42

Arıza Tutarlılığı: FSCK ve günlük kaydı (Crash Consistency: FSCK and Journaling)

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 **kalıcı(persist)** olmalıdır, yani güç kaybına rağmen verileri tutan cihazlarda (sabit diskler veya flash tabanlı SSD'ler gibi) saklanarak uzun süre hayatta kalmalıdırlar.

Bir dosya sisteminin karşılaştığı en büyük zorluklardan biri**, güç kaybı (power loss)** veya sistem **çökmesine (Crash)** rağmen kalıcı veri yapılarının nasıl güncelleneceğidir. Spesifik olarak, disk üzerindeki yapıları güncellemenin tam ortasında birisi güç kablosuna takılırsa ve makine güç kaybederse ne olur? Veya işletim sistemi bir hatayla karşılaşır ve çöker mi? Güç kayıpları ve çökmeler nedeniyle, kalıcı bir veri yapısını güncellemek oldukça zor olabilir ve dosya sistemi uygulamasında **kilitlenme tutarlılığı (crash-consistency problems)** sorunu olarak bilinen yeni ve ilginç bir soruna yol açar.

Bu sorunun anlalıması oldukça basittir. Belirli bir işlemi tamamlamak için disk üzerindeki iki yapıyı, A ve B'yi güncellemeniz gerektiğini hayal edin. Disk aynı anda yalnızca tek bir isteğe hizmet verdiğinden, A veya B isteklerinden biri diske önce ulaşır. Bir yazma tamamlandıktan sonra sistem çökerse veya güç kaybederse, disk üzerindeki yapı **tutarsız(inconsistent)** bir durumda kalacaktır. Bu nedenle, tüm dosya sistemlerinin çözmesi gereken bir sorunumuz var.

Sistem çökmesine rağmen güncelleme nasıl yapılır

Sistem herhangi iki yazma işlemi arasında çökebilir veya güç kaybedebilir ve bu nedenle diskteki durum yalnızca kısmen güncellenebilir. Çökmeden sonra sistem, dosya sistemini tekrar monte etmek ve ön yüklemesini yapmak ister (dosyalara erişmek için). Çökmelerin herhangi bir zamanda meydana gelebileceği göz önüne alındığında, dosya sisteminin disk üzerindeki görüntüyü makul bir durumda tutmasını nasıl sağlayabiliriz?

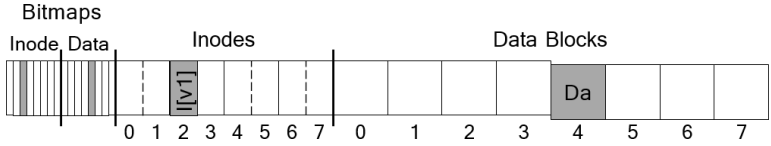
c

Bu bölümde, bu sorunu daha ayrıntılı olarak açıklayacağız ve dosya sistemlerinin bunu aşmak için kullandığı bazı yöntemler inceleyeceğiz. Konuya **FSCK** veya **dosya sistemi denetleyicisi (File system checker)** olarak bilinen eski dosya sistemlerinin yaklaşımını inceleyerek başlayacağız. Sonradan dikkatimizi **günlük kaydı (journaling)** olarak bilinen başka bir konuya çevireceğiz (**önceden yazma günlüğü(write-ahead logging)** olarak da bilinir). Her yazma işlemine biraz ek yük ekleyen ancak çökmelerden veya güç kayıplarından daha hızlı kurtulan bir tekniktir. Linux ext3'ün uyguladığı birkaç farklı günlük kaydı türü de dahil olmak üzere günlük kaydının temel mekanizmasını tartışacağız (modern bir günlük kaydı dosya sistemi gibidir).

42.1- Detaylı bir örnek

Günlük kaydı araştırmamızı başlatmak için bir örneğe bakalım. Disk yapılarını bir şekilde güncelleyen bir **iş yükü (workload)** kullanmamız gerekecek. Burada iş yükünün basit olduğunu varsayın: mevcut bir dosyaya bir veri bloğu eklenmesi. Ekleme, dosyanın açılması, dosya uzaklığını dosyanın sonuna taşımak için lseek() öğesinin çağrılması ve ardından dosyayı kapatmadan önce dosyaya tek bir 4KB yazma işlemi yapılmasıyla gerçekleştirilir.

Ayrıca, daha önce gördüğümüz dosya sistemlerine benzer şekilde, disk üzerinde standart basit dosya sistemi yapılarını kullandığımızı varsayalım. Bu küçük örnek, bir **inode bit eşlemi(inode bitmap)** içerir (inode başına yalnızca 8 bit). Bir **veri bitmap(data bitmap)** (veri bloğu(data block) başına yalnızca 8 bit), düğümler (toplam 8, 0 ila 7 arasında numaralandırılmış ve dört bloğa yayılmış) ve veri blokları (toplam 8, 0 ila 7 arasında numaralandırılmış). İşte bu dosya sisteminin bir diyagramı:



Yukarıdaki resimde de görüldüğü gibi inode bitmap yapısında tek bir inode tahsis edilmiştir(inode numarası 2) ve veri bitmap'inde işaretlenen tek bir tahsis edilmiş veri bloğu (veri bloğu 4). İnode, bu inode' un ilk versiyonu olduğu için I[v1] olarak gösterilir, yakında güncellenecektir (yukarıda açıklanan iş yükü nedeniyle)

Bu basitleştirilmiş düğümün içine de bir göz atalım (I[v1]):

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

Bu basitleştirilmiş düğümde, dosyanın boyutu 1'dir (tahsis edilmiş bir blok), ilk önce doğrudan işaretçi blok 4'ü gösterir (dosyanın ilk veri bloğu, Da) ve diğer 3 pointer, null olarak atanır(kullanılmadıklarını gösterir). Elbette gerçek düğümlerin daha birçok alanıvardır; daha fazla bilgi için önceki bölümlere bakın.

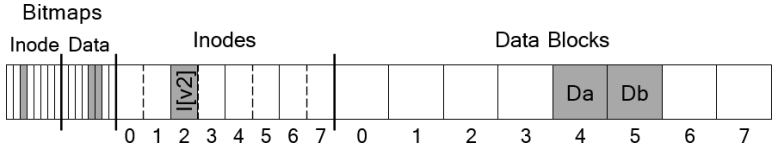
Bir dosyaya ekleme yaptığımızda, ona yeni bir veri bloğu ekliyoruz ve bu nedenle disk üzerindeki üç yerde güncellememiz gerekiyor: Düğüm (yeni bloğu işaret etmeli veekleme nedeniyle yeni ve daha büyük boyutu kaydetmeli), yeni veri bloğu Db ve yeni veri bloğunun tahsis edildiğini belirtmek için veri bit eşleminin yeni bir sürümü (diğer adı b[v2]).

Böylece sistemin hafızasında diske yazmamız gereken üç blok vardır. Güncelleneninode (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 bit eşlemi (B[v2]) şöyledir: 00001100. Son olarak, kullanıcıların dosyalara koyduklarıyla dolu olan veri bloğu (Db) vardır. Belki de çalıntı müzik vardır içinde?

Dosya sisteminin disk üzerindeki son görüntüsünün şöyle görünmesini istiyoruz:



Bu değişimi kaydetmek için, dosya sistemi disk üzerine üç farklı yazma işlemi gerçekleştirmesi gerekir. İnode bölümüne(I[2]), bitmap bölümüne (B[v2]) ve veri bloğuna (Db). Bu yazma işlemlerinin genellikle, kullanıcı bir write() sistem çağrısı yaptığında hemen gerçekleşmediğine dikkat edin; bunun yerine, kirli giriş, bit eşlem ve yeni veriler bir süre önce ana bellekte (**sayfa önbelleğinde(page cache)** veya **arabellek önbelleğinde(buffer cache))** kalacaktır. Daha sonra, dosya sistemi nihayet bunları diske yazmaya karar verdiğinde (5 veya 30 saniye sonra), dosya sistemi gerekli yazma isteklerini diske gönderir. Ne yazık ki, bir çökme meydana gelebilir, bu nedenle diskteki bu güncellemelere müdahale edilebilir. Özellikle, bu yazmalardan biri veya ikisi gerçekleştikten sonra bir çökme olursa, dosya sistemi komik bir durumda kalabilir.

c

Çökme Senaryoları

Problemi daha iyi anlamak için bazı çökme senaryolarına bakalım. Sadece tek bir yazmanın başarılı olduğunu hayal edin; dolayısıyla burada listelediğimiz üç olası sonuçvardır:

- **Sadece veri bloğu (Db) diske yazılır ise**: Bu durumda, verilerdisktedir, ancak ona işaret eden bir düğüm ve hatta bloğun tahsisedildiğini söyleyen hiçbir bitmap yoktur. Böylece yazma işlemi hiç gerçekleşmemiş gibidir. Bu durum sistem çökme tutarlılığı perspektifinden bakılır ise bir sorun teşkil etmez[1].
- **Yalnızca güncellenen düğüm (Inode-I[v2]) diske yazılır ise**: Bu durumda düğüm, Db'nin yazılmak üzere olduğu, ancak Db'nin henüz orada yazılmamış olduğu disk adresini(5) gösterir. Böylece, bu işaretçiye güvenirsek, diskten **çöp (garbage)** verileri okuyacağız (disk adresi 5'in eski içeriği).

