43

Günlük-yapılı Dosya Sistemleri (Log-structured File Systems)

90'lı yılların başında Berkeley'de Profesör John Ousterhout ve yüksek lisans öğrencisi Mendel Rosenblum liderliğindeki bir grup, günlük yapılı dosya sistemi [RO91] olarak bilinen yeni bir dosya sistemi geliştirdi. Bunu yapma motivasyonları aşağıdaki gözlemlere dayanıyordu:

- **Sistem bellekleri büyüyor (System memories are growing)**: Bellek büyüdükçe, bellekte daha fazla veri önbelleğe alınabilir. Daha fazla veri önbelleğe alındıkça, okumalara önbellek tarafından hizmet verildiği için disk trafiği giderek artan bir şekilde yazmalardan oluşur. Bu nedenle, dosya sistemi performansı büyük ölçüde yazma performansı ile belirlenir.
- **Rastgele G / Ç performansı ile sıralı G / Ç performansı arasında büyük bir boşluk vardır (There is a large gap between random I/O performance and se- quential I/O performance)**: Sabit sürücü aktarım bant genişliği yıllar içinde büyük ölçüde artmıştır [S 98]; Bir sürücünün yüzeyine daha fazla bit paketlendikçe, söz konusu bitlere erişirken bant genişliği artar. Bununla birlikte, arama ve dönme gecikmesi maliyetleri yavaş yavaş azalmıştır; Ucuz ve küçük motorların plakaları daha hızlı döndürmesini veya disk kolunu daha hızlı hareket ettirmesini sağlamak zordur. Bu nedenle, diskleri sıralı bir şekilde kullanabiliyorsanız, aramalara ve dönüşlere neden olan yaklaşımlara göre oldukça büyük bir performans avantajı elde edersiniz.
- **Mevcut dosya sistemleri birçok yaygın iş yükünde kötü performans gösterir (Existing file systems perform poorly on many common workloads)**:Örneğin, FFS [MJLF84], bir blok boyutunda yeni bir dosya oluşturmak için çok sayıda yazma gerçekleştirir: biri yeni bir inode için, biri inode bit eşlemini güncellemek için, biri dosyanın bulunduğu dizin veri bloğuna, biri onu güncellemek için dizin inode'una, biri yeni dosyanın bir parçası olan yeni veri bloğu ve veri bloğunu ayrılmış olarak işaretlemek için veri bit eşlemine bir tane. Bu nedenle, FFS tüm bu blokları aynı blok grubuna yerleştirmesine rağmen, FFS birçok kısa aramaya ve ardından dönme gecikmesine neden olur ve bu nedenle performans, en yüksek sıralı bant genişliğinin çok gerisinde kalır.
- **File systems are not RAID-aware**: Örneğin, hem RAID-4 hem de
- RAID-5, tek bir bloğa mantıksal bir yazmanın 4 fiziksel G / Ç'nin gerçekleşmesine neden olduğu küçük yazma sorununa (**small-write problem)** sahiptir. Mevcut dosya sistemleri bu en kötü durum RAID yazma davranışından kaçınmaya çalışmaz.

İPUCU: AYRINTILAR ÖNEMLİDİR

Tüm ilginç sistemler birkaç genel fikir ve bir dizi ayrıntıdan oluşur. Bazen, bu sistemleri öğrenirken, kendi kendinize "Ah, genel fikri anladım; gerisi sadece detaylar" diye düşünürsünüz ve bunu işlerin gerçekte nasıl yürüdüğünü yalnızca yarısı öğrenmek için kullanırsınız. Bunu yapma! Çoğu zaman, detaylar kritiktir. LFS'de göreceğimiz gibi, genel fikri anlamak kolaydır, ancak gerçekten çalışan bir sistem oluşturmak için tüm zor durumları düşünmeniz gerekir.

Bu nedenle ideal bir dosya sistemi yazma performansına odaklanır ve diskin sıralı bant genişliğini kullanmaya çalışır. Ayrıca, yalnızca veri yazmakla kalmayıp aynı zamanda disk üzerindeki meta veri yapılarını sık sık güncelleyen yaygın iş yüklerinde iyi performans gösterir. Son olarak, tek disklerin yanı sıra baskınlarda da iyi çalışır.

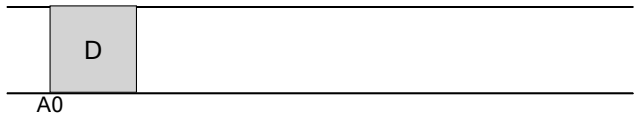
Rosenblum ve Ousterhout'un tanıdığı yeni dosya sistemi türü, **Günlük yapı Dosya Sisteminin (Log-structured File System)** kısaltması olan **LFS** olarak ilan edildi.Disk yazarken, LFS önce tüm güncellemeleri arabelleğe alır (meta veriler dahil!) bir girişte-bellek **segmenti(segment)**; segment dolduğunda, diskin kullanılmayan bir bölümüne uzun, sıralı bir aktarımla diske yazılır.LFS hiçbir zaman mevcut verilerin üzerine yazmaz, bunun yerine segmentleri *her zaman* boş konumlara yazar. Segmentler büyük olduğundan, disk (veya RAID) verimli bir şekilde kullanılır ve dosya sisteminin performansı zirveye yaklaşır.

DÖNÜM:
TÜM YAZILARI SIRALI YAZMALAR NASIL YAPILIR?

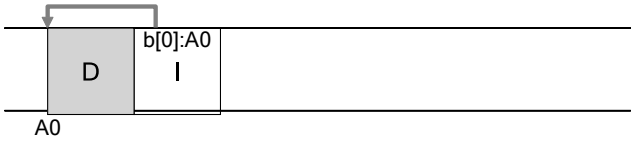
Bir dosya sistemi tüm yazıları sıralı yazmalara nasıl dönüştürebilir? Okunacak istenen blok diskin herhangi bir yerinde olabileceğinden, okumalar için bu görev imkansızdır. Ancak yazmalar için dosya sisteminin her zaman bir seçeneği vardır ve tam da bu seçimden yararlanmayı umuyoruz.

43.1 Diske Sırayla Yazma(Writing To Disk Sequentially)

Bu nedenle ilk zorluğumuz var: dosya sistemi durumuna yapılan tüm güncellemeleri diske bir dizi sıralı yazmaya nasıl dönüştürebiliriz? Bunu daha iyi anlamak için basit bir örnek kullanalım. Bir dosyaya bir veri bloğu D yazdığımızı düşünün. Veri bloğunun diske yazılması, aşağıdaki disk düzeni ile sonuçlanabilir: D disk adresinde yazılmıştır A0:



Ancak, bir kullanıcı bir veri bloğu yazdığında, yalnızca diske yazılan veriler değildir; Güncellenmesi gereken başka meta veriler (**metadata)** de vardır. Bu durumda, dosyanın inode'unu (**inode)** (İ) diske de yazalım ve D veri bloğuna işaret etmesini sağlayalım. Diske yazıldığında, veri bloğu ve inode böyle bir şeye benzeyecektir (inode'un genellikle durum böyle olmayan veri bloğu kadar büyük göründüğünü unutmayın; Çoğu sistemde veri bloklarının boyutu 4 kb'dir, oysa bir inode çok daha küçüktür, yaklaşık 128 bayttır):



Bu, tüm güncellemeleri (veri blokları, inode'lar vb.) Basitçe yazmanın temel fikridir. diske sırayla, LFS'nin kalbinde oturur. Eğer anlarsan bu, temel fikri elde edersiniz. Ancak tüm karmaşık sistemlerde olduğu gibi, şeytan ayrıntılarda gizlidir.

