerekir.

41.7 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. Dolayısıyla bu **dahili parçalanma (internal fragmentation)**, 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 (örneğin 1 KB boyutunda), iki alt bloğu kaplar ve böylece 4 KB'lik bir bloğun tamamını boşa harcamaz.



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

Dosya büyüdükçe, dosya sistemi tam 4 KB veri edinene kadar dosyaya 512 baytlık bloklar 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 çalışma gerektirdiğini (özellikle kopyalamayı gerçekleştirmek için çok fazla G/Ç) gözlemleyebilirsiniz. Ve yine haklısınız! 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 uzmanlığından 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 bilgisayar CPU'sunun 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, sorun sıralı okumalar sırasında 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), diskin başını geçmeden, sonraki bloğu istemek için yeterli zamana sahip olur. Aslında FFS, ekstra dönüşlerden kaçınmak için belirli bir disk için *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 (parameterization)** adı verildi.

Şöyle düşünüyor olabilirsiniz: bu şema 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. Ne yazık ki, modern diskler çok daha akıllıdır: dahili olarak tüm izi okurlar ve onu dahili bir disk önbelleğinde arabelleğe alırlar (tam da bu nedenle genellikle **iz arabelleği (track buffer)** olarak adlandırılır). Ardından, izin sonraki okumalarında, disk istenen verileri önbelleğinden döndürür.

İPUCU: SİSTEMİ KULLANILABİLİR HALE GETİRİN

Muhtemelen FFS'den çıkarılacak en temel ders, yalnızca diske duyarlı düzenin kavramsal olarak iyi bir fikrini tanıtmakla kalmayıp, aynı zamanda sistemi daha kullanışlı hale getiren bir dizi özellik eklemesidir. Uzun dosya adları, sembolik bağlantılar ve atomik olarak çalışan bir yeniden adlandırma işlemi, tümü bir sistemin yararını artırdı; 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.

Böylece dosya sistemleri artık bu inanılmaz derecede düşük düzeyli ayrıntılar hakkında endişelenmek zorunda kalmıyor. 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 dosya sisteminde geleneksel sabit boyutlu yaklaşım (örneğin, 8 karakter) yerine daha anlamlı adlar sağladı. Ayrıca, **sembolik bağlantı (symbolic link)** adı verilen yeni bir kavram tanıtıldı. Önceki bir bölümde [AD14b] tartışıldığı gibi, sabit bağlantılar sınırlıdır, çünkü her ikisi de dizinlere işaret edememektedir (dosya sistemi hiyerarşisinde döngüler getirme korkusuyla) ve yalnızca aynı birimdeki dosyalara işaret edebilmektedirler (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 `rename()` işlemi başlattı. Temel teknolojinin ötesindeki kullanılabilirlik iyileştirmeleri, FFS'ye daha güçlü bir kullanıcı tabanı kazandırdı.

41.8 Özet

FFS'nin tanıtılması, dosya yönetimi sorununun bir işletim sistemindeki en ilginç konulardan biri olduğunu açıkça ortaya koyduğu ve cihazların en önemlisi olan sabit diskle nasıl başa çıkmaya başlanabileceğini gösterdiği için, dosya sisteminde 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). 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 onhis work building FFS. In his acceptance speech, he discussed the original FFS software: only 1200lines of code! Modern versions are a little more complex, e.g., the BSD FFS descendant now is in the50-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 BENİOKU'ya 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 görünür? Kontrol etmek için `-c` ile çalıştırın.

```
enes@ubuntu:~/ostep-homework/file-ffs$ ./ffs.py -f in.largefile -c -L 4

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

`-f in.largefile` giriş dosyasını `-L 4` bayraklarıyla çalıştırdığımızda göründüğü gibi her grubun dört 'a' veri bloğu vardır.

2. Şimdi `-L 30` ile çalıştırın. Ne görmeyi beklersiniz? Bir kez daha, haklı olup olmadığınızı görmek için `-c` 'yi açın. `/a` dosyasına tam olarak hangi blokların ayrıldığını görmek için `-S` komutunu da kullanabilirsiniz.

