42

Kilitlenme Tutarlılığı: FSCK ve Günlük Kaydı

Şimdiye kadar gördüğümüz gibi, dosya sistemi beklenen soyutlamaları uygulamak için bir dizi veri yapısını yönetir: dosyalar, dizinler ve bir dosya sisteminden beklediğimiz temel soyutlamayı desteklemek için gereken diğer tüm meta veriler. Çoğu veri yapısının aksine (örneğin, çalışan bir programın belleğinde bulunanlar), dosya sistemi veri yapıları **persist**, yani, güç kaybına rağmen verileri tutan cihazlarda (sabit diskler veya flash tabanlı SSD'ler gibi) saklanan uzun mesafe boyunca hayatta kalmaları gerekir. Bir dosya sisteminin karşılaştığı en büyük zorluklardan biri, bir dosya sisteminin varlığına rağmen persis-tent veri yapılarının nasıl güncelleneceğidir.

power loss or **system crash**. Özellikle, disk üzerindeki yapıları güncellemenin tam ortasında, birisi güç kablosunun üzerinden geçerse ve makine güç kaybederse ne olur? Veya işletim sistemi bir hatayla karşılaşır ve çöker? Güç kayıpları ve çökmeler nedeniyle, kalıcı bir veri yapısını güncellemek oldukça zor olabilir ve dosya sistemi uygulamasında yeni ve ilginç bir soruna yol açar**.crash-consistency problem**.

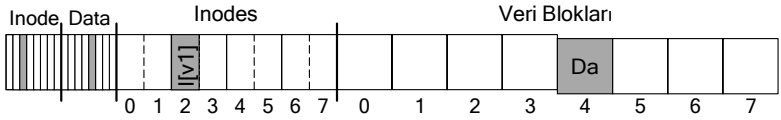
Bu sorunun anlaşılması oldukça basittir. Belirli bir işlemi tamamlamak için iki disk üzerindeki yapıyı, A ve B'yi güncellemeniz gerektiğini düşünün. Disk aynı anda yalnızca tek bir isteğe hizmet ettiğinden, bu isteklerden biri önce diske ulaşır (A veya B). Bir yazma işlemi tamamlandıktan sonra sistem çökerse veya güç kaybederse, diskteki yapı **inconsistent** devlet. Ve böylece, tüm dosya sistemlerinin çözmesi gereken bir sorunumuz var:

CRUX: ÇÖKMELERE RAĞMEN DİSK NASIL GÜNCELLENİR Sistem herhangi iki yazma arasında çökebilir veya güç kaybedebilir ve bu nedenle diskteki durum yalnızca kısmen güncellenebilir. Çökmeden sonra, sistem önyükleme yapar ve dosya sistemini tekrar monte etmek ister (dosyalara ve benzerlerine erişmek için). Kilitlenmelerin zaman içinde rastgele noktalarda meydana gelebileceği göz önüne alındığında, dosya sisteminin diskteki görüntüyü makul bir durumda tutmasını nasıl sağlayabiliriz?

Bu bölümde, bu sorunu daha ayrıntılı olarak açıklayacağız ve dosya sistemlerinin üstesinden gelmek için kullandığı bazı yöntemlere bakacağız. fsck veya dosya sistemi denetleyicisi olarak bilinen eski dosya sistemleri tarafından benimsenen yaklaşımı inceleyerek başlayacağız. Daha sonra dikkatimizi, günlük kaydı (önceden yazma günlüğü olarak da bilinir) olarak bilinen, her yazıya biraz ek yük ekleyen, ancak çökmelerden veya güç kayıplarından daha hızlı bir şekilde iyileşen bir teknik olan başka bir yaklaşıma çevireceğiz. Linux ext3 [T98,PAA05] (nispeten modern bir günlük dosyası sistemi) tarafından uygulanan birkaç farklı günlük kaydı da dahil olmak üzere günlük tutmanın temel mekanizmasını tartışacağız.

42.1 A Detailed Example

Günlük tutma araştırmamızı başlatmak için, bir örneğe bakalım. Disk üzerindeki yapıları bir şekilde güncelleyen bir iş yükü kullanmamız gerekir. Burada iş yükünün basit olduğunu varsayalım: tek bir veri bloğunun varolan bir dosyaya eklenmesi. Ekleme, dosya açılarak, dosya uzaklığını dosyanın sonuna taşımak için lseek() çağrılarak ve ardından kapatmadan önce dosyaya tek bir 4 KB yazma işlemi verilerek gerçekleştirilir. Ayrıca, daha önce gördüğümüz dosya sistemlerine benzer şekilde, diskte standart basit dosya sistemi yapıları kullandığımızı varsayalım. Bu küçük örnek, bir inode bitmap'i (yalnızca 8 bit, inode başına bir tane), bir veri bitmap'i (ayrıca 8 bit, veri bloğu başına bir tane), inode'ları (8 toplam, 0'dan 7'ye kadar numaralandırılmış ve dört bloğa yayılmış) ve veri bloklarını (8 toplam, 0 ila 7 arasında numaralandırılmış) içerir. İşte bu dosya sisteminin bir diyagramı: Bit eşlemler



Resimdeki yapılara bakarsanız, inode bitmap'te işaretlenmiş tek bir inode (inode numarası 2) ve veri bitmap'inde de işaretlenmiş tek bir tahsis edilmiş veri bloğunun (veri bloğu 4) tahsis edildiğini görebilirsiniz. Bu inode'un ilk versiyonu olduğu için inode I [v1] olarak gösterilir; yakında güncellenecektir (yukarıda açıklanan iş yükü nedeniyle). Bu basitleştirilmiş inode'un içine de göz atalım. I[v1]'in içinde şunları görüyoruz:

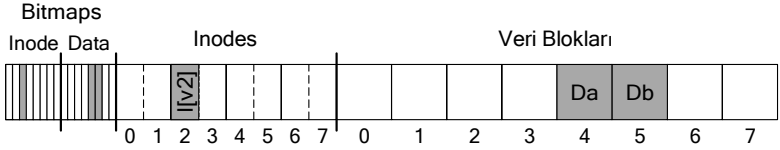
```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

Bu basitleştirilmiş inode'da, dosyanın boyutu 1'dir (bir blok tahsis edilmiştir), ilk doğrudan işaretçi blok 4'e işaret eder (ilk veri bloğu
dosya, Da) ve diğer üç doğrudan işaretçinin tümü null olarak ayarlanır

(kullanılmadıklarını gösterir). Tabii ki, gerçek inode'ların daha birçok alanı vardır; Daha fazla bilgi için önceki bölümlere bakın. Dosyaya eklediğimizde, ona yeni bir veri bloğu ekliyoruz ve bu nedenle üç disk yapısını güncellememiz gerekiyor: inode (yeni bloğa işaret etmeli ve ekleme nedeniyle yeni daha büyük boyutu kaydetmeli), yeni veri bloğu Db ve yeni veri bloğunun tahsis edildiğini belirtmek için veri bitmap'inin yeni bir sürümü (B [v2] olarak adlandırın). Böylece, sistemin hafızasında, diske yazmamız gereken üç bloğumuz var. Güncellenmiş inode (inode sürüm 2 veya kısaca I [v2]) şimdi şöyle görünüyor:

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

Güncellenmiş veri bitmap'i (B[v2]) artık şöyle görünür: 00001100. Son olarak, kullanıcıların dosyalara koyduğu her şeyle doldurulmuş veri bloğu (Db) vardır. Belki de çalınan müzik? İstediğimiz şey, dosya sisteminin son disk üzerindeki görüntüsünün şöyle görünmesidir:



Bu geçişi başarmak için, dosya sistemi diske her biri inode (I[v2]), bitmap (B[v2]) ve veri bloğu (Db) için olmak üzere üç ayırma hızında yazma işlemi gerçekleştirmelidir. Bu yazmaların, kullanıcı write() sistem çağrısı yaptığında genellikle hemen gerçekleşmediğini unutmayın; bunun yerine, kirli in-ode, bitmap ve yeni veriler önce bir süre ana bellekte (sayfa önbelleğinde veya arabellek önbelleğinde) oturacaktır; daha sonra, dosya sistemi nihayet bunları diske yazmaya karar verdiğinde (örneğin 5 saniye veya 30 saniye sonra), dosya sistemi gerekli yazma isteklerini diske gönderir. Ne yazık ki, bir kilitlenme oluşabilir ve bu nedenle diske yapılan bu güncelleştirmeleri engelleyebilir. Özellikle, bu yazılardan bir veya ikisi gerçekleştikten sonra bir çökme meydana gelirse, ancak üçü de gerçekleşmezse, dosya sistemi komik bir durumda bırakılabilir.

Kilitlenme Senaryoları

Sorunu daha iyi anlamak için, bazı kaza sce- narios'larına bakalım. Sadece tek bir yazının başarılı olduğunu hayal edin; Bu nedenle, burada listelediğimiz üç olası sonuç vardır:

- **Sadece veri bloğu (Db) diske yazılır.** Bu durumda, veriler disktedir, ancak ona işaret eden bir inode ve bloğun ayrıldığını bile söyleyen bir bitmap yoktur. Böylece, sanki yazma hiç gerçekleşmemiş gibidir. Bu durum, dosya sistemi kilitlenme tutarlılığı açısından hiç de sorun değildir1.

- **Sadece güncellenmiş inode (I [v2]) diske yazılır.** Bu durumda, inode, Db'nin on yazılmak üzere olduğu disk adresine (5) işaret eder, ancak Db henüz oraya yazılmamıştır. Bu nedenle, bu işaretçiye güvenirsek, çöp verilerini diskten okuyacağız (disk adresi 5'in eski içeriği). Dahası, dosya sistemi tutarsızlığı dediğimiz yeni bir sorunumuz var. Disk üzerindeki bitmap bize veri bloğu 5'in tahsis edilmediğini söylüyor, ancak inode olduğunu söylüyor. Bitmap ve inode arasındaki anlaşmazlık, dosya sisteminin veri yapılarında bir tutarsızlıktır; dosya sistemini kullanmak için bu sorunu bir şekilde çözmeliyiz (aşağıda daha fazlası).

- **Yalnızca güncellenmiş bitmap (B[v2]) diske yazılır.** Bu durumda, bitmap blok 5'in ayrıldığını gösterir, ancak buna işaret eden bir inode yoktur. Böylece dosya sistemi yine tutarsızdır; çözülmeden bırakılırsa, blok 5 dosya sistemi tarafından asla kullanılmayacağından bu yazma bir boşluk sızıntısına neden olur. Diske üç blok yazma girişiminde üç kilitlenme senaryosu daha vardır. Bu gibi durumlarda, iki yazma başarılı olur ve sonuncusu başarısız olur:
- **inode (I[v2]) ve bitmap (B[v2]) diske yazılır, ancak veriye yazılmaz (Db).** Bu durumda, dosya sistemi meta verileri tamamen tutarlıdır: inode 5'i engellemek için bir işaretçiye sahiptir, bitmap 5'in kullanımda olduğunu gösterir ve böylece dosya sisteminin meta verilerinin perspektifinden her şey yolunda görünür. Ancak bir sorun var: 5 tanesinde yine çöp var.

- **inode (I[v2]) ve veri bloğu (Db) yazılır, ancak bitmap (B[v2] yazılmaz) .** Bu durumda, diskteki cor-rect verilerine işaret eden inode'a sahibiz, ancak yine de bitmap'in (B1) in-ode ve eski sürümü arasında bir tutarsızlık var. Bu nedenle, dosya sistemini kullanmadan önce sorunu bir kez daha çözmemiz gerekiyor.

- **Bit eşlem (B[v2]) ve veri bloğu (Db) yazılır, ancak inode (I[v2] yazılmaz) .** Bu durumda, yine inode ve veri bitmap'i arasında bir tutarsızlık var. Bununla birlikte, blok yazılmış olmasına ve bitmap kullanımını göstermesine rağmen, hiçbir inode dosyaya işaret etmediğinden, hangi dosyaya ait olduğu hakkında hiçbir fikrimiz yoktur.

1Ancak, bazı verileri kaybeden kullanıcı için bir sorun olabilir!

Kilitlenme Tutarlılığı Sorunu

42.2 Umarım, bu kilitlenme senaryolarından, çökmeler nedeniyle disk üzerindeki dosya sistemi görüntümüzde oluşabilecek birçok sorunu görebilirsiniz: dosya sistemi veri yapılarında tutarsızlık olabilir; uzay sızıntılarımız olabilir; çöp verilerini bir kullanıcıya iade edebiliriz; ve benzerleri. İdeal olarak yapmak istediğimiz şey, dosya sistemini tutarlı bir durumdan (örneğin, dosya eklenmeden önce) atomik olarak başka bir duruma taşımaktır (örneğin, inode, bitmap ve yeni veri bloğu diske yazıldıktan sonra). Ne yazık ki, bunu kolayca yapamayız, çünkü disk bir seferde yalnızca bir yazma işlemi gerçekleştirir ve bu güncellemeler arasında çökmeler veya güç kaybı oluşabilir. Bu genel soruna kilitlenme tutarlılığı sorunu diyoruz (buna tutarlı güncelleştirme sorunu da diyebiliriz).

Çözüm #1: Dosya Sistemi Denetleyicisi

İlk dosya sistemleri, kilitlenme tutarlılığı için basit bir yaklaşım benimsedi. Temel olarak, tutarsızlıkların olmasına izin vermeye ve daha sonra (yeniden başlatırken) bunları düzeltmeye karar verdiler. Bu tembel yaklaşımın klasik bir örneği, bunu yapan bir araçta bulunur: fsck2. fsck, bu tür tutarsızlıkları bulmak ve onarmak için kullanılan bir UNIX aracıdır [MK96]; Bir disk bölümünü kontrol etmek ve onarmak için benzer araçlar farklı sistemlerde mevcuttur. Böyle bir yaklaşımın tüm sorunları çözemeyeceğini unutmayın; Örneğin, dosya sisteminin tutarlı göründüğü ancak inode'un çöp verilerine işaret ettiği yukarıdaki durumu düşünün. Tek gerçek amaç, dosya sistemi meta verilerinin dahili olarak tutarlı olduğundan emin olmaktır. fsck aracı, McKusick ve Kowalski'nin makalesinde [MK96] özetlendiği gibi birkaç aşamada çalışır. Dosya sistemi bağlanmadan ve kullanılabilir hale getirilmeden önce çalıştırılır (fsck, çalışırken başka hiçbir dosya sistemi etkinliğinin devam etmediğini varsayar); tamamlandığında, diskteki dosya sistemi tutarlı olmalı ve böylece kullanıcılar tarafından erişilebilir hale getirilmelidir.

İşte fsck'nin yaptıklarının temel bir özeti:

- **Süper blok:** fsck önce üst bloğun makul görünüp görünmediğini kontrol eder, çoğunlukla dosya sistemi boyutunun tahsis edilen blok sayısından daha büyük olduğundan emin olmak gibi akıl sağlığı kontrolleri yapar. Sonuç olarak, bu akıl sağlığı kontrollerinin amacı, şüpheli (yozlaşmış) bir süper blok bulmaktır; bu durumda, sistem (veya yönetici) süper bloğun alternatif bir kopyasını kullanmaya karar verebilir.
- **Ücretsiz bloklar:** Daha sonra, fsck şu anda dosya sisteminde hangi blokların tahsis edildiğini anlamak için inode'ları, dolaylı blokları, çift dolaylı blokları vb. tarar. Bu bilgiyi, ayırma bitmap'lerinin doğru bir sürümünü üretmek için kullanır; Böylece, bitmap'ler ve inode'lar arasında herhangi bir tutarsızlık varsa, inode'lar içindeki bilgilere güvenilerek çözülür. Tüm inode'lar için aynı tür bir kontrol gerçekleştirilir ve kullanımda gibi görünen tüm inode'ların inode bitmap'lerinde bu şekilde işaretlendiğinden emin olunur.