43.2 Sıralı ve Etkili Yazma(Writing Sequentially And Effectively)

Ne yazık ki, diske sırayla yazmak (tek başına) yeterli değildir verimli yazmayı garanti edin. Örneğin, bir single yazdığımızı hayal edin T zamanında A adresine engelle. Sonra bir süre bekleriz ve şöyle yazarız: A + 1 adresindeki disk (sıralı sıradaki sonraki blok adresi), ama zamanla T + δ. Ne yazık ki, birinci ve ikinci yazılar arasında, disk döndürüldü; ikinci yazıyı verdiğinizde, bu şekilde bekleyecektir taahhüt edilmeden önce bir rotasyonun çoğu için (özellikle rotasyon zaman alır $T_{rotation(dönme)}$, disk bekleyecektir $T_{rotation(dönme)} - \delta$ taahhüt etmeden önce disk yüzeyine ikinci yazma). Ve böylece umarım görebilirsiniz diske sıralı sırayla yazmanın bunu başarmak için yeterli olmadığı en yüksek performans; bunun yerine, çok sayıda bitişik yayınlamanız gerekir iyi yazma elde etmek için sürücüye yazar (veya büyük bir yazma) performans.

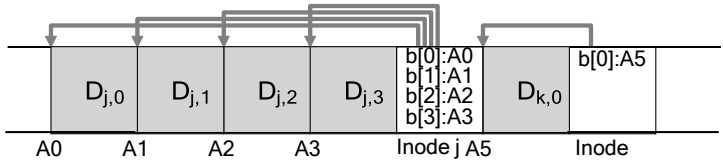
Bu amaca ulaşmak için LFS, yazma olarak bilinen eski bir teknik kullanır Arabellek (**write buffering[1]).** LFS, diske yazmadan önce bellekteki güncellemeleri takip eder; Yeterli sayıda güncelleme aldığında bunları bir kerede diske yazar ve böylece diskin verimli kullanılmasını sağlar.

LFS'nin bir defada yazdığı güncellemelerin büyük bir kısmı bir segmentin (**segment**) adıyla anılır. Bu terim bilgisayarda aşırı kullanılmasına ragmen sistemler, burada sadece LFS'nin gruplamak için kullandığı büyük bir yığın anlamına gelir yazar. Bu nedenle, diske yazarken, LFS bellek içi bir segmentteki güncelleştirmeleri arabelleğe alır ve ardından segmenti bir kerede diske yazar.

[1] Gerçekten de, muhtemelen birçok kişi tarafından icat edildiğinden, bu fikir için iyi bir alıntı bulmak zor1 ve bilgisayar tarihinin çok erken dönemlerinde. Yazma arabelleğinin yararları üzerine bir çalışma için, bkz. Solworth ve Orji [YANİ 90]; potansiyel zararları hakkında bilgi edinmek için bkz. Mogul [M94].

Sürece segment yeterince büyük, bu yazarlar verimli olacak.

İşte LFS'nin iki güncelleme kümesini arabelleğe aldığı bir örnek: küçük segment; gerçek segmentler daha büyüktür (birkaç MB). İlk güncelleme dört bloktan j dosyasına yazılır; ikincisi, k dosyasına eklenen bir bloktur. LFS daha sonra yedi bloğun tüm bölümünü aynı anda diske aktarır. Bu blokların ortaya çıkan disk düzeni aşağıdaki gibidir:



43.3 Tamponlama Ne Kadar? (How Much To Buffer?)

Bu, şu soruyu gündeme getirir: LFS, diske yazmadan önce kaç güncelleştirmeyi arabelleğe almalıdır? Cevap elbette diske bağlı kendisi, özellikle konumlandırma ek yükünün aşağıdakilere kıyasla ne kadar yüksek olduğu aktarım hızı; Benzer bir analiz için FFS bölümüne bakın.

Örneğin, konumlandırmanın (yani döndürme ve genel gider arama) olduğunu varsayalım. her yazmadan önce kabaca $T_{position}$ birkaç saniye sürer. Daha fazla varsayın disk aktarım hızının n R_{peak} MB / sn olduğunu. LFS arabelleği ne kadar olmalı böyle bir diskte çalışırken yazmadan önce?

Bunu düşünmenin yolu, her yazdığınızda, bir konumlandırma maliyetinin sabit yükü. Böylece, ne kadar var bu maliyeti amorti **(amortize)** etmek için yazmak mı? Ne kadar çok yazarsan o kadar iyi (açıkçası) ve en yüksek bant genişliğine ulaşmaya ne kadar yaklaşırsanız. Somut bir cevap elde etmek için D MB yazdığımızı varsayalım. Bu veri yığınını yazma zamanı (T_{Write}) konumlandırma zamanıdır $T_{position}$ artı D aktarma zamanı (D / R_{peak}), veya:

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \tag{43.1}$$

Ve böylece etkili yazma oranı ($R_{effective}$), ki bu sadece yazılan veri miktarı, onu yazmak için toplam süreye bölünür:

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}. \tag{43.2}$$

İlgilendiğimiz şey, efektif oranın ($R_{effective}$) kapanmasıdıen yüksek orana. Spesifik olarak, efektif oranın bir miktar kesir olmasını istiyoruz F, 0 <F <1 olan en yüksek oranın (tipik bir F 0,9 veya % 90'ı olabilir. en yüksek oran). Matematiksel formda, bu istediğimiz anlamına gelir. $R_{effective}$ = F * R_{peak} cBu noktada, D için çözebiliriz:

$$R_{effective} = \frac{R_{effective}}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak} \quad (43.3)$$

$$D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}}) \quad (43.4)$$

$$D = (F \times R_{peak} \times T_{position}) + (F \times R_{peak} \times \frac{D}{R_{peak}}) \quad (43.5)$$

$$D = \frac{F}{1 - F} \times R_{peak} \times T_{position} \quad (43.6)$$

Konumlandırma süresi 10 milisaniye olan bir diskle bir örnek verelim ve 100 MB / s'lik en yüksek aktarım hızı; etkili bir zirvenin % 90'ının bant genişliği (F = 0.9). Bu durumda, D = (0,9 / 0,1) * 100 MB / s * 0,01 saniye = 9 MB. Görmek için bazı farklı değerler deneyin en yüksek bant genişliğine yaklaşmak için ne kadar arabelleğe almamız gerekiyor. Zirvenin % 95'ine ulaşmak için ne kadar gereklidir? 99%?

43.4 Sorun:İnode'ları Bulma(Problem: Finding Inodes)

LFS'de bir inodu nasıl bulduğumuzu anlamak için, tipik bir UNİX dosya sisteminde bir inodu nasıl bulacağımızı kısaca gözden geçirelim. FFS gibi tipik bir dosya sisteminde ve hatta eski UNİX dosya sisteminde, inode'ları bulmak kolaydır, çünkü bunlar bir dizi halinde düzenlenir ve sabit konumlarda diske yerleştirilir.

Örneğin, eski UNİX dosya sistemi tüm düğümleri diskin sabit bir bölümünde tutar. Böylece, bir inode numarası ve başlangıç adresi verildiğinde, belirli bir inodu bulmak için, inode numarasını bir inodun boyutuyla çarparak ve bunu disk üzerindeki dizinin başlangıç adresine ekleyerek tam disk adresini hesaplayabilirsiniz; Bir inode numarası verilen dizi tabanlı indeksleme, hızlı ve basittir.

FFS'de bir inode numarası verilen bir inode bulmak yalnızca biraz daha karmaşıktır, çünkü FFS inode tablosunu parçalara ayırır ve her silindir grubuna bir grup inode yerleştirir. Bu nedenle, her bir düğüm parçasının ne kadar büyük olduğunu ve her birinin başlangıç adreslerini bilmek gerekir. Bundan sonra hesaplamalar benzer ve aynı zamanda kolaydır.

LFS'de hayat daha zordur. Neden? İnode'ları tüm diske dağıtmayı başardık! Daha da kötüsü, asla yerine yazmayız ve bu nedenle bir inode'un en son sürümü (yani istediğimiz sürüm) hareket etmeye devam eder.