```
enes@ubuntu:~/ostep-homework/file-ffs$ ./ffs.py -f in.largefile -c -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

     00000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

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

`-L 30 ile` çalıştırdığımızda ve –c ile control ettiğimizde yukarıdaki bu sonuçlara ulaşırız.

```
enes@ubuntu:~/ostep-homework/file-ffs$ ./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

     00000000000000000000 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
```

Ayrıca -S komutunu kullanarak /a dosyasına hangi blokların ayrıldığını yukarıdaki şekilde görebiliriz.

Sonuç olarak bu, büyük dosya istisnası seçeneğini eklememekle aynı şeydir.

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 (dosya genişliği)* dediğimiz bir şey. `/a` 'nın flespan'ini (dosya genişliğini) hesaplayın. Ne olduğunu görmek için `ffs.py -f in.largefile -L 4 -T –c` komutunu çalıştırın. Aynısını `-L 100` ile yapın. Büyük dosya özel durum parametresi düşük değerlerden yüksek değerlere değiştikçe flsapen 'de (dosya genişliğinde) ne gibi bir fark beklersiniz?

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./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
```

ffs.py -f in.largefile -L 4 -T –c komutu çalıştığında yukarıdaki sonuç çıkar.

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./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
```

Aynısını -L 100 iile yaptığımızda ise yukarıdaki sonucu görürüz.

Büyük dosya istisna parametresi düşük değerlerden yüksek değerlere değiştikçe flaspen (dosya genişliği) azalmalıdır.

4. Şimdi yeni bir giriş dosyasına bakalım, `in.manyfiles`. FFS politikasının bu dosyaları gruplar arasında nasıl yerleştireceğini 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.

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./ffs.py -f in.manyfiles -v
op create /a [size:2] ->success
op create /b [size:2] ->success
op create /c [size:2] ->success
op create /d [size:2] ->success
op create /e [size:2] ->success
op create /f [size:2] ->success
op create /g [size:2] ->success
op create /h [size:2] ->success
op create /i [size:2] ->success
op mkdir /j ->success
op mkdir /t ->success
op create /t/u [size:3] ->success
op create /j/l [size:1] ->success
op create /t/v [size:3] ->success
op create /j/m [size:1] ->success
op create /t/w [size:3] ->success
op create /j/n [size:1] ->success
op create /t/x [size:3] ->success
op create /j/o [size:1] ->success
op create /t/y [size:3] ->success
op create /j/p [size:1] ->success
op create /t/z [size:3] ->success
op create /j/q [size:1] ->success
op create /t/A [size:3] ->success
op create /j/r [size:1] ->success
op create /t/B [size:3] ->success
op create /j/C [size:3] ->success