- **Inode state:** Her inode yolsuzluk veya diğer sorunlar için kontrol edilir. Örneğin, fsck, tahsis edilen her inode'un geçerli bir tür alanına sahip olduğundan emin olur (örneğin, normal dosya, dizin, sembolik bağlantı, vb.). Inode alanlarıyla ilgili kolayca düzeltilemeyen sorunlar varsa, inode şüpheli olarak kabul edilir ve fsck tarafından temizlenir; inode bitmap buna uygun olarak güncellenir.
- **Inode links:** fsck ayrıca tahsis edilen her bir in-ode'nin bağlantı sayısını da doğrular. Hatırlayabileceğiniz gibi, bağlantı sayısı, bu par-tiküler dosyaya bir başvuru (yani bir bağlantı) içeren farklı dizinlerin sayısını gösterir. Bağlantı sayısını doğrulamak için fsck, kök dizinden başlayarak en-tire dizin ağacını tarar ve dosya sistemindeki her dosya ve dizin için kendi bağlantı sayılarını oluşturur. Yeni hesaplanan sayım ile bir inode bulunan arasında bir uyumsuzluk varsa, genellikle inode içindeki sayıyı sabitleyerek düzeltici önlem alınmalıdır. Ayrılmış bir inode bulunursa, ancak hiçbir dizin buna başvurmazsa, kayıp + bulunan dizine taşınır.
- **Duplicates:** fsck ayrıca yinelenen işaretçileri, yani iki farklı inod'un aynı bloğa başvurduğu durumları da kontrol eder. Bir inode belirsiz bir şekilde kötüyse, temizlenebilir. Alternatif olarak, işaret edilen blok kopyalanabilir, böylece her inode istenildiği gibi kendi kopyasını verir.
- **Bad blocks:** Tüm işaretçilerin listesi taranırken hatalı blok işaretçileri için bir kontrol de gerçekleştirilir. Bir işaretçi, geçerli aralığının dışındaki bir şeye açıkça işaret ediyorsa "kötü" olarak kabul edilir, örneğin, katılım boyutundan daha büyük bir bloğa atıfta bulunan bir adrese sahiptir. Bu durumda, fsck çok zekice bir şey yapamaz; sadece işaretçiyi inode veya dolaylı bloktan kaldırır (temizler).
- **Directory checks:** fsck kullanıcı dosyalarının içeriğini anlamıyor; ancak, dizinler dosya sisteminin kendisi tarafından oluşturulan özel olarak biçimlendirilmiş bilgileri tutar. Bu nedenle, fsck her dizinin içeriği üzerinde ek bütünlük denetimleri gerçekleştirir, "." ve ".." girişlerinin ilk girişler olduğundan, bir dizin girişinde başvurulan her inode'un ayrıldığından ve hiçbir dizinin tüm hiyerarşide birden fazla kez bağlanmadığından emin olur.

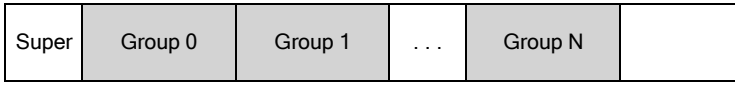
42.3 Gördüğünüz gibi, çalışan bir fsck oluşturmak, dosya sistemi hakkında karmaşık bilgi gerektirir; böyle bir kod parçasının her durumda doğru çalıştığından emin olmak zor olabilir [G+08]. Bununla birlikte, fsck'nin (ve benzer yaklaşımların) daha büyük ve belki de daha temel bir sorunu vardır: çok yavaştırlar. Çok büyük bir disk birimiyle, ayrılan tüm blokları bulmak ve dizin ağacının tamamını okumak için tüm diski taramak birkaç dakika veya saat sürebilir. Disklerin kapasitesi arttıkça ve RAID'lerin popülaritesi arttıkça fsck'nin performansı engelleyici hale geldi (son gelişmelere rağmen [M+13]). Daha yüksek bir seviyede, fsck'nin temel öncülü sadece biraz irra-tional görünüyor. Diske yalnızca üç bloğun yazıldığı yukarıdaki örneğimizi düşünün; sadece üç blokluk bir güncelleme sırasında meydana gelen sorunları düzeltmek için tüm diski taramak inanılmaz derecede pahalıdır. Bu durum, anahtarlarınızı yatak odanızda yere düşürmeye ve daha sonra anahtarlar için tüm evi aramaya başlamaya benzer.

42.4

Çözüm #2: Günlük Tutma (veya Önceden Yazma Günlüğü)

Muhtemelen tutarlı güncelleme sorununun en popüler çözümü, veritabanı yönetim sistemleri dünyasından bir fikir çalmaktır. Önceden yazma günlüğü olarak bilinen bu fikir, tam olarak bu tür bir sorunu çözmek için icat edildi. Dosya sistemlerinde, genellikle tarihsel nedenlerden dolayı önceden yazma günlüğü jour-naling diyoruz. Bunu yapan ilk dosya sistemi Cedar [H87] idi, ancak Linux ext3 ve ext4, reiserfs, IBM'in JFS'si, SGI'nın XFS ve Windows NTFS dahil olmak üzere birçok modern dosya sistemi bu fikri kullanıyor. Temel fikir aşağıdaki gibidir. Diski güncellerken, yapıların üzerine yazmadan önce, önce ne yapmak üzere olduğunuzu açıklayan küçük bir not (diskte başka bir yerde, iyi bilinen bir yerde) yazın. Bu notu yazmak "ileriye yaz" kısmıdır ve biz de bunu "günlük" olarak düzenlediğimiz bir yapıya yazarız; bu nedenle, önceden yazma günlüğü.

Notu diske yazarak, güncelleştirmekte olduğunuz yapıların güncellenmesi (üzerine yazılması) sırasında bir kilitlenme meydana gelirse, geri dönüp yaptığınız nota bakıp tekrar deneyebileceğinizi garanti edersiniz; Böylece, tüm diski taramak yerine, bir çökmeden sonra tam olarak neyi düzelteceğinizi (ve nasıl düzelteceğinizi) bileceksiniz. Tasarım gereği, günlük kaydı bu nedenle kurtarma sırasında gereken iş miktarını büyük ölçüde azaltmak için güncellemeler sırasında biraz iş ekler. Şimdi popüler bir günlük kaydı dosya sistemi olan Linux ext3'ün günlüklemeyi dosya sistemine nasıl dahil ettiğini açıklayacağız. Disk üzerindeki yapıların çoğu Linux ext2 ile aynıdır, örneğin, disk blok gruplarına ayrılmıştır ve her blok grubu bir inode bitmap, veri bitmap, inode ve veri blokları içerir. Yeni anahtar yapı, bölüm içinde veya başka bir cihazda az miktarda yer kaplayan günlüğün kendisidir. Böylece, bir ext2 dosya sistemi (günlük kaydı olmadan) şöyle görünür:

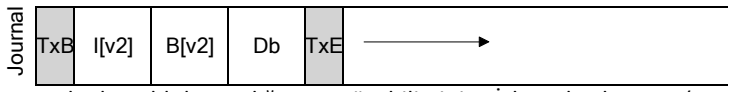


Günlük aynı dosya sistemi görüntüsüne yerleştirildiğini varsayarsak (bazen ayrı bir aygıta veya dosya sistemi içindeki bir dosya olarak yerleştirilse de), günlüğü olan bir ext3 dosya sistemi şöyle görünür:

The real difference is just the presence of the journal, and of course, how it is used.

Veri Günlüğe Kaydetme

Veri günlüğünün nasıl çalıştığını anlamak için basit bir örneğe bakalım. Veri günlüğü, bu tartışmanın çoğunun dayandığı Linux ext3 dosya sistemi ile bir mod olarak kullanılabilir.Diyelim ki inode (I[v2]), bitmap (B[v2]) ve veri bloğunu (Db) tekrar diske yazmak istediğimiz kanonik güncellememizi tekrar aldık. Onları son disk konumlarına yazmadan önce, şimdi onları günlüğe (diğer adıyla günlük) yazacağız. Bu, günlükte böyle görünecek:



Burada beş blok yazdığımızı görebilirsiniz. İşlem başlangıcı (TxB), dosya sistemine yapılan güncelleme (örneğin, I[v2], B[v2] ve Db bloklarının son adresleri) ve bir tür işlem tanımlayıcısı (TID) dahil olmak üzere bu güncelleme hakkında bilgi verir. Orta dle üç blok sadece blokların kendilerinin tam içeriğini içerir; bu, güncelleştirmenin tam fiziksel içeriğini günlüğe kaydettiğimiz için fiziksel günlük kaydı olarak bilinir (alternatif bir fikir, mantıksal günlükleme, güncelleştirmenin daha kompakt bir mantıksal temsilini günlükte koyar, örneğin, "bu güncelleştirme, X dosyasına veri bloğu Db eklemek istiyor", bu biraz daha karmaşıktır, ancak günlükte yer tasarrufu sağlayabilir ve belki de performansı artırabilir). Son blok (TxE), bu transaksiyonun sonunun bir göstergesidir ve ayrıca TID'yi de içerecektir. Bu işlem diskte güvenli bir şekilde olduğunda, dosya sistemindeki eski yapıların üzerine yazmaya hazırız; bu işleme denetim noktası adı verilir. Bu nedenle, dosya sistemini kontrol etmek için (yani, dergideki düzeltme güncellemesiyle güncel hale getirmek), yukarıda görüldüğü gibi I[v2], B[v2] ve Db yazılarını disk konumlarına göndeririz; Bu yazmalar başarıyla tamamlanırsa, dosya sistemini başarıyla kontrol ettik ve temel olarak yapıldı. Böylece, ilk işlem sıramız:

Journal write: İşlem başlangıç bloğu, bekleyen tüm veriler ve meta veri güncellemeleri ve bir işlem sonu bloğu dahil olmak üzere işlemi günlüğe yazın; bu yazmaların tamamlanmasını bekleyin.

Checkpoint: Bekleyen meta verileri ve veri güncellemelerini dosya sistemindeki son konumlarına yazın.

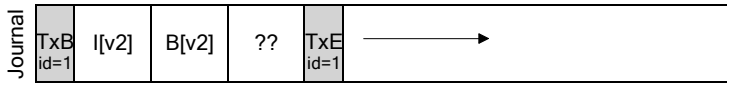
Örneğimizde, dergiye önce TxB, I[v2], B[v2], Db ve TxE yazacağız. Bu yazma işlemleri tamamlandığında, I[v2], B[v2] ve Db'yi diskteki son konumlarına denetleyerek güncelleştirmeyi tamamlarız. Dergiye yazılan yazılar sırasında bir çökme meydana geldiğinde işler biraz daha zorlaşır. Burada, transaksiyondaki bloklar kümesini (örneğin, TxB, I [v2], B [v2], Db, TxE) diske yazmaya çalışıyoruz. Bunu yapmanın basit bir yolu, her birini bir kerede yayınlamak, her birinin tamamlanmasını beklemek ve ardından bir sonrakini yayınlamak olacaktır. Ancak, bu yavaştır. İdeal olarak, yayınlamak istiyoruz

