43

Günlük Yapılı Dosya Sistemleri

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

- **Sistem Bellekleri Büyüyor**: Bellek büyüdükçe, bellekte daha fazla veri önbelleğe alınabilir. Daha fazla veri önbelleğe alındıkça, disk trafiği giderek artan bir şekilde yazma işlemlerinden oluşur ve okumalara önbellek tarafından hizmet verilir. 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:** Sabit disk aktarım bant genişliği yıllar içinde oldukça arttı [P98]; Bir sürücünün yüzeyine daha fazla bit paketlendikçe, söz konusu bitlere erişirken bant genişliği artar. Bununla birlikte, arama ve rotasyon gecikme maliyetleri yavaş yavaş azalmıştır; Ucuz ve küçük motorların plakaları daha hızlı döndürmesini veya disk kolunu daha hızlı hareket ettirmesini sağlamak zordur. Böylece, diskleri sıralı bir şekilde kullanabiliyorsanız, aramalara ve rotasyonlara neden olan yaklaşımlara göre büyük bir performans avantajı elde edersiniz.

- **Mevcut dosya sistemleri birçok yaygın iş yükünde düşük performans gösterir:** Örneğin, FFS [MJLF84], bir blok boyutunda yeni bir dosya oluşturmak için çok sayıda yazma işlemi gerçekleştirir: biri yeni bir inode için, biri inode bitmap'i güncellemek için, biri dosyanın içinde bulunduğu dizin veri bloğuna, biri güncellemek için inode dizinine, biri yeni dosyanın bir parçası olan yeni veri bloğuna, ve veri bloğunu ayrılmış olarak işaretlemek için veri bitmap'ine bir tane. Bu nedenle, FFS tüm bu blokları aynı blok grubuna yerleştirse de, FFS birçok kısa aramaya ve ardından dönme gecikmelerine neden olur ve bu nedenle performans tepe sıralı bant genişliğinin çok altında kalır.

- **Dosya sistemleri RAID uyumlu değildir:** Örneğin, hem RAID-4 hem de RAID-5, **small-write problem** (küçük yazma sorununa) sahiptir; tek blok, 4 fiziksel I/O'nun gerçekleşmesine neden olur. Mevcut dosya sistemleri, bu en kötü RAID yazma davranışından kaçınmaya çalışmaz.

İPUCU: DETAYLAR ÖNEMLİDİR

Tüm ilginç sistemler birkaç genel fikirden ve bir dizi ayrıntıdan oluşur. Bazen, bu sistemleri öğrenirken, kendinize "Ah, genel fikri anlıyorum; gerisi sadece detaylar" diyor ve bunu işlerin gerçekte nasıl yürüdüğünü sadece yarı yarıya öğrenmek için kullanıyorsunuz. Bunu yapma! Çoğu zaman ayrıntılar önemlidir. LFS'de göreceğimiz gibi, genel fikrin anlaşılması kolaydır, ancak gerçekten çalışan bir sistem oluşturmak için *tüm* zor durumları düşünmeniz gerekir.

İdeal bir dosya sistemi bu nedenle yazma performansına odaklanır ve diskin sıralı bant genişliğinden yararlanmaya çalışır. Ayrıca, yalnızca veri yazmakla kalmayıp aynı zamanda disk üzerindeki meta veri yapılarını sık sık güncelleştiren yaygın iş yüklerinde de iyi performans gösterir. Son olarak, RAID'lerde ve tek disklerde iyi çalışır.

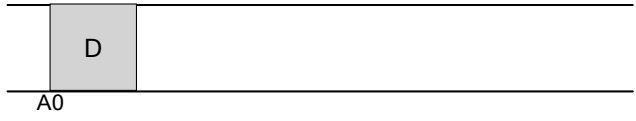
Rosenblum ve Ousterhout'un tanıttığı yeni dosya sistemi türü, **Günlük Yapılandırılmış Dosya** Sistemi'nin kısaltması olan LFS olarak adlandırıldı. Diske yazarken , LFS önce tüm güncellemeleri bellek içi bir **segmentte** arabelleğe alır (meta veriler dahil!); segment dolduğunda, diskin kullanılmayan bir bölümüne uzun, sıralı bir aktarımla diske yazılır. LFS asla mevcut verilerin üzerine yazmaz, bunun yerine *her zaman* segmentleri boş yerlere yazar. Segmentler büyük olduğundan, disk (veya RAID) verimli bir şekilde kullanılır ve dosya sisteminin performansı zirveye yaklaşır.