Ayrıca, **dosya sistemi tutarsızlığı (a file-system in- consistency)** dediğimiz yeni bir sorunumuz var. Disk üzerindeki bitmap bize veri bloğu 5'in tahsis edilmediğini söylüyor, ancak inode tahsis edildiğini söyler. Bitmap ve düğüm arasındaki uyuşmazlık, dosya sisteminin veri yapılarında bir tutarsızlığa yol açar; dosya sistemini kullanmak için bu sorunu bir şekilde çözmeliyiz (daha fazlası aşağıdaki yazılarda).

- **Sadece güncellenmiş bitmap (B[v2]) diske yazılır ise:** Bu durumda, bitmap, blok 5'in tahsis edildiğini gösterir ama buna işaret eden bir düğüm yoktur. Böylece dosya sistemi tekrar tutarsız hale gelir. Bu sorun çözülmeden bırakılır ise, blok 5 dosya sistemi tarafından asla kullanılmayacağından, bu yazma **alan sızıntısına (space leak)** neden olur.

Diske üç blok yazma girişiminde ayrıca üç kilitlenme senaryosu daha vardır. Budurumlarda, iki yazma başarılı ve sonuncusu başarısız olur:

- **İnode (I[v2]) ve bitmap (B[v2]) diske yazılır, ancak veri (Db) yazılmaz:** Bu durumda, dosya sistemi meta verileri tamamen tutarlıdır: düğümde blok 5 için bir işaretçi vardır, bitmap 5'in kullanımda olduğunu gösterir ve bu nedenle dosya sisteminin meta verileri açısından her şey yolunda görünür. Ama bir sorun var: 5'in içinde yine çöp var.

1:Ancak, bazı verilerini kaybetmiş olan kullanıcı için bir sorun olabilir!

c

- **İnode (I[v2]) ve veri bloğu (Db) yazılır, ancak bitmap (B[v2]) yazılmaz:** Bu durumda, inode diskteki doğru verilere işaret ediyor, ancak yine inode ile bitmap'in eski sürümü (B1) 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, . inode ve veri bit eşlemi arasında bir tutarsızlık var. Ancak blok olmasına rağmen yazıldı ve bit eşlem kullanımını gösteriyor, hiçbir inode dosyayı işaret etmediği için hangi dosyaya ait olduğu hakkında hiçbir fikrimiz yok.

Çökme(Crash) Tutarlılığı Sorunu

Umarım, bu çökme senaryolarından, çökmeler nedeniyle diskteki dosya sistemi görüntümüzde oluşabilecek birçok sorunu görebilirsiniz. Dosya sistemi veri yapılarında tutarsızlık olabilir; alan sızıntıları yaşayabiliriz, bir kullanıcıya çöp verileri döndürebiliriz ve benzeri sorunlar karşımıza çıkabilir. İdeal olarak yapmak istediğimiz şey, dosya sistemini bir tutarlı durumdan (örneğin, dosya eklenmeden önce) diğerine **atomik olarak (atomically)** (örneğin, inode, bitmap ve yeni veri bloğu diske yazıldıktan sonra) taşımaktır. Ne yazık ki bunu kolayca yapamıyoruz çünkü disk bir seferde yalnızca bir yazma işlemi yapıyor ve bu güncellemeler arasında çökmeler veya güç kaybı meydana gelebilir. Bu genel soruna **çökme tutarlılığı sorunu (crash-consistency problem)** diyoruz (buna **tutarlı güncelleme (consistent-update problem)** sorunu da diyebiliriz).

42.1 Çözüm #1: Dosya Sistemi Denetleyicisi

Erken dosya sistemleri, çökme tutarlılığı için basit bir yaklaşım benimsemişlerdi. Temel olarak, tutarsızlıkların olmasına izin vermeye ve daha sonra (yeniden başlatırken) düzeltmeye karar verirlerdi. Bu tembel yaklaşımın klasik bir örneği, bunu yapan bir araçta bulunur: FSCK[2]. FSCK, bu tür tutarsızlıkları bulmak ve onarmak için 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 düğümün çöp verilere işaret ettiği yukarıdaki durumu düşünün. Tek asıl amaç, dosya sistemi meta verilerinin dahili olarak tutarlı olduğundan emin olmaktır.

McKusick ve Kowalski'nin makalesinde [MK96] özetlendiği gibi FSCK aracı birkaç aşamada çalışır. Dosya sistemi kurulmadan ve kullanıma sunulmadan önce çalıştırılır (fsck, çalışırken başka hiçbir dosya sistemi etkinliğinin devam etmediğini varsayar); bittiğinde, diskteki dosya sistemi tutarlı olmalı ve böylece kullanıcılar tarafından erişilebilir hale getirilmelidir.

İşte FSCK aracının ne yaptığının temel bir özeti:

- **Süper blok (Superblock):** FSCK, önce süper bloğun makul görünüp görünmediğini kontrol eder, çoğunlukla dosya sistemi boyutunun tahsis edilen blok sayısından daha büyük olduğundan emin olmak gibi mantıklı kontrolleri yapar. Genellikle bu mantık içeren kontrollerin amacı şüpheli (yozlaşmış) bir süper blok bulmaktır; bu durumda, sistem (veya yönetici) süper bloğun alternatif bir kopyasını kullanmaya karar verebilir.
- **Özgür bloklar:** Ardından FSCK, o anlık erişilebilen tahsis edilmiş blokların olduğunu anlamak için düğümleri, dolaylı blokları, çift dolaylı blokları vb. yapıları tarar. Bu bilgiyi, ayırma bit eşlemlerinin doğru bir sürümünü üretmek için kullanır; bu nedenle, bitmap' ler ve düğümler arasında herhangi bir tutarsızlık varsa, düğümlerdeki bilgilere güvenilerek çözülür. Tüm düğümler için aynı tip tip kontrolü yapılır ve kullanımdaymış gibi görünen tüm düğümlerin inode bitmaplerinde bu şekilde işaretlendiğinden emin olunur.
- **Düğüm durumu(inode state):** Her düğüm, bozulma veya diğer sorunlar açısından kontrol edilir. Örneğin, fsck, tahsis edilen her düğümün geçerli bir tür alanına (örneğin, normal dosya, dizin, sembolik bağlantı, vb.) sahip olmasını sağlar. İnode alanlarıyla ilgili kolayca düzeltilemeyen sorunlar varsa, inode şüpheli kabul edilir ve fsck tarafından temizlenir; inode bitmapi buna uygun olarak güncellenir.
- **İnode bağlantıları(inode links):** FSCK ayrıca tahsis edilen her düğümün bağlantı sayısını da doğrular. Hatırlayabileceğiniz gibi, bağlantı sayısı, bu belirli dosyaya bir referans (yani bir bağlantı) içeren farklı dizinlerin sayısını gösterir. FSCK bağlantı sayısını doğrulamak için, kök dizinden başlayarak tüm dizin ağacını tarar ve dosya sistemindeki her dosya ve dizin için kendi bağlantı sayılarını oluşturur. Yeni hesaplanan sayı ile bir düğümde bulunan sayı arasında bir uyumsuzluk varsa, genellikle sayıyı düğüm içinde sabitleyerek düzeltici önlem alınmalıdır. Tahsis edilmiş bir inode bulunursa, ancak hiçbir dizin ona başvuruda bulunmadıysa, kayıp + bulunan (lost + found)dizine taşınır.
- **Yinelenenler(Duplicates):** FSCK ayrıca yinelenen işaretçileri, yani şu durumlarda kontrol eder: iki farklı düğüm aynı bloğa atıfta bulunur. Bir düğüm bariz şekilde kötüyse, temizlenebilir. Alternatif olarak, işaret edilen blok kopyalanabilir, böylece her düğüme istendiği gibi kendi kopyası verilir.
- **Kötü bloklar(bad blocks):** Tüm işaretçiler listesinde tarama yapılırken, hatalı blok işaretçileri için de bir tarama gerçekleştirilir. Bir işaretçi, geçerli aralığının dışında bir şeye açıkça işaret ediyorsa "kötü" olarak kabul edilir. Örneğin, boyutundan daha büyük bir bloğa atıfta bulunan bir adresi vardır. Bu durumda, fsck çok akıllıca bir şey yapamaz; sadece işaretçiyi düğümdenveya dolaylı bloktan kaldırır (temizler).

2: "eff-ess-see-kay", "eff-ess-check" veya beğenmediyseniz "ef-sak" olarak telaffuz edilir. Evet, ciddi

c

- **Dizin kontrolleri(directory checks):** FSCK, kullanıcı dosyalarının içeriğini anlamaz; ancak, dizinler özel olarak biçimlendirilmiş bilgileri tutar dosya sisteminin kendisi tarafından oluşturulur. Böylece fsck, her dizinin içeriği üzerinde ek bütünlük kontrolleri gerçekleştirir ve "." olduğundan emin olur ve "..", bir dizin girişinde atıfta bulunulan her düğümün tahsis edildiği ve tüm hiyerarşide hiçbir dizinin bir kereden fazla bağlanmamasını sağlayan ilk girişlerdir.