BIR KENARA: DISKE YAZMAYI ZORLAMA

İki disk yazma işlemi arasında sıralamayı zorlamak için, modern dosya sistemlerinin birkaç ek önlem alması gerekir. Eski zamanlarda, iki yazma arasında, A ve B arasında sıralamayı zorlamak kolaydı: sadece A'nın diske yazılmasını sağlayın, yazma tamamlandığında diskin işletim sistemini kesmesini bekleyin ve ardından B'nin yazmasını yayınlayın. Disklerdeki yazma önbelleklerinin artan kullanımı nedeniyle işler biraz daha karmaşık hale geldi. Yazma arabelleğe alma etkinken (bazen anında raporlama olarak da adlandırılır), bir disk, diskin bellek önbelleğine yerleştirildiğinde ve henüz diske ulaşmadığında işletim sistemine yazma işleminin tamamlandığını bildirir. İşletim sistemi daha sonra bir sonraki yazma işlemini yayınlarsa, önceki yazmalardan sonra diske ulaşması garanti edilmez; böylece yazılar arasında sıralama korunmaz. Çözümlerden biri, yazma arabelleğe almayı devre dışı bırakmaktır. Bununla birlikte, daha modern sistemler ekstra önlemler alır ve açık yazma engelleri çıkarır; Böyle bir engel, tamamlandığında, engelden önce verilen tüm yazmaların, engelden sonra verilen herhangi bir yazmadan önce diske ulaşmasını garanti eder.

Tüm bu makineler, diskin doğru şekilde çalıştırılmasına büyük güven gerektirir. Ne yazık ki, son zamanlarda yapılan araştırmalar, bazı disk üreticilerinin, "daha yüksek performanslı" diskler sunmak amacıyla, yazma engeli isteklerini açıkça görmezden geldiklerini ve böylece disklerin görünüşte daha hızlı çalışmasını sağladığını, ancak yanlış çalışma riski altında olduğunu göstermektedir [C + 13, R + 11]. Kahan'ın dediği gibi, oruç yanlış olsa bile, oruç neredeyse her zaman yavaşı yener.

(2) beş bloğun tümü aynı anda yazar, çünkü bu, beş yazmayı tek bir sıralı yazıya dönüştürür ve böylece daha hızlı olur. Bununla birlikte, takip eden nedenden dolayı bu güvenli değildir: bu kadar büyük bir yazma göz önüne alındığında, disk dahili olarak zamanlama yapabilir ve büyük yazının küçük parçalarını herhangi bir sırayla tamamlayabilir. Bu nedenle, disk dahili olarak (1) TxB, I [v2], B [v2] ve TxE yazabilir ve ancak daha sonra Db yazabilir.



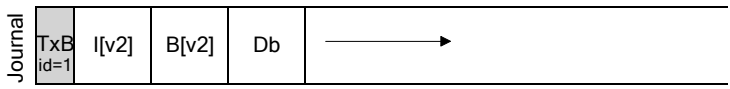
Bu neden bir sorun? İşlem geçerli bir trans-action gibi görünüyor (eşleşen sıra numaralarıyla bir başlangıcı ve bir sonu var). Dahası, dosya sistemi bu dördüncü bloğa bakamaz ve yanlış olduğunu bilemez; sonuçta, keyfi kullanıcı verileridir. Bu nedenle, sistem şimdi yeniden başlatılır ve kurtarmayı çalıştırırsa, bu işlemi tekrar oynatacak ve çöp bloğunun içeriğini cahilce kopyalayacaktır. Db'nin yaşaması gereken yere. Bu, bir dosyadaki rasgele kullanıcı verileri için kötüdür; dosya sistemini monte edilemez hale getirebilecek süper blok gibi kritik bir dosya sistemi parçasına dönüşürse çok daha kötüdür.

BIR KENARA: GÜNLÜK YAZMA IŞLEMLERINI OPTIMIZE ETME

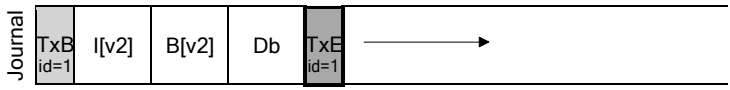
Günlüğe yazmanın belirli bir verimsizliğini fark etmiş olabilirsiniz. Yani, dosya sistemi öncelikle işlem-başlangıç bloğunu ve işlemin içeriğini yazmalıdır; ancak bu yazma işlemleri tamamlandıktan sonra dosya sistemi işlem sonu bloğunu diske gönderebilir. Bir diskin nasıl çalıştığını düşünürseniz, performans anlaşması açıktır: genellikle ekstra bir rotasyon oluşur (nedenini düşünün). Eski yüksek lisans öğrencilerimizden biri olan Vijayan Prabhakaran'ın bu sorunu çözmek için basit bir fikri vardı [P + 05]. Dergiye bir işlem yazarken, derginin içeriğinin bir sağlama toplamını başlangıç ve bitiş bloklarına ekleyin. Bunu yapmak, dosya sisteminin beklemeye gerek kalmadan tüm işlemi bir kerede yazmasını sağlar; kurtarma sırasında, dosya sistemi işlemde depolanan sağlama toplamı ile hesaplanan sağlama toplamında bir uyumsuzluk görürse, işlemin yazılması sırasında bir kilitlenme oluştuğu sonucuna varabilir ve böylece dosya sistemi güncelleştirmesini atabilir. Böylece, yazma protokolünde ve kurtarma sisteminde küçük bir ince ayar ile, bir dosya sistemi daha hızlı ortak durum performansı elde edebilir; bunun da ötesinde, sistem biraz daha güvenilirdir, çünkü dergiden gelen tüm okumalar artık bir sağlama toplamı ile korunmaktadır.

Bu basit düzeltme, Linux dosya sistemi geliştiricilerinin dikkatini çekecek kadar çekiciydi ve daha sonra onu yeni nesil Linux dosya sistemine dahil ettiler (tahmin ettiniz!) Linux ext4. Şimdi Android el tipi platform da dahil olmak üzere dünya çapında milyonlarca makineye gönderiliyor. Böylece, birçok Linux tabanlı sistemde diske her yazdığınızda, Wisconsin'de geliştirilen küçük bir kod, sisteminizi biraz daha hızlı ve daha güvenilir hale getirir.

Bu sorunu önlemek için, dosya sistemi işlem yazma işlemini iki adımda verir. İlk olarak, TxE bloğu dışındaki tüm blokları dergiye yazar, bu yazıların hepsini bir kerede yayınlar. Bu yazmalar tamamlandığında, günlük şöyle bir şeye benzeyecektir (ekleme iş yükümüzü tekrar varsayarsak):



Bu yazmalar tamamlandığında, dosya sistemi TxE bloğunun yazımını yayınlar ve böylece günlüğü bu son, güvenli durumda bırakır.:



Bu işlemin önemli bir yönü, disk tarafından sağlanan atomiklik garantisidir. Diskin, herhangi bir 512 baytlık yazmanın gerçekleşeceğini veya gerçekleşmeyeceğini (ve asla yarı yazılı olmayacağını) garanti ettiği ortaya çıktı; Bu nedenle, TxE'nin yazılmasının atomik olduğundan emin olmak için, onu tek bir 512 baytlık blok haline getirmelisiniz. Böylece, dosya sistemini güncellemek için mevcut protokolümüz, üç aşamasının her biri etiketlenmiş olarak:

1. **Journal write:** İşlemin içeriğini (TxB, meta veriler ve veriler dahil) günlüğe yazın; bu yazmaların tamamlanmasını bekleyin.
2. **Journal commit:** İşlem taahhüt bloğunu (TxE içeren) günlüğe yazın; yazmanın tamamlanmasını bekleyin; işlemin yapıldığı söylenir.
3. **Checkpoint:** Güncellemenin içeriğini (meta veriler ve veriler) diskteki son konumlarına yazma

.

Kurtarma

Şimdi bir dosya sisteminin bir çökmeden kurtulmak için jour-nal içeriğini nasıl kullanabileceğini anlayalım. Bu güncelleştirme dizisi sırasında herhangi bir zamanda kilitlenme meydana gelebilir. Çökme, işlem kütüğe güvenli bir şekilde yazılmadan önce (yani, yukarıdaki Adım 2 tamamlanmadan önce) gerçekleşirse, işimiz kolaydır: bekleyen güncelleme basitçe atlanır. Kilitlenme, işlem günlüğe kaydedildikten sonra gerçekleşirse, ancak kontrol noktası tamamlanmadan önce, dosya sistemi güncellemeyi aşağıdaki gibi kurtarabilir. Sistem önyüklendiğinde, dosya sistemi kurtarma işlemi günlüğü tarar ve diske kaydedilmiş işlemleri arar; Böylece bu işlemler (sırayla) yeniden oynatılır, dosya sistemi işlemdeki blokları diskteki son konumlarına yazmaya çalışır. Bu günlük biçimi, var olan en basit biçimlerden biridir ve günlüğü yinele olarak adlandırılır. Dosya sistemi, günlükte gerçekleştirilen işlemleri kurtararak, diskteki yapıların tutarlı olmasını sağlar ve böylece dosya sistemini bağlayıp kendisini yeni istekler için hazırlayarak devam edebilir.

Kontrol noktası sırasında herhangi bir noktada, blokların son konumlarındaki bazı güncellemeler tamamlandıktan sonra bile bir çökmenin meydana gelmesinin sorun olmadığını unutmayın. En kötü durumda, bu güncellemelerden bazıları kurtarma sırasında tekrar oluşturulur. Kurtarma nadir bir işlem olduğundan (yalnızca beklenmeyen bir sistem çökmesinden sonra gerçekleşir), birkaç gereksiz yazma işlemi endişelenecek bir şey değildir3.

Günlük Güncelleştirmelerini Toplu Hale Getirme

Temel protokolün çok fazla ekstra disk trafiği ekleyebileceğini fark etmiş olabilirsiniz. Örneğin, aynı dizinde dosya1 ve dosya2 adlı iki dosyayı arka arkaya oluşturduğumuzu varsayalım. Bir dosya oluşturmak için, aşağıdakiler de dahil olmak üzere bir dizi disk yapısını güncellemek gerekir: in-ode bitmap (yeni bir inode ayırmak için), dosyanın yeni oluşturulan inode'u,

