43

Günlük yapılandırılmış 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 olarak bilinen yeni bir dosya sistemi geliştirdi [RO91]. Bunu yapmak için motivasyonları aşağıdaki gözlemlere dayanıyordu:

- **Sistem bellekleri büyüyor (System memories are growing):** Bellek büyüdükçe, bellekte daha fazla veri önbelleğe alınabilir. Daha fazla veri önbelleğe alındıkça, okumalar önbellek tarafından servis edildiği için disk trafiği giderek artan bir şekilde yazmalardan oluşur. Bu nedenle, dosya sistemi performansı büyük ölçüde yazma performansı tarafından belirlenir.
- **Rastgele G/Ç performansı ile sıralı G/Ç performansı arasında büyük bir fark vardır (There is large gap between random I/O performance and sequential I/O performance):** Sabit sürücü aktarım bant genişliği yıllar içinde büyük ölçüde artmıştır [P98]; bir sürücünün yüzeyine daha fazla bit yerleştirildikçe, söz konusu bitlere erişim sırasındaki bant genişliği de artmaktadır. Ancak arama ve dönme gecikmesi maliyetleri yavaşça 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, arama ve döndürmeye neden olan yaklaşımlara göre büyük bir performans avantajı elde edersiniz.
- **Mevcut dosya sistemleri birçok yaygın iş yükünde düşük performans göstermektedir (Existing file systems perform poorly on many common workloads):** Örneğin, FFS [MJLF84] bir blok büyüklüğünde yeni bir dosya oluşturmak için çok sayıda yazma işlemi gerçekleştirir: biri yeni bir düğüm için, biri düğüm bitmapini güncellemek için, biri dosyanın içinde bulunduğu dizin veri bloğuna, biri dizin düğümünü güncellemek için, biri yeni dosyanın bir parçası olan yeni veri bloğuna ve biri de veri bloğunu tahsis edilmiş olarak işaretlemek için veri bitmapine. Bu nedenle, FFS tüm bu blokları aynı blok grubuna yerleştirse de, FFS birçok kısa aramaya ve ardından

rotasyonel gecikmelere neden olur ve bu nedenle performans, en yüksek sıralı bant genişliğinin çok altında kalır.

- **Dosya sistemleri RAID ile uyumlu değildir(File systems are not RAID-aware):** Örneğin, hem RAID-4 hem de RAID-5, tek bir bloğa

mantıksal yazmanın 4 fiziksel I/O'nun gerçekleşmesine neden olduğu küçük yazma sorununa **(small-write problem)**sahiptir. Mevcut dosya sistemleri bu en kötü durum RAID yazma davranışından kaçınmaya çalışmaz.

İPUCU : ÖNEMLİ DETAYLAR

Tüm ilginç sistemler birkaç genel fikir ve bir dizi ayrıntıdan oluşur. Bazen bu sistemleri öğrenirken kendi kendinize "Oh, genel fikri anladım; gerisi sadece detaylar" diye düşünürsünüz ve bunu işlerin gerçekten nasıl yürüdüğünü sadece yarım yamalak öğrenmek için kullanırsınız. Bunu yapmayın! Çoğu zaman ayrıntılar kritik önem taşır. 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 genişliğinden faydalanmaya çalışır. Ayrıca, yalnızca veri yazmakla kalmayıp aynı zamanda disk üzerindeki meta veri yapılarını da sık sık güncelleyen yaygın iş yüklerinde iyi performans gösterecektir. Son olarak, RAID'lerin yanı sıra tek diskler üzerinde de iyi çalışacaktır.

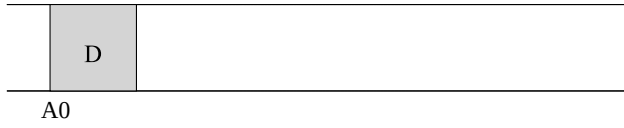
Rosenblum ve Ousterhout'un tanıttığı yeni dosya sistemi türüne Günlük yapılandırılmış Dosya Sistemi'nin **(Log-structured File System)** kısaltması olan LFS**(LFS)** adı verildi. Diske yazarken, LFS önce tüm güncellemeleri (meta veriler dahil!) bellek içi bir segmentte arabelleğe alır; segment dolduğunda, diskin kullanılmayan bir bölümüne uzun ve sıralı bir aktarımla diske yazılır. LFS asla mevcut verilerin üzerine yazmaz, bunun yerine her zaman segmentleri boş konumlara yazar. Çünkü segmentler büyük olduğu için disk (veya RAID) verimli bir şekilde kullanılır ve dosya sisteminin performansı zirveye yaklaşır.

CAN ALICI NOKTA: TÜM YAZMALAR NASIL SIRALI YAZMALAR HALİNE GETİRİLİR?

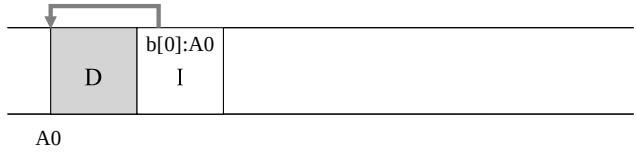
Bir dosya sistemi tüm yazmaları nasıl sıralı yazmalara dönüştürebilir? Okumalar için bu görev imkansızdır çünkü okunmak istenen blok disk üzerinde herhangi bir yerde olabilir. Ancak yazmalar için dosya sisteminin her zaman bir seçeneği vardır ve biz de tam olarak bu seçenekten yararlanmayı umuyoruz.

43.1 Diske Sıralı Olarak Yazma (Writing To Disk Sequentially)

Böylece ilk zorluğumuz ortaya çıkıyor: dosya sistemi durumundaki tüm güncellemeleri diske bir dizi sıralı yazmaya nasıl dönüştürebiliriz? Bunu daha iyi anlamak için basit bir örnek kullanalım. Bir dosyaya bir D veri bloğu yazdığımızı düşünün. Veri bloğunun diske yazılması, D'nin A0 disk adresine yazılmasıyla aşağıdaki disk içi düzenle sonuçlanabilir:



Ancak, bir kullanıcı bir veri bloğu yazdığında, diske yazılan yalnızca veri değildir; güncellenmesi gereken başka **meta veriler(metadata)** de vardır. Bu durumda, dosyanın dizin **düğümünü (inode)** (I) de diske yazalım ve bunun D veri bloğunu göstermesini sağlayalım. Diske yazıldığında, veri bloğu ve dizin düğümü aşağıdaki gibi görünecektir (dizin düğümü veri bloğu kadar büyük göründüğüne dikkat edin, ki bu genellikle böyle değildir; Çoğu sistemde veri blokları 4 KB boyutundayken bir dizin düğümü çok daha küçüktür, yaklaşık 128 bayt):



Tüm güncellemelerin (veri blokları, dizin düğümleri vb.) sırayla diske yazılması şeklindeki bu temel fikir, LFS'nin kalbinde yer alır. Bunu anlarsanız, temel fikri de anlamış olursunuz. Ancak tüm karmaşık sistemlerde olduğu gibi, şeytan ayrıntıda gizlidir.

43.2 Sıralı ve Etkili Yazma (Writing Sequentially And Effectively)

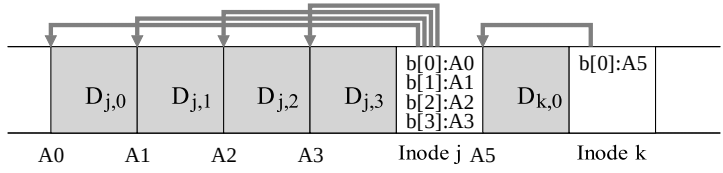
Ne yazık ki, diske sırayla yazmak verimli yazmayı garanti etmek için (tek başına) yeterli değildir. Örneğin, T zamanında A adresine tek bir blok yazdığımızı düşünün. Daha sonra bir süre bekleriz ve diske A + 1 adresine (sıralı düzende bir sonraki blok adresi), ancak T + δ zamanında yazarız. Birinci ve ikinci yazmalar arasında, ne yazık ki disk dönmüştür; ikinci yazmayı yayınladığınızda, bu nedenle işlenmeden önce bir dönüşün çoğunu bekleyecektir (özellikle, dönüş $T_{rotation}$ zamanını alırsa, disk ikinci yazmayı disk yüzeyine işlemeden önce $T_{rotation} - \delta$ bekleyecektir). Böylece, diske yalnızca sırayla yazmanın en yüksek performansı elde etmek için yeterli olmadığını, bunun yerine iyi bir yazma performansı elde etmek için sürücüye

çok sayıda bitişik yazma (veya tek bir büyük yazma) yapmanız gerektiğini umarım görebilirsiniz.

Bu amaca ulaşmak için LFS, **ara belleğe yazma (write buffering)** [1] olarak bilinen eski bir teknik kullanır. LFS, diske yazmadan önce güncellemeleri bellekte takip eder; yeterli sayıda güncelleme aldığında, hepsini bir kerede diske yazar ve böylece diskin verimli kullanılmasını sağlar. LFS'nin tek seferde yazdığı büyük güncelleme yığınına **bölüm (segment)** adı verilir. Bu terim bilgisayar sistemlerinde aşırı kullanılsa da, burada sadece LFS'nin yazmaları gruplamak için kullandığı büyük bir yığın anlamına gelir. Böylece, diske yazarken, LFS güncellemeleri bellek içi bir segment'e yazar

ve ardından bu segment'i bir kerede diske yazar. Segment yeterince büyük olduğu sürece, bu yazmalar verimli olacaktır.

Burada LFS'nin iki güncelleme setini küçük bir segmentte tamponladığı bir örnek verilmiştir; gerçek segmentler daha büyüktür (birkaç MB).İlk güncellemede j dosyasına dört blok yazılır; ikincisinde ise k dosyasına bir blok eklenir. LFS daha sonra yedi blokluk segmentin tamamını tek seferde diske işler. Bu blokların disk üzerinde ortaya çıkan düzeni aşağıdaki gibidir:



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

Bu da şu soruyu gündeme getirmektedir: LFS diske yazmadan önce kaç güncellemeyi arabelleğe almalıdır? Elbette cevap diskin kendisine, özellikle de konumlandırma ek yükünün aktarım hızına kıyasla ne kadar yüksek olduğuna bağlıdır; benzer bir analiz için FFS bölümüne bakın. Örneğin, her yazma işleminden önce konumlandırmanın (yani döndürme ve arama işlemlerinin) kabaca $T_{position}$ saniye sürdüğünü varsayın. Ayrıca disk aktarım hızının R_{peak} MB/s olduğunu varsayalım. Böyle bir disk üzerinde çalışırken LFS yazmadan önce ne kadar tamponlama yapmalıdır?

Bunu düşünmenin yolu, her yazdığınızda, konumlandırma maliyetinin sabit bir ek yükünü ödediğinizdir. Dolayısıyla, bu maliyeti amorti**(amortize)** etmek için ne kadar yazmanız gerekir? Ne kadar çok yazarsanız, o kadar iyi ve en yüksek bant genişliğine ulaşmaya o kadar

[1] Aslında, bu fikir için iyi bir alıntı bulmak zordur, çünkü muhtemelen birçok kişi tarafından ve bilgisayar tarihinin çok erken dönemlerinde icat edilmiştir. Yazma tamponlamasının faydaları hakkında bir çalışma için Solworth ve Orji'ye [SO90]; potansiyel zararları hakkında bilgi edinmek için Mogul'a [M94] bakın.

yakın olursunuz.