Önemli Nokta:
TÜM YAZILARI SIRALI YAZILAR HALİNE NASIL GETİRİLİR?

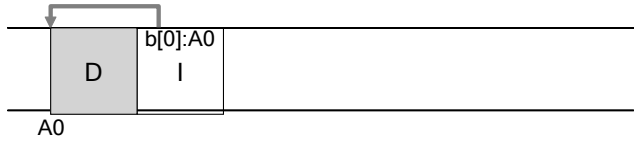
Bir dosya Sistemleri tüm yazma işlemlerini sıralı yazma işlemine nasıl dönüştürebilir? Okumalar için, bu görev imkansızdır, çünkü okunması istenen blok diskte herhangi bir yerde olabilir. Bununla birlikte, yazma işlemleri için, dosya sisteminin her zaman bir seçeneği vardır ve tam olarak bu seçimden yararlanmayı umuyoruz.

43.1 Diske Sıralı Olarak Yazma

Bu nedenle ilk zorluğumuzla karşı karşıyayız: Dosya sistemi durumuna yapılan tüm güncellemeleri diske bir dizi sıralı yazmaya nasıl dönüştürebiliriz? Bunu daha iyi anlamak için basit bir örnek kullanalım. Bir dosyaya D veri bloğu yazdığımızı hayal edin. Veri bloğunun diske yazılması, A0 disk adresinde D'nin yazılması ile aşşağıdaki disk üzerinde düzene neden olabilir:



Bununla birlikte, bir kullanıcı bir veri bloğu yazdığında, yalnızca diske yazılan veriler değildir; güncellenmesi gereken başka **metadata** (meta veriler) vardır. Bu durumda, dosyanın **inode'unu** (*I*) diske de yazalım ve D veri bloğuna işaret etmesini *sağlayalım*. Diske yazıldığında, veri bloğu ve inode böyle bir şeye benzeyecektir (inode'un veri bloğu kadar büyük göründüğünü unutmayın, ki bu genellikle böyle değildir; Çoğu sistemde, veri blokları 4 KB boyutundadır, oysa bir inode çok fazladır. daha küçük, yaklaşık 128 bayt):



Tüm güncellemeleri (veri blokları, inode'lar vb.) diske sırayla yazma konusundaki bu temel fikir, LFS'nin kalbinde yer alır. Bunu çözerseniz , temel fikri elde edersiniz. Ancak tüm karmaşık sistemlerde olduğu gibi, şeytan ayrıntılarda gizlidir.

43.2 Sıralı ve Etkili Bir Şekilde Yazma

Ne yazık ki, diske sırayla yazmak, verimli yazmaları garanti etmek için (tek başına) yeterli değildir. Örneğin, T zamanında A'ya hitap etmek için tek bir blok yazdığımızı hayal edin. Daha sonra biraz bekleriz ve diske A + 1 adresinde (sıralı sırayla bir sonraki blok adresi) yazarız, ancak $T + \delta$ zamanında yazıyoruz. Birinci ve ikinci yazmalar arasında, ne yazık ki, disk dönmüştür; ikinci yazmayı yayınladığınızda, işlenmeden önce bir dönüşün çoğunu bekleyecektir (özellikle, döndürme zaman alırsa T dönüşü alırsa , disk ikinci yazmayı disk yüzeyine işlemeden önce δ $T_{dönmesini}$ bekleyecektir). Ve böylece umarım diske sıralı sırayla yazmanın en yüksek performansı elde etmek için yeterli olmadığını görebilirsiniz; bunun yerine, iyi yazma performansı elde etmek için sürücüye çok sayıda *bitişik* yazma (veya bir büyük yazma) vermeniz gerekir.

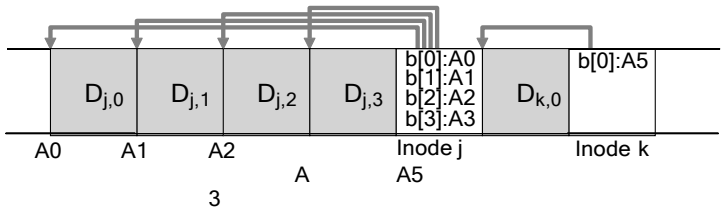
Bu amaca ulaşmak için, LFS, yazma olarak bilinen eski bir tekniği kullanır. Bu teknik **Buffering** (arabelleğe alma) yöntemidir. Diske yazmadan önce, LFS bellekteki güncellemeleri takip eder; Yeterli sayıda güncelleme aldığında, hepsini bir kerede diske yazar, böylece diskin verimli bir şekilde kullanılmasını sağlar.