43.5 Yönlendirme Yoluyla Çözüm: İnode Haritası (Solution Through Indirection: The Inode Map)

Bunu düzeltmek için, LFS tasarımcıları,**inode haritası (ımap) (inodemap(imap))** adı verilen bir veri yapısı aracılığıyla inode numaraları ile düğümler arasında bir **yönlendirme düzeyi (level of indirection)** getirdiler. Imap, bir inode numarası alan bir yapıdır girdi olarak ve inode'un en son sürümünün disk adresini üretir

İPUCU: BİR YÖNLENDİRME DÜZEYİ KULLANIN

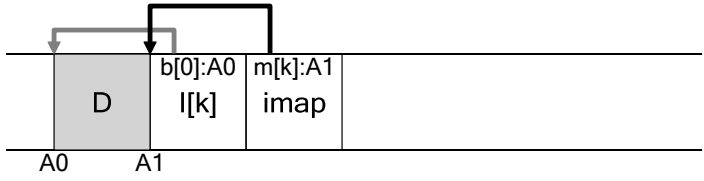
İnsanlar genellikle Bilgisayar Bilimlerindeki tüm sorunların çözümünün sadece bir yönlendirme düzeyi(**level of indirection)** olduğunu söylerler. Bu açıkça doğru değil; Bu sadece çoğu sorunun çözümüdür (evet, bu hala çok güçlü bir yorumdur, ancak noktayı anlıyorsunuz). Çalıştığımız her sanallaştırmayı, örneğin sanal belleği veya bir dosya kavramını kesinlikle bir yönlendirme düzeyi olarak düşünebilirsiniz. Ve kesinlikle LFS'deki inode haritası, inode numaralarının sanallaştırılmasıdır. Umarım bu örneklerde yönlendirmenin büyük gücünü görebilirsiniz, bu da her referansı değiştirmek zorunda kalmadan yapıları (VM örneğindeki sayfalar veya lfs'deki inode'lar gibi) serbestçe hareket ettirmemize izin verir. Tabii ki, yönde bir dezavantajı olabilir: **ekstra ek yük (extra overhead)**. Bu nedenle, bir dahaki sefere bir sorununuz olduğunda, dolaylı olarak çözmeyi deneyin, ancak önce bunu yapmanın genel giderlerini düşündüğünüzden emin olun. Wheeler'ın ünlü bir şekilde dediği gibi, "Bilgisayar bilimlerindeki tüm problemler, elbette çok fazla dolaylı sorun dışında, başka bir dolaylı düzeyle çözülebilir."

Bu nedenle, genellikle giriş başına 4 bayt (bir disk işaretçisi) ile basit bir *dizi*, olarak uygulanacağını hayal edebilirsiniz. Diske herhangi bir inode yazıldığında, harita yeni konumu ile güncellenir.

Imap maalesef kalıcı tutulmalıdır (yani diske yazılmalıdır); Bunu yapmak, lfs'nin çökmeler arasında inode'ların konumlarını takip etmesine ve böylece istenildiği gibi çalışmasına izin verir. Bu nedenle, bir soru: ımap diskte nerede bulunmalıdır?

Elbette diskin sabit bir bölümünde yaşayabilir. Ne yazık ki, sık sık güncellendiğinden, bu daha sonra dosya yapılarına yönelik güncellemelerin ardından ımap'ye yazmaların yapılmasını gerektirecek ve bu nedenle performans düşecektir (yani, her güncelleme ile ımap'ın sabit konumu arasında daha fazla disk araması olacaktır).

Bunun yerine, LFS, inode haritasının parçalarını diğer tüm yeni bilgileri yazdığı yerin hemen yanına yerleştirir. Bu nedenle, bir dosyaya bir veri bloğu eklerken k, LFS aslında yeni veri bloğunu, onun inode'unu ve inode eşlemesinin bir parçasını hep birlikte diske aşağıdaki gibi yazar:



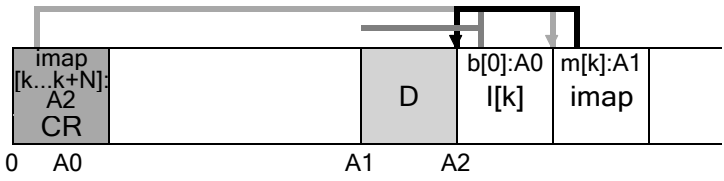
Bu resimde,ımap işaretli blokta depolanan ımap dizisinin parçası, LFS'ye k kodunun A1 disk adresinde olduğunu söyler; Bu inode ise LFS'ye D veri bloğunun A0 adresinde olduğunu söyler.

43.6 Çözümün Tamamlanması: Kontrol Noktası Bölgesi (Completing The Solution: The Checkpoint Region)

Zeki okuyucu (bu sensin, değil mi?) burada bir sorun fark etmiş olabilir. İnode haritasını nasıl bulabiliriz, şimdi parçaları da diske yayılmış durumda mı? Sonunda sihir yoktur: dosya aramasına başlamak için dosya sisteminin diskte sabit ve bilinen bir konuma sahip olması gerekir.

LFS,bunun için diskte **kontrol noktası bölgesi** (**check-point region (CR))** olarak bilinen sabit bir yere sahiptir. Kontrol noktası bölgesi, inode haritasının en son parçalarına işaretçiler (ör. adreslerini) içerir ve bu nedenle inode harita parçaları önce CR okunarak bulunabilir. Not denetim noktası bölgesi yalnızca düzenli aralıklarla güncelleştirilir (örneğin, her 30 saniyede bir) ve bu nedenle performans kötü etkilenmezBu nedenle, disk üzerindeki mizanpajın genel yapısı bir kontrol noktası bölgesi içerir (ode haritasının en son parçalarına işaret eder); inode harita parçalarının her biri inode'ların adreslerini içerir; inode'lar, tipik UNİX dosya sistemleri gibi dosyalara (ve dizinlere) işaret eder.

Denetim noktası bölgesinin bir örneği(diskin başında,adres 0'da sonuna kadar olduğuna dikkat edin) ve tek bir ımap parçası, inode ve veri bloğu. Gerçek bir dosya sistemi elbette çok daha büyük bir CR(aslında daha sonra anlayacağımız gibi iki tane olurdu), birçok ımap parçasına ve elbette daha birçok inoda, veri bloğuna vb. sahip olacaktır.

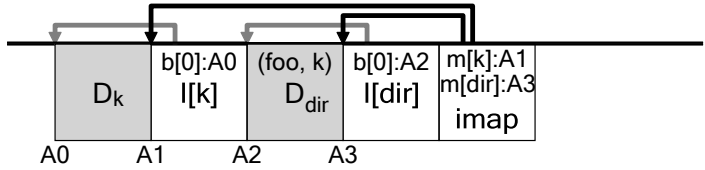


43.7 Diskten Dosya Okuma:Özet (Reading A File From Disk: A Recap)

LFS'nin nasıl çalıştığını anladığınızdan emin olmak için, şimdi bir dosyayı diskten okumak için neler olması gerektiğini inceleyelim. Başlamak için hafızamızda hiçbir şey olmadığını varsayalım. Okumamız gereken ilk disk üzerindeki veri yapısı kontrol noktası bölgesidir. Kontrol noktası bölgesi, tüm inode haritasına işaretçiler (örnek disk adresleri) içerir ve böylece LFS daha sonra tüm iode haritasını okur ve bellekte önbelleğe alır.Bu noktadan sonra, bir dosyanın inode numarası verildiğinde, LFS, imap'de inode-numarasını inode-disk-adresi eşlemesine arar ve inode'un en son sürümünde okur. Bu noktada, dosyadan bir bloğu okumak için LFS, doğrudan işaretçiler veya dolaylı işaretçiler veya gerektiğinde iki kat dolaylı işaretçiler kullanarak tam olarak tipik bir UNİX dosya sistemi olarak ilerlerYaygın durumda, LFS, diskten bir dosya okunurken tipik bir dosya sistemiyle aynı sayıda G / Ç gerçekleştirmelidir; tüm ımap önbelleğe alınır ve bu nedenle LFS'nin okuma sırasında yaptığı fazladan iş, imap'te kodun adresini aramaktır.