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].Ancak FSCK (ve benzer yaklaşımlar) daha büyük ve belki de daha temel bir soruna sahiptir: Çok yavaşlar. Çok büyük bir disk hacmiyle, tahsis edilen tüm blokları bulmak ve tüm dizin ağacını okumak için tüm diski taramak dakikalar veya saatler sürebilir. Disklerin kapasitesi ve RAID'lerin popülaritesi arttıkça FSCK aracının performansı yasaklayıcı hale geldi (son gelişmelere [M+13] rağmen).

Daha yüksek bir düzeyde, FSCK aracının temel önermesi biraz mantıksız görünüyor. Diske sadece üç bloğun yazıldığı yukarıdaki örneğimizi düşünün; sadece üç blokluk bir güncelleme sırasında oluşan sorunları düzeltmek için tüm diski taramak inanılmaz derecede pahalıdır. Bu durum, yatak odanızda anahtarlarınızı yere düşürmeye benzer ve ardından bodrumdan başlayarak ve her odada yolunuza devam ederek, tüm evi anahtarlar için arama kurtarma algoritmasını başlatın. Çalışır ama zaman israftır. Böylece diskler (ve RAID'ler) büyüdükçe, araştırmacılar ve uygulayıcılar başka çözümler aramaya başladılar.

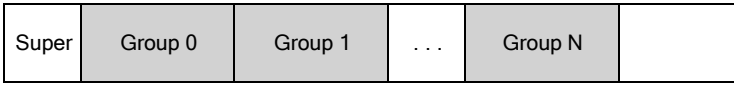
Çözüm #2: Günlük kaydı veya İleri Yazma Günlüğü (Journaling)

Tutarlı güncelleme sorununun muhtemelen en popüler çözümü, veritabanı yönetim sistemleri dünyasından bir fikir çalmaktır. **İleriye yazma günlüğü (write-ahead logging)** olarak bilinen bu fikir, tam olarak bu tür bir sorunu çözmek için icat edildi. Dosya sistemlerinde, tarihsel nedenlerle genellikle ileriye dönük **günlük kaydı (journaling)** olarak adlandırırız. Bunu yapan ilk dosya sistemi Cedar [H87] idi, ancak Linux ext3 ve ext4, reiserfs, IBM'in JFS'si, SGI'nin XFS'si ve Windows NTFS dahil olmak üzere birçok modern dosya sistemi bu fikri kullanıyor.

Basit fikir aşağıdakiler gibidir. Diski güncellerken, yapıların üzerine yazmadan önce, yapmak üzere olduğunuz şeyi açıklayan küçük bir not yazın (diskte başka bir yerde, iyi bilinen bir yerde). Bu notu yazmak "önceden yazmak" kısmıdır ve "log" olarak düzenlediğimiz bir yapıya yazıyoruz; bu nedenle, adı ileri yazma günlüğüdür. Notu diske yazarak, güncelleme yaptığınız yapıların güncelleme (üzerine yazma) sırasında bir çökme olursa, geri dönüp aldığınız nota bakıp tekrar deneyebileceksiniz; böylece, bir çökmeden sonra tüm diski taramak yerine tam olarak neyi düzelteceğinizi (ve nasıl düzelteceğinizi) bileceksiniz. Tasarım gereği, günlük kaydı, kurtarma sırasında gereken çalışma miktarını büyük ölçüde azaltmak için güncellemeler sırasında biraz iş ekler.

c

Şimdi popüler bir günlük kaydı dosya sistemi olan Linux ext3'ün nasıl çalıştığını açıklayacağız. Dosya sisteminde günlük kaydını içerir. Disk üzerindeki yapıların çoğu Linux ext2 ile aynıdır. Örneğin disk blok gruplarına bölünmüştür ve her blok grubu bir inode bitmap, veri bitmap, inode' lar ve veri blokları içerir. Yeni anahtar yapı, bölüm içinde veya başka bir aygıtta az miktarda yer kaplayan günlüğün kendisidir. Böylece, bir ext2 dosya sistemi (günlük kaydı olmadan) şöyle görünür:



Günlüğün aynı dosya sistemi görüntüsüne yerleştirildiğini varsayarsak (bazen ayrı bir aygıta veya dosya sistemi içinde bir dosya olarak yerleştirilse de), günlük içeren bir ext3 dosya sistemi şöyle görünür:

Asıl fark, sadece derginin varlığı ve tabiki nasıl kullanıldığıdır.

Veri Günlüğü

Veri günlüğü (data journaling) tutmanın nasıl çalıştığını anlamak için basit bir örneğe bakalım. Veri günlüğü tutma, bu tartışmanın çoğunun dayandığı Linux ext3 dosya sistemiyle bir mod olarak mevcuttur.

İnode (I[v2]), bitmap (B[v2]) ve veri bloğunu (Db) tekrar diske yazmak istediğimiz kanonik güncellememizi tekrar yaptığımızı varsayalım. Onları son disk konumlarına yazmadan önce, onları önce günlüğe (A.K.A günlüğü) yazacağız. Bu,günlük şöyle görünecek:

Burada beş blok yazdığımızı görebilirsiniz. İşlem başlangıcı (TxB), dosya sistemine yapılan bekleyen güncelleme hakkında bilgiler (örneğin, I[v2], B[v2] ve Db bloklarının son adresleri) ve bazı **işlem tanımlayıcısı (transaction identifier - TID)** türleri dahil olmak üzere bu güncelleme hakkında bilgi verir. Ortadaki üç blok sadece blokların tam içeriğini içerir. Bu, güncellemenin tam fiziksel içeriğini günlüğe koyduğumuz için **fiziksel günlüğe (physical logging)** kaydetme olarak bilinir (alternatif bir fikir**, mantıksal günlüğe (logical logging)** kaydetme, güncellemenin daha kompakt bir mantıksal gösterimini günlüğe koyar, örneğin, "bu güncelleme veri eklemek istiyor Db'yi X dosyasına engelle", bu biraz daha karmaşıktır ancak günlükte yerden tasarruf edebilir ve belki de performansı artırabilir). Son blok (TxE), bu işlemin sonunun bir işaretidir ve ayrıca TID' yi de içerir.

Bu işlem güvenli bir şekilde diskte olduğunda, dosya sistemindeki eski yapıların üzerine yazmaya hazırdır; bu işleme **kontrol noktası (checkpoint)** denir. Böylece, dosya sistemini **kontrol etmek** için (yani, onu dergide bekleyen güncelleme ile güncel hale getirmek), I[v2], B[v2] ve Db yazmalarını yukarıda görüldüğü gibi disk konumlarına göndeririz; bu yazma işlemleri başarıyla tamamlanırsa, dosya sistemini başarıyla kontrol eder ve temelde bitmiş olur. Böylece, ilk işlem sıramız:

1. **Günlük yazmak (Journal write):** Bir işlem başlangıcı bloğu, bekleyen tüm veriler ve meta veri güncellemeleri ve bir işlem bitiş bloğu dahil olmak üzere işlemi günlüğe yazın; bu yazma işlemlerinin tamamlanmasını bekleyin.
2. **Kontrol noktası (Checkpoint):** Bekleyen meta verileri ve veri güncellemelerini onların dosya sistemindeki son konumları yazılır. Örneğimizde, önce günlüğe TxB, I[v2], B[v2], Db ve TxE yazacaktık. Bu yazma işlemleri tamamlandığında, I[v2], B[v2] ve Db'yi diskteki son konumlarına kontrol ederek güncellemeyi tamamlarız.

Günlüğe yazılan yazılar sırasında bir çökme meydana geldiğinde işler biraz daha zorlaşıyor. Burada işlemdeki blok setini (örn. TxB, I[v2], B[v2], Db, TxE) diske yazmaya çalışıyoruz. Bunu yapmanın basit bir yolu, her birini birer birer yayınlamak, her birinin tamamlanmasını beklemek ve ardından bir sonrakini yayınlamaktır. Beş bloğun tümü aynı anda yazılır, çünkü bu, beş yazmayı tek bir sıralı yazmaya dönüştürür ve böylece daha hızlı olur.

Diske yazmayı zorlamak

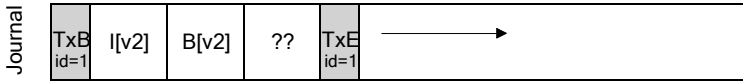
İki disk yazma işlemi arasında sıralamayı zorlamak için modern dosya sistemlerinin birkaç ekstra önlem alması gerekir. Eski zamanlarda, iki yazma, A ve B arasında sıralama yapmaya zorlamak kolaydı: sadece A'nın yazılmasını diske verin, yazma tamamlandığında diskin işletim sistemini kesmesini bekleyin ve ardından B'nin yazısınıyayınlayın.