Somut bir cevap elde etmek için, D MB yazdığımızı varsayalım. Bu veri yığınını yazma süresi (T_{write}), konumlandırma süresi Tposition artı aktarma süresidir,), D ($\frac{D}{R_{peak}}$ veya:

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

Ve böylece, yazılan veri miktarının toplam yazma süresine bölünmesiyle elde edilen etkin yazma oranı ($R_{effective}$) elde edilir:

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

İlgilendiğimiz şey, efektif oranı ($R_{effective}$) en yüksek orana yaklaştırmaktır. Özellikle, etkin oranın, 0 < F < 1 (tipik bir F 0,9 veya en yüksek oranın %90'ı olabilir) olmak üzere, en yüksek oranın bir F kesri olmasını istiyoruz. Matematiksel formda bu $R_{effective} = F \times R_{peak}$ istediğimiz anlamına gelir.

Bu noktada, D'yi çözebiliriz:

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

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

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

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

Örnek olarak, konumlandırma süresi 10 milisaniye ve en yüksek aktarım hızı 100 MB/s olan bir diskle çalışalım; en yüksek hızın %90'ı kadar etkin bir bant genişliği istediğimizi varsayalım (F = 0,9). Bu durumda , $D = \frac{0.9}{0.1} \times$ 100 *MB/s* × 0.01 saniye = 9 *MB*. En yüksek bant genişliğine yaklaşmak için ne kadar arabelleğe almamız gerektiğini görmek için bazı farklı değerler deneyin. Zirvenin %95'ine ulaşmak için ne kadar gerekir? 99%?

43.4 Sorun: Inode'ları Bulma

LFS'de bir dizi düğümünü nasıl bulduğumuzu anlamak için, tipik bir UNIX dosya sisteminde bir düğümü nasıl bulunduğunu kısaca gözden geçirelim. FFS gibi tipik bir dosya sisteminde, hatta eski UNIX dosya sisteminde, dizi düğümlerini bulmak kolaydır, çünkü bunlar bir dizi halinde düzenlenir ve disk üzerinde sabit konumlara yerleştirilir.

Örneğin, eski UNIX dosya sistemi tüm düğümleri diskin sabit bir bölümünde tutar. Böylece, bir dizi düğümün numarası ve başlangıç adresi verildiğinde, belirli bir dizi düğümünü bulmak için, dizin düğüm numarasını bir düğümün boyutuyla çarparak ve bunu disk dizisinin başlangıç adresine ekleyerek tam disk adresini hesaplayabilirsiniz; bir düğüm numarası verildiğinde dizi tabanlı indeksleme hızlı ve basittir.

FFS'de bir düğüm numarası verilen bir düğümü bulmak sadece biraz daha karmaşıktır, çünkü FFS düğüm tablosunu parçalara böler ve her silindir grubunun içine bir grup düğüm yerleştirir Bu nedenle, her bir düğüm yığınının ne kadar büyük olduğunu ve her birinin başlangıç adreslerini bilmek gerekir. Bundan sonra, hesaplamalar benzer ve aynı zamanda kolaydır.

43.5 Dolaylı Çözüm: Inode Haritası

Bunu düzeltmek için LFS tasarımcıları, **düğüm haritası (inode map)** (imap) adı verilen bir veri yapısı aracılığıyla düğüm numaraları ve düğümler arasında bir **dolaylama düzeyi (level of indirection)** getirmiştir. Imap, girdi olarak bir düğüm numarası alan ve düğümlerin en son sürümünün disk adresini üreten bir yapıdır.

IPUCU: BIR DOLAYLAMA DÜZEYI KULLANIN

İnsanlar genellikle Bilgisayar Bilimlerindeki tüm sorunların çözümünün basitçe bir **dolaylama seviyesi (level of indirection)** olduğunu söylerler. Bu kesinlikle doğru değildir; bu sadece çoğu sorunun çözümüdür(evet, bu hala çok güçlü bir yorum, ama ne demek istediğimi anladınız). İncelediğimiz her sanallaştırmayı, örneğin sanal bellek ya da dosya kavramını, basitçe bir dolaylama düzeyi olarak düşünebilirsiniz.Ve kesinlikle LFS'deki düğüm haritası düğüm numaralarının sanallaştırılmasıdır. Umarım bu örneklerde dolaylamanın büyük gücünü görebilirsiniz, bu da yapıları (VM örneğindeki sayfalar veya LFS'deki düğümler gibi) her referansı değiştirmek zorunda kalmadan serbestçe hareket ettirmemizi sağlar. Elbette dolaylamanın da bir dezavantajı olabilir: **fazladan ek yük (extre overhead).** Bu nedenle, bir dahaki sefere bir sorunla karşılaştığınızda, bunu dolaylama ile çözmeyi deneyin, ancak önce bunu yapmanın genel giderlerini düşündüğünüzden emin olun. Wheeler'ın ünlü sözünde dediği gibi, "Bilgisayar bilimindeki tüm sorunlar başka bir dolaylama seviyesiyle çözülebilir, tabii ki çok fazla dolaylama sorunu hariç."

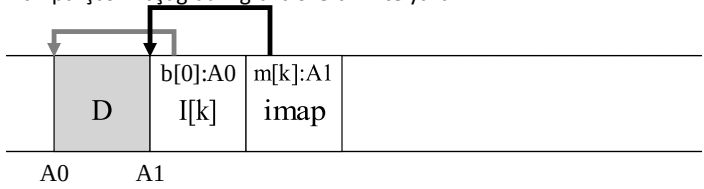
Bu nedenle, genellikle giriş başına 4 bayt (bir disk işaretçisi) ile basit bir dizi olarak uygulanacağını hayal edebilirsiniz. Bir düğüm diske her yazıldığında, imap yeni konumuyla güncellenir.

Ne yazık ki, imap'in kalıcı tutulması (yani diske yazılması) gerekir; bunu yapmak LFS'nin çökmeler boyunca düğümlerin konumlarını takip etmesini ve

böylece istenen şekilde çalışmasını sağlar. Bu nedenle, bir soru: imap disk üzerinde nerede bulunmalıdır?

Elbette diskin sabit bir bölümünde yaşayabilir. Ne yazık ki, sık sık güncellendiği için, bu durum dosya yapılarındaki güncellemeleri imap'e yazmaların takip etmesini gerektirecek ve dolayısıyla performans düşecektir (yani, her güncelleme ile imap'in sabit konumu arasında daha fazla disk araması olacaktır).

Bunun yerine LFS, düğüm haritasının parçalarını diğer tüm yeni bilgileri yazdığı yerin hemen yanına yerleştirir. Bu nedenle, bir k dosyasına bir veri bloğu eklerken, LFS aslında yeni veri bloğunu, düğümünü ve düğüm haritasının bir parçasını aşağıdaki gibi diske birlikte yazar:



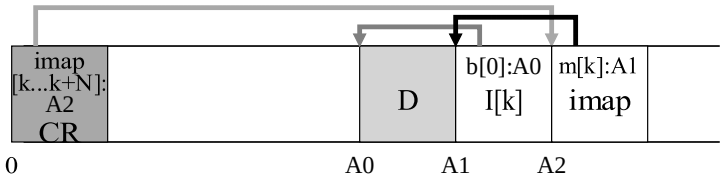
Bu resimde, imap dizisinin imap olarak işaretlenmiş blokta saklanan parçası LFS'ye k düğümünün A1 disk adresinde olduğunu söyler; bu düğüm de LFS'ye D veri bloğunun A0 adresinde olduğunu söyler.

43.6 Çözümün Tamamlanması: Kontrol Noktası Bölgesi

Zeki bir okuyucu (bu sizsiniz, değil mi?) burada bir sorun olduğunu fark etmiş olabilir. Artık parçaları da diske yayıldığına göre düğüm haritasını nasıl bulacağız? Sonuçta, sihirli bir şey yoktur: dosya sistemi, bir dosya aramaya başlamak için disk üzerinde sabit ve bilinen bir konuma sahip olmalıdır.

LFS'de bunun için disk üzerinde **kontrol noktası bölgesi (checkpoint region) (CR)** olarak bilinen sabit bir yer vardır. Kontrol noktası bölgesi, düğüm haritasının en son parçalarının işaretçilerini (yani adreslerini) içerir ve böylece düğüm haritası parçaları önce CR okunarak bulunabilir. Kontrol noktası bölgesinin yalnızca periyodik olarak güncellendiğini (örneğin her 30 saniyede bir) ve bu nedenle performansın kötü etkilenmediğini unutmayın. Böylece, disk üzerindeki yerleşimin genel yapısı bir kontrol noktası bölgesi içerir (düğüm haritasının en son parçalarına işaret eder); düğüm haritası parçalarının her biri düğümlerin adreslerini içerir; düğümler tipik UNIX dosya sistemlerinde olduğu gibi dosyalara (ve dizinlere) işaret eder.

İşte kontrol noktası bölgesinin bir örneği (diskin en başında, 0 adresinde olduğuna dikkat edin) ve tek bir imap yığını, düğüm ve veri bloğu. Gerçek bir dosya sistemi elbette çok daha büyük bir CR'ye (aslında, daha sonra anlayacağımız gibi iki tane olacaktır), birçok imap parçasına ve elbette çok daha fazla düğüme, veri bloğuna vb. sahip olacaktır.



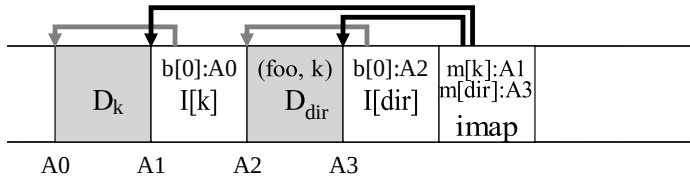
43.7 Diskten Dosya Okuma: Özet

LFS'nin nasıl çalıştığını anladığınızdan emin olmak için, şimdi bir dosyayı diskten okumak için ne olması gerektiğini gözden geçirelim. Başlangıç için hafızamızda hiçbir şey olmadığını varsayalım. Disk üzerinde okumamız gereken ilk veri yapısı kontrol noktası bölgesidir. Kontrol noktası bölgesi tüm düğüm haritasına işaretçiler (yani disk adresleri) içerir ve bu nedenle LFS daha sonra tüm düğüm haritasını okur ve bellekte önbelleğe alır. Bu noktadan sonra, bir dosyanın düğüm numarası verildiğinde, LFS basitçe imap'teki düğüm numarası ile düğüm disk adresi eşlemesine bakar ve düğümün en son sürümünü okur. Bu noktada, dosyadan bir blok okumak için LFS, gerektiğinde doğrudan işaretçiler veya dolaylı işaretçiler veya çift dolaylı işaretçiler kullanarak tipik bir UNIX dosya sistemi gibi ilerler. Genel durumda, LFS bir dosyayı diskten okurken tipik bir dosya sistemiyle aynı sayıda G/Ç gerçekleştirmelidir; imap'in tamamı önbelleğe alınır ve bu nedenle LFS'nin okuma sırasında yaptığı ekstra iş, imap'te inode'un adresini aramaktır.