43.8 Dizinler Ne Olacak?(What About Directories?)

Şimdiye kadar, yalnızca inode'ları ve veri bloklarını dikkate alarak tartışmamızı biraz basitleştirdik. Ancak dosya sistemindeki bir dosyaya erişmek için (favori sahte dosya adlarımızdan biri olan/home/remzi/foo, gibi) bazı dizinlere de erişilmesi gerekir. Peki LFS dizin verilerini nasıl depolar? Neyse ki, dizin yapısı temel olarak klasik UNİX dosya sistemleriyle aynıdır, çünkü bir dizin yalnızca bir (ad, kod numarası) eşlemeleri topluluğudur. Örneğin, diskte bir dosya oluştururken, LFS'nin hem yeni bir düğüm, bazı veriler hem de bu dosyaya başvuran dizin verileri ve onun kodunu yazması gerekir. LFS'nin bunu diskte sırayla yapacağını unutmayın (güncellemeleri bir süre arabelleğe aldıktan sonra). Böylece, bir dosya foo oluşturma dizin, diskte aşağıdaki yeni yapılara yol açacaktır:



Inode eşlemesinin parçası, hem dizin dosyası dizininin hem de yeni oluşturulan f dosyasının konumu hakkında bilgi içerir. Bu nedenle, foo dosyasına erişirken (k numaralı inode ile), dizin dizininin (A3) inode'unun konumunu bulmak için önce inode haritasına (genellikle bellekte önbelleğe alınır) bakarsınız; ardından, dizin verilerinin konumunu (A2) veren dizin kodunu okursunuzbu veri bloğunu okumak size (`foo`, k) 'nin addan kod numarasına eşlemesini verir. Ardından, k (A 1) kod numarasının konumunu bulmak için inode haritasına tekrar bakın ve son olarak A0 adresinden istediğiniz veri bloğunu okuyun.

LFS'de, **özyinelemeli güncelleştirme sorunu(recursive update problem)** [Z + 12] olarak bilinen, inode eşlemesinin çözdüğü başka bir ciddi sorun vardır. Sorun, hiçbir zaman yerinde güncellenmeyen (LFS gibi), bunun yerine güncellemeleri diskteki yeni konumlara taşıyan herhangi bir dosya sisteminde ortaya çıkar.

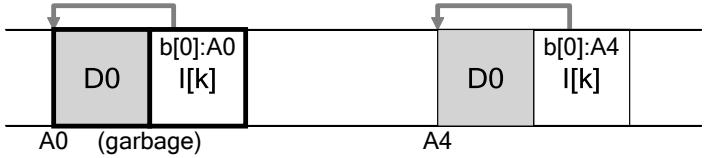
Özellikle, bir inode güncellendiğinde, diskteki konumu değişirDikkatli olmasaydık, bu aynı zamanda bu dosyaya işaret eden dizinde bir güncelleme yapılmasını gerektirecekti, bu da o dizinin üst öğesinde bir değişiklik yapılmasını zorunlu kılacaktı, vb. Dosya sistemi ağacına kadar.

LFS, inode haritasıyla bu sorunu akıllıca önler. Bir inode'un konumu değişebilse de, değişiklik hiçbir zaman dizinin kendisine yansıtılmaz; bunun yerine, dizin aynı addan kod numarasına eşlemeyi tutarken ımap yapısı güncelleştirilir. Bu nedenle, yönlendirme yoluyla LFS, özyinelemeli güncelleme sorununu önler.

43.9 Yeni Bir Sorun:Çöp Toplama(A New Problem: Garbage Collection)

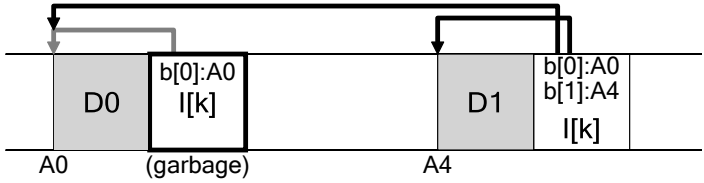
LFS ile ilgili başka bir sorun fark etmiş olabilirsiniz; bir dosyanın en son sürümünü (inode ve verileri dahil) diskteki yeni konumlara art arda yazar. Bu işlem, yazmaları verimli tutarken, lfs'nin dosya yapılarının eski sürümlerini diske dağılmış halde bıraktığı anlamına gelir. Biz (oldukça belirsiz bir şekilde) bu eski sürümlere **çöp(garbage**) diyoruz.

Örneğin, tek bir veri bloğu D0'a işaret eden k inode numarasına göre yeniden eklenmiş mevcut bir dosyamız olduğu durumu hayal edelim. Şimdi bu bloğu güncelleyerek hem yeni bir inode hem de yeni bir veri bloğu oluşturuyoruz. Sonuçta ortaya çıkan LFS disk düzeni şöyle bir şeye benzeyecektir (basitlik için ımap'yi ve diğer yapıları atladığımızı unutmayın; yeni inode'u işaret etmek için yeni bir ımap parçasının da diske yazılması gerekir):



Diyagramda, hem inode hem de veri bloğunun diskte iki sürümü olduğunu görebilirsiniz, biri eski (soldaki) ve bir akım ve böylece **canlı** (sağdaki)(**live** (the one on the right)). Bir veri bloğunu (mantıksal olarak) basit bir şekilde güncelleme eylemiyle, bir dizi yeni yapının LFS tarafından sürdürülmesi ve böylece söz konusu blokların eski sürümlerinin diskte bırakılması gerekir.

Başka bir örnek olarak, bunun yerine o orijinal dosyaya bir blok eklediğimizi hayal edin k. Bu durumda, inode'un yeni bir sürümü oluşturulur, ancak eski veri bloğu hala inode tarafından işaret edilir. Bu nedenle, hala hayatta ve mevcut dosya sisteminin çok büyük bir parçası:



Peki inode'ların, veri bloklarının vb. Bu eski sürümleriyle ne yapmalıyız? Bu eski sürümler etrafta kalabilir ve kullanıcıların eski dosya sürümlerini geri yüklemelerine izin verilebilir (örneğin, yanlışlıkla bir dosyanın üzerine yazdıklarında veya sildiklerinde, bunu yapmak oldukça kullanışlı olabilir); böyle bir dosya sistemi, bir dosyanın farklı sürümlerini takip ettiği için **sürüm oluşturma dosya sistemi(versioning file system)** olarak bilinir. Ancak bunun yerine LFS, bir dosyanın yalnızca en son canlı sürümünü saklar; bu nedenle (arka planda), LFS, dosya verilerinin, kodların ve diğer yapıların bu eski ölü sürümlerini periyodik olarak bulmalıdır, ve onları **temizleyin (clean);** temizlik yapmalı

böylece, sonraki yazmalarda kullanılmak üzere diskteki blokları tekrar boş hale getirin. Temizleme işleminin, programlar için kullanılmayan programları otomatik olarak boşaltan programlama dillerinde ortaya çıkan bir teknik olan **bir çöp toplama (garbage collection)** biçimi olduğunu unutmayın.

Daha önce, büyük yazmaların lfs'de diske yazılmasını sağlayan mekanizma olduğu kadar önemli segmentleri de tartıştık. Anlaşıldığı üzere, aynı zamanda etkili temizliğin oldukça ayrılmaz bir parçasıdırlar. LFS temizleyicisi basitçe tek veri bloklarını, inode'ları vb. Geçip serbest bırakırsa ne olacağını hayal edin., temizlik sırasında. Sonuç: diskte ayrılan alan arasında bir miktar boş **delik(holes)** bulunan bir dosya sistemi. LFS, diske sırayla ve yüksek performansla yazmak için geniş bir bitişik bölge bulamayacağından yazma performansı önemli ölçüde düşecektir.

Bunun yerine, LFS temizleyici segmentlere göre segmentler bazında çalışır, böylece daha sonra yazmak için büyük alan parçalarını temizler. Temel temizleme işlemi aşağıdaki gibi çalışır. Periyodik olarak, LFS temizleyici bir dizi eski (kısmen kullanılmış) segmentte okur, bu segmentler içinde hangi blokların canlı olduğunu belirler ve ardından yalnızca içindeki canlı bloklarla yeni bir segment seti yazar ve eskileri yazmak için serbest bırakır. Özellikle, temizleyicinin mevcut segmentleri okumasını, içeriklerini N yeni segmentlere (burada N <M) **sıkıştırmasını(compact)** ve ardından N segmentlerini yeni konumlarda diske yazmasını bekleriz. Eski M segmentleri daha sonra serbest bırakılır ve sonraki yazmalar için dosya sistemi tarafından kullanılabilir.