Disklerdeki yazma önbelleklerinin artan kullanımı nedeniyle işler biraz daha karmaşık hale geldi. Yazma arabelleğe alma etkinleştirildiğinde (**anında raporlama (immediate reportling)** olarak da adlandırılır), bir disk, diskin bellek önbelleğine yerleştirildiğinde ve henüz diske ulaşmadığında, işletim sistemine yazmanın tamamlandığını bildirir. İşletim sistemi daha sonra bir yazma işlemi yaparsa, önceki yazmalardan sonra diske ulaşacağı garanti edilmez; bu nedenle yazmalar arasındaki sıralama korunmaz. Bir çözüm, yazma arabelleğini devre dışı bırakmaktır. Ancak, daha modern sistemler ekstra önlemler alır ve açık **yazma engelleri (write barries)** yayınlar; bariyer, bariyerden önce verilen tüm yazmaların, bariyerden sonra yayınlanan herhangi bir yazmadan önce diske ulaşmasını garanti eder.

Tüm bu makineler, diskin doğru çalışmasına büyük ölçüde güvenmeyi gerektirir. Ne yazık ki, son araştırmalar bazı disk üreticilerinin "daha yüksek performanslı" diskler sunma çabasıyla yazma engeli isteklerini açıkça görmezden geldiğini ve böylece disklerin görünüşte daha hızlı çalışmasını sağladığını ancak yanlış çalışma riski altında olduğunu gösteriyor [C+13, R +11]. Kahan'ın dediği gibi, hızlı, yanlış olsa bile, neredeyse her zaman yavaş

c

Ancak bu, aşağıdaki nedenden dolayı güvenli değildir: böyle büyük bir yazma işlemi yapıldığında, disk dahili olarak zamanlama yapabilir ve büyük yazmanın küçük parçalarını herhangi bir sırada tamamlayabilir. Böylece, disk dahili olarak (1) TxB, I[v2], B[v2] veTxE yazabilir ve ancak daha sonra (2) Db yazabilir. Ne yazık ki, disk (1) ve (2) arasında güçkaybederse, diskte şu olur:



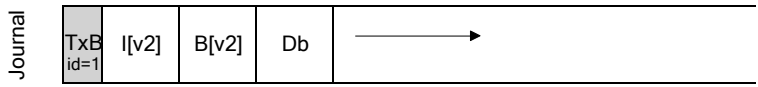
Bu niçin bir problemdir? İşlem, geçerli bir işlem gibi görünüyor (birbiriyle eşleşen sıra numaralarıyla bir başlangıcı ve bir sonu var). Ayrıca, dosya sistemi o dördüncü bloğa bakamaz ve bunun yanlış olduğunu bilemez; sonuçta, keyfi kullanıcı verileridir. Böylece, sistem şimdi yeniden başlatılır ve kurtarma çalıştırılırsa, bu işlemi yeniden yürütecek ve çöp bloğunun '??' içeriğini, Db'nin olması gereken konuma bilgisizce kopyalayacaktır. Bu, bir dosyadaki rastgele kullanıcı verileri için kötüdür; süper blok gibi dosya sistemini çözülemez hale getirebilecek kritik bir dosya sisteminin başına gelirse çokdaha kötüdür.

GÜNLÜK YAZIMLARININ OPTİMİZE EDİLMESİ

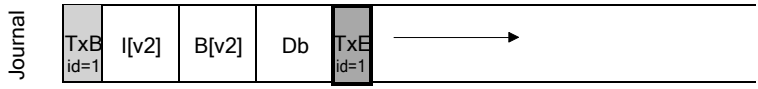
Günlüğe yazmanın belirli bir verimsizliğini fark etmiş olabilirsiniz. Yani, dosya sistemi önce işlem başlama bloğunu ve işlemin içeriğini yazmalıdır; ancak bu yazma işlemleri tamamlandıktan sonra dosya sistemi işlem sonu bloğunu diske gönderebilir. Bir diskin nasıl çalıştığını düşünüyorsanız, performans etkisi açıktır: genellikle fazladan bir dönüş yapılır (nedenini düşünün).

Eski lisansüstü öğrencilerimizden biri olan Vijayan Prabhakaran'ın bu sorunu çözmek için basit bir fikri vardı [P+05]. Günlüğe bir işlem yazarken, başlangıç ve bitiş bloklarına günlük içeriğinin bir sağlama toplamını eklememizi söylemişti. 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 çökme meydana geldiği sonucuna varabilir ve böylece dosya sistemi güncellemesini iptal edebilir. Böylece, yazma protokolünde ve kurtarma sisteminde küçük bir ince ayar ile bir dosya sistemi daha hızlı genel durum performansı elde edebilir; bunun da ötesinde, günlükten yapılan okumalar artık bir sağlamatoplamı ile korunduğundan, sistem biraz daha güvenilirdir. Bu basit düzeltme, Linux dosya sistemi geliştiricilerinin dikkatini çekecek kadar çekiciydi ve daha sonra onu (tahmin ettiniz!) **Linux ext4** olarak adlandırılan yeni nesil Linux dosya sistemine dahil ettiler. Artık Android platformu da dahil olmak üzere dünya çapında milyonlarca makinede vardır. Bu nedenle, 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 işlemsel yazmayı iki adımda gerçekleştirir. İlk olarak, TxE bloğu dışındaki tüm blokları günlüğe yazar ve bunların hepsini bir kerede yayınlar. Bu yazma işlemleri tamamlandığında, günlük şöyle görünecektir (ekleme iş yükümüzü tekrar varsayarsak):



Bu yazma işlemleri tamamlandığında, dosya sistemi TxE bloğunun yazılmasını yayınlar ve böylece günlük yazma sonlanır, güvenli durum aşağıdaki gibidir:



Bu sürecin önemli bir yönü, disk tarafından sağlanan atomsallık garantisidir. Görünüşe göre disk, herhangi bir 512 byte yazmanın gerçekleşeceğini veya gerçekleşmeyeceğini (ve asla yarım yazılmayacağını) garanti eder; bu nedenle, TxE'nin yazılmasının atomik olduğundan emin olmak için, onu tek bir 512 baytlık blok yapmalıdır. Bu nedenle, dosya sistemini güncellemek için mevcut protokolümüzün üç aşamasının her biri şu şekilde etiketlenmiştir:

1- **Günlük yazma (journal write):** İşlemin içeriğini (TxB, meta veriler ve verilerdahil) günlüğe yazın; bu yazıların tamamlanması beklenmelidir.
2- **Günlük işlemek (journal commit):** İşlem tamamlama bloğunu (TxE içeren) günlüğe yazın; yazmanın tamamlanmasını bekleyin, işlem, işlemenin yapıldığı söyleniyor.
3- **Kontrol noktası (checkpoint):** Güncellemenin içeriğini, disk üzerindeki sonkonumlarına yazın (meta veriler ve veriler).

Kurtarma (Recovery)

Şimdi bir dosya sisteminin bir çökmeden **kurtulmak (recovery)** için günlüğün içeriğini nasıl kullanabileceğini anlayalım. Bir güncelleme dizisi sırasında herhangi bir zamanda bir çökme meydana gelebilir. Çökme, işlem günlüğe güvenli bir şekilde yazılmadan önce gerçekleşirse (yani, yukarıdaki Adım 2 tamamlanmadan önce), o zaman işimiz kolaydır: bekleyen güncelleme basitçe atlanır. Çökme, işlem günlüğe kaydedildikten sonra, ancak kontrol noktası tamamlanmadan önce gerçekleşirse, dosya sistemi güncellemeyi aşağıdaki gibi **kurtarabilir (recover)**. Sistem ön yükleme yaptığında, dosya sistemi kurtarma işlemi günlüğünü tarar ve diske taahhüt edilen işlemleri arar; bu işlemler sırasıyla **tekrar oynatılır (replayed)**, dosya sistemi tekrar işlemdeki blokları disk üzerindeki son konumlarına yazmaya çalışır. Bu günlük kaydı biçimi, mevcut en basit biçimlerden biridir ve **yeniden günlük kaydı (redo logging)** olarak adlandırılır. Günlükte taahhüt edilen işlemleri kurtararak, dosya sistemi disk üzerindeki yapıların tutarlı olmasını sağlar ve böylece dosya sistemini kurarak ve yeni istekler için kendini hazırlayarak ilerletebilir.

Blokların son konumlarında yapılan bazı güncellemeler tamamlandıktan sonra bile, kontrol noktası sırasında herhangi bir noktada bir çökme yaşanmasının

c

normal olduğunu unutmayın. En kötü durumda, bu güncellemelerin bazıları kurtarma sırasında basitçe yeniden gerçekleştirilir.

Kurtarma nadir yapılan bir işlem olduğundan (yalnızca beklenmeyen bir sistem çökmesinden sonra yapılır), birkaç yedekli yazma işlemi için endişelenecek[3] bir şey yoktur.

Günlük Güncellemelerini Toplu İşleme (Batching Log Updates)

Temel protokolün çok fazla disk trafiği ekleyebileceğini fark etmiş olabilirsiniz. Örneğin,aynı dizinde dosya1 ve dosya2 adlı bir satırda iki dosya oluşturduğumuzu hayal edin. Bir dosya oluşturmak için, aşağıdakiler dahil olmak üzere bir dizi disk üzerindeki yapıyı güncellemek gerekir: inode bitmap (yeni bir inode tahsis etmek için), dosyanın yeni oluşturulan inode'u, yeni dizin girişini içeren üst dizinin veri bloğu ve (şimdi yeni bir değişiklik zamanı olan) üst dizin inode'u oluşturur. Günlük kaydıyla, iki dosya oluşturma işlemimizin her biri için tüm bu bilgileri mantıksal olarak günlüğe aktarırız; dosyalar aynı dizinde olduğundan ve aynı inode bloğu içinde düğümleri olduğunu varsayarsak, bu, dikkatli olmazsak, aynı blokları tekrar tekrar yazmak zorunda kalacağımız anlamına gelir.