43.8 Peki ya Dizinler?

Şimdiye kadar sadece düğümleri ve veri bloklarını ele alarak tartışmamızı biraz basitleştirdik. Ancak, bir dosya sistemindeki bir dosyaya erişmek için (favori sahte dosya isimlerimizden biri olan /home/remzi/foo gibi), bazı dizinlere de erişilmesi gerekir. Peki LFS dizin verilerini nasıl saklar?

Neyse ki, dizin yapısı temelde klasik UNIX dosya sistemleriyle aynıdır, çünkü bir dizin sadece (isim, düğüm numarası) eşlemelerinin bir koleksiyonudur. Örneğin, disk üzerinde bir dosya oluştururken, LFS hem yeni bir düğüm, hem bazı veriler, hem de bu dosyaya atıfta bulunan dizin verileri ve düğümlerini yazmalıdır. LFS'nin bunu disk üzerinde sırayla yapacağını unutmayın (güncellemeleri bir süre tamponladıktan sonra). Böylece, bir dizinde bir foo dosyası oluşturmak diskte aşağıdaki yeni yapılara yol açacaktır:



Düğüm haritasının parçası hem dir dizin dosyasının hem de yeni oluşturulan f dosyasının konum bilgisini içerir. Böylece, foo dosyasına (düğüm numarası k olan) erişirken, önce dir dizininin düğümünün konumunu bulmak için düğüm haritasına (genellikle bellekte önbelleğe alınır) bakarsınız (A3); daha sonra dizin düğümünü okursunuz, bu da size dizin verilerinin konumunu verir (A2); bu veri bloğunu okumak size (foo, k) isim-düğüm numarası eşlemesini verir. Daha sonra k numaralı düğümün (A1) yerini bulmak için düğüm haritasına tekrar başvurursunuz ve son olarak A0 adresindeki istenen veri bloğunu okursunuz.

LFS'de düğüm haritasının çözdüğü, **özyinelemeli güncelleme problemi (recursive update problem)** olarak bilinen bir başka ciddi problem daha vardır [Z+12]. Sorun, hiçbir zaman yerinde güncelleme yapmayan (LFS gibi), bunun yerine güncellemeleri diskteki yeni konumlara taşıyan herhangi bir dosya sisteminde ortaya çıkar.

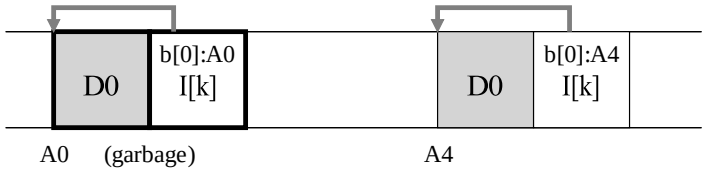
Özellikle, bir düğüm güncellendiğinde, diskteki konumu değişir. Eğer dikkatli olmasaydık, bu aynı zamanda bu dosyayı işaret eden dizinin de güncellenmesini gerektirecekti, bu da o dizinin üst dizininde bir değişiklik yapılmasını zorunlu kılacaktı ve bu şekilde dosya sistemi ağacında yukarıya doğru devam edecekti.

LFS, bölüm haritası ile bu sorunu akıllıca önler. Bir düğümün konumu değişse bile, değişiklik asla dizinin kendisine yansıtılmaz; bunun yerine, dizin aynı ad-bölüm numarası eşlemesini tutarken imap yapısı güncellenir. Böylece, dolaylama yoluyla, LFS özyinelemeli güncelleme sorununu önler.

43.9 Yeni Bir Sorun: Çöp Toplama

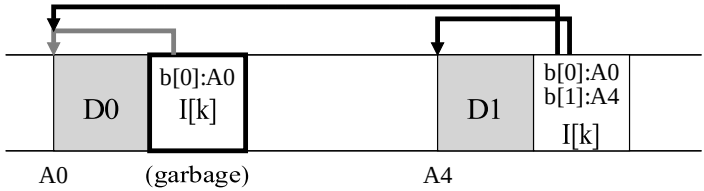
LFS ile ilgili başka bir sorunu fark etmiş olabilirsiniz; bir dosyanın en son sürümünü (düğüm ve verileri dahil) sürekli olarak diskteki yeni konumlara yazar. Bu işlem, yazma işlemini verimli kılarken, LFS'nin dosya yapılarının eski sürümlerini diske dağılmış halde bıraktığı anlamına gelir. Bu eski sürümleri (oldukça kaba bir şekilde) **çöp (garbage)** olarak adlandırıyoruz.

Örneğin, tek bir D0 veri bloğuna işaret eden k düğüm numarasıyla anılan mevcut bir dosyamız olduğunu düşünelim. Şimdi bu bloğu güncelleyerek hem yeni bir düğüm hem de yeni bir veri bloğu oluşturuyoruz. Sonuçta ortaya çıkan LFS'nin disk üzerindeki düzeni aşağıdaki gibi olacaktır (basitlik için imap ve diğer yapıları atladığımıza dikkat edin; yeni düğümü işaret etmek için diske yeni bir imap yığınının da yazılması gerekecektir):



Diyagramda,hem düğüm hem de veri bloğunun diskte biri eski (soldaki) ve diğeri güncel ve dolayısıyla **canlı (live)** (sağdaki) olmak üzere iki sürümü olduğunu görebilirsiniz. Bir veri bloğunun (mantıksal olarak) basit bir şekilde güncellenmesiyle, bir dizi yeni yapı LFS tarafından kalıcı hale getirilmeli, böylece söz konusu blokların eski sürümleri diskte bırakılmalıdır.

Başka bir örnek olarak, bunun yerine orijinal k dosyasına bir blok eklediğimizi düşünün. Bu durumda, düğümün yeni bir sürümü oluşturulur, ancak eski veri bloğu hala düğüm tarafından işaret edilir. Dolayısıyla, hala canlıdır ve mevcut dosya sisteminin bir parçasıdır:



Peki düğümlerin, veri bloklarının ve benzerlerinin bu eski sürümleriyle ne yapmalıyız? Bu eski sürümler saklanabilir 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 **sürümleme dosya sistemi (versioning file system)** olarak bilinir.

Ancak, LFS bunun yerine bir dosyanın yalnızca en son canlı sürümünü tutar; bu nedenle (arka planda), LFS dosya verilerinin, düğümlerinin ve diğer yapıların bu eski ölü sürümlerini periyodik olarak bulmalı ve temizlemelidir; **temizlik (clean)** böylece diskteki blokları sonraki yazmalarda kullanılmak üzere yeniden boş hale getirmelidir. Temizleme işleminin, kullanılmayan belleği programlar için otomatik olarak serbest bırakan programlama dillerinde ortaya çıkan bir teknik olan **çöp toplama (garbage collection)** işleminin bir biçimi olduğunu unutmayın.

Daha önce segmentlerin LFS'de diske büyük yazma işlemlerini mümkün kılan mekanizma olarak öneminden bahsetmiştik. Görünüşe göre, etkili temizlik için de oldukça ayrılmazlar. LFS temizleyicinin temizlik sırasında tek veri bloklarını, düğümlerini vb. basitçe gözden geçirip serbest bırakması durumunda ne olacağını hayal edin. Sonuç: diskte ayrılmış alanların arasına karışmış bir miktar boş boşluğa**(holes)** sahip 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 segment bazında çalışır, böylece sonraki yazma için büyük alan parçalarını temizler. Temel temizleme işlemi aşağıdaki gibi çalışır. Periyodik olarak, LFS temizleyici bir dizi eski (kısmen kullanılan) segmenti okur, bu segmentler içinde hangi blokların canlı olduğunu belirler ve ardından içinde sadece canlı bloklar bulunan yeni bir segment kümesi yazarak eskilerini yazma için serbest bırakır. Özellikle, temizleyicinin mevcut M segmenti okumasını, içeriklerini N yeni segmente **sıkıştırmasını (compact)** (burada N < M) ve ardından N segmenti yeni konumlarda diske yazmasını bekliyoruz. Eski M segmentleri daha sonra serbest bırakılır ve dosya sistemi tarafından sonraki yazmalar için kullanılabilir.

Ancak şimdi iki sorunla karşı karşıyayız. Birincisi mekanizma: LFS bir segment içindeki hangi blokların canlı, hangilerinin ölü olduğunu nasıl söyleyebilir? İkincisi ise politika: temizleyici ne sıklıkla çalışmalı ve temizlemek için hangi segmentleri seçmeli?

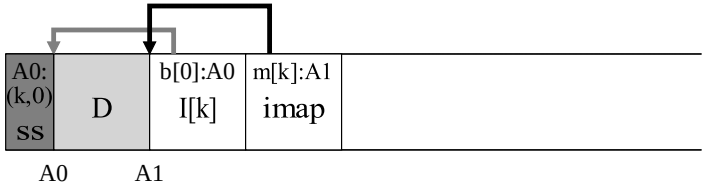
43.10 Blok Canlılığını Belirleme

İlk olarak mekanizmayı ele alıyoruz. Bir S disk segmenti içinde bir D veri bloğu verildiğinde, LFS'nin D'nin canlı olup olmadığını belirleyebilmesi gerekir. Bunu yapmak için, LFS her bloğu tanımlayan her segmente biraz ekstra bilgi ekler. Özellikle, LFS her veri bloğu D için düğüm numarasını (hangi dosyaya ait olduğu) ve ofsetini (dosyanın hangi bloğu olduğu) içerir. Bu bilgiler segmentin başında bulunan ve **segment özeti bloğu (segment summary block)** olarak bilinen 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 özet bloğuna bakın ve N düğüm numarasını ve T ofsetini bulun. Ardından, N'nin nerede yaşadığını bulmak için imap'e bakın ve N'yi diskten okuyun (belki de zaten bellektedir, bu daha da iyidir). Son olarak, T ofsetini kullanarak, düğümünün bu dosyanın T. bloğunun diskte nerede olduğunu düşündüğünü görmek için düğüme (veya dolaylı bir bloğa) bakın.Tam olarak A disk adresini gösteriyorsa, LFS D bloğunun canlı olduğu sonucuna varabilir. Başka bir yeri işaret ediyorsa, LFS D'nin kullanımda olmadığı (yani ölü olduğu) sonucuna varabilir ve böylece bu sürüme artık ihtiyaç olmadığını bilir. İşte bir sözde kod özeti:

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // D bloğu yaşıyor

else

   // D bloğu çöp
```

Burada, segment özet bloğunun (SS işaretli) A0 adresindeki veri bloğunun aslında 0 ofsetindeki k dosyasının bir parçası olduğunu kaydettiği mekanizmayı gösteren bir diyagram bulunmaktadır. k için imap'i kontrol ederek, düğümü bulabilir ve gerçekten de bu konuma işaret ettiğini görebilirsiniz.



LFS'nin canlılığı belirleme sürecini daha verimli hale getirmek için kullandığı bazı kısayollar vardır. Örneğin, bir dosya kesildiğinde veya silindiğinde, LFS **sürüm numarasını (version number)** artırır ve yeni sürüm numarasını imap'e kaydeder. LFS, disk üzerindeki segmente sürüm numarasını da kaydederek, sadece disk üzerindeki sürüm numarasını imap'teki bir sürüm numarasıyla karşılaştırarak yukarıda açıklanan daha uzun kontrolü kısa devre yapabilir ve böylece ekstra okumalardan kaçınabilir.

43.11 Bir Politika Sorusu: Hangi Bloklar Ne Zaman Temizlenmeli?

Yukarıda açıklanan mekanizmaya ek olarak, LFS hem ne zaman temizleneceğini hem de hangi blokların temizlenmeye değer olduğunu belirlemek için bir dizi ilke içermelidir. Ne zaman temizleyeceğinizi belirlemek daha kolaydır; periyodik olarak, boşta kaldığınızda ya da disk dolu olduğu için temizlemek zorunda olduğunuzda.

Hangi blokların temizleneceğini belirlemek daha zordur ve birçok araştırma makalesine konu olmuştur. Orijinal LFS makalesinde [RO91], yazarlar sıcak ve soğuk segmentleri ayırmaya çalışan bir yaklaşım tanımlamaktadır. Sıcak segment, içeriğin sık sık üzerine yazıldığı segmenttir; dolayısıyla, böyle bir segment için en iyi politika, temizlemeden önce uzun süre beklemektir, çünkü giderek daha fazla blok üzerine yazılmakta (yeni segmentlerde) ve böylece kullanım için serbest bırakılmaktadır. Bunun aksine, soğuk bir segmentte birkaç ölü blok olabilir ancak içeriğinin geri kalanı nispeten sabittir. Bu nedenle yazarlar soğuk segmentlerin daha erken,

sıcak segmentlerin ise daha geç temizlenmesi gerektiği sonucuna varmış ve tam olarak bunu yapan bir sezgisel geliştirmişlerdir. Ancak, çoğu politikada olduğu gibi, bu politika da mükemmel değildir; sonraki yaklaşımlar daha iyisinin nasıl yapılacağını göstermektedir [MR+97].

43.12 Çökme Kurtarma ve Günlük

Son bir sorun: LFS diske yazarken sistem çökerse ne olur? Günlük tutma ile ilgili önceki bölümde hatırlayabileceğiniz gibi, güncellemeler sırasında çökmeler dosya sistemleri için zordur ve bu nedenle LFS'nin de dikkate alması gereken bir şeydir.

Normal çalışma sırasında, LFS bir segmentteki yazıları arabelleğe alır ve ardından (segment dolduğunda veya belirli bir süre geçtiğinde) segmenti diske yazar. LFS bu yazmaları bir günlükte**(log)** 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. Bu işlemlerden herhangi biri sırasında (bir segmente yazma, CR'ye yazma) çökmeler meydana gelebilir. Peki LFS bu yapılara yazma işlemi sırasında meydana gelen çökmeleri nasıl ele alır?

Önce ikinci durumu ele alalım. CR güncellemesinin atomik olarak gerçekleşmesini sağlamak için LFS aslında biri diskin her iki ucunda olmak üzere iki CR tutar ve bunlara dönüşümlü olarak yazar. LFS ayrıca CR'yi düğüm haritasının ve diğer bilgilerin en son işaretçileriyle güncellerken dikkatli bir protokol uygular; özellikle, önce bir başlık (zaman damgasıyla), ardından CR'nin gövdesi ve son olarak son bir blok (yine zaman damgasıyla) yazar. CR güncellemesi sırasında sistem çökerse, LFS bunu tutarsız bir zaman damgası çifti görerek tespit edebilir. LFS her zaman tutarlı zaman damgalarına sahip en son CR'yi kullanmayı seçer ve böylece CR'nin tutarlı bir şekilde güncellenmesi sağlanır.

Şimdi ilk durumu ele alalım. LFS CR'yi her 30 saniyede bir yazdığından, dosya sisteminin son tutarlı anlık görüntüsü oldukça eski olabilir. Bu nedenle, yeniden başlatma sonrasında LFS sadece kontrol noktası bölgesini, işaret ettiği imap parçalarını ve sonraki dosya ve dizinleri okuyarak kolayca kurtarabilir; ancak son birkaç saniyelik güncellemeler kaybolacaktır.

Bunu iyileştirmek için LFS, veritabanı topluluğunda **ileri sarma (roll forward)** olarak bilinen bir teknikle bu segmentlerin çoğunu yeniden oluşturmaya çalışır. Temel fikir, son kontrol noktası bölgesiyle başlamak, günlüğün sonunu bulmak (CR'ye dahil olan) ve ardından bunu sonraki segmentleri okumak ve içinde geçerli güncellemeler olup olmadığına bakmak için kullanmaktır. Eğer varsa, LFS dosya sistemini buna göre günceller ve böylece son kontrol noktasından bu yana yazılan veri ve meta verilerin çoğunu kurtarır. Ayrıntılar için Rosenblum'un ödüllü tezine bakınız [R92].

43.13 Özet

LFS diskin güncellenmesine yeni bir yaklaşım getirmektedir. Yer yer dosyaların üzerine yazmak yerine, LFS her zaman diskin kullanılmayan bir bölümüne yazar ve daha sonra temizleme yoluyla bu eski alanı geri alır. Veritabanı sistemlerinde **gölge sayfalama (shadow paging)** [L77] olarak adlandırılan ve dosya sisteminde bazen **yazma üzerine kopyalama (copy-on-write)** olarak adlandırılan bu yaklaşım, LFS tüm güncellemeleri bellekteki bir segmentte toplayabildiği ve daha sonra bunları sırayla birlikte yazabildiği için yüksek verimli yazma sağlar.

İPUCU : HATALARINI ÖZELLİKLERİNE DÖNÜŞTÜR

Sisteminizin temel bir kusuru olduğunda, bunu bir özelliğe ya da faydalı bir şeye dönüştürebilir misiniz diye bakın. NetApp'ın WAFL'ı bunu eski dosya içerikleriyle yapıyor; eski sürümleri kullanılabilir hale getirerek, WAFL'ın artık sık sık temizleme konusunda endişelenmesine gerek kalmıyor (yine de eski sürümleri eninde sonunda arka planda siliyor) ve böylece harika bir özellik sağlıyor ve LFS temizleme sorununun çoğunu harika bir şekilde ortadan kaldırıyor. Sistemlerde bunun başka örnekleri de var mı? Kuşkusuz, ancak bunları kendiniz düşünmek zorunda kalacaksınız, çünkü bu bölüm büyük bir "O" ile bitti.

LFS'nin ürettiği büyük yazmalar birçok farklı cihazda performans açısından mükemmeldir. Sabit disklerde büyük yazımlar konumlandırma süresinin en aza indirilmesini sağlar; RAID-4 ve RAID-5 gibi parite tabanlı RAID'lerde ise küçük yazma sorununu tamamen ortadan kaldırır. Hatta son araştırmalar Flash tabanlı SSD'lerde yüksek performans için büyük G/Ç'lerin gerekli olduğunu göstermiştir [H+17]; bu nedenle, belki de şaşırtıcı bir şekilde, LFS tarzı dosya sistemleri bu yeni ortamlar için bile mükemmel bir seçim olabilir.

Bu yaklaşımın dezavantajı çöp üretmesidir; verilerin eski kopyaları diskin her yerine dağılmıştır ve sonraki kullanım için bu alanı geri kazanmak isteniyorsa, eski bölümleri periyodik olarak temizlemek gerekir. Temizlik, LFS'de birçok tartışmanın odağı haline geldi ve temizlik maliyetleri [SS+95] ile ilgili endişeler belki de LFS'nin alandaki ilk etkisini sınırladı. Ancak NetApp'ın **WAFL** [HLM94], Sun'ın **ZFS** [B07] ve Linux **btrfs** [R+13] gibi bazı modern ticari dosya sistemleri ve hatta modern **flash tabanlı (flash-based) SSD'ler** [AD14], diske yazmada benzer bir kopyalama yaklaşımını benimsemektedir ve dolayısıyla LFS'nin fikri mirası bu modern dosya sistemlerinde yaşamaya devam etmektedir. Özellikle WAFL, temizlik sorunlarını bir özelliğe dönüştürerek aştı; dosya sisteminin eski sürümlerini anlık görüntüler **(snapshots)** aracılığıyla sağlayarak, kullanıcılar 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 the chapter 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 Rosenblum 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, buffering for too long can be harmful: see Mogul [M94] for details.*

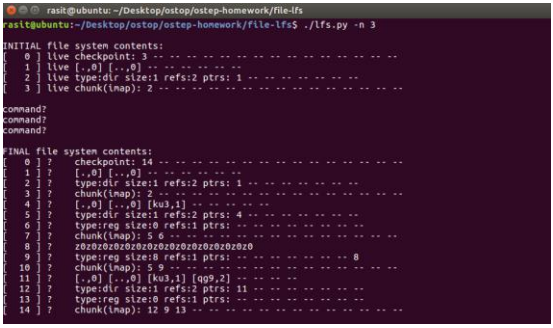
[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes" by Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '13, San Jose, California, February 2013. *Our paper on a new way to build flash-based storage devices, to avoid redundant mappings in the file 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.*

Ev Ödevi (Simülasyon)

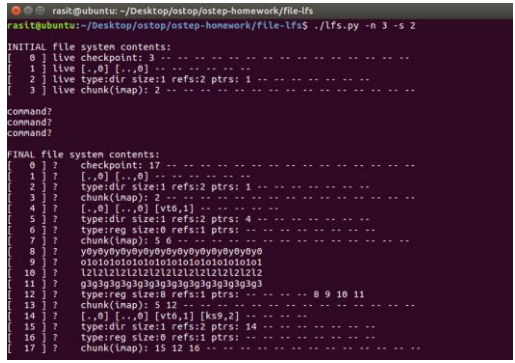
Bu bölümde, LFS tabanlı bir dosya sisteminin nasıl çalıştığını daha iyi anlamak için kullanabileceğiniz basit bir LFS simülatörü olan lfs.py tanıtılmaktadır. Simülatörün nasıl çalıştırılacağına ilişkin ayrıntılar için README'yi okuyun.

Sorular

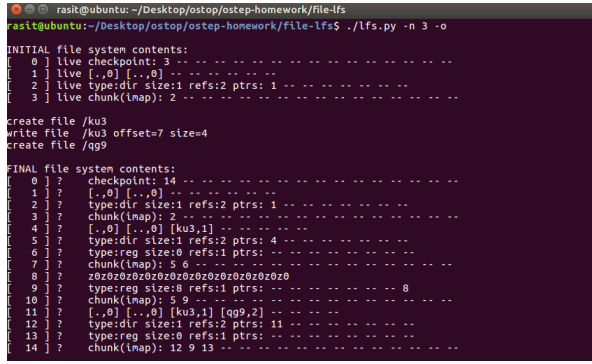
1. ./lfs.py -n 3 dosyasını çalıştırın, belki çekirdeği (-s) değiştirebilirsiniz. Son dosya sistemi içeriğini oluşturmak için hangi komutların çalıştırıldığını bulabilir misiniz? Bu komutların hangi sırayla verildiğini söyleyebilir misiniz? Son olarak, son dosya sistemi durumundaki her bloğun canlılığını belirleyebilir misiniz? Hangi komutların çalıştırıldığını göstermek için -o ve son dosya sistemi durumunun canlılığını göstermek için -c kullanın. Verilen komutların sayısını artırdıkça (örneğin -n 3'ü -n 5 olarak değiştirdiğinizde) görev sizin için ne kadar zorlaşıyor?



Kodumuzdaki -n 3 bayrağı kaç tane işlem yapacağımızı belirtiyor. Burada 3 tane işlem yapıyoruz. Checkpoint bizim kontrol noktamızdır (CR). Kontrol noktamız bize imapların bulunduğu konumu gösterir. Chunk(imap) bize inode'ların bulunduğu konumu gösterir. Inode'lar ise bizim data'mızı gösterir.



-s 2 bayrağı dosya sisteminin boyutunu bloklar halinde belirtir. Bu durumda her bloğun boyutu 2 bayttır. Bu bayrak, dosya sistemini belirtilen boyutta başlatmak için kullanılır. Kod parçasında, dosya sisteminin boyutu 6 bloktur. Bu bayrak olmadan, dosya sistemi boyutu varsayılan yapılandırmaya veya kullanıcı tarafından belirtilen seçeneklere göre belirlenir.



Kodların hangi sırayla çalıştığını görmek için -o bayrağını kullanıyoruz. Bu kod bloğunda, dosya sisteminde bazı işlemler yapıldığı gösterilmektedir. İlk olarak /ku3 adlı dosya oluşturulmuş, daha sonra bu dosyanın içine offset 7'de boyutu 4 olan veri yazılmış. En son /qg9 adlı dosya oluşturulmuş. Bu işlemlerin sonucunda dosya sistemi içeriğinin güncellendiğini görmekteyiz.