Ancak şimdi iki sorunla karşı karşıyayız. Birincisi mekanizmadır: LFS, bir segment içindeki hangi blokların canlı ve hangilerinin ölü olduğunu nasıl söyleyebilir? İkincisi politikadır: temizleyici ne sıklıkta çalışmalı ve hangi bölümleri temizlemek için seçmelidir?

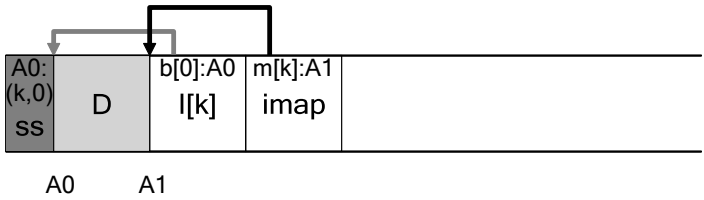
43.10 Blok Canlılığının Belirlenmesi(Determining Block Liveness)

Önce mekanizmayı ele alıyoruz. Bir disk üzerindeki segmentler içindeki bir veri bloğu D verildiğinde, LFS, D'nin canlı olup olmadığını belirleyebilmelidir. Bunu yapmak için LFS, her bloğu tanımlayan her segmente biraz daha fazla bilgi ekler. Özellikle, LFS, her veri bloğu D için, onun kod numarasını (hangi dosyaya ait olduğunu) ve ofsetini (dosyanın hangi bloğu olduğunu) içerir. Bu bilgiler, **segment özeti bloğu(segment summary block)** olarak bilinen segmentin başındaki bir yapıya kaydedilir.