LFS'nin bir kerede yazdığı güncellemelerin büyük bir kısmı, bir **segmentin** adıyla anılır. Bu terim bilgisayar sistemlerinde aşırı kullanılmasına rağmen, burada sadece LFS'nin yazmaları gruplandırmak için kullandığı büyük bir yığın anlamına gelir. Böylece, diske yazarken , LFS güncellemeleri bir bellek içi arabelleğe alır.

[1]Indeed, it is hard to find a good citation for this idea, since it was likely invented by many and very early on in the history of computing. For a study of the benefits of write buffering, see Solworth and Orji [SO90]; to learn about its potential harms, see Mogul [M94].

Segmentine ayırır ve ardından segmentin tümünü bir kerede diske yazar. Segment yeterince büyük olduğu sürece, bu yazma işlemleri verimli olacaktır.

LFS'nin iki güncelleme kümesini küçük bir segmentte arabelleğe aldığı bir örnek aşağıda verilmiştir; gerçek segmentler daha büyüktür (birkaç MB). İlk güncelleme, dört blogun j dosyasına yazması; ikincisi , k dosyasına eklenen bir bloktur. LFS daha sonra yedi bloğun tüm segmentini bir kerede diske işler . Bu blokların ortaya çıkan disk içi düzeni aşağıdaki gibidir :



43.3 Buffer'a (Arabellek) Alma Miktarı Ne Kadardır?

Bu, şu soruyu gündeme getirir: LFS, diske yazmadan önce kaç güncelleştirmeyi arabelleğe almalıdır? Cevap, elbette, diskin kendisine, özelliklev de konumlandırma yükünün aktarım hızına kıyasla ne kadar yüksek olduğuna bağlıdır; benzer bir analiz için FFS bölümüne bakınız.

Örneğin, her yazma işleminden sonra konumlandırmanın(yani, döndürme ve baş üstü arama)kabaca $T_{position}$ saniyeleri aldığını varsayalım. Disk aktarım hızının R_{peak} MB/*sn* olduğunu varsayalım. Böyle bir diskte çalışırken LFS'nin yazmadan önce arabelleğe alması ne kadar olmalıdır?

Bunu düşünmenin yolu, her yazma işleminde, konumlandırma maliyetinin sabit bir ek yükünü ödemenizdir. Bu nedenle, bu maliyeti **amortize** etmek için ne kadar yazmanız gerekiyor? Ne kadar çok yazarsanız, o kadar iyi (açıkçası) ve en yüksek bant genişliğine ulaşmaya o kadar yakın olursunuz.

Somut bir cevap elde etmek için,D MB'yi yazdığımızı varsayalım. Bu veri yığınını yazma zamanı (T_{write}) konumlandırma zamanıdır.
$T_{position}$ daha Ve Saat Hedef aktarmak, D(D/R_{peak}) veya :

$$T_{write} = T_{position} + \frac{D}{Rpeak} \qquad (43.1)$$

Ve böylece, yazılan veri miktarının onu yazmak için toplam süreye bölünmesiyle elde edilen etkin yazma *oranı* ($R_{effective}$) şöyledir:

$$R_{effective} = \frac{D}{Twrite} = \frac{D}{Tpositon + \frac{D}{Rpeak}} \qquad (43.2)$$

İlgilendiğimiz şey,effective oranı ($R_{effective}$) zirve oranına yaklaştırmaktır. Spesifik olarak effective oranının bir kesir olmasını istiyoruz. 0<F<1(tipik bir F 0,9 veya %90 olabilir) matematiksel olarak bu,$R_{effective}=F\times R_{peak}$ istediğimiz anlamına gelir.

Bu noktada, D için çözebiliriz:

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

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

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

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

Konumlandırma süresi 10 milisaniye ve tepe aktarım hızı 100 MB/s olan bir diskle bir örnek yapalım; tepe noktasının %90'ı kadar etkili bir bant genişliği istediğimizi varsayalım (F=0,9). Bu durumda D = $\frac{0,9}{0,1}$ × 100 MB/s × 0.01 saniye = 9 MB tepe bant genişliğine yaklaşmak için ne kadar ara belleğe almamız gerektiğini görmek için bazı farklı değerler deneyin. %95'e ulaşmak için ne kadar gerekli ve Zirve %99 'a?