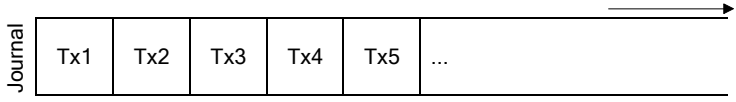
[3]Unless you worry about everything, in which case we can't help you. Stop worrying so much, it is unhealthy! But now you're probably worried about over-worrying.

yeni dizin en- try'yi içeren üst dizinin veri bloğu ve üst dizin inode (şimdi yeni bir değişiklik zamanına sahip). Günlük kaydı ile, tüm bu bilgileri mantıksal olarak iki dosya oluşturmamızın her biri için dergiye taahhüt ederiz; çünkü dosyalar aynı dizindedir ve aynı in-ode bloğu içinde inode'ları bile olduğunu varsayarsak, bu, dikkatli olmazsak, aynı blokları tekrar tekrar yazacağımız anlamına gelir.

Bu sorunu gidermek için, bazı dosya sistemleri her güncellemeyi diske teker teker yapmaz (örneğin, Linux ext3); bunun yerine, tüm güncellemeleri küresel bir işlemde arabelleğe alabilirsiniz. Yukarıdaki örneğimizde, iki dosya oluşturulduğunda, dosya sistemi yalnızca bellek içi inode bitmap'ini, dosyaların inode'larını, dizin verilerini ve dizin inode'unu kirli olarak işaretler ve bunları geçerli işlemi oluşturan bloklar listesine ekler. Sonunda bu blokları diske yazma zamanı geldiğinde (örneğin, 5 saniyelik bir zaman aşımından sonra), yukarıda açıklanan tüm güncellemeleri içeren bu tek genel işlem gerçekleştirilir. Bu nedenle, güncellemeleri arabelleğe alarak, bir dosya sistemi birçok durumda diske aşırı yazma trafiğini önleyebilir.

Kütük Sonlu Yapmak

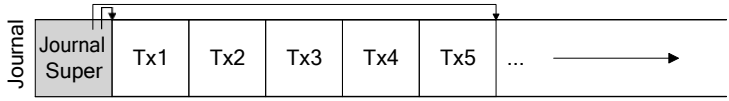
Böylece, disk üzerindeki dosya sistemi yapılarını güncellemek için temel bir protokole ulaştık. Dosya sistemi bellekteki güncelleştirmeleri bir süreliğine arabelleğe alır; nihayet diske yazma zamanı geldiğinde, dosya sistemi önce işlemin ayrıntılarını dikkatlice günlüğe yazar (diğer adıyla ileri yazma günlüğü); İşlem tamamlandıktan sonra, dosya sistemi bu blokları diskteki son konumlarına kadar kontrol eder. Ancak, günlük sonlu bir boyuttadır. Buna işlem eklemeye devam edersek (bu şekilde olduğu gibi), yakında dolacaktır. Sizce o zaman ne olur?



Günlük dolduğunda iki sorun ortaya çıkar. Birincisi daha basit, ancak daha az kritiktir: günlük ne kadar büyük olursa, kurtarma işleminin kurtarmak için günlükteki tüm işlemleri (sırayla) yeniden yürütmesi gerektiğinden, kurtarma işlemi o kadar uzun sürer. İkincisi daha çok bir sorundur: günlük dolu olduğunda (veya neredeyse doluyken), diske başka işlem yapılamaz, böylece dosya sistemi "kullanışlıdan daha az" (yani işe yaramaz) hale gelir.

Bu sorunları gidermek için, günlük kaydı dosya sistemleri günlüğü dairesel bir veri yapısı olarak ele alır ve tekrar tekrar kullanır; bu nedenle dergiye bazen dairesel bir günlük denir. Bunu yapmak için, dosya sisteminin bir denetim noktasından bir süre sonra harekete geçmesi gerekir. Özellikle, bir işlem kontrol noktası belirlendikten sonra, dosya sistemi günlük içinde işgal ettiği alanı boşaltmalı ve günlük alanının yeniden kullanılmasına izin vermelidir. Bu amaca ulaşmanın birçok yolu vardır; örneğin, basitçe

bir dergi süper bloğundaki günlükteki en eski ve en yeni kontrol noktası olmayan işlemler; diğer tüm alanlar ücretsizdir. İşte grafiksel bir tasvir:



Dergi süper bloğunda (ana dosya sistemi süper bloğu ile karıştırılmamalıdır), günlük kaydı sistemi hangi işlemlerin henüz kontrol edilmediğini bilmek için yeterli bilgiyi kaydeder ve böylece yeniden oluşturma süresini azaltır ve günlüğün dairesel bir şekilde yeniden kullanılmasını sağlar. Ve böylece temel protokolümüze bir adım daha ekliyoruz:

1. **Journal write:** İşlemin içeriğini (TxB ve güncellemenin içeriğini içeren) günlüğe yazın; bu yazmaların tamamlanmasını bekleyin.
2. **Journal commit:** İşlem taahhüt bloğunu (TxE içeren) günlüğe yazın; yazma işleminin tamamlanmasını bekleyin; işlem artık gerçekleştirilmiştir.
3. **Checkpoint:** Güncellemenin içeriğini dosya sistemi içindeki son konumlarına yazma.
4. **Free:** Bir süre sonra, dergi süper bloğunu güncelleyerek işlemi günlükte ücretsiz olarak işaretleyin.

Böylece nihai veri günlüğü protokolümüze sahibiz. Ancak yine de bir sorun var: Her veri bloğunu diske iki kez yazıyoruz, bu da özellikle sistem çökmesi kadar nadir görülen bir şey için ağır bir ödeme maliyeti. Verileri iki kez yazmadan tutarlılığı korumanın bir yolunu bulabilir misiniz?

Meta Veri Günlüğe Kaydetme

Kurtarma şimdi hızlı olsa da (günlüğü taramak ve tüm diski taramak yerine birkaç işlemi yeniden yürütmek), dosya sisteminin normal çalışması istediğimizden daha yavaştır. Özellikle, diske her yazma için, şimdi de önce dergiye yazıyoruz, böylece yazma trafiğini iki katına çıkarıyoruz; Bu iki katına çıkma, artık sürücünün en yüksek yazma bant genişliğinin yarısında ilerleyecek olan sıralı yazma iş yükleri sırasında özellikle acı vericidir. Ayrıca, günlüğe yazma ve ana dosya sistemine yazma arasında, bazı iş yükleri için gözle görülür bir ek yük ekleyen maliyetli bir arama vardır.

> Her veri bloğunu diske iki kez yazmanın yüksek maliyeti nedeniyle, insanlar performansı hızlandırmak için birkaç farklı şey denediler. Örneğin, yukarıda açıkladığımız günlük tutma moduna, tüm kullanıcı verilerini (dosya sisteminin meta verilerine ek olarak) günlüğe kaydettiği için genellikle veri günlüğü (Linux ext3'te olduğu gibi) denir. Daha basit (ve daha yaygın) bir günlük tutma biçimine bazen sıralı günlük kaydı (veya yalnızca meta veri günlüğü) denir ve kullanıcı verilerinin dergiye yazılmaması dışında neredeyse aynıdır. Böylece, yukarıdaki güncellemenin aynısı yapılırken aşağıdaki bilgiler dergiye yazılacaktır:

Daha önce günlüğe yazılan veri bloğu Db, bunun yerine dosya sistemine uygun şekilde yazılır ve fazladan yazmayı önler; Diske giden G/Ç trafiğinin çoğunun veri olduğu göz önüne alındığında, verileri iki kez yazmamak, günlük kaydının G/Ç yükünü önemli ölçüde azaltır. Değişiklik ilginç bir soruyu gündeme getiriyor: veri bloklarını diske ne zaman yazmalıyız?

Sorunu daha iyi anlamak için bir dosyanın örnek ekini tekrar ele alalım. Güncelleme üç bloktan oluşur: I[v2], B[v2] ve Db. İlk ikisi de meta verilerdir ve günlüğe kaydedilir ve ardından kontrol noktası gösterilir; ikincisi dosya sistemine yalnızca bir kez yazılacaktır. Db'yi diske ne zaman yazmalıyız? Önemli mi?

Görünüşe göre, veri yazmanın sıralanması meta veriler için önemlidir - yalnızca günlük kaydı. Örneğin, trans- action (I[v2] ve B[v2] içeren) tamamlandıktan sonra diske Db yazarsak ne olur? Ne yazık ki, bu ap- proach'ın bir sorunu var: dosya sistemi tutarlı, ancak I [v2] çöp verilerine işaret edebilir. Özellikle, I[v2] ve B[v2] yazıldığı ancak Db'nin diske girmediği durumu düşünün. Dosya sistemi daha sonra kurtarmaya çalışacaktır. Db günlükte olmadığından, dosya sistemi I[v2] ve B[v2]'ye yazmaları yeniden yürütür ve tutarlı bir dosya sistemi oluşturur (dosya sistemi meta verileri perspektifinden). Bununla birlikte, I [v2] çöp verilerine, yani Db'nin yöneldiği yuvada ne varsa ona işaret ediyor olacağım.

Bu durumun ortaya çıkmamasını sağlamak için, bazı dosya sistemleri (örneğin, Linux ext3), ilgili meta veriler diske yazılmadan önce, önce diske veri blokları (normal dosyalardan) yazar. Özellikle, protokol aşağıdaki gibidir:

1. **Data write:** Verileri son konuma yazın; tamamlanmayı bekleyin (bekleme isteğe bağlıdır; ayrıntılar için aşağıya bakın).
2. **Journal metadata write:** Başlangıç bloğunu ve meta verileri günlüğe yazın; yazmaların tamamlanmasını bekleyin.
3. **Journal commit:** İşlem taahhüt bloğunu (TxE içeren) günlüğe yazın; yazma işleminin tamamlanmasını bekleyin; işlem (in-cluding data) artık gerçekleştirilmiştir.
4. **Checkpoint metadata:** Meta veri güncelleştirmesinin içeriğini dosya sistemi içindeki son konumlarına yazın.
5. **Free:** Daha sonra, işlemi dergi üst bloğunda ücretsiz olarak işaretleyin.

Bir dosya sistemi, önce verileri yazmaya zorlayarak, işaretçinin hiçbir zaman çöpe işaret etmeyeceğini garanti edebilir. Gerçekten de, bu "işaret edilen nesneyi işaret eden nesneden önce yazın" kuralı, çarpışma tutarlılığının özündedir ve diğer kilitlenme tutarlılığı şemaları [GP94] tarafından daha da fazla kullanılır (ayrıntılar için aşağıya bakın).