```
rasit@ubuntu: ~/Desktop/ostop/ostep-homework/file-lfs
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -n 3 -c

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 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

command?
command?
command?

FINAL file system contents:
[   0 ] live checkpoint: 14 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ]      [.,0] [..,0] [ku3,1] -- -- -- -- --
[   5 ]      type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] live z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0
[   9 ] live type:reg size:8 refs:1 ptrs: -- -- -- -- -- -- -- 8
[  10 ]      chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  11 ] live [.,0] [..,0] [ku3,1] [qg9,2] -- -- -- --
[  12 ] live type:dir size:1 refs:2 ptrs: 11 -- -- -- -- -- -- --
[  13 ] live type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  14 ] live chunk(imap): 12 9 13 -- -- -- -- -- -- -- -- -- -- -- -- --
```

Live kelimesinin anlamı, bu dosya sistemi içeriğindeki dosyaların ve dizinlerin canlı olduğunu, dosyaların kullanılabilir olduğunu belirmektedir. Bu canlı dosyaları görebilmek için -c bayrağı kullanılır. Burada canlı olan dosyalar [0,8,9,11,12,13,14], ölü olan dosyalar [1,2,3,4,5,6,7,10]. Ölü olan dosyalar çöp toplayıcısı tarafından silinir.

```
rasit@ubuntu: ~/Desktop/ostop/ostep-homework/file-lfs
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -n 5 -c -o

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 /ku3
write file  /ku3 offset=7 size=4
create file /qg9
link file   /qg9 /is8
create dir  /cl6

FINAL file system contents:
[   0 ] live checkpoint: 23 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ]      [.,0] [..,0] [ku3,1] -- -- -- -- --
[   5 ]      type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] live x0x0x0x0x0x0x0x0x0x0x0x0x0x0x0
[   9 ] live type:reg size:8 refs:1 ptrs: -- -- -- -- -- -- -- 8
[  10 ]      chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  11 ]      [.,0] [..,0] [ku3,1] [qg9,2] -- -- -- --
[  12 ]      type:dir size:1 refs:2 ptrs: 11 -- -- -- -- -- -- --
[  13 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  14 ]      chunk(imap): 12 9 13 -- -- -- -- -- -- -- -- -- -- -- -- --
[  15 ]      [.,0] [..,0] [ku3,1] [qg9,2] [is8,2] -- -- --
[  16 ]      type:dir size:1 refs:2 ptrs: 15 -- -- -- -- -- -- --
[  17 ] live type:reg size:0 refs:2 ptrs: -- -- -- -- -- -- -- --
[  18 ]      chunk(imap): 16 9 17 -- -- -- -- -- -- -- -- -- -- -- -- --
[  19 ] live [.,0] [..,0] [ku3,1] [qg9,2] [is8,2] [cl6,3] -- --
[  20 ] live [.,3] [..,0] -- -- -- -- -- --
[  21 ] live type:dir size:1 refs:3 ptrs: 19 -- -- -- -- -- -- --
[  22 ] live type:dir size:1 refs:2 ptrs: 20 -- -- -- -- -- -- --
[  23 ] live chunk(imap): 21 9 17 22 -- -- -- -- -- -- -- -- -- -- --
```

-n bayrağımızı 5 yaptığımızda, 5 tane dosya işlemi yapacağımızı gösterir. İlk başta /ku3 adlı bir dosya oluşturur, daha sonra write file /ku3 offset=7 size =4 komudu çalıştırıldığında, dosyaya 7 baytlık bir offsette dört bayt veri yazar. /qg9 adlı başka bir dosya oluşturur. Oluşturduktan sonra /qg9'u /is8 adlı bir dosyaya bağlar ve /cl6 adlı bir dizin oluşturur. Yani -n bayrağı arttıkça dosyaya yapılacak işlemler artıyor, daha karmaşık hale geliyor.

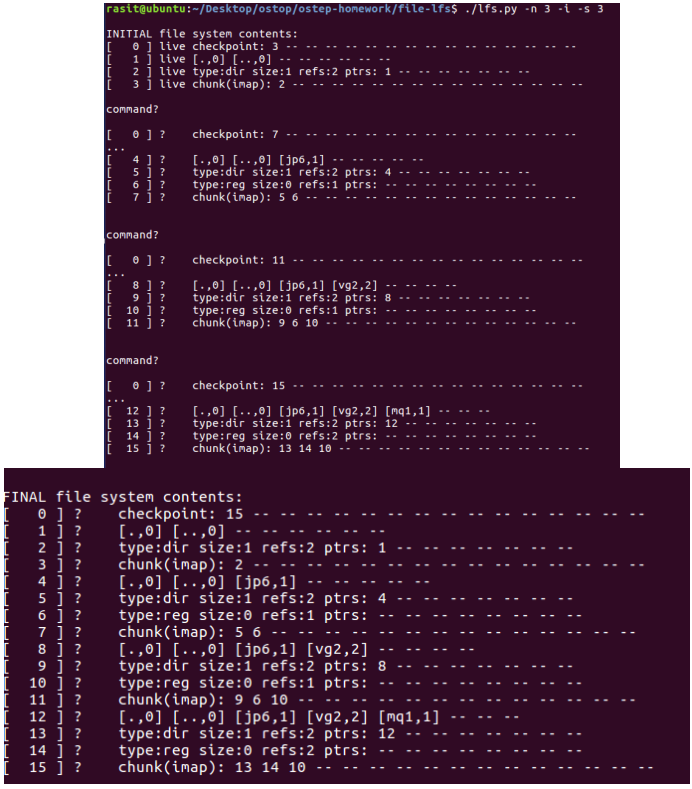
2. Yukarıdakileri acı verici buluyorsanız, her bir komutun neden olduğu güncelleme kümesini göstererek kendinize biraz yardımcı olabilirsiniz. Bunu yapmak için ./lfs.py -n 3 -i dosyasını çalıştırın. Şimdi her bir

komutun ne olması gerektiğini anlamanın daha kolay olup olmadığına bakın. Farklı komutların yorumlanmasını sağlamak için rastgele tohumu değiştirin (örneğin, -s 1, -s 2, -s 3, vb.).

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -n 3 -i

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 -- -- -- -- -- -- -- -- -- -- -- -- -- --

command?

[   0 ] ?    checkpoint: 7 -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[   4 ] ?    [.,0] [..,0] [ku3,1] -- -- -- -- --
[   5 ] ?    type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] ?    chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- --


command?

[   0 ] ?    checkpoint: 10 -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[   8 ] ?    z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0
[   9 ] ?    type:reg size:8 refs:1 ptrs: -- -- -- -- -- -- -- 8
[  10 ] ?    chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- --


command?

[   0 ] ?    checkpoint: 14 -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[  11 ] ?    [.,0] [..,0] [ku3,1] [qg9,2] -- -- -- --
[  12 ] ?    type:dir size:1 refs:2 ptrs: 11 -- -- -- -- -- -- --
[  13 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  14 ] ?    chunk(imap): 12 9 13 -- -- -- -- -- -- -- -- -- -- --
```

```
FINAL file system contents:
[   0 ] ?    checkpoint: 14 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] ?    [.,0] [..,0] -- -- -- -- -- --
[   2 ] ?    type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] ?    chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] ?    [.,0] [..,0] [ku3,1] -- -- -- -- --
[   5 ] ?    type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] ?    chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] ?    z0z0z0z0z0z0z0z0z0z0z0z0z0z0z0
[   9 ] ?    type:reg size:8 refs:1 ptrs: -- -- -- -- -- -- -- 8
[  10 ] ?    chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- --
[  11 ] ?    [.,0] [..,0] [ku3,1] [qg9,2] -- -- -- --
[  12 ] ?    type:dir size:1 refs:2 ptrs: 11 -- -- -- -- -- -- --
[  13 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  14 ] ?    chunk(imap): 12 9 13 -- -- -- -- -- -- -- -- -- -- --
```

-i bayrağının işlevi, kullanıcının dosya sistemini adım adım inceleyebilmesine ve komutların etkilerini görmesine yardımcı olur. Örneğin, kullanıcı ilk komudunu girip bunun etkisini gördükten sonra ikinci komutu girebilir ve böylece dosya sisteminde yapılan değişiklikleri adım adım izleyebilir.



-s bayrağını 3 yaparak dosyanın boyutunu değiştirdik ve -i bayrağını kullanarak kod bloklarımızı adım adım inceleyerek daha kolay yorum yapmamızı artırdık. Kodu incelersek ilk komut create file /jp6 dosya sisteminde /jp6 adlı bir dosya oluşturur. İkinci komut create dir /vg2 dosya sisteminde /vg2 adlı bir dizin oluşturur. Üçüncü komut link file /jp6/vg2/mq1 dosya sisteminde /vg2 dizininin içinde /mg1 adında bir bağıntı oluşturur. Bu bağlantı /jp6 dosyasına işaret eder. Dördüncü komut unlink/vg2/mq1 dosya sisteminde 7vg2 dizininin içindeki /mq1 bağlantısını siler. Böylece adım adım incelemiş olduk. -o bayrağını kullanarak daya ayrıntılı görebilirsiniz.

3. Her bir komut tarafından diskte hangi güncellemelerin yapıldığını bulma becerinizi daha fazla test etmek için aşağıdakileri çalıştırın: ./lfs.py -o -F -s 100 (ve belki birkaç başka rastgele tohum). Bu sadece bir dizi komutu gösterir ve size dosya sisteminin son durumunu GÖSTERMEZ. Dosya sisteminin son durumunun ne olması gerektiği hakkında mantık yürütebilir misiniz?