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
```

-v ile çalıştırdığımızda gördüğümüz sonuç yukarıdadır.

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./ffs.py -f in.manyfiles -v -c
op create /a [size:2] ->success
op create /b [size:2] ->success
op create /c [size:2] ->success
op create /d [size:2] ->success
op create /e [size:2] ->success
op create /f [size:2] ->success
op create /g [size:2] ->success
op create /h [size:2] ->success
op create /i [size:2] ->success
op mkdir /j ->success
op mkdir /t ->success
op create /t/u [size:3] ->success
op create /j/l [size:1] ->success
op create /t/v [size:3] ->success
op create /j/m [size:1] ->success
op create /t/w [size:3] ->success
op create /j/n [size:1] ->success
op create /t/x [size:3] ->success
op create /j/o [size:1] ->success
op create /t/y [size:3] ->success
op create /j/p [size:1] ->success
op create /t/z [size:3] ->success
op create /j/q [size:1] ->success
op create /t/A [size:3] ->success
op create /j/r [size:1] ->success
op create /t/B [size:3] ->success
op create /j/C [size:3] ->success

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

Sonuç olarak aynı klasördeki dosyaların düğümleri ve veri blokları aynı grupta olmalıdır.

5. FFS'yi değerlendirmek için kullanılan bir metriğe *dirspan* denir. Bu metrik, belirli bir dizindeki dosyaların yayılımını, özellikle dizindeki tüm dosyaların düğümleri ve veri blokları ile dizinin kendisinin düğümü 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?

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./ffs.py -f in.manyfiles -c -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

     0000000000000000000 1111111111 2222222222
     01234567890123456789 0123456789 0123456789

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

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

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

Dispran değerleri yukarıda göründüğü gibidir.

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'ı nasıl etkiler?

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./ffs.py -f in.manyfiles -T -c -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
     01234567890123456789 0123456789 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------- ---------- ----------

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

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

Yukarıda göründüğü üzere aynı klasördeki dosyalar farklı gruplara dağılmıştır.

Boyutu bu şekilde değiştirmek dirspan 'i artıracaktır.

7. FFS yeni bir dizinin düğümünü hangi gruba yerleştirmelidir? Varsayılan (simülatör) politikası, en fazla boş düğüme sahip grubu arar. Farklı bir politika ise, en fazla boş düğüme sahip bir grup kurar. Örneğin, `-A 2` ile çalıştırırsanız, yeni bir dizin tahsis ederken, simülatör gruplara çiftler halinde bakar ve tahsis için en iyi çifti seçer. Bu stratejiyle tahsisin 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?

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./ffs.py -f in.manyfiles -I 5 -A 2 -c -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?    True
spread data?      False
contig alloc:     1

     0000000000000000000 1111111111 2222222222
     0123456789012345678 9 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 ---------- ---------- ---------- ----------

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

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

Bu işlem farklı klasörleri birbirindenuzak tutar. Dirspan 'i azaltacaktır. Grup başına düğüm sayısı küçük olduğunda, aynı klasördeki dosyaları daha yakın hale getirir.

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 doğrulamak için `-c` ile çalıştırın. `/i` dosyasının veri düzeni hakkında ilginç olan nedir? Bu neden sorunlu?

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./ffs.py -f in.fragmented -v -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:     1

     000000000000000000 1111111111 2222222222
     012345678901234567890 0123456789 0123456789

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

./ffs.py -f in.fragmented -v komutunu çalıştırıp –c ile control ettiğimizde yukarıdaki snuçlara ulaşırız.

Bu işlem sonucunda bloklar, sürekli / aralıksız bir blok yerine bölünür. Dosyayı okumak veya yazmak yavaş olacaktır. Sorun budur.

9. *Bitişik tahsis* (`-C`) adını verdiğimiz yeni bir politika, her dosyanın bitişik olarak tahsis edilmesini sağlamaya çalışır. Spesifik olarak, `-C n` ile dosya sistemi, bir bloğu yerleştirmeden 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 düzen nasıl değişir? Son olarak, `-C` filespan ve dirspan'ı nasıl etkiler?

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./ffs.py -f in.fragmented -v -c -T
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 /ib-d-f-h- /ibidifihi iii------- ----------
    1 ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ----------
    3 ---------- ---------- ---------- ----------
    4 ---------- ---------- ---------- ----------
    5 ---------- ---------- ---------- ----------
    6 ---------- ---------- ---------- ----------
    7 ---------- ---------- ---------- ----------
    8 ---------- ---------- ---------- ----------
    9 ---------- ---------- ---------- ----------

span: files
  file:         /i  filespan:  21
  file:         /h  filespan:  10
  file:         /f  filespan:  10
  file:         /d  filespan:  10
  file:         /b  filespan:  10
               avg  filespan:  12.20

span: directories
  dir:            /  dirspan:  22
                avg  dirspan:  22.00
```

Önceki sorudaki komutlara göre bulunan dispran 22 dir.

```
enes@ubuntu:~/Desktop/ostep-homework-master/ostep-homework-master/file-ffs$ ./ffs.py -f in.fragmented -v -C 2 -c -T
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 9 0123456789 0123456789

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

span: files
  file:         /i  filespan:  25
  file:         /h  filespan:  10
  file:         /f  filespan:  10
  file:         /d  filespan:  10
  file:         /b  filespan:  10
               avg  filespan:  13.00

span: directories
  dir:           /  dirspan:  26
               avg  dirspan:  26.00
```

-C ile bulunan yeni dispran 26 dır.

Ve şimdi `/i` dosyası bitişik olarak yerleştirilmiştir.

-C, filespan ve dirspan'i artıracaktır.