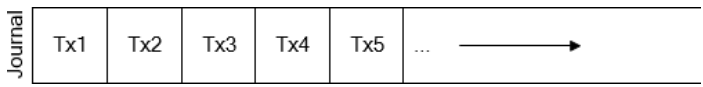
Bu sorunu çözmek için, bazı dosya sistemleri her güncellemeyi birer birer diske göndermez (örneğin, Linux ext3); bunun yerine, tüm güncellemeler global bir işlemde arabelleğe alınabilir. Yukarıdaki örneğimizde, iki dosya oluşturulduğunda, dosya sistemi sadece bellek içi inode bitmap' i, dosyaların inode'larını, Dizin verilerini ve dizin inode'u kirli olarak işaretler ve bunları erişilebilen bloklar listesine ekler.

Sonunda bu blokları diske yazma zamanı geldiğinde (örneğin, 5 saniye geçtikten sonra) bu tek global işlem yukarıda açıklanan tüm güncellemeleri taahhüt eder. Böylece, güncellemeleri arabelleğe alarak, diske aşırı yazma trafiğini önleyebilir.

Günlüğü Sonlu Yapmak (Making The Log Finite)

Disk üzerindeki dosya sistemi yapılarını güncellemek için temel bir protokole ulaştık. Dosya sistemi güncellemeleri bir süre için bellekte(memory) arabelleğe(buffer) alır; nihayet diske yazma zamanı geldiğinde, dosya sistemi önce işlemin ayrıntılarını dikkatlice günlüğe yazar (A.K.A. ileri yazma günlüğü); işlem tamamlandıktan sonra, dosya sistemi bu blokların diskteki sonkonumlarını kontrol eder.

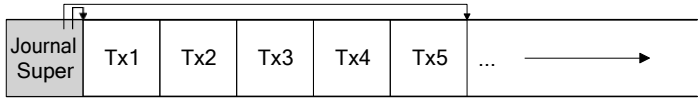
Ancak, günlük sonlu bir boyuttadır. Buna işlem eklemeye devam edersek (aşağıdaki şekilde olduğu gibi) yakında dolacak. Sence ne zaman olur?



Günlük dolduğunda iki sorun ortaya çıkar. Birincisi daha basittir ve daha az kritiktir: günlük ne kadar büyükse, kurtarma işlemi o kadar uzun sürer, çünkü kurtarma işleminin yapılabilmesi için günlük içindeki tüm işlemleri sırasıyla tekrar etmesi gerekir.

3: Her şey için endişelenmediğiniz sürece, bu durumda size yardımcı olamayız. Bu kadar endişelenmeyi bırak, bu sağlıksız! Ama şimdi muhtemelen aşırı endişelenmekten endişe ediyorsun.

İkincisi daha önemli bir sorundur: günlük dolduğunda (veya neredeyse dolu olduğunda), diske başka hiçbir işlem yapılamaz, bu da dosya sistemini "işe yaramaz" hale getirir.Bu sorunları çözmek için, günlük kaydı dosya sistemleri, günlüğü döngüsel bir veri yapısı olarak ele alır ve onu tekrar tekrar kullanır; bu nedenle günlüğe bazen **dairesel kayıt (circular log)** denir. Bunu yapmak için, dosya sistemi bir kontrol noktasından bir süre sonra harekete geçmelidir. Spesifik olarak, bir işlemin kontrol noktası yapıldıktan sonra, dosya sistemi günlük içinde kapladığı alanı boşaltmalı ve günlük alanının yeniden kullanılmasına izin vermelidir. Bu amaca ulaşmanın birçok yolu vardır; örneğin, günlükteki en eski ve en yeni kontrol noktası olmayan işlemleri bir **günlük süper bloğunda(journal superblock)** işaretleyebilirsiniz; diğer tüm alan ücretsizdir İşte bir grafik tasviri:



Günlük süper bloğunda (journal superblock) (ana dosya sistemi süper bloğu ile karıştırılmamalıdır), günlük kaydı sistemi, hangi işlemlerin henüz kontrol noktasına getirilmediğini bilmek için yeterli bilgiyi kaydeder ve böylece kurtarma süresini kısaltır ve günlüğün yeniden dairesel bir şekilde kullanılmasını sağlar. Ve böylece temel protokolümüze bir adım daha ekliyoruz:

1- **Günlük yazma (Journal write):** İşlemin içeriğini (TxB, meta veriler ve veriler dahil)günlüğe yazın; bu yazıların tamamlanması beklenmelidir.
2- **Günlük işlemek (Journal commit):** İşlem tamamlama bloğunu yazın (içeren TxE) günlüğe; yazmanın tamamlanmasını bekleyin, işlem şimdi taahhüt edilmiştir.
3- **Kontrol noktası (Checkpoint):** Güncellemenin içeriğini dosya sistemi içinde nihai konumlarına yazın.
4- **Özgür (Free):** Bir süre sonra, günlük süper bloğunu güncelleyerek işlemi günlükte serbest olarak işaretleyin.

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

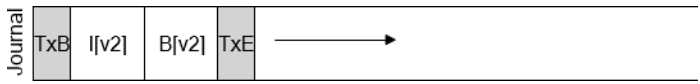
Meta Veri Günlüğü (metadata journaling)

Kurtarma işlemi artık hızlı olsa da (tüm diski taramak yerine günlüğü taramak ve birkaç işlemi yeniden yürütmek), dosya sisteminin normal çalışması istediğimizden daha yavaştır. Özellikle, her diske yazma işlemi için, artık önce günlüğe de yazıyoruz, böylece yazma trafiğini iki katına çıkarıyoruz; bu ikiye katlama, özellikle şimdi sürücünün en yüksek yazma bant genişliğinin yarısında devam edecek olan sıralı yazma iş yükleri sırasında acı vericidir.

c

Ayrıca, günlüğe yazma ve ana dosya sistemine yazma işlemleri arasında, bazı işyükleri için dikkate değer 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 denedi. Örneğin, yukarıda tanımladığımız günlük kaydı modu, tüm kullanıcı verilerini (dosya sisteminin meta verilerine ek olarak) günlüğe kaydettiği için genellikle **veri günlük kaydı(data journaling)** olarak adlandırılır (Linux ext3'te olduğu gibi). Daha basit (ve daha yaygın) bir günlük kaydı biçimine bazen **sıralı günlük kaydı(ordered journaling)** (veya yalnızca **meta veri günlük kaydı (metadata journaling)**) denir ve kullanıcı verilerinin günlüğe yazılmaması dışında neredeyse aynıdır. Böylece, yukarıdakiyle aynı güncellemeyi gerçekleştirirken, günlüğe aşağıdaki bilgiler yazılacaktır:



Daha önce günlüğe yazılan veri bloğu Db, bunun yerine uygun dosya sistemine yazılacak ve fazladan yazmadan kaçınılacaktır; diske gelen çoğu G/Ç trafiğinin veri olduğu göz önüne alındığında, verileri iki kez yazmamak, günlük kaydının G/Ç yükünü önemli ölçüde azaltır. Bu, ilginç bir soruyu gündeme getiriyor: Veri bloklarını diske ne zaman yazmalıyız?

Görünen o ki, veri yazmanın sırası, yalnızca meta veri günlük kaydı için önemlidir. Örneğin, işlem (I[v2] ve B[v2]'yi içeren) tamamlandıktan sonra Db'yi diske yazarsak ne olur? Ne yazık ki, bu yaklaşımın bir sorunu vardır: dosya sistemi tutarlıdır ama I[v2] çöp verilere işaret ediyor olabilir. Spesifik olarak, I[v2] ve B[v2]'nin yazıldığı ancak Db'nin diske girmediği durumu düşünün. Dosya sistemi, Db günlükte olmadığı için, sistemi daha sonra kurtarmaya çalışacaktır. Dosya sistemi I[v2] ve B[v2]'ye yazma işlemlerini yeniden yürütecek ve tutarlı bir dosya sistemi üretecektir (dosya sistemi meta verileri açısından). Bununla birlikte, I[v2], çöp verilere, yani Db'nin yöneldiği yuvada ne varsa onu işaret ediyor olacaktır.

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

1. **Veri yazma (Data write):** Verileri nihai konuma yazın; tamamlanmasını bekleyin(bekleme isteğe bağlıdır; ayrıntılar için aşağıya bakın).
2. **Günlük meta verileri yazma (Journal metadata write):** Başlangıç bloğunu ve metaverileri günlüğe yazın; yazmaların tamamlanmasını bekleyin.
3. **Günlük tamamlama (Journal commit):** İşlem tamamlama bloğunu (TxE içeren) günlüğe yazın; yazmanın tamamlanmasını bekleyin, işlem (veriler dahil) şimdi tamamlanmıştır.
4. **Kontrol noktası meta verileri:** Meta veri güncellemesinin içeriğini dosya sistemi içindeki son konumlarına yazın.
5. **Serbest (Free):** Daha sonra, günlük süper bloğunda işlemi ücretsiz olarak işaretleyin.