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -o -F -s 100

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 /us7
write file  /us7 offset=4 size=0
write file  /us7 offset=7 size=7
```

İlk başta /us7 adlı bir dosya oluşturulur. Bu dosyanın başlangıçta boş olduğunu söyleyebiliriz. Daha sonra /us7 adlı dosyaya offset 4'ten başlayarak boyutu 0 olan bir veri yazdırılır. Bu komut dosya sisteminde hiçbir değişikliğe neden olmaz çünkü veri boyutu 0'dır ve hiçbir şey yazılmaz. Daha sonra offset değeri 7'den başlayarak boyutu 7 olan veri yazılır. Tahmin ederek final durumunu yazmaya çalışıyım.

[0] live checkpoint 3

[1] live [.,0][..,0]

[2] live type:dir size:1 refs:2 ptrs:1

[3] live chunk(imap2)

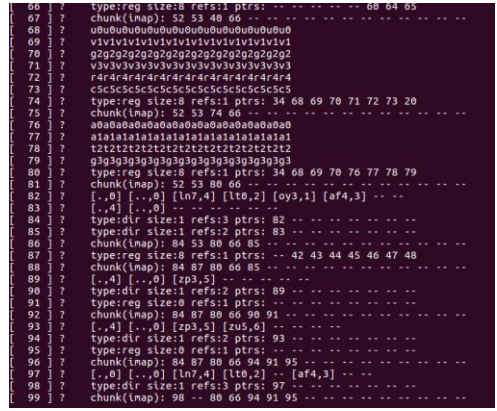
[4] live type:file size:7 resf:1 ptrs:1

[5] live chunk(data):7

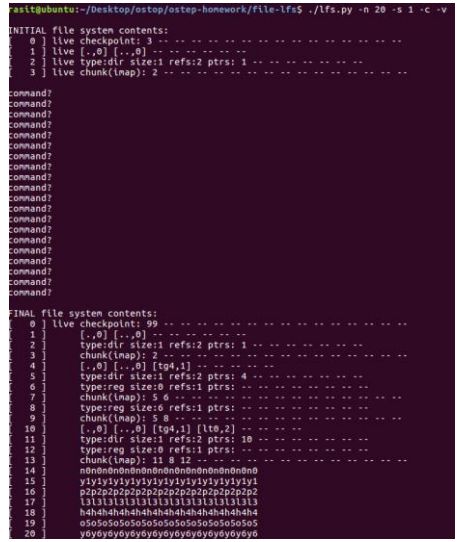
Kısaca yürüttüğüm mantığı açıklarsam, dosya sistemi içerisinde /us7 adlı bir dosya oluşturulmuş ve dosyaya veri yazılmış olur.

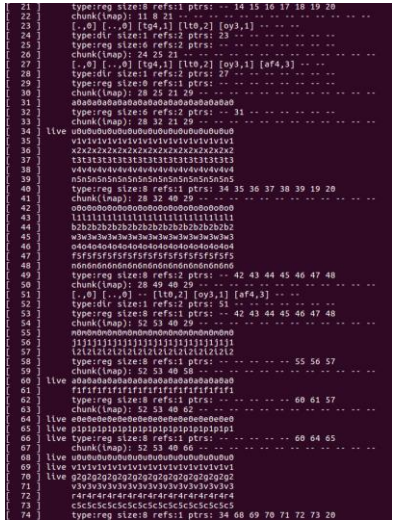
4. Şimdi bir dizi dosya ve dizin işleminden sonra hangi dosya ve dizinlerin canlı olduğunu belirleyip belirleyemeyeceğinize bakın. tt ./lfs.py -n 20 -s 1 dosyasını çalıştırın ve ardından son dosya sistemi durumunu inceleyin. Hangi yol adlarının geçerli olduğunu bulabilir misiniz? Sonuçları görmek için tt ./lfs.py -n 20 -s 1 -c -v komutunu çalıştırın. Rastgele komutlar dizisi verildiğinde cevaplarınızın eşleşip eşleşmediğini görmek için -o ile çalıştırın. Daha fazla sorun elde etmek için farklı rastgele tohumlar kullanın.

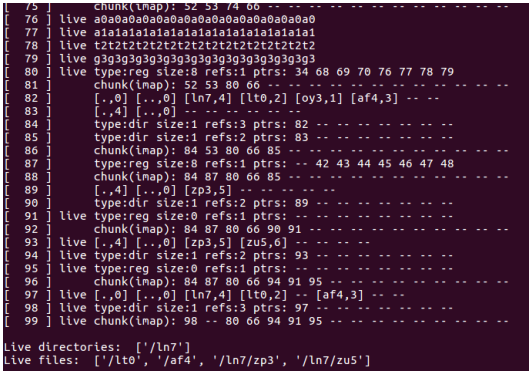
```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -n 20 -s 1

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 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?
command?

FINAL file system contents:
[   0 ] ?    checkpoint: 99 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] ?    [.,0] [..,0] -- -- -- -- -- --
[   2 ] ?    type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] ?    chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] ?    [.,0] [..,0] [tg4,1] -- -- -- -- --
[   5 ] ?    type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] ?    chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] ?    type:reg size:6 refs:1 ptrs: -- -- -- -- -- -- -- --
[   9 ] ?    chunk(imap): 5 8 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  10 ] ?    [.,0] [..,0] [tg4,1] [lt0,2] -- -- -- --
[  11 ] ?    type:dir size:1 refs:2 ptrs: 10 -- -- -- -- -- -- --
[  12 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  13 ] ?    chunk(imap): 11 8 12 -- -- -- -- -- -- -- -- -- -- -- -- --
[  14 ] ?    n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0
[  15 ] ?    y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1
[  16 ] ?    p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2
[  17 ] ?    l3l3l3l3l3l3l3l3l3l3l3l3l3l3l3
[  18 ] ?    h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4
[  18 ] ?    h4h4h4h4h4h4h4h4h4h4h4h4h4h4h4
[  19 ] ?    o5o5o5o5o5o5o5o5o5o5o5o5o5o5o5
[  20 ] ?    y6y6y6y6y6y6y6y6y6y6y6y6y6y6y6
[  21 ] ?    type:reg size:8 refs:1 ptrs: -- 14 15 16 17 18 19 20
[  22 ] ?    chunk(imap): 11 8 21 -- -- -- -- -- -- -- -- -- -- -- -- --
[  23 ] ?    [.,0] [..,0] [tg4,1] [lt0,2] [oy3,1] -- -- --
[  24 ] ?    type:dir size:1 refs:2 ptrs: 23 -- -- -- -- -- -- --
[  25 ] ?    type:reg size:6 refs:2 ptrs: -- -- -- -- -- -- -- --
[  26 ] ?    chunk(imap): 24 25 21 -- -- -- -- -- -- -- -- -- -- -- -- --
[  27 ] ?    [.,0] [..,0] [tg4,1] [lt0,2] [oy3,1] [af4,3] -- --
[  28 ] ?    type:dir size:1 refs:2 ptrs: 27 -- -- -- -- -- -- --
[  29 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  30 ] ?    chunk(imap): 28 25 21 29 -- -- -- -- -- -- -- -- -- -- -- --
[  31 ] ?    a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0
[  32 ] ?    type:reg size:6 refs:2 ptrs: -- 31 -- -- -- -- -- --
[  33 ] ?    chunk(imap): 28 32 21 29 -- -- -- -- -- -- -- -- -- -- -- --
[  34 ] ?    u0u0u0u0u0u0u0u0u0u0u0u0u0u0u0u0
[  35 ] ?    v1v1v1v1v1v1v1v1v1v1v1v1v1v1v1
[  36 ] ?    x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2
[  37 ] ?    t3t3t3t3t3t3t3t3t3t3t3t3t3t3t3
[  38 ] ?    v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4
[  39 ] ?    n5n5n5n5n5n5n5n5n5n5n5n5n5n5n5
[  40 ] ?    type:reg size:8 refs:1 ptrs: 34 35 36 37 38 39 19 20
[  41 ] ?    chunk(imap): 28 32 40 29 -- -- -- -- -- -- -- -- -- -- -- --
[  42 ] ?    o0o0o0o0o0o0o0o0o0o0o0o0o0o0o0o0
[  43 ] ?    l1l1l1l1l1l1l1l1l1l1l1l1l1l1l1l1
[  44 ] ?    b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2
[  45 ] ?    w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3w3
[  46 ] ?    o4o4o4o4o4o4o4o4o4o4o4o4o4o4o4o4
[  47 ] ?    f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5
[  48 ] ?    n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6
[  49 ] ?    type:reg size:8 refs:2 ptrs: -- 42 43 44 45 46 47 48
[  50 ] ?    chunk(imap): 28 49 40 29 -- -- -- -- -- -- -- -- -- -- -- --
[  51 ] ?    [.,0] [..,0] -- [lt0,2] [oy3,1] [af4,3] -- --
[  52 ] ?    type:dir size:1 refs:2 ptrs: 51 -- -- -- -- -- -- --
[  53 ] ?    type:reg size:8 refs:1 ptrs: -- 42 43 44 45 46 47 48
[  54 ] ?    chunk(imap): 52 53 40 29 -- -- -- -- -- -- -- -- -- -- -- --
[  55 ] ?    m0m0m0m0m0m0m0m0m0m0m0m0m0m0m0m0
[  56 ] ?    j1j1j1j1j1j1j1j1j1j1j1j1j1j1j1j1
[  57 ] ?    i2i2i2i2i2i2i2i2i2i2i2i2i2i2i2i2
[  58 ] ?    type:reg size:8 refs:1 ptrs: -- -- -- -- -- 55 56 57
[  59 ] ?    chunk(imap): 52 53 40 58 -- -- -- -- -- -- -- -- -- -- -- --
[  60 ] ?    a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0
[  61 ] ?    f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1
[  62 ] ?    type:reg size:8 refs:1 ptrs: -- -- -- -- -- 60 61 57
[  63 ] ?    chunk(imap): 52 53 40 62 -- -- -- -- -- -- -- -- -- -- -- --
[  64 ] ?    e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e0
[  65 ] ?    p1p1p1p1p1p1p1p1p1p1p1p1p1p1p1p1
[  66 ] ?    type:reg size:8 refs:1 ptrs: -- -- -- -- -- 60 64 65
```



Yukarıdaki koda baktığımızda insanın gözü korkuyor. Burada hangi dosyaların canlı olup olmadığını görmek zordur. Çünkü sondaki dosya sistemin içeriği, dosya sisteminde çalıştırılan komutlara bağlı olarak farklı olacaktır. Bu komutların ne olduğunu bilmeden, sondaki dosya sisteminin içeriğinin ne olacağını söylemek zordur.







Burada hangi satırların canlı olduğunu görüyoruz. [0,34,60,64,65,66,70, 76,77 ,78,79,80 ,91,93,94, 95,97,98,99]. -c bayrağı hangi satırların canlı olduğunu gösterir. -v bayrağı ise bize özel bir çıktı verir. Hangi dosyanın ve dizinin canlı olduğunu gösterir. Canlı olan dizi [‘/ln7’]. Canlı olan dosya ise [‘/lt0’ , ‘/af4’ , ‘/ln7/zp3’ , ‘/ln7/zu5’].

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -n 20 -s 1 -c -v -o

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 /tg4
write file  /tg4 offset=6 size=0
create file /lt0
write file  /lt0 offset=1 size=7
link file   /tg4 /oy3
create file /af4
write file  /tg4 offset=1 size=1
write file  /lt0 offset=0 size=6
write file  /oy3 offset=1 size=7
delete file /tg4
write file  /af4 offset=5 size=7
write file  /af4 offset=5 size=2
write file  /af4 offset=6 size=4
write file  /lt0 offset=1 size=6
write file  /lt0 offset=4 size=5
create dir  /ln7
write file  /oy3 offset=3 size=0
create file /ln7/zp3
create file /ln7/zu5
delete file /oy3
```

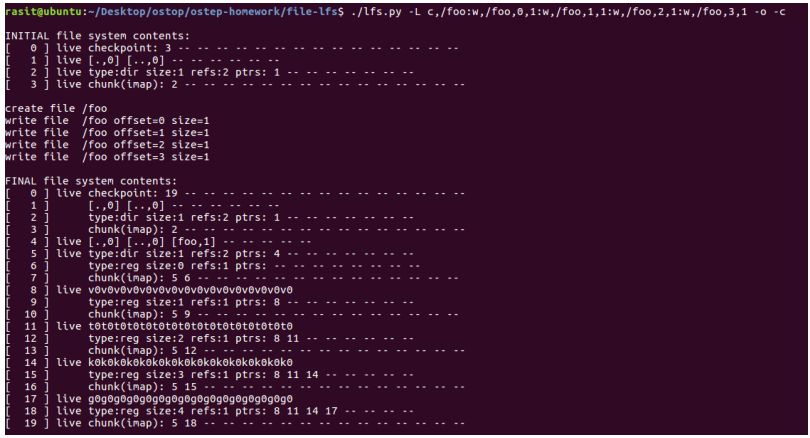
Kalabalık görülmesin diye final kısmını almadım. -o bayrağıyla dosyalarda hangi adımların gerçekleştiğini görebiliriz. Yukarıya baktığımızda /tg4,/lt0,/oy3,/af4,/In7/zu7 dosyalarının oluşturulması ve daha sonra yazma ve silme işlemlerinin yapıldığını görüyoruz.