Çoğu sistemde, meta veri günlüğü oluşturma (ext3'ün sıralı günlük kaydına benzer), tam veri günlüğünden daha popülerdir. Örneğin, Windows NTFS ve SGI'nın XFS'sinin her ikisi de bir tür meta veri günlüğü kullanır. Linux ext3 size veri, sıralı veya sırasız modları seçme seçeneği sunar (sıralanmamış modda, veriler istediğiniz zaman yazılabilir). Tüm bu modlar meta verileri tutarlı tutar; veri için semantiklerinde farklılık gösterirler.

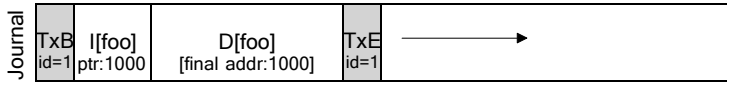
Son olarak, dergiye yazmaları (Adım 2) vermeden önce veri yazmayı tamamlamaya zorlamanın (Adım 1), yukarıdaki protokolde belirtildiği gibi doğruluk için gerekli olmadığını unutmayın. Özellikle, aynı anda verilere, işlem başlangıç bloğuna ve günlüğe kaydedilen meta verilere yazma işlemleri yayınlamak iyi olur; tek gerçek gereklilik, dergi taahhüt bloğunun yayınlanmasından önce Adım 1 ve 2'nin tamamlanmasıdır (Adım 3).

Zor Durum: Yeniden Kullanımı Engelle

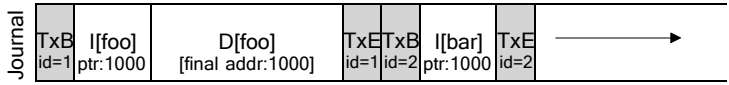
Günlük tutmayı daha zor hale getiren bazı ilginç köşe vakaları vardır ve bu nedenle tartışmaya değer. Bunların bir kısmı blok yeniden kullanımı etrafında döner; Stephen Tweedie'nin (ext3'ün arkasındaki ana güçlerden biri) dediği gibi:

> "Tüm sistemin kısmı nedir? ... Dosyaları silmektir. Sil ile ilgili her şey tüylü. Sil ile ilgili her şey... bloklar silinip sonra yeniden tahsis edilirse ne olacağı konusunda kabuslar görüyorsunuz." [T00]

Tweedie'nin verdiği özel örnek aşağıdaki gibidir. Bir tür meta veri günlüğü kullandığınızı varsayalım (ve bu nedenle dosyalar için veri blokları günlüğe kaydedilmez). Foo adında bir dizininiz olduğunu varsayalım. Kullanıcı foo'ya bir girdi ekler (örneğin bir dosya oluşturarak) ve böylece foo'nun içeriği (dizinler meta veri olarak kabul edildiğinden) günlüğe yazılır; foo dizin verilerinin konumunun blok 1000 olduğunu varsayalım. Günlük bu nedenle şöyle bir şey içerir:



Bu noktada, kullanıcı dizindeki her şeyi ve di- rectory'nin kendisini siler ve 1000 numaralı bloğu yeniden kullanım için serbest bırakır. Son olarak, kullanıcı yeni bir dosya (örneğin çubuk) oluşturur ve bu da foo'ya ait olan aynı bloğu (1000) yeniden kullanır. Çubuğun inode'u, verileri gibi diske de bağlıdır; Bununla birlikte, meta veri günlüğü kullanımda olduğundan, günlüğe yalnızca çubuğun inode'unun bağlı olduğunu unutmayın; dosya çubuğundaki blok 1000'de yeni yazılan veriler günlüğe kaydedilmez.



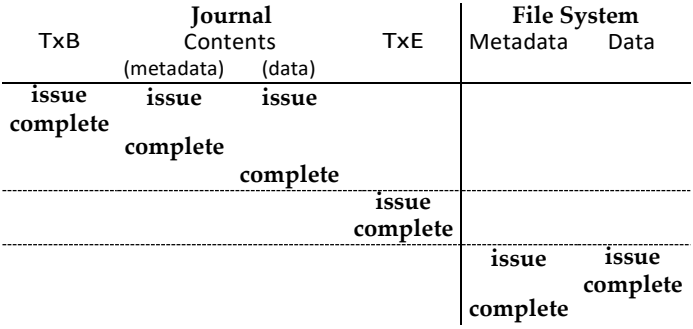


Figure 42.1: **Data Journaling Timeline**

Şimdi bir kilitlenme oluştuğunu ve tüm bu bilgilerin hala günlükte olduğunu varsayalım. Yeniden oynatma sırasında, kurtarma işlemi, blok 1000'deki dizin verilerinin yazılması da dahil olmak üzere günlükteki her şeyi yeniden yürütür; böylece tekrar, eski dizin içeriğiyle mevcut dosya çubuğunun kullanıcı verilerinin üzerine yazar! Açıkçası bu doğru bir kurtarma eylemi değildir ve kesinlikle dosya çubuğunu okurken kullanıcıya bir sür-prise olacaktır.

Bu sorunun bir dizi çözümü vardır. Birincisi, eski bol miktarda, söz konusu blokların silinmesi dergiden kontrol edilene kadar blokları asla tekrar kullanamaz. Linux ext3'ün bunun yerine yaptığı şey, günlüğe iptal kaydı olarak bilinen yeni bir kayıt türü eklemektir. Yukarıdaki durumda, dizinin silinmesi günlüğe bir iptal kaydının yazılmasına neden olur. Günlüğü tekrar oynatırken, sistem önce bu tür yeniden çağırma kayıtlarını tarar; Bu tür iptal edilen veriler asla tekrar oynatılmaz, böylece yukarıda belirtilen sorundan kaçınılır.

Günlük Kaydını Tamamlama: Bir Zaman Çizelgesi

Günlük tutma tartışmamıza son vermeden önce, tartıştığımız protokolleri her birini gösteren zaman çizelgeleriyle özetliyoruz. Şekil 42.1, verileri ve meta verileri günlüğe kaydederken protokolü gösterirken, Şekil 42.2 yalnızca meta verileri günlüğe kaydederken protokolü göstermektedir.

Her şekilde, zaman aşağı yönde artar ve şekildeki her satır, bir yazmanın verilebileceği veya tamamlanabileceği mantıksal zamanı gösterir. Örneğin, veri günlüğü protokolünde (Şekil 42.1), işlem başlangıç bloğunun (TxB) yazmaları ve trans-eylemin içeriği mantıksal olarak aynı anda verilebilir ve böylece herhangi bir sırayla tamamlanabilir; ancak, işlem bitiş bloğuna (TxE) yazma, söz konusu önceki yazılar tamamlanana kadar verilmemelidir. Benzer şekilde, denetim noktası verilere yazar ve meta veri blokları, aktarım eylemi uç bloğu işlenene kadar başlayamaz. Yatay kesikli çizgiler, yazma sırası gereksinimlerine nerede uyulması gerektiğini gösterir.

Meta veri günlüğü protokolü için benzer bir zaman çizelgesi gösterilir.

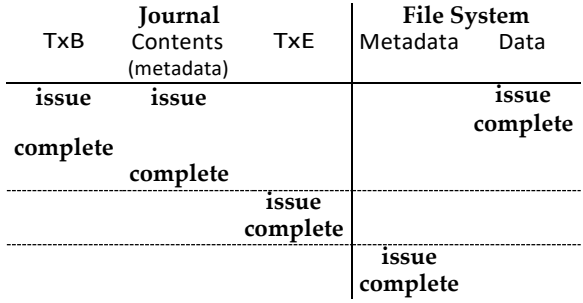


Figure 42.2: **Metadata Journaling Timeline**

Veri yazmanın, işleme yazılan yazılar ve derginin içeriği ile aynı anda mantıksal olarak verilebileceğini unutmayın; ancak, işlem sona ermeden önce düzenlenmeli ve tamamlanmalıdır.

Son olarak, zaman çizelgelerindeki her yazma için işaretlenen tamamlanma zamanının keyfi olduğunu unutmayın. Gerçek bir sistemde, tamamlanma süresi, performansı artırmak için yazmaları yeniden sıralayabilen G/Ç alt sistemi tarafından belirlenir. Sahip olduğumuz siparişle ilgili tek garanti, protokol doğruluğu için uygulanması gerekenlerdir (ve şekillerdeki yatay kesikli çizgilerle gösterilmiştir).

42.5 Çözüm #3: Diğer Yaklaşımlar

Şimdiye kadar dosya sistemi meta verilerini tutarlı tutmak için iki seçenek açıkladık: fsck'ye dayalı tembel bir yaklaşım ve günlük kaydı olarak bilinen daha etkin bir yaklaşım. Ancak, bunlar sadece iki yaklaşım değildir. Soft Updates [GP94] olarak bilinen böyle bir yaklaşım, Ganger ve Patt tarafından tanıtıldı. Bu yaklaşım, disk üzerindeki yapıların hiçbir zaman tutarsız bir çadır durumunda bırakılmamasını sağlamak için dosya sistemine tüm yazmaları dikkatli bir şekilde sıralar. Örneğin, kendisine işaret eden inode'dan önce diske işaret eden bir veri bloğu yazarak, inode'un asla çöpe işaret etmemesini sağlayabiliriz; dosya sisteminin tüm yapıları için benzer kurallar türetilebilir. Bununla birlikte, Yazılım Güncellemelerini uygulamak zor olabilir; Yukarıda açıklanan günlük kaydı katmanı, tam dosya sistemi yapıları hakkında nispeten az bilgi ile uygulanabilirken, Soft Updates her dosya sistemi veri yapısı hakkında karmaşık bilgi gerektirir ve bu nedenle sisteme adil miktarda karmaşıklık ekler.

> Başka bir yaklaşım yazma üzerine kopyalama (evet, COW) olarak bilinir ve Sun'ın ZFS'si [B07] de dahil olmak üzere bir dizi popüler dosya sisteminde kullanılır. Bu teknik bilgi hiçbir zaman yerinde dosya veya dizinlerin üzerine yazmaz; bunun yerine, diskte daha önce kullanılmayan konumlara yeni güncelleştirmeler yerleştirir. Bir dizi yükseltme tamamlandıktan sonra, COW dosya sistemleri yeni güncelleştirilen yapılara işaretçiler eklemek için dosya sisteminin kök

yapısını çevirir. Bunu yapmak, dosya sisteminin tutarlı kalmasını kolaylaştırır. Gelecekteki bir bölümde günlük yapılandırılmış dosya sistemini (LFS) tartışırken bu teknik hakkında daha fazla şey öğreneceğiz; LFS, bir İNEĞIN erken bir örneğidir.

Başka bir yaklaşım da Wisconsin'de yeni geliştirdiğimiz bir yaklaşımdır. Arka işaretçi tabanlı tutarlılık (veya BBC) başlıklı bu teknikte, yazmalar arasında sıralama yapılmaz. Tutarlılık sağlamak için, sistemdeki her bloğa ek bir geri işaretçi eklenir; örneğin, her veri bloğunun ait olduğu inode için bir referansı vardır. Bir dosyaya erişirken, dosya sistemi, ileri işaretçinin (örneğin, inode veya doğrudan bloktaki adres) kendisine geri başvuran bir bloğa işaret edip etmediğini kontrol ederek dosyanın tutarlı olup olmadığını belirleyebilir. Eğer öyleyse, her şey güvenli bir şekilde diske ulaşmış olmalı ve böylece dosya tutarlıdır; değilse, dosya tutarsızdır ve bir hata döndürülür. Dosya sistemine geri işaretçiler ekleyerek, yeni bir tembel kilitlenme tutarlılığı biçimi elde edilebilir [C+12].