Önce verileri yazmaya zorlayarak, bir dosya sistemi bir işaretçinin hiçbir zaman çöpü işaret etmeyeceğini garanti edebilir. Gerçekten de "işaret edilen nesneyi, onu gösteren nesneden önce yaz" kuralı, çökme tutarlılığının merkezinde yer alır ve diğer çökme tutarlılığı şemaları [GP94] tarafından daha da fazla kullanılır (ayrıntılar için aşağıya bakın).

Çoğu sistemde, meta veri günlük kaydı (ext3'ün sıralı günlük kaydına benzer) tam veri günlük kaydından daha popülerdir. Örneğin, Windows NTFS ve SGI'nin XFS'si bir tür meta veri günlüğü kullanır. Linux ext3 size veri, sıralı veya sırasız modlardan birini seçme seçeneği sunar (sırasız modda veriler herhangi bir zamanda yazılabilir). Bu modların tümü, meta verileri tutarlı tutar; veri için anlam bilimlerinde farklılık gösterirler.

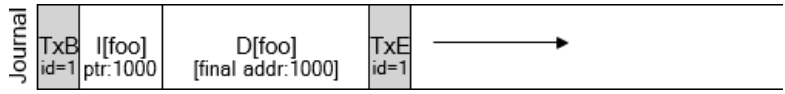
Son olarak, yukarıdaki protokolde belirtildiği gibi, dergiye yazı göndermeden önce (Adım 2) veri yazma işlemini tamamlamaya zorlamanın (1. Adım) doğruluk için gerekli olmadığını unutmayın. Spesifik olarak, verilere, işlem başlatma bloğuna ve günlüğe kaydedilmiş meta verilere aynı anda yazma yapmak iyi olur; tek gerçek gereksinim, günlük kesinleştirme bloğunun yayınlanmasından önce Adım 1 ve 2'nin tamamlanmasıdır (3.Adım).

Zor Durum: Yeniden Kullanımı Engelle (Tricky Case: Block Reuse)

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

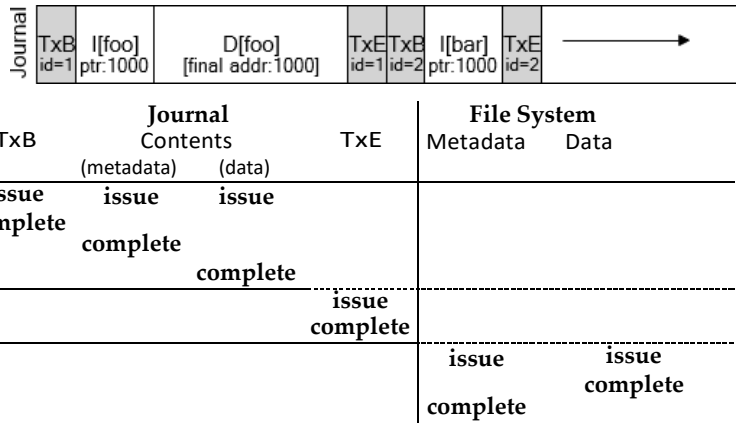
> "Bütün sistemin iğrenç kısmı nedir? ... Dosyaları siliyor. Silme ile ilgili her şey kıldır. Silme ile ilgili her şey... bloklar silinir ve sonra yeniden tahsis edilirse ne olacağı konusunda kabuslar görürsünüz." [T00]

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



Bu noktada, kullanıcı dizindeki her şeyi ve dizinin kendisini siler ve blok 1000'i yeniden kullanım için serbest bırakır. Son olarak, kullanıcı, foo'ya ait olan aynı bloğu (1000) yeniden kullanan yeni bir dosya (say çubuğu) oluşturur. Çubuğun (bar) düğümü, verileri gibi diske bağlıdır; ancak, meta veri günlük kaydı kullanımda olduğundan, yalnızca bar düğümünün günlüğe kaydedildiğini unutmayın; dosya çubuğunda blok 1000'e yeni yazılan veriler günlüğe kaydedilmez.

c



Figür: 42.1: **Veri Günlüğü Zaman Çizelgesi (Data Journaling Timeline)**

Şimdi bir çökme meydana geldiğini ve tüm bu bilgilerin hala günlükte olduğunu varsayalım. Yeniden yürütme sırasında, kurtarma işlemi, blok 1000'deki dizin verilerinin yazılması da dahil olmak üzere günlükteki her şeyi yeniden yürütür; böylece yeniden oynatma, eski dizin içerikleriyle mevcut dosya çubuğunun kullanıcı verilerinin üzerine yazar! Açıkça bu doğru bir kurtarma eylemi değildir ve dosya çubuğunu okurken kesinlikle kullanıcı için bir sürpriz olacaktır.

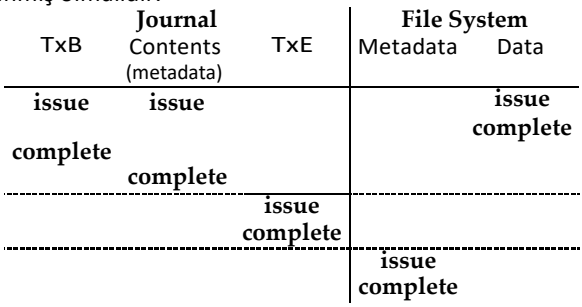
Bu sorunun bir dizi çözümü var. Örneğin, söz konusu blokların silinmesi günlükten kontrol noktası alınana kadar bloklar asla yeniden kullanılamaz. Bunun yerine Linux ext3'ün yaptığı, günlüğe **iptal kaydı (revoke)** olarak bilinen yeni bir kayıt türü eklemektir. Yukarıdaki durumda, dizinin silinmesi, dergiye bir iptal kaydının yazılmasına neden olacaktır. Günlüğü yeniden oynatırken, sistem önce bu tür iptal kayıtlarını tarar; bu tür iptal edilen veriler hiçbir zaman tekrar oynatılmaz, böylece yukarıda bahsedilen sorundan kaçınılır.

Günlük Toparlamak: Bir Zaman Çizelgesi (Wrapping Up Journaling: A Timeline)

Günlük kaydı tartışmamızı bitirmeden önce, tartıştığımız protokolleri her birini gösteren zaman çizelgeleriyle özetliyoruz. Şekil 42.1 verileri ve meta verileri günlüğe kaydetme protokolünü gösterir, Şekil 42.2 yalnızca meta verileri günlüğe kaydetme protokolünü gösterir.

Her şekilde, zaman aşağı yönde artar ve şekildeki her satır, bir yazmanın yapılabileceği veya tamamlanabileceği mantıksal zamanı gösterir. Örneğin, veri günlüğü protokolünde (Şekil 42.1), işlem başlangıç bloğu (TxB) ve işlemin içeriği mantıksal olarak aynı anda yayınlanabilir ve böylece herhangi bir sırayla tamamlanabilir; ancak, işlem bitiş bloğuna (TxE) yazma işlemi, söz konusu önceki yazma işlemleri tamamlanana kadar yapılmamalıdır. Benzer şekilde, kontrol noktası verilere yazar ve meta veri blokları, işlem bitiş bloğu taahhüt edilene kadar başlayamaz. Yatay kesikli çizgiler, yazma sırası gereksinimlerine uyulması gereken yerleri gösterir.

Meta veri günlük kaydı protokolü için benzer bir zaman çizelgesi gösterilir. Veri yazma işleminin mantıksal olarak işleme yazma işlemlerinin başlaması ve günlüğün içeriği ile aynı anda verilebileceğini unutmayın; ancak, işlem sonu düzenlenmeden önce düzenlenmiş ve tamamlanmış olmalıdır.



Figüre 42.2: **Meta Veri Günlüğü Zaman Çizelgesi**

Son olarak, zaman çizelgelerinde her yazma için işaretlenen tamamlama zamanının isteğe bağlı olduğunu unutmayın. Gerçek bir sistemde tamamlanma süresi, performansı artırmak için yazmaları yeniden sıralayabilen I/O (Giriş/Çıkış) alt sistemi tarafından belirlenir. Sahip olduğumuz siparişle ilgili garantiler, protokolün doğruluğu için uygulanması gereken garantilerdir (ve şekillerde yatay kesikli çizgilerle gösterilmiştir).