Bu bilgi göz önüne alındığında, bir bloğun canlı mı yoksa ölü mü olduğunu belirlemek kolaydır. Diskte A adresinde bulunan bir D bloğu için segment özeti bloğuna bakın ve N ve ofset T kod numarasını bulun. Ardından, N'nin nerede yaşadığını bulmak ve N'yi diskten okumak için ımap'ye bakın (belki de zaten bellekte, ki bu daha da iyi). Son olarak, T ofsetini kullanarak, inode'un bu dosyanın T. bloğunun diskte nerede olduğunu düşündüğünü görmek için inode'a (veya bazı dolaylı bloklara) bakın. Tam olarak A disk adresini gösteriyorsa, LFS, D bloğunun canlı olduğu sonucuna varabilir. Başka bir yere işaret ederse, LFS, D'nin kullanımda olmadığı sonucuna varabilir (yani öldü) ve böylece bu sürüme artık ihtiyaç duyulmadığını bilir. İşte bir sözde kod özeti:

```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

Burada, segment özet bloğunun (SS işaretli) A0 adresindeki veri bloğunun aslında 0 ofsetindeki *k* dosyasının bir parçası olduğunu kaydettiği mekanizmayı gösteren bir diyagram bulunmaktadır. Imap'ı *k* için kontrol ederek, inodu bulabilir ve gerçekten o konuma işaret ettiğini görebilirsiniz.



Canlılığı belirleme sürecini daha verimli hale getirmek için lfs'nin kullandığı bazı kısayollar vardır. Örneğin, bir dosya kesildiğinde veya silindiğinde, LFS **sürüm numarasını (version number)** artırır ve yeni sürüm numarasını ımap'ye kaydeder. Sürüm numarasını disk üzerindeki bölüme de kaydederek, LFS, disk üzerindeki sürüm numarasını ımap'deki bir sürüm numarasıyla karşılaştırarak yukarıda açıklanan daha uzun kontrolü kısa devre yapabilir, böylece fazladan okumalardan kaçınabilir.

43.11 Bir Politika Sorusu: Hangi Bloklar Temizlenmeli ve Ne Zaman?(A Policy Question: Which Blocks To Clean, And When?)

Yukarıda açıklanan mekanizmanın yanı sıra, LFS hem ne zaman temizleneceğini hem de hangi blokların temizlenmeye değer olduğunu belirlemek için bir dizi ilke içermelidir. Ne zaman temizleneceğini belirlemek daha kolaydır; periyodik olarak, boşta kalma süresi boyunca veya disk dolu olduğu için yapmanız gerektiğinde.

Hangi blokların temizleneceğini belirlemek daha zordur ve birçok araştırma makalesine konu olmuştur. Orijinal LFS makalesinde [RO91] yazarlar, sıcak ve soğuk segmentleri ayırmaya çalışan bir yaklaşımı tanımlamaktadır. Sıcak bölüm, içeriğin sık sık üzerine yazıldığı bölümdür; Bu nedenle, böyle bir bölüm için en iyi politika, temizlemeden önce uzun süre beklemektir, çünkü giderek daha fazla blok yazıldığından (yeni bölümlerde) ve böylece kullanım için serbest bırakıldığından. Soğuk bir segment, aksine, birkaç ölü bloğa sahip olabilir, ancak içeriğinin geri kalanı nispeten kararlıdır. Bu nedenle yazarlar, soğuk segmentlerin er ya da geç temizlenmesi ve tam olarak bunu yapan bir sezgisel geliştirmesi gerektiği sonucuna varırlar. Ancak, çoğu politikada olduğu gibi, bu politika mükemmel değildir; Daha sonraki ap- proach'lar nasıl daha iyi yapılacağını gösterir [MR + 97].

43.12 Kilitlenme Kurtarma ve Günlük (Crash Recovery And The Log)

One Son bir sorun: LFS diske yazarken sistem çökerse ne olur? Önceki bölümde hatırladığınız gibi, güncellemeler sırasında çökmeler dosya sistemleri için zordur ve bu nedenle lfs'nin de dikkate alması gereken bazı şeyler vardır.

Normal çalışma sırasında, arabelleklerin bir segmentte yazar ve sonra (segment dolduğunda veya belirli bir süre geçtiğinde), segmenti diske yazar. LFS bu yazmaları bir günlükte düzenler, yani kontrol noktası bölgesi bir baş ve kuyruk segmentine işaret eder ve her segment yazılacak bir sonraki segmente işaret eder. LFS ayrıca kontrol noktası bölgesini periyodik olarak günceller. Çökmeler, bu işlemlerden herhangi biri sırasında açıkça ortaya çıkabilir (bir segmente yazın, cr'ye yazın). Peki LFS, bu yapılara yazma sırasında çökmeleri nasıl ele alır?

Önce ikinci davayı ele alalım. CR güncellemesinin atomik olarak gerçekleşmesini sağlamak için, LFS aslında diskin her iki ucunda birer tane olmak üzere iki Araba tutar ve bunlara dönüşümlü olarak yazar. LFS ayrıca, cr'yi inode haritasına ve diğer bilgilere en son işaretçilerle güncellerken dikkatli bir protokol uygular; Özellikle, önce bir başlık (zaman damgalı), ardından cr'nin gövdesi ve ardından son olarak son bir blok (ayrıca zaman damgalı) yazar. Bir CR güncellemesi sırasında sistem çökerse, LFS tutarsız bir çift zaman damgası görerek bunu algılayabilir. LFS her zaman tutarlı zaman damgalarına sahip en son cr'yi kullanmayı seçer ve böylece cr'nin tutarlı bir şekilde güncellenmesi sağlanır.

Şimdi ilk davayı ele alalım. LFS, cr'yi her 30 saniyede bir yazdığından, dosya sisteminin son tutarlı anlık görüntüsü oldukça eski olabilir. Bu nedenle, yeniden başlatıldığında, LFS, yalnızca kontrol noktası bölgesinde, işaret ettiği ımap parçalarını ve sonraki dosya ve dizinleri okuyarak kolayca kurtarılabilir; Ancak, güncellemelerin son saniyeleri kaybolacaktır.

Bunu iyileştirmek için LFS, veritabanı topluluğunda **ileri sarma (roll forward)** olarak bilinen bir teknikle bu bölümlerin çoğunu yeniden oluşturmaya çalışır. Temel fikir, son kontrol noktası bölgesi ile başlamak, günlüğün sonunu (cr'ye dahil olan) bulmak ve ardından bunu sonraki bölümleri okumak ve içinde geçerli güncellemeler olup olmadığını görmek için kullanmaktır. Varsa, LFS dosya sistemini buna göre günceller ve böylece son kontrol noktasından bu yana yazılan verilerin ve meta verilerin çoğunu kurtarır. Ayrıntılar için Rosenblum'un ödüllü tezine bakın [R92].

43.13 Özet (Summary)

LFS, diski güncellemek için yeni bir yaklaşım sunar. Dosyaları yerlere aşırı yazmak yerine, LFS her zaman diskin kullanılmayan bir bölümüne yazar ve daha sonra bu eski alanı temizleme yoluyla talep eder. Veritabanı sistemlerinde **gölge sayfalama (shadow paging**)[L77] ve dosya sistemi konuşmasında bazen **kopyala-yaz (copy-on-write)** olarak adlandırılan bu yaklaşım, LFS tüm güncellemeleri bir bellek içi segmentte toplayabileceğinden ve ardından bunları sırayla birlikte yazabileceğinden, oldukça verimli yazmayı sağlar.

İPUCU: KUSURLARI ERDEMLERE DÖNÜŞTÜRÜN
Sisteminizin temel bir kusuru olduğunda, onu bir özelliğe mi yoksa kullanışlı bir şeye mi dönüştürebileceğinize bakın. Netapp'ın wafl'si bunu eski dosya içeriğiyle yapar; Eski sürümleri kullanıma sunarak, wafl'in artık sık sık temizlik konusunda endişelenmesine gerek kalmaz (sonunda arka planda eski sürümleri silmesine rağmen) ve böylece harika bir özellik sağlar ve LFS temizleme sorununun çoğunu ortadan kaldırır. harika bir bükülme. Sistemlerde bunun başka örnekleri var mı? Kuşkusuz, ama onları kendiniz düşünmeniz gerekecek, çünkü bu bölüm büyük bir "O" ile sona erdi. Üzerinde. Bitti. Bozuk. Çıktık. Barış!

LFS'nin ürettiği büyük yazılar, birçok farklı cihazda performans için mükemmeldir. Sabit sürücülerde büyük yazmalar, konumlandırma süresinin en aza indirilmesini sağlar; RAID-4 ve RAID-5 gibi eşlik tabanlı baskınlarda, küçük yazma sorununu tamamen önlerler. Son araştırmalar, Flash tabanlı Ssd'lerde yüksek performans için büyük G / Ç'lerin gerekli olduğunu bile göstermiştir [H + 17]; Bu nedenle, belki de şaşırtıcı bir şekilde, LFS tarzı dosya sistemi terimleri, bu yeni ortamlar için bile mükemmel bir seçim olabilir.

Bu yaklaşımın dezavantajı, çöp üretmesidir; Verilerin eski kopyaları diske dağılmıştır ve daha sonra kullanmak üzere bu tür bir alanı yeniden talep etmek istiyorsa, eski bölümleri periyodik olarak temizlemelidir. Temizlik, lfs'de birçok tartışmanın odağı haline geldi ve temizlik maliyetleriyle ilgili endişeler [SS + 95] belki de lfs'nin saha üzerindeki ilk etkisini sınırladı. Bununla birlikte, Netapp'ın **WAFL** [HLM94], Sun'ın **ZFS** [B07] ve Linux **btrfs** [R + 13] ve hatta modern **flash tabanlı SSD'ler (flash-based SSDs**)[AD14] dahil olmak üzere bazı modern ticari dosya sistemleri, diske yazmaya benzer bir kopyala-yaz yaklaşımı benimser ve böylece lfs'nin entelektüel mirası bu modern dosya sistemlerinde yaşar. Özellikle WAFL, temizleme sorunlarını bir özelliğe dönüştürerek atlattı; Kullanıcılar, dosya sisteminin eski sürümlerini **anlık görüntüler (snapshots**) aracılığıyla sağlayarak, mevcut dosyaları yanlışlıkla sildiklerinde eski dosyalara erişebiliyorlardı.

.

References

[AD14] "Operating Systems: Three Easy Pieces" (Chapter: Flash-based Solid State Drives) by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Arpaci-Dusseau Books, 2014. *A bit gauche to refer you to another chapter in this very book, but who are we to judge?*

[B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Copy Available: `http://www.ostep.org/Citations/zfs_last.pdf`. *Slides on ZFS; unfortunately, there is no great ZFS paper (yet). Maybe you will write one, so we can cite it here?*

[H+17] "The Unwritten Contract of of Solid State Drives" by Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. EuroSys '17, April 2017. *Which unwritten rules one must follow to extract high performance from an SSD? Interestingly, both request scale (large or parallel requests) and locality still matter, even on SSDs. The more things change ...*

[HLM94] "File System Design for an NFS File Server Appliance" by Dave Hitz, James Lau, Michael Malcolm. USENIX Spring '94. *WAFL takes many ideas from LFS and RAID and puts it into a high-speed NFS appliance for the multi-billion dollar storage company NetApp.*

[L77] "Physical Integrity in a Large Segmented Database" by R. Lorie. ACM Transactions on Databases, Volume 2:1, 1977. *The original idea of shadow paging is presented here.*

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM TOCS, Volume 2:3, August 1984. *The original FFS paper; see thechapter on FFS for more details.*

[MR+97] "Improving the Performance of Log-structured File Systems with Adaptive Methods" by Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, Thomas E. Anderson. SOSP 1997, pages 238-251, October, Saint Malo, France. *A more recent paper detailing better policies for cleaning in LFS.*

[M94] "A Better Update Policy" by Jeffrey C. Mogul. USENIX ATC '94, June 1994. *In this paper, Mogul finds that read workloads can be harmed by buffering writes for too long and then sending them to the disk in a big burst. Thus, he recommends sending writes more frequently and in smaller batches.*

[P98] "Hardware Technology Trends and Database Opportunities" by David A. Patterson. ACM SIGMOD '98 Keynote, 1998. Available online here: `http://www.cs.berkeley.edu/~pattrsn/talks/keynote.html`. *A great set of slides on technology trends in computer systems. Hopefully, Patterson will create another of these sometime soon.*

[R+13] "BTRFS: The Linux B-Tree Filesystem" by Ohad Rodeh, Josef Bacik, Chris Mason. ACM Transactions on Storage, Volume 9 Issue 3, August 2013. *Finally, a good paper on BTRFS, a modern take on copy-on-write file systems.*

[RO91] "Design and Implementation of the Log-structured File System" by Mendel Rosen- blum and John Ousterhout. SOSP '91, Pacific Grove, CA, October 1991. *The original SOSP paper about LFS, which has been cited by hundreds of other papers and inspired many real systems.*

[R92] "Design and Implementation of the Log-structured File System" by Mendel Rosenblum. http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-696.pdf. *The award-winning dissertation about LFS, with many of the details missing from the paper.*

[SS+95] "File system logging versus clustering: a performance comparison" by Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan. USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995. *A paper that showed the LFS performance sometimes has problems, particularly for workloads with many calls to `fsync()` (such as database workloads). The paper was controversial at the time.*

[SO90] "Write-Only Disk Caches" by Jon A. Solworth, Cyril U. Orji. SIGMOD '90, Atlantic City, New Jersey, May 1990. *An early study of write buffering and its benefits. However, bufferingfor too long can be harmful: see Mogul [M94] for details.*

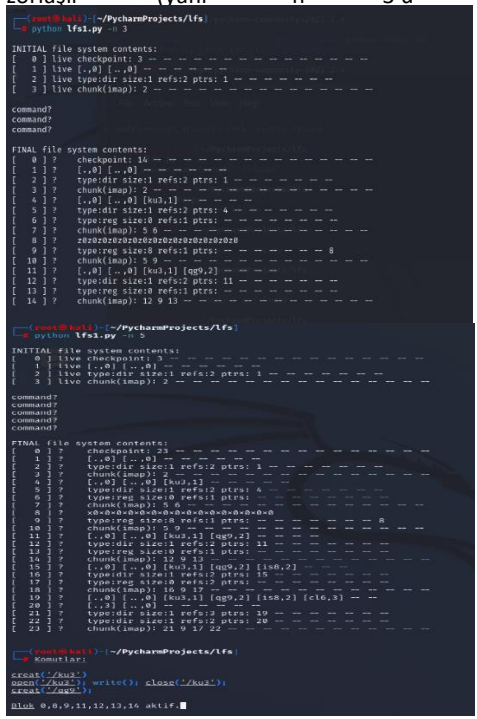
[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes" by Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '13, San Jose, California, February 2013. *Our paper on a new way to build flash-based storage devices, to avoid redundant mappings in the file system and FTL. The idea is for the device to pick the physical location of a write, and return the address to the file system, which stores the mapping.*

Ödev (Simülasyon) Homework (Simulation)

Bu bölüm tanıtır lfs.py , LFS tabanlı bir dosya sisteminin nasıl çalıştığını daha iyi anlamak için kullanabileceğiniz basit bir LFS simülatörü. Simülatörün nasıl çalıştırılacağına ilişkin ayrıntılar için README bölümünü okuyun.

Sorular (Questions)

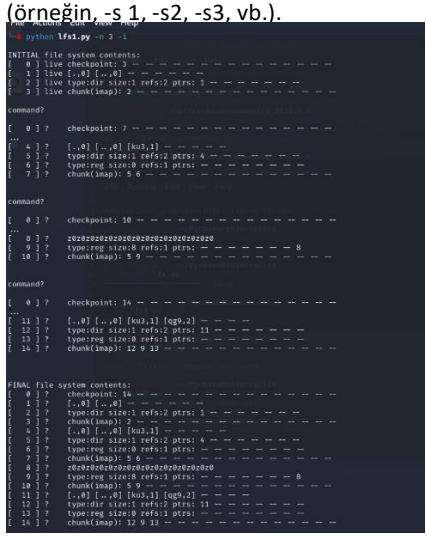
1. Run ./lfs.py -n 3, belki de seed (ler) i değiştirir. Son dosya sistemi içeriğini oluşturmak için hangi komutların çalıştırıldığını bulabilir misiniz? Bu emirlerin hangi emirle verildiğini söyleyebilir misiniz? Son olarak, son dosya sistemi durumundaki her bloğun canlılığını belirleyebilir misiniz? Hangi komutların çalıştırıldığını göstermek için -o ve son dosya sistemi durumunun canlılığını göstermek için - c kullanın. Verilen komut sayısını artırdıkça görev sizin için ne kadar zorlaşır (yani -n 3'ü 5'e değiştirin)?



Blok 0,8,9,11,12,13,14 aktif.

2. Yukarıdakileri acı verici bulursanız, her bir komutun neden olduğu güncelleme kümesini göstererek kendinize biraz yardımcı olabilirsiniz. Bunu yapmak için run./lfs.py -n 3 -i. Şimdi her komutun ne olması gerektiğini anlamanın daha kolay olup olmadığına bakın. Yorumlanacak farklı komutlar almak için rastgele tohumu değiştirin

(örneğin, -s 1, -s2, -s3, vb.).



- İ bayrağı kesinlikle komutları anlamayı kolaylaştırır. Komutlar hakkında bilgi verir.

3. Her komut tarafından diske hangi güncelleştirmelerin yapıldığını bulma yeteneğinizi daha fazla sınamak için aşağıdakileri çalıştırın: ./lfs.py -o -F -s 100 (ve belki birkaç tane daha rastgele seed). Bu yalnızca bir komut kümesini gösterir ve size dosya sisteminin son durumunu göstermez. Dosya sisteminin son durumunun ne olması gerektiği konusunda bir nedeniniz olabilir mi?

4. Şimdi, bir dizi dosya ve dizin işleminden sonra hangi dosya ve dizinlerin canlı olduğunu belirleyip belirleyemeyeceğinize bakın. Tt'yi çalıştırın./lfs.py - n 20 -s 1 ve ardından son dosya sistemi durumunu inceleyin. Hangi yol adlarının geçerli olduğunu bulabilir Tt'yi çalıştırın./lfs.py -n20 -s 1 -c -v sonuçları görmek için. Cevaplarınızın

olup olmadığını görmek için -o ile çalıştırın rastgele komutlar dizisi göz önüne alındığında eşleştirin. Farklı rastgele kullanındaha fazla sorun almak için tohumlar. misin?