Son olarak, bir günlük protokolünün disk yazmalarının tamamlanmasını bekleme sayısını azaltmak için teknikleri de araştırdık. İyimser kilitlenme tutarlılığı [C+13] başlıklı bu yeni yaklaşım, işlem sağlama toplamının genelleştirilmiş bir biçimini [P+05] kullanarak diske mümkün olduğunca çok sayıda yazma işlemi yapar ve ortaya çıkmaları durumunda tutarsızlıkları tespit etmek için birkaç başka teknik içerir. Bazı iş yükleri için bu iyimser teknikler performansı büyüklük sırasına göre artırabilir. Ancak, gerçekten iyi çalışması için, biraz farklı bir disk arayüzü gereklidir [C + 13].

42.6 Özet

Kilitlenme tutarlılığı sorununu ortaya koyduk ve bu soruna saldırmak için çeşitli yaklaşımları tartıştık. Bir dosya sistemi denetleyicisi oluşturmanın eski yaklaşımı işe yarar, ancak modern sistemlerde kurtarılamayacak kadar yavaştır. Bu nedenle, birçok dosya sistemi artık günlük kaydı kullanmaktadır. Günlük tutma, kurtarma süresini O'dan (disk biriminin boyutu) O'ya (günlük boyutu) düşürür, böylece bir çökme ve yeniden başlatmadan sonra kurtarma işlemini önemli ölçüde hızlandırır. Bu nedenle, birçok modern dosya sistemi günlük kaydı kullanır. Günlük tutmanın birçok farklı biçimde gelebileceğini de gördük; en yaygın kullanılanı, hem dosya sistemi meta verileri hem de kullanıcı verileri için makul tutarlılık garantilerini korurken günlüğe gelen trafik miktarını azaltan sıralı meta veri günlüğüdür. Sonunda, kullanıcı verileri üzerinde güçlü garantiler muhtemelen sağlanması gereken en önemli şeylerden biridir; İşin garibi, son araştırmaların gösterdiği gibi, bu alan devam eden bir çalışma olmaya devam etmektedir [P + 14].

References

[B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Available online: `http://www.ostep.org/Citations/zfs_last.pdf`. *ZFS uses copy-on-write and journaling, actually, as in some cases, logging writes to disk will perform better.*

[C+12] "Consistency Without Ordering" by Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '12, San Jose, California. *A recent paper of ours about a new form of crash consistency based on back pointers. Read it for the exciting details!*

[C+13] "Optimistic Crash Consistency" by Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau . SOSP '13, Nemacolin Woodlands Resort, PA, November 2013. *Our work on a more optimistic and higher performance journaling protocol. For workloads that call `fsync()` a lot, performance can be greatly improved.*

[GP94] "Metadata Update Performance in File Systems" by Gregory R. Ganger and Yale N. Patt. OSDI '94. *A clever paper about using careful ordering of writes as the main way to achieve consistency. Implemented later in BSD-based systems.*

[G+08] "SQCK: A Declarative File System Checker" by Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. OSDI '08, San Diego, California. *Our own paper on a new and better way to build a file system checker using SQL queries. We also show some problems with the existing checker, finding numerous bugs and odd behaviors, a direct result of the complexity of `fsck`.*

[H87] "Reimplementing the Cedar File System Using Logging and Group Commit" by Robert Hagmann. SOSP '87, Austin, Texas, November 1987. *The first work (that we know of) that applied write-ahead logging (a.k.a. journaling) to a file system.*

[M+13] "ffsck: The Fast File System Checker" by Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '13, San Jose, California, February 2013. *A recent paper of ours detailing how to make fsck an order of magnitude faster. Some of the ideas have already been incorporated into the BSD file system checker [MK96] and are deployed today.*

[MK96] "Fsck – The UNIX File System Check Program" by Marshall Kirk McKusick and T. J. Kowalski. Revised in 1996. *Describes the first comprehensive file-system checking tool, the eponymous `fsck`. Written by some of the same people who brought you FFS.*

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM Transactions on Computing Systems, Volume 2:3, August 1984. *You already know enough about FFS, right? But come on, it is OK to re-reference important papers.*

[P+14] "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications" by Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. OSDI '14, Broomfield, Colorado, October 2014. *A paper in which we study what file systems guarantee after crashes, and show that applications expect something different, leading to all sorts of interesting problems.*

[P+05] "IRON File Systems" by Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. SOSP '05, Brighton, England, October 2005. *A paper mostly focused on studying how file systems react to disk failures. Towards the end, we introduce a transaction checksum to speed up logging, which was eventually adopted into Linux ext4.*

[PAA05] "Analysis and Evolution of Journaling File Systems" by Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. USENIX '05, Anaheim, California, April 2005. *An early paper we wrote analyzing how journaling file systems work.*

[R+11] "Coerced Cache Eviction and Discreet-Mode Journaling" by Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. DSN '11, Hong Kong, China, June 2011. *Our own paper on the problem of disks that buffer writes in a memory cache instead of forcing them to disk, even when explicitly told not to do that! Our solution to overcome this problem: if you want **A** to be written to disk before **B**, first write **A**, then send a lot of "dummy" writes to disk, hopefully causing **A** to be forced to disk to make room for them in the cache. A neat if impractical solution.*

[T98] "Journaling the Linux ext2fs File System" by Stephen C. Tweedie. The Fourth Annual Linux Expo, May 1998. *Tweedie did much of the heavy lifting in adding journaling to the Linux ext2 file system; the result, not surprisingly, is called ext3. Some nice design decisions include the strong focus on backwards compatibility, e.g., you can just add a journaling file to an existing ext2 file system and then mount it as an ext3 file system.*

[T00] "EXT3, Journaling Filesystem" by Stephen Tweedie. Talk at the Ottawa Linux Symposium, July 2000. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html *A transcript of a talk given by Tweedie on ext3.*

[T01] "The Linux ext2 File System" by Theodore Ts'o, June, 2001.. Available online here: `http://e2fsprogs.sourceforge.net/ext2.html`. *A simple Linux file system based on the ideas found in FFS. For a while it was quite heavily used; now it is really just in the kernel as an example of a simple file system.*

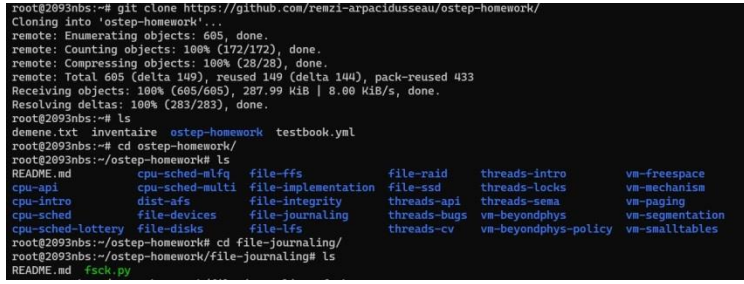
Ödev (Simülasyon)

Bu bölümde, dosya sistemi bozulmalarının nasıl tespit edilebileceğini (ve sürekli olarak onarılabileceğini) daha iyi anlamak için kullanabileceğiniz basit bir simülatör olan fsck.py tanıtılmaktadır. Simülatörün nasıl çalıştırılacağı hakkında ayrıntılar için lütfen ilgili README'ye bakın.

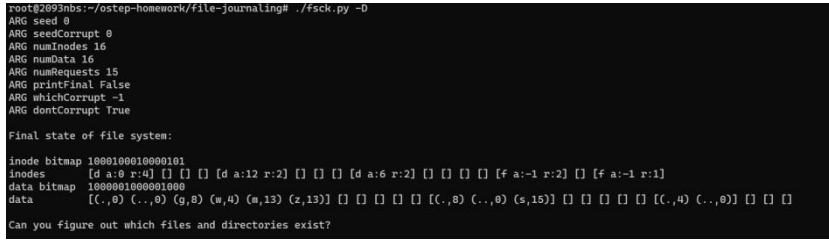
Questions

1. İlk olarak, fsck.py -D komutunu çalıştırın; Bu bayrak herhangi bir bozulmayı kapatır ve böylece rastgele bir dosya sistemi oluşturmak için kullanabilir ve orada hangi dosyaların ve dizinlerin bulunduğunu belirleyip belirleyemeyeceğinizi görebilirsiniz. Öyleyse, devam edin ve bunu yapın! Haklı olup olmadığınızı görmek için -p bayrağını kullanın. Çekirdek (-s) değerini 1, 2 ve 3 gibi farklı değerlere ayarlayarak rastgele oluşturulmuş birkaç farklı dosya sistemi için bunu deneyin.

Projeyi klon yapıp cd ile kendi ödevime gittim



-D komutunu çalıştırdık



-p bayrağını kullandık

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -p
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal True
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:7 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?


Summary of files, directories::
  Files:       ['/m', '/z', '/g/s']
  Directories: ['/', '/g', '/w']
```