42.1 Çözüm #3: Diğer yaklaşımlar

Şimdiye kadar dosya sistemi meta verilerini tutarlı tutmak için iki seçenek tanımladık: fsck'e dayalı tembel bir yaklaşım ve günlük kaydı olarak bilinen daha aktif bir yaklaşım. Ancak, bunlar sadece iki yaklaşım değildir. Yumuşak Güncellemeler [GP94] olarak bilinen böyle bir yaklaşım, Ganger ve Patt tarafından tanıtıldı. Bu yaklaşım, disk üzerindeki yapıların asla tutarsız bir durumda bırakılmamasını sağlamak için dosya sistemine yapılan tüm yazma işlemlerini dikkatli bir şekilde düzenler. Örneğin, diske kendisine işaret eden inode'dan önce bir veri bloğu yazarak, inode'un hiçbir zaman çöpü işaret etmemesini sağlayabiliriz; dosya sisteminin tüm yapıları için benzer kurallar türetilebilir. Ancak Yazılım Güncellemelerini uygulamak zor olabilir; Yukarıda açıklanan günlük kaydı katmanı, tam dosya sistemi yapıları hakkında nispeten az bilgi ile uygulanabilirken, Yazılımsal Güncellemeler, her dosya sistemi veri yapısı hakkında karmaşık bilgi gerektirir ve bu nedenle sisteme makul miktarda karmaşıklık ekler.

Başka bir yaklaşım**, yazma üzerine kopyalama (copy-on write)** (COW) olarak bilinir ve Sun'ın ZFS'si [B07] dahil olmak üzere bir dizi popüler dosya sisteminde kullanılır Bu teknik hiçbir zaman dosyaların veya dizinlerin üzerine yazmaz; bunun yerine, diskte daha önce kullanılmayan konumlara yeni güncellemeler yerleştirir. Bir dizi güncelleme tamamlandıktan sonra, COW dosya sistemleri, yeni güncellenen yapılara

c

işaretçiler eklemek için dosya sisteminin kök yapısını çevirir. Bunu yapmak, dosya sistemini tutarlı tutmayı kolaylaştırır. Gelecekteki bir bölümde günlük yapılandırılmış dosya sistemini (LFS) tartışırken bu teknik hakkında daha fazla şey öğreneceğiz; LFS, COW' un erken bir örneğidir.

Başka bir yaklaşım, Wisconsin'de yeni geliştirdiğimiz yaklaşımdır. **Backpointer tabanlı tutarlılık (backpointer-based consistency(**yada **BBC))** başlıklı bu teknikte, yazma işlemleri arasında herhangi bir sıralama uygulanmaz. Tutarlılığı sağlamak için sistemdeki her bloğa ek bir **geri işaretçi (backpointer)** eklenir; örneğin, her veri bloğunun ait olduğu düğümün bir referansı vardır. Bir dosyaya erişirken, dosya sistemi, iletme işaretçisinin (örneğin, düğümdeki 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 bu nedenle dosya tutarlı olmalıdır; değilse, dosya tutarsızdır ve bir hata döndürülür. Dosya sistemine geri işaretçiler ekleyerek, yeni bir tembel çökme tutarlılığı elde edilebilir [C+12].

Son olarak, bir günlük protokolünün disk yazma işlemlerinin tamamlanması için beklemesi gereken süreyi azaltan teknikleri de araştırdık. **İyimser kilitlenme tutarlılığı (optimistic crash consistency)** [C+13] başlıklı bu yeni yaklaşım, **işlem sağlama toplamı (transaction checksum)** genelleştirilmiş bir biçimini [P+05] kullanarak diske mümkün olduğunca çok yazma işlemi yapar ve ortaya çıkması durumunda tutarsızlıkları tespit etmek için birkaç başka teknik içerir. Bazı iş yükleri için bu iyimser teknikler, performansı büyük ölçüde iyileştirebilir. Ancak, gerçekten iyi çalışması için biraz farklı bir disk arayüzü gereklidir [C+13].

42.1 Özet

Kilitlenme tutarlılığı sorununu anlattık ve bu soruna saldırmak için çeşitli yaklaşımları tartıştık. Bir dosya sistemi denetleyicisi oluşturmaya yönelik eski yaklaşım işe yarar ancak modern sistemlerde kurtarılması muhtemelen çok yavaştır. Bu nedenle, birçok dosya sistemi artık günlük kaydı kullanıyor. Günlük kaydı, kurtarma süresini O'dan (disk hacminin boyutu) O'ya (günlüğün boyutu) düşürür, böylece bir çökme ve yeniden başlatma sonrasında kurtarmayı önemli ölçüde hızlandırır. Bu nedenle, birçok modern dosya sistemi günlük kaydı kullanır. Günlük tutmanın birçok farklı biçimde olabileceğini de gördük; en yaygın olarak kullanılan hem dosya sistemi meta verileri hem de kullanıcı verileri için makul tutarlılık garantilerini korurken, günlüğe gelen trafik miktarını azaltan sıralı meta veri günlük kaydıdır. Sonunda, kullanıcı verileri üzerinde güçlü garantiler, muhtemelen sağlanması gereken en önemli şeylerden biridir; Garip bir şekilde, son araştırmaların gösterdiği gibi, bu alan halen devam eden bir çalışmadır [P+14].

Referanslar

[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.*

c

[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.*

Ev ödevi (Simülasyon)

Bu bölümde, dosya sistemi bozulmalarının nasıl tespit edilebileceğini (ve potansiyel olarak onarılabileceğini) daha iyi anlamak için kullanabileceğiniz basit bir simülatör olan fsck.py tanıtılmaktadır. Simülatörün nasıl çalıştırılacağına ilişkin ayrıntılar için lütfen ilgiliBENİOKU'ya (README) bakın.

Sorular

1. İlk önce fsck.py -D'yi çalıştırın; bu bayrak herhangi bir bozulmayı kapatır ve böyleceonu rastgele bir dosya sistemi oluşturmak için kullanabilir ve orada hangi dosya ve dizinlerin olduğunu belirleyip belirleyemeyeceğinizi görebilirsiniz. Öyleyse devamet ve bunu yap! Haklı olup olmadığınızı görmek için -p bayrağını kullanın. Tohumu (-s) 1, 2 ve 3 gibi farklı değerlere ayarlayarak bunu rastgele oluşturulmuş birkaç farklı dosya sistemi için deneyin.

D bayrağı dosyada varsa herhangi bir bozulmayı kapatır.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -D
ARG seed 0
ARG seedCorrupt 0
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt True

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) (..,0)] [] [] []

Can you figure out which files and directories exist?
```

p bayrağı dosyayı bozuk haliyle önümüze çıkarır.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 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']
```

c

-s bayrağı farklı dosya sistemleri oluşturur.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -s 1
ARG seed 1
ARG seedCorrupt 0
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 1000100110010001
inodes       [d a:0 r:4] [] [] [] [f a:-1 r:1] [] [] [d a:10 r:2] [d a:6 r:2] [] [] [f a:-1 r:3] [] [] [] [f a:-1 r:1]
data bitmap  1000000000100001
data         [(.,0) (..,0) (m,7) (a,8) (g,11)] [] [] [] [] [] [] [] [] [] [(.,7) (..,0) (m,15) (e,11)] [] [] [] [] [(.,8) (..,0) (r,4)
(w,11)]

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

-s 1 ile rastgele oluşturulan dosya sisteminde inode bloktaki 8. inode, data bloktaki boş bir dizini gösteriyor.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -s 2
ARG seed 2
ARG seedCorrupt 0
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 1000000100110101
inodes       [d a:0 r:3] [] [] [] [] [] [d a:4 r:3] [] [] [f a:-1 r:1] [f a:-1 r:2] [] [d a:11 r:3] [] [d a:7 r:2]
data bitmap  1000100000010001
data         [(.,0) (..,0) (c,13)] [] [] [] [(.,7) (..,13) (u,15) (q,11)] [] [] [] [] [] [(.,13) (..,0) (o,7)] [] [] [] [(.,15) (..,
7) (q,11) (e,10)]

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

-s 2 ile oluşturulan rastgele dosya sisteminde inodes bloktaki 15 numaralı indise sahip inode, data blokta boş olan 7. indise sahip bloğu işaret etmekte.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -s 3
ARG seed 3
ARG seedCorrupt 0
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 1011000000000001
inodes       [d a:0 r:3] [] [d a:13 r:3] [d a:6 r:2] [] [] [] [] [] [] [] [] [] [] [f a:15 r:2]
data bitmap  1000000001000101
data         [(.,0) (..,0) (f,15) (x,15) (r,2)] [] [] [] [] [] [] [] [] [] [(.,3) (..,2)] [] [] [] [(.,2) (..,0) (s,3)] [] [w]

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

-s 3 ile oluşturulan dosya sisteminde inodes bloktaki 3 indisli inode, data blokta boş olan 6. indisli bloğu işaretlemekte.

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

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 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?
```

-S 1 ile oluşturulan dosya sistemindeki hata, inodes blokta 13. İndisli alan dolu olmasına rağmen inode bitmapte dolu olduğu gösterilmemiştir.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 1 -c
ARG seed 0
ARG seedCorrupt 1
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 BITMAP corrupt bit 13

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)] [] [] []
```

-c bayrağı ile de çalıştırdığında da 13. Bitmap alanında bozukluk olduğunu söylüyor. Burdaki çözüm bitmapte 13. Biti 1 yapmaktır.

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

Bu kısımda karşılaştırmaları yapabilmek için -S19 ' u ilk başta bayraksız sonra -D bayrağı ile bozulmayı kapatarak sonra da cevaba bakabilmek için -c bayrağı ile çalıştırmamız gereklidir.