```
┌──(root㉿kali)-[~/PycharmProjects/lfs]
└─# tt ./lfs.py -n 20 -s 1
/usr/bin/tt:51:in `<main>': invalid option: -n (OptionParser::InvalidOption)
/usr/bin/tt:51:in `<main>': invalid option: n (OptionParser::InvalidOption)

┌──(root㉿kali)-[~/PycharmProjects/lfs]
└─# tt ./lfs.py -n 20 -s 1 -c -v
/usr/bin/tt:51:in `<main>': invalid option: -n (OptionParser::InvalidOption)
/usr/bin/tt:51:in `<main>': invalid option: n (OptionParser::InvalidOption)

┌──(root㉿kali)-[~/PycharmProjects/lfs]
└─# tt ./lfs.py -n 20 -s 1 -c -v -o
/usr/bin/tt:51:in `<main>': invalid option: -n (OptionParser::InvalidOption)
/usr/bin/tt:51:in `<main>': invalid option: n (OptionParser::InvalidOption)

┌──(root㉿kali)-[~/PycharmProjects/lfs]
└─# Etkin dosyaları ve dizinleri bulmak için son ımapye bakmak gerekir.
```

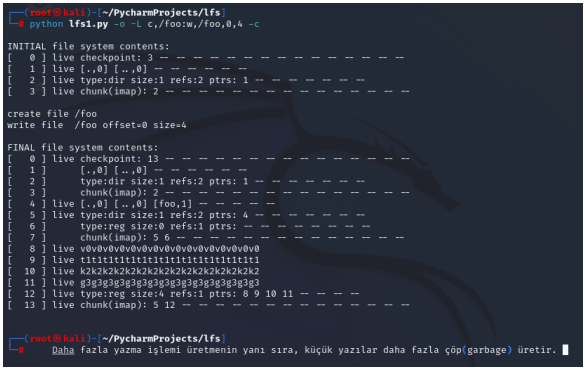
Etkin dosyaları ve dizinleri bulmak için son ımap'ye bakmak gerekir.

5. Şimdi bazı özel komutlar verelim. İlk önce bir dosya oluşturalım ve tekrar tekrar yaz. Bunu yapmak için, aşağıdakileri yapmanızı sağlayan -L bayrağını kullanın yürütülecek belirli komutları belirtin. ”/foo” dosyasını oluşturalım ve ona dört kez yaz: -L c, /foo:w, /foo, 0, 1:w, /foo, 1, 1:w, /foo, 2, 1:w, /foo, 3, 1-o . Son dosya sisteminin canlılığını belirleyip belirleyemeyeceğinize bakın durum; Cevaplarınızı kontrol etmek için -c kullanın.

```
┌──(root㉿kali)-[~/PycharmProjects/lfs]
└─# python lfs1.py -o -L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:w,/foo,3,1

INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

create file /foo
write file  /foo offset=0 size=1
write file  /foo offset=1 size=1
write file  /foo offset=2 size=1
write file  /foo offset=3 size=1

FINAL file system contents:
[   0 ] ?    checkpoint: 19 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] ?    [.,0] [..,0] -- -- -- -- -- --
[   2 ] ?    type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] ?    chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] ?    [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ] ?    type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] ?    chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] ?    v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ] ?    type:reg size:1 refs:1 ptrs: 8 -- -- -- -- -- -- --
[  10 ] ?    chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- --
[  11 ] ?    t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0
[  12 ] ?    type:reg size:2 refs:1 ptrs: 8 11 -- -- -- -- -- --
[  13 ] ?    chunk(imap): 5 12 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  14 ] ?    k0k0k0k0k0k0k0k0k0k0k0k0k0k0k0k0
[  15 ] ?    type:reg size:3 refs:1 ptrs: 8 11 14 -- -- -- -- --
[  16 ] ?    chunk(imap): 5 15 -- -- -- -- -- -- -- -- -- -- -- -- --
[  17 ] ?    g0g0g0g0g0g0g0g0g0g0g0g0g0g0g0g0
[  18 ] ?    type:reg size:4 refs:1 ptrs: 8 11 14 17 -- -- -- --
[  19 ] ?    chunk(imap): 5 18 -- -- -- -- -- -- -- -- -- -- -- -- --