-S'e 1,2,3 gibi farklı değerler verdik

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 1
ARG seed 0
ARG seedCorrupt 1
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000001
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 2
ARG seed 0
ARG seedCorrupt 2
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [f a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 3
ARG seed 0
ARG seedCorrupt 3
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:2]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

2. Şimdi, bir yolsuzluğun ortaya koyalım. Başlamak için fsck.py -S 1 komutunu çalıştırın. Hangi tutarsızlığın ortaya çıktığını görebiliyor musunuz? Gerçek bir dosya sistemi onarım aracında nasıl düzeltirsiniz? Haklı olup olmadığınızı kontrol etmek için -c kullanın.

İnode bitmap ve iode verileri bit 13'te tutarsız

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 1
ARG seed 0
ARG seedCorrupt 1
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000001
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -c
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Initial state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

CORRUPTION::INODE 8 points to dead block 7

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:7 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

3. Tohumu -S 3 veya -S 19 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? Cevabınızı kontrol etmek için -c komutunu kullanın. Bu iki durumda farklı olan nedir?

-s 3:inodes bloğu dosyanın /m iki referansı olduğunu söylüyor ancak very bloğu bunun bir olduğunu söylüyor
-s 19:dizinin /g iki dizinini var ama inode sadece bir referansı olduğunu gösteriyor
İki durum dosyaya bir bağlantı oluşturuken olabilir /m ancak dizinin ikiden fazla referası olabilir

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 19
ARG seed 0
ARG seedCorrupt 19
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:1] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -c
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Initial state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

CORRUPTION::INODE 8 points to dead block 7

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:7 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

4. Tohumu -S 5 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? Bu sorunu otomatik bir şekilde çözmek ne kadar zor olurdu? Cevabınızı kontrol etmek için -c komutunu kullanın. Ardından, -S 38 ile benzer bir tutarsızlık getirin; bunu tespit etmek daha zor/mümkün mü? Son olarak, -S 642; bu tutarsızlık tespit edilebilir mi? Eğer öyleyse, dosya sistemini nasıl düzeltirsiniz?

5. Tohumu -S 6 veya -S 13 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? Cevabınızı kontrol etmek için -c komutunu kullanın. Bu iki durum arasındaki fark nedir? Böyle bir durumla karşılaştığınızda onarım aracı ne yapmalıdır?

   -s 16 :12.inode un herhangi bir referansı yok
   -s 13:10. inode un herhangi bir referansı yok
   Birincisi bir dizin ,ikincisi klasördür onları silerim

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 6
ARG seed 0
ARG seedCorrupt 6
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [d a:-1 r:1] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -c
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Initial state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

CORRUPTION::INODE 8 points to dead block 7

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:7 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

6. Tohumu -S 9 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? Cevabınızı kontrol etmek için -c komutunu kullanın. Bu durumda bir kontrol ve onarım aracı hangi bilgi parçasına güvenmelidir?

Dosya /m bir dizinine değiştirildi very blokları

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 9
ARG seed 0
ARG seedCorrupt 9
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [d a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -c
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Initial state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

CORRUPTION::INODE 8 points to dead block 7

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:7 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

7. Tohumu -S 15 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? Cevabınızı kontrol etmek için -c komutunu kullanın. Bu durumda bir onarım aracı ne yapabilir? Onarım mümkün değilse, ne kadar veri kaybolur

/dosya olarak değiştirildi inodeu dizin türüne değiştirin tüm veriler kaybolacaktır

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 15
ARG seed 0
ARG seedCorrupt 15
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [f a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -c
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Initial state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

CORRUPTION::INODE 8 points to dead block 7

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:7 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

8. Tohumu -S 10 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? Cevabınızı kontrol etmek için -c komutunu kullanın. Buradaki dosya sistemi yapısında onarıma yardımcı olabilecek artıklık var mı?
   /w var olmayan bir klasöre taşındı /w hangi klasörün inode'u onardığını bulmak için very bloklarını control edin

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 10
ARG seed 0
ARG seedCorrupt 10
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,3)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -c
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Initial state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

CORRUPTION::INODE 8 points to dead block 7

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:7 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

9. Tohumu -S 16 ve -S 20 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? Cevabınızı kontrol etmek için -c komutunu kullanın. Onarım aracı sorunu nasıl çözmelidir?

   -s 16:Dosyanın very bloğu /m boş bir blok olarak değiştirldi dosyayı silin

   -s 20: Dizininin very bloğu /g boş bir blok olarak değiştirildi very bloklarını arayın hangisinin bir dizin olabileceğini ve işaret edildiğini bulun /g için

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 16
ARG seed 0
ARG seedCorrupt 16
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:7 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -S 20
ARG seed 0
ARG seedCorrupt 20
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:11 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

Can you figure out how the file system was corrupted?
```

```
root@2093nbs:~/ostep-homework/file-journaling# ./fsck.py -c
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt False

Initial state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:6 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []

CORRUPTION::INODE 8 points to dead block 7

Final state of file system:

inode bitmap 1000100010000101
inodes       [d a:0 r:4] [] [] [] [d a:12 r:2] [] [] [] [d a:7 r:2] [] [] [] [] [f a:-1 r:2] [] [f a:-1 r:1]
data bitmap  1000001000001000
data         [(.,0) (..,0) (g,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```