Bayraksız normal çalıştırma.

```
mrt@mrt-virtual-machine:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S19
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)] [] [] []
```

c

-D bayrağı ile çalıştırma.

```
mrt@mrt-virtual-machine:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S19 -D
ARG seed 0
ARG seedCorrupt 19
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt True

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) (..,0)] [] [] []

Can you figure out which files and directories exist?
```

Bayraksız ve -D bayrağı ile çalıştırılımış kodun çıktıları karşılaştırıldığında inodes blokta 8. İnode un referans count değerinin farklı olduğu görülür. Buradaki nozukluk : /g dizininin iki dizini var ama inode onun yalnızca bir referansı olduğunu gösteriyor.

```
mrt@mrt-virtual-machine:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S19 -c
ARG seed 0
ARG seedCorrupt 19
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 refcnt decreased

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)] [] [] []
```

-c bayrağı ile de çalıştırıldığında bizim yorumumuzun doğru olduğu görülüyor.

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

-S 5 ile çalıştırdıktan sonra karşılaştırma yapabilmek için -S5' -D bayrağı ile de çalıştırmamız gerekir.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 5
ARG seed 0
ARG seedCorrupt 5
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) (y,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 5 -D
ARG seed 0
ARG seedCorrupt 5
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt True

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) (..,0)] [] [] []
```

İki çıktı karşılaştırıldığında inode bloktaki 8. İndisli inode un işaret ettiği data bloktaki 6. İndisli data verisi olan (s, 15), y olarak değişmiş durumda. Bu da dosya sisteminde bozulmaya neden olur. Cevabı görmek için -c komutunu çalıştıralım.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 5 -c
ARG seed 0
ARG seedCorrupt 5
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 with directory [('.', 8), ('..', 0), ('s', 15)]:
  entry ('s', 15) altered to refer to different name (y)

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) (y,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

-S38 için de karşılaştırma yapılırsa

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 38
ARG seed 0
ARG seedCorrupt 38
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) (b,0)] [] [] []
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 38 -D
ARG seed 0
ARG seedCorrupt 38
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt True

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) (..,0)] [] [] []
```

c

-S38 de ise inode blokta 4. inode, 12. Data bölgesini gösteriyor ve oraya bakıldığında (.., 0) olan verinin, (b, 0) olarak değiştiğini görüyoruz.. Cevabın doğru olduğunu görmek için -c bayrağı ile çalıştıralım

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 38 -c
ARG seed 0
ARG seedCorrupt 38
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 4 with directory [('.', 4), ('..', 0)]:
  entry ('..', 0) altered to refer to different name (b)

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) (b,0)] [] [] []
```

-S642 için de karşılaştırma yapılırsa

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 642
ARG seed 0
ARG seedCorrupt 642
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) (w,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 642 -D
ARG seed 0
ARG seedCorrupt 642
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt True

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) (..,0)] [] [] []
```

-S38' deki gibi aynı hata ile karşılaşıyoruz, inode bloktaki 0. inode data blokta 0. datayı işaret ediyor ve bu yerdeki verilerden biri (g, 8) şeklinde olması gerekirken (w, 8) şeklindendir. Bu durum dosya sisteminde bozulmalara neden olur. Buradaki sorun veri bölgesinde olduğu için kullanıcı tarafından değiştirilebilir. Cevabı görmek için -c bayrağı ile çalıştıralım.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 642 -c
ARG seed 0
ARG seedCorrupt 642
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 0 with directory [('.', 0), ('..', 0), ('g', 8), ('w', 4), ('m', 13), ('z', 13)]:
  entry ('g', 8) altered to refer to different name (w)

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) (w,8) (w,4) (m,13) (z,13)] [] [] [] [] [] [(.,8) (..,0) (s,15)] [] [] [] [] [] [(.,4) (..,0)] [] [] []
```

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

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 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)] [] [] []
```

-S6 ile oluşturulan dosyadaki hata: inodes blokta bir inode var iken inode bitmapte bu inode biti 1 olarak işaretlenmemiştir. Bu durumda araç dolu olan indis için inode bitmapte de 1 olarak işaretlemelidir. -c bayrağı ile cevaba bakalım.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 6 -c
ARG seed 0
ARG seedCorrupt 6
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 12 orphan

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)] [] [] []
```

-S13

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 13
ARG seed 0
ARG seedCorrupt 13
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: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)] [] [] []
```

İnodes blokta 10. İndste inode var iken inode bitmapte bu alan 1 olarak işaretlenmemiştir. -c bayrağı ile cevaba bakalım

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 13 -c
ARG seed 0
ARG seedCorrupt 13
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 10 orphan

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: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)] [] [] []
```

c

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

Karşılaştırma yapabilmek için bayraksız ve -D bayrağı ile çalıştıralım.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 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)] [] [] []

mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 9 -D
ARG seed 0
ARG seedCorrupt 9
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt True

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) (..,0)] [] [] []
```

Burada inodes bloktaki 13. indisli inode, bir dosya(file) türünde olması gerekirken dizin(directory) türündedir, bu dosya sisteminde bozulmalara neden olur. Cevabın doğru olduğunu anlamak için -c bayrağı ile çalıştıralım.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 9 -c
ARG seed 0
ARG seedCorrupt 9
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 13 was type file, now dir

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)] [] [] []
```

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

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 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)] [] [] []

mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 15 -D
ARG seed 0
ARG seedCorrupt 15
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt True

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) (..,0)] [] [] []
```

İnodes blokta 0. indisli inode bir dizin(directory) olarak işaretlenmesi gerekirken dosya(file) olarak işaretlenmiş. Burdaki hata tüm verinin kaybolmasına neden olur. -c ile doğruluğun bakalım.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 15 -c
ARG seed 0
ARG seedCorrupt 15
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 0 was type file, now dir

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)] [] [] []
```

8.Tohumu -S 10 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? cevabınızı kontrol etmek için -c bayrağını kullanın. Buradaki dosya sistemi yapısında onarıma yardımcı olabilecek fazlalık var mı?

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 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)] [] [] []

mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 10 -D
ARG seed 0
ARG seedCorrupt 10
ARG numInodes 16
ARG numData 16
ARG numRequests 15
ARG printFinal False
ARG whichCorrupt -1
ARG dontCorrupt True

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) (..,0)] [] [] []
```

c

Burada data bloktaki verilerin içindeki parantezlerde 2. Eleman inodes bloku işaretler. Data bloktaki 13. İndisli data daki 2. Parantezin 2. Elemanı boş olan bir inodes blok indise işaret etmekte. Yani /w var olmayan bir klasöre taşındı. Bu sorun şöyle çözülür: Hangi klasörde /w olduğunu bulmak için data blokları kontrol edilir ve inode onarılır.
-c Bayrağı ile çalıştırıp hatayı tespit edebildik mi bakalım.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 10 -c
ARG seed 0
ARG seedCorrupt 10
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 4 with directory [('.', 4), ('..', 0)]:
  entry ('..', 0) altered to refer to unallocated inode (3)

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)] [] [] []
```

-c bayrağıyla çalıştırılınca hatayı doğru tespit ettiğimizi görüyoruz.

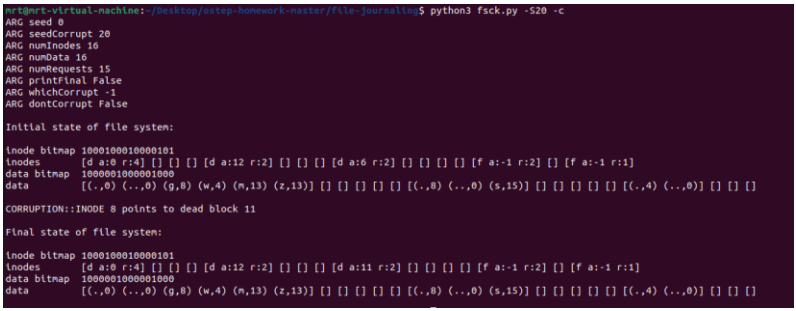
9.Tohumu -S 16 ve -S 20 olarak değiştirin; hangi tutarsızlığı görüyorsunuz? Cevabınızı kontrol etmek için -c kullanın. Onarım aracı sorunu nasıl çözmelidir?

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 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)] [] [] []
```

Burada inodes blokta 13. indisli inode data blokta boş olan 7. İndisli alana işaret ediyor. Burada çözüm dosyayı silmektir. Cevabı görmek için -c bayrağıyla çalıştırdığımızda bize yanlış yere işaret ettiğini gösteriyor.

```
mrt@ubuntu:~/Desktop/ostep-homework-master/file-journaling$ python3 fsck.py -S 16 -c
ARG seed 0
ARG seedCorrupt 16
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 13 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: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)] [] [] []
```

-S20 çalıştırdığımızda da aynı türde bir hatayı alıyoruz. İnodes blokta 8 indisli inode, data blokta boş olan 11 indisli dataya işaret ediyor. Burada \g dizinin veri bloğu boş blok olarak değiştirilmiş oluyor. Burada çözüm veri bloklarını aramak, hangisinin bir dizin olabileceğini bulmak ve .(nokta) \g 'ye işaret edilir.



-c bayrağı ile çalıştırdığımızda bize tekrardan 8. İnode un ölü blok(boş blok) 11' e işaret ettiğini söylüyor.