┌──(root㉿kali)-[~/PycharmProjects/lfs]
└─# Blok 0,4,5,8,11,14,17,18,19 yayında. Yine son ımap yardımcı olur.
```

Blok 0,4,5,8,11,14,17,18,19 yayında. Yine son ımap yardımcı olur.

6. Şimdi, aynı şeyi yapalım, ancak bunun yerine tek bir yazma işlemiyle dört kişiden. Çalıştırın: /lfs.py -o -L c, /foo: w, /foo, 0, 4 için "/ foo" dosyası oluşturun ve tek bir yazma işlemiyle 4 blok yazın. Canlılığı tekrar hesaplayın ve -c ile doğru olup olmadığınızı kontrol edin. Bir dosyayı bir kerede yazmak arasındaki temel fark nedir (örneğin burada yapıyoruz) her seferinde bir blok yapmaya karşı (yukarıdaki gibi)?. Ne bu size güncellemelerin ana olarak arabelleğe alınmasının önemini anlatıyor mu gerçek LFS'nin yaptığı gibi hafıza mı?

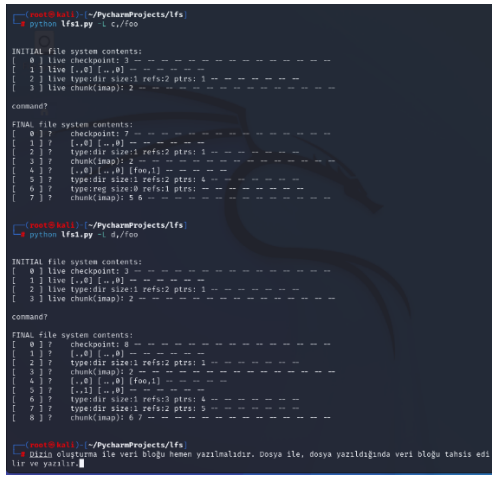


Daha fazla yazma işlemi üretmenin yanı sıra, küçük yazılar daha fazla çöp(garbage) üretir.

7. Başka bir özel örnek yapalım. İlk önce aşağıdakileri çalıştırın: ./lfs.py -L c, /foo: w, /foo, 0,1. Bu komut seti ne yapar? Şimdi, çalıştırın: ./lfs.py -L, c, /foo: w, /foo, 7,1. Bu ne anlama geliyor komutlar kümesi yapar mı? İkisi nasıl farklı? Ne söyleyebilirsin bu iki komut kümesinden inode'daki boyut alanı hakkında?

Daha sonra, boyutu inode ofseti+yazma boyutunda yapan bir ofsetle yazar.

8. Şimdi açıkça dosya oluşturmaya ve dizin oluşturmaya bakalım. Bir dosya ve ardından bir dizin oluşturmak için simülasyonları çalıştırın: /lfs.py -L c, /foo ve ./lfs.py -L d, /foo. Bu koşularda benzer olan nedir ve farklı olan nedir?

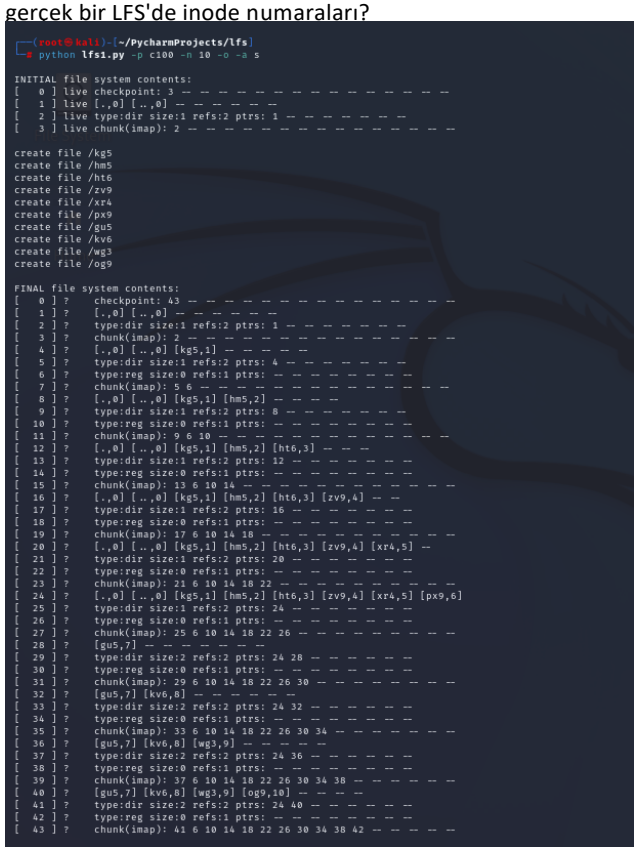


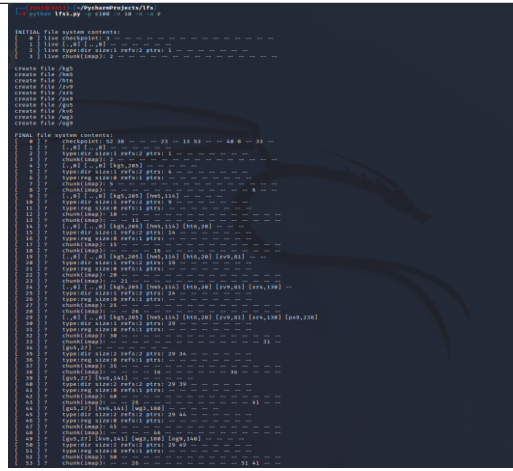
Dizin oluşturma ile veri bloğu hemen yazılmalıdır. Dosya ile, dosya yazıldığında veri bloğu tahsis edilir ve yazılır.

9. LFS simülatörü sabit bağlantıları da destekler. Nasıl çalıştıklarını incelemek için aşağıdakileri çalıştırın: ./lfs.py -L c, /foo: l, /foo, / bar:l, /foo, /goo -o -i. Bir sabit bağlantı oluşturulduğunda hangi bloklar yazılır? Nasıl olduğunu bu sadece yeni bir dosya oluşturmaya benzer ve nasıl farklı? Bağlantılar oluşturuldukça referans sayısı alanı nasıl değişir?

Her iki durumda da, üst dizine yeni bir girdi eklenir. Ancak bir bağlantı ile yeni inode'lara gerek yoktur. Eski inode güncellenmeli ve böylece yeni ref sayısı ile kopyalanmalıdır.

10. LFS birçok farklı politika kararı verir. Birçoğunu burada keşfetmiyoruz - belki de gelecek için kalan bir şey - ama burada keşfettiğimiz basit bir şey var: inode numarası seçimi. İlk önce çalıştırın: ./lfs.py -p c100 -n 10 -o -a s olağan davranışı göstermek için ücretsiz kullanmaya çalışan "sıralı" tahsis politikası ile sıfıra en yakın inode numaraları. Ardından, "rastgele" bir ilkeye geçin ve çalıştırın: ./lfs.py -p c100 -n 10 -o -a r (bu -p c100 bayrak, rastgele işlemlerin yüzde 100'ünün dosya oluşturma olmasını sağlar.) Rasgele bir ilke, sıralı bir ilke ile disk üzerindeki farklar nelerdir politika sonuçlandı mı? Bu, seçimin önemi hakkında ne diyor gerçek bir LFS'de inode numaraları?





Rastgele bir politika daha fazla parçalanma yaratır.

11. Varsaydığımız son bir şey, LFS simülatörü daima her güncellemeden sonra kontrol noktası bölgesini güncellemesidir. Gerçek LFS'de durum böyle değildir: uzun aramalardan kaçınmak için periyodik olarak güncellenir. Çalıştırın: ./lfs.py -N -i -o -s 1000. bazı işlemleri görmek için ve dosya sisteminin ara ve son durumları ne zaman denetim noktası bölgesi diske zorlanmaz. Denetim noktası bölgesi hiçbir zaman güncellenmezse ne olur? Periyodik olarak güncellenirse ne olur? Günlükte ilerleyerek dosya sistemini en son duruma nasıl kurtaracağınızı bulabilir misiniz?