5. Şimdi bazı özel komutlar verelim. İlk olarak, bir dosya oluşturalım ve ona tekrar tekrar yazalım. Bunu yapmak için, yürütülecek belirli komutları belirtmenizi sağlayan -L bayrağını kullanın. "/foo" dosyasını oluşturalım ve ona dört kez yazalım:
`-L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:w,/foo,3,1 -o`. Son dosya sistemi durumunun canlılığını belirleyip belirleyemeyeceğinize bakın; yanıtlarınızı kontrol etmek için -c kullanın.

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:w,/foo,3,1 -o

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 ] ?    v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   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 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

İlk başta 0. İndisteki kontrol noktasına (CR) bakarım. Kontrol noktamız bizim canlı bloğumuzdur (0. İndis canlı). CR noktamız bize 19. İndisteki imap bloğumuzu gösterir. Imap bloğumuz canlıdır (19. İndis canlı). Imap bizi ilk

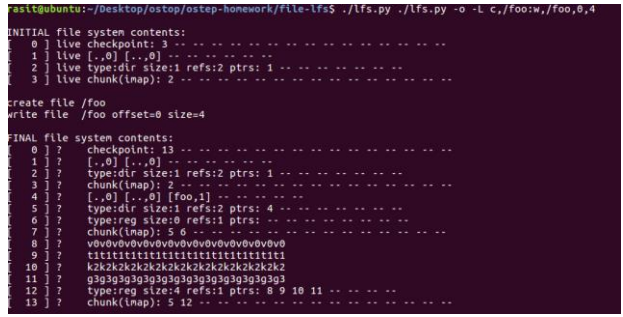
başta 5. İndisteki inode bloğumuzu gösterir (5.indis canlı). Inode ise 4. İndisi işaret eder be burada dosyamız oluşur (4. İndis canlı). Daha sonra tekrardan imap bloğuna geliyoruz. Bu sefer 18. İndisteki inode bloğuna gider (18. İndis canlı). Inode ilk başta 8. İndise giderek dosyayı yazar (8. İndis canlı). İnode 11. İndise gider tekrardan dosyayı yazar (11. İndis canlı). Inode 14. İndise gider ve dosya tekrardan yazılır (14. İndis canlı). En son 17. İndise gider ve dosyayı tekrardan yazar (17. İndis canlı). En son canlı blokları yazarsam [0,19,5 ,4,18,8,11,14,17]



-c bayrağını kullanarak hangi blokların canlı olduğunu görebiliriz. Yorumladığımız dışında bir şey çıkmadı. Canlı bloklar [0,4,5,8,11,14,17,18,19]

6. Şimdi aynı şeyi dört yerine tek bir yazma işlemi ile yapalım. "/foo" dosyasını oluşturmak ve tek bir yazma işlemiyle 4 blok yazmak için ./lfs.py -o -L c,/foo:w,/foo,0,4 dosyasını çalıştırın.

Canlılığı tekrar hesaplayın ve -c ile haklı olup olmadığınızı kontrol edin. Bir dosyayı tek seferde yazmak (burada yaptığımız gibi) ile her seferinde bir blok yazmak (yukarıdaki gibi) arasındaki temel fark nedir? Bu size güncellemeleri gerçek LFS'nin yaptığı gibi ana bellekte tamponlamanın önemi hakkında ne söylüyor?



```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py ./lfs.py -o -L c,/foo:w,/foo,0,4 -c

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=4

FINAL file system contents:
[   0 ] live checkpoint: 13 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ]      [.,0] [..,0] -- -- -- -- -- --
[   2 ]      type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ]      chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] live [.,0] [..,0] [foo,1] -- -- -- -- -- --
[   5 ] live type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ]      type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ]      chunk(imap): 5 6 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] live v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ] live t1t1t1t1t1t1t1t1t1t1t1t1t1t1t1t1
[  10 ] live k2k2k2k2k2k2k2k2k2k2k2k2k2k2k2k2
[  11 ] live g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3
[  12 ] live type:reg size:4 refs:1 ptrs: 8 9 10 11 -- -- -- --
[  13 ] live chunk(imap): 5 12 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

-c bayrağını çalıştırarak hangi blokların canlı olduğunu gördük. Şimdi gelelim asıl soruya. Bir dosyayı tek seferde yazmak, dosyayı bir kerede tümüyle yazmak anlamına gelir. Bu, dosyanın tüm içeriğinin aynı anda ana bellekte tamponlanarak yazılmasını sağlar. Bu, dosyayı yazarken daha hızlı bir şekilde yazılmasını ve daha az disk erişimi gerektiğini gösterir. Her seferinde bir blok yazmak ise, dosya parçalara ayrılır ve her parça ayrı ayrı yazılır. Bu dosyanın daha küçük parçalara bölünmesini ve her seferinde sadece bir parçanın yazılmasını sağlar. Ancak bu yöntem daha yavaş bir yazma hızına ve daha disk erişimine neden olur.

7. Başka bir spesifik örnek verelim. İlk olarak aşağıdakileri çalıştırın: `./lfs.py -L c,/foo:w,/foo,0,1`. Bu komut seti ne işe yarar? Şimdi./lfs.py -L c,/foo:w,/foo,7,1 dosyasını çalıştırın. Bu komut seti ne işe yarar? İkisi nasıl farklıdır? Bu iki komut setinden inode'daki boyut alanı hakkında ne söyleyebilirsiniz?

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -L c,/foo:w,/foo,0,1

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 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

command?
command?

FINAL file system contents:
[   0 ] ?    checkpoint: 10 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   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 ] ?    v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ] ?    type:reg size:1 refs:1 ptrs: 8 -- -- -- -- -- -- --
[  10 ] ?    chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

./lfs.py -L c,/foo:w,/foo,0,1 komutu çalıştırdığımızda, bir dosya sisteminde bir dizin oluşturur ve bu dizine bir dosya yazar. Dizin adın /foo ve dosya adı da /foo olarak belirlenmiştir. Dosyaya yazılan veri, offset 0 ve boyutu 1 olan v0v0v0v0v0v0v0v0v0v0v0v0 adlı veridir.

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -L c,/foo:w,/foo,7,1

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 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

command?
command?

FINAL file system contents:
[   0 ] ?    checkpoint: 10 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   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 ] ?    v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ] ?    type:reg size:8 refs:1 ptrs: -- -- -- -- -- -- -- 8
[  10 ] ?    chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

./lfs.py -L c,/foo:w,/foo,0,1 komutu çalıştırdığımızda yine bir dizin oluşturur ve bu dizine bir dosya yazar. Dizin adı /foo ve dosya adı /foo olarak belirlenmiştir. Dosyaya yazılan veri, offset 7 ve boyutu 1 olan v0v0v0v0v0v0v0v0v0v0v0v0 adlı veridir. Bu veri, dosyanın 8. Karakterine yazılır ve dosyanın boyutu 8'e yükseltilir.

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

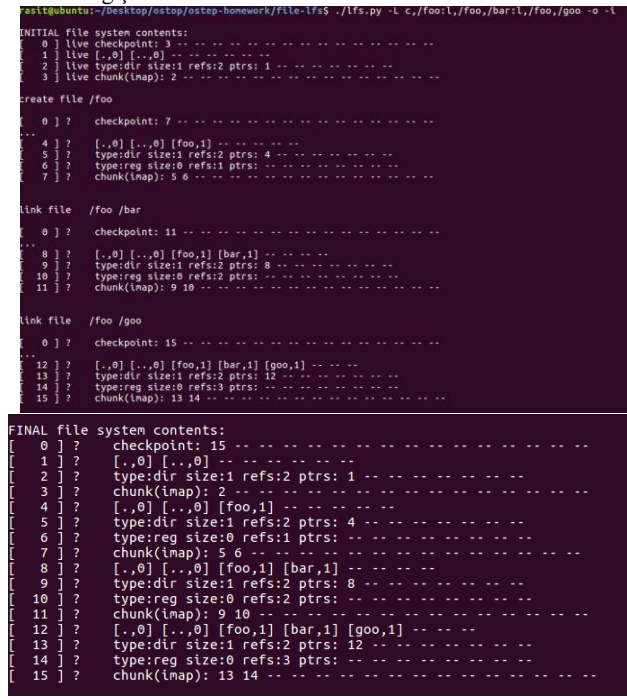
```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -L c,/foo

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 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

command?

FINAL file system contents:
[   0 ] ?    checkpoint: 7 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   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 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Komut satırındaki c dosya oluşturmak istediğimizi belirtiyor. Yukarıdaki çıktıda /foo adında bir dosya oluşturuyoruz.

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -L d,/foo

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 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

command?

FINAL file system contents:
[   0 ] ?    checkpoint: 8 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] ?    [.,0] [..,0] -- -- -- -- -- --
[   2 ] ?    type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] ?    chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] ?    [.,0] [..,0] [foo,1] -- -- -- -- --
[   5 ] ?    [.,1] [..,0] -- -- -- -- -- --
[   6 ] ?    type:dir size:1 refs:3 ptrs: 4 -- -- -- -- -- -- --
[   7 ] ?    type:dir size:1 refs:2 ptrs: 5 -- -- -- -- -- -- --
[   8 ] ?    chunk(imap): 6 7 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Komut satırındaki d dizin oluşturmak istediğimizi belirtiyor. Yukarıdaki çıktıda bir dosya sisteminde bir dizin oluşturur. Dizin adın /foo olarak belirtilmiştir. Bu dizin, boş bir dizindir ve hiçbir dosya ve alt dizin içermemektedir.

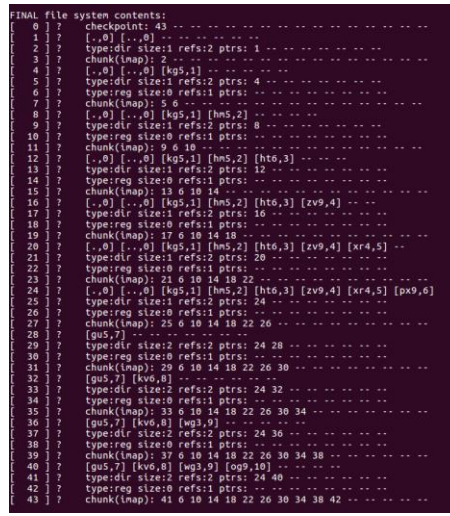
9. LFS simülatörü sabit bağlantıları da destekler. Nasıl çalıştıklarını incelemek için aşağıdakileri çalıştırın: ./lfs.py -L c,/foo:l,/foo,/bar:l,/foo,/goo -o -i. Bir sabit bağlantı oluşturulduğunda hangi bloklar yazılır? Bu sadece yeni bir dosya oluşturmaya nasıl benzer ve nasıl farklıdır? Bağlantılar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



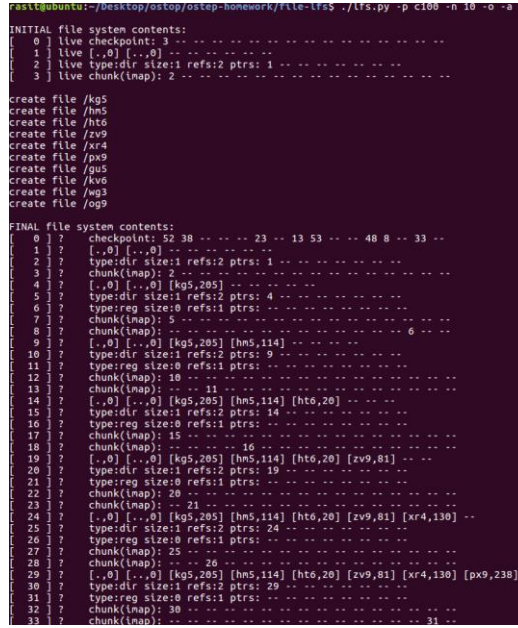
Yukarıdaki çıktıda dosya sistemi /foo adında yeni bir dosya oluşturacak, ardından bu dosyaya /bar adında bir bağlantı oluşturacak ve ardından /goo adında dosyaya başka bir bağlantı oluşturacak şekilde değiştirilmektedir. Dosya sistemi, dosyaları bir bilgisayara organize etmek ve saklama yöntemidir. Bu durumda, dosya sistemi dosyaları ve dizinleri takip etmek için bağlantılı bir liste yapısı kullanmaktadır. Foo dosyası oluşturulduğunda, dizin listesine yeni bir girdi eklenir ver dosya için yeni bir inode oluşturulur. Inode, dosya hakkında türü, boyutu ve ona yapılan referansların sayısı gibi meta verileri içerir. Bu durumda, /foo dosyası kendine bir referans içeren boş bir normal dosyadır. Bar bağlantısı oluşturulduğunda, dizin listesine yeni bir girdi eklenir ve /foo dosyasının referans sayısı bir artırılır. Bu, /foo dosyasına artık iki referans olduğu anlamına gelir. Biri /foo girişi ve diğeri /bar girişi aracılığıyla. Goo bağlantısı oluşturulduğunda, işlem öncekiyle aynıdır. Dizin listesine yeni bir girdi eklenir ve /foo dosyası için referans sayısı bir artırılır. Bu, /foo dosyasına artık üç referans olduğu anlamına gelir. Biri /foo girişi, biri /bar girişi ve biri de /goo girişi aracılığıyla. Genel olarak, dosya sistemi içeriği /foo dosyasının oluşturulmasını ve bu dosyaya /bar ve /goo olmak üzere iki bağlantı oluşturulmasını yansıtacak şekilde değiştirilir.

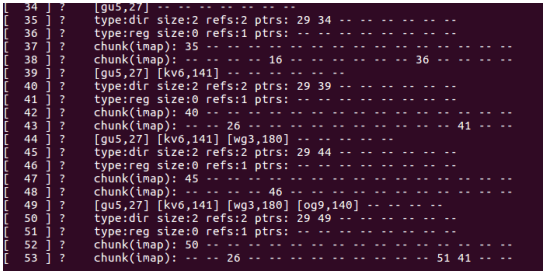
10. LFS birçok farklı politika kararı alır. Burada bunların çoğunu incelemiyoruz - belki de geleceğe bırakılmış bir şey - ama burada basit bir tanesini inceleyeceğiz: inode numarası seçimi. İlk olarak ./lfs.py -p c100 -n 10 -o -a s komutunu çalıştırarak sıfıra en yakın boş inode numaralarını kullanmaya çalışan "sıralı" tahsis politikası ile olağan davranışı gösterin. Ardından, ./lfs.py -p c100 -n 10 -o -a r komutunu çalıştırarak "rastgele" ilkesine geçin (-p c100 bayrağı rastgele işlemlerin yüzde 100'ünün dosya oluşturma olmasını sağlar). Rastgele bir politika ile sıralı bir politika arasında disk üzerinde ne gibi farklar vardır? Bu, gerçek bir LFS'de inode numaralarını seçmenin önemi hakkında ne söylüyor?

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -p c100 -n 10 -o -a s

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 /kg5
create file /hm5
create file /ht6
create file /zv9
create file /xr4
create file /px9
create file /gu5
create file /kv6
create file /wg3
create file /og9
```



./lfs.py -p c100 -n 10 -o -a s komutu çalıştırdığımızda yukarıdaki çıktıyı alıyoruz. Bu çıktı sıralı bir çıktıdır.





./lfs.py -p c100 -n 10 -o -a r komutu çalıştırdığımızda yukarıdaki çıktıyı alıyoruz. Bu çıktı rastgele bir çıktıdır. Çıktıya baktığımızda arasındaki farkı zaten anlıyoruz. Rastgele çıktı daha karışık geliyor. Şimdi gelelim soruları cevaplamaya. Rastgele bir politika ile sıralı politika arasındaki temel fark, dosya sisteminde inode numaralarının nasıl atandığıdır. Rastgele bir politikada, inode numaraları rastgele olarak atanır. Sıralı politika ise, dosya sistemi inode numaralarını dosyaların fiziksel konumuna göre seçer. Bu dosyaların disk üzerinden birbirlerine en yakın fiziksel konumda tutulmasını sağlar ve bu da disk erişim hızını artırır. Gerçek bir LFS dosya sisteminde inode numaralarının seçimi çok önemlidir çünkü inode numaraları dosya sistemindeki dosyaların konumunu belirtir. Bu nedenle, inode numaralarının doğru bir şekilde atanması dosya sisteminde dosyaların bulunması ve yönetilmesini kolaylaştırır. Ayrıca, inode numaralarının düzgün bir şekilde atanması dosya sistemindeki hızlı bir erişim sağlamaya yardımcı olur.

11. Varsaydığımız son bir şey de LFS simülatörünün her güncellemeden sonra kontrol noktası bölgesini her zaman güncellediğidir. Gerçek LFS'de durum böyle değildir: uzun arayışlardan kaçınmak için periyodik olarak güncellenir. Kontrol noktası bölgesi diske zorlanmadığında bazı işlemleri ve dosya sisteminin ara ve son durumlarını görmek için ./lfs.py -N -i -o -s 1000 dosyasını çalıştırın. Kontrol noktası bölgesi hiç güncellenmezse ne olur? Ya periyodik olarak güncellenirse? Günlükte ileri sararak dosya sistemini en son duruma nasıl kurtaracağınızı bulabilir misiniz?

```
rasit@ubuntu:~/Desktop/ostop/ostep-homework/file-lfs$ ./lfs.py -N -i -o -s 1000

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 dir  /jm5

[   4 ] ?    [.,0] [..,0] [jm5,1] -- -- -- -- --
[   5 ] ?    [.,1] [..,0] -- -- -- -- -- --
[   6 ] ?    type:dir size:1 refs:3 ptrs: 4 -- -- -- -- -- -- --
[   7 ] ?    type:dir size:1 refs:2 ptrs: 5 -- -- -- -- -- -- --
[   8 ] ?    chunk(imap): 6 7 -- -- -- -- -- -- -- -- -- -- -- -- -- --


create file /jm5/jm2

[   9 ] ?    [.,1] [..,0] [jm2,2] -- -- -- -- --
[  10 ] ?    type:dir size:1 refs:2 ptrs: 9 -- -- -- -- -- -- --
[  11 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  12 ] ?    chunk(imap): 6 10 11 -- -- -- -- -- -- -- -- -- -- -- -- --


create dir  /lb9

[  13 ] ?    [.,0] [..,0] [jm5,1] [lb9,3] -- -- -- --
[  14 ] ?    [.,3] [..,0] -- -- -- -- -- --
[  15 ] ?    type:dir size:1 refs:4 ptrs: 13 -- -- -- -- -- -- --
[  16 ] ?    type:dir size:1 refs:2 ptrs: 14 -- -- -- -- -- -- --
[  17 ] ?    chunk(imap): 15 10 11 16 -- -- -- -- -- -- -- -- -- -- --
```

```
FINAL file system contents:
[    0 ] ?    checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[    1 ] ?    [.,0] [..,0] -- -- -- -- -- --
[    2 ] ?    type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[    3 ] ?    chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[    4 ] ?    [.,0] [..,0] [jm5,1] -- -- -- -- --
[    5 ] ?    [.,1] [..,0] -- -- -- -- -- --
[    6 ] ?    type:dir size:1 refs:3 ptrs: 4 -- -- -- -- -- -- --
[    7 ] ?    type:dir size:1 refs:2 ptrs: 5 -- -- -- -- -- -- --
[    8 ] ?    chunk(imap): 6 7 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[    9 ] ?    [.,1] [..,0] [jm2,2] -- -- -- -- --
[   10 ] ?    type:dir size:1 refs:2 ptrs: 9 -- -- -- -- -- -- --
[   11 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   12 ] ?    chunk(imap): 6 10 11 -- -- -- -- -- -- -- -- -- -- -- -- --
[   13 ] ?    [.,0] [..,0] [jm5,1] [lb9,3] -- -- -- --
[   14 ] ?    [.,3] [..,0] -- -- -- -- -- --
[   15 ] ?    type:dir size:1 refs:4 ptrs: 13 -- -- -- -- -- -- --
[   16 ] ?    type:dir size:1 refs:2 ptrs: 14 -- -- -- -- -- -- --
[   17 ] ?    chunk(imap): 15 10 11 16 -- -- -- -- -- -- -- -- -- -- --
```

LFS kontrol noktası bölgesi, dosya sisteminin güncel durumunu tutan bir alandır. Eğer bu bölge güncellenmezse, dosya sisteminde meydana gelen değişiklikler kaydedilemez ve bu da dosya sisteminin korunmasına yardımcı olan özelliklerini kaybetmesine neden olabilir. Bu, dosya sisteminde veri kaybına yol açabilir. Periyodik olarak güncellenmesi, LFS kontrol noktası bölgesinin düzenli olarak güncellenmesini sağlar ve bu da dosya sisteminin sağlıklı bir şekilde çalışmasını destekler. Günlük ileri sararak dosya sistemini en son duruma nasıl kurtaracağınızı bulmak için, dosya sistemine ait bir yedekleme sistemini kullanmanız önerilebilir. Bu sayede, dosya sisteminde meydana gelen bir sorun sonucunda kaybolan verilerinizi geri yükleyebilirsiniz.