43.4 Problem: Inodes Bulma

LFS'de bir inode'u nasıl bulduğumuzu anlamak için, tipik bir UNIX dosya sisteminde bir inode'u nasıl bulacağımızı gözden geçirelim. FFS veya hatta eski UNIX dosya sistemi gibi tipik bir dosya sisteminde, inode'ları bulmak kolaydır, çünkü bunlar bir dizide düzenlenir ve sabit konumlarda diske yerleştirilir.

Örneğin, eski UNIX dosya sistemi tüm inode'ları diskin sabit bir bölmesinde tutar. Böylece, bir inode numarası ve başlangıç adresi verildiğinde, belirli bir inode'u bulmak için, tam disk adresini, inode numarasını bir inode boyutuyla çarparak ve bunu diskteki dizinin başlangıç adresine ekleyerek hesaplayabilirsiniz; Bir inode numarası verilen dizi tabanlı indeksleme hızlı ve basittir.

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

LFS'de hayat daha zordur. Neden? Pekala, inode'ları diskin her tarafına dağıtmayı başardık! Daha da kötüsü, asla yerinde üzerine yazmayız ve böylece bir inode'un en son sürümü (yani istediğimiz sürüm) hareket etmeye devam eder.

43.5 Dolaylı Çözüm : Inode Haritası

Bunu düzeltmek için, LFS tasarımcıları, inode **haritası (imap**) adı verilen bir veri yapısı aracılığıyla inode numaraları ve inode'lar arasında bir inode map (imap) tanıttı. Imap, bir inode numarasını giriş olarak alan ve en son sürümünün disk adresini üreten bir yapıdır.

İPUCU: BİR DOLAYLI SEVİYE KULLANIN

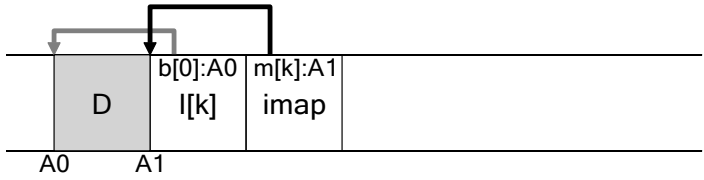
İnsanlar genellikle bilgisayar bilimlerindeki tüm problemlerin çözümünün sadece bir **level of indirection** (yönlendirme seviyesi) olduğunu söylerler. Bu açıkça doğru değil; bu sadece *çoğu* sorunun çözümüdür (evet, bu hala bir yorumun çok güçlü olduğudur, ama konuyu anlıyorsunuz). İncelediğimiz her sanallaştırmayı, örneğin sanal belleği veya bir dosya kavramını, basit biryönlendirme seviyesi olarak düşünebilirsiniz. Ve kesinlikle LFS'deki inode haritası, inode sayılarının sanallaştırılmasıdır. Umarım bu örneklerde yönlendirmenin büyük gücünü görebilir ve yapıları (VM örneğindeki sayfalar veya LFS'deki idüğümleri gibi) her referansı değiştirmek zorunda kalmadan serbestçe hareket ettirmemize izin verebilirsiniz. Tabii ki, yönlendirmenin de bir dezavantajı olabilir: **extra overhead** (ekstra ek yük)**.** Bu nedenle, bir dahaki sefere bir sorununuz olduğunda, onu yönlendirmeyle çözmeyi deneyin, ancak önce bunu yapmanın ek yüklerini düşündüğünüzden emin olun. Wheeler'ın ünlü dediği gibi, "Bilgisayar bilimlerindeki tüm problemler, elbette çok fazla yönlendirme problemi dışında, başka bir yönlendirme seviyesi ile çözülebilir."

Inode. Böylece, genellikle giriş başına 4 bayt (bir disk işaretçisi) ile basit bir dizi olarak uygulanacağını hayal edebilirsiniz. Diske her inode yazıldığında, imap yeni konumuyla güncellenir.

İmap'in ne yazık ki, kalıcı tutulması gerekir (yani, diske yazılır); Bunu yapmak, LFS'nin çökmeler boyunca inodların yerlerini takip etmesini ve böylece istenildiği gibi çalışmasını sağlar. Bu nedenle, bir soru: Imap diskte nerede bulunmalıdır?

Elbette diskin sabit bir bölümünde yaşayabilir. Ne yazık ki, sık sık güncellendiği için, bu daha sonra imap'e yazmalar tarafından takip edilecek dosya yapılarındaki güncellemeleri gerektirecek ve bu nedenle performans düşecektir (yani, each güncellemesi ile imap'ın sabit konumu arasında daha fazla disk arayışı olacaktır).

Bunun yerine, LFS inode haritasının parçalarının bulunduğu yerin hemen yanına diğer tüm yeni bilgileri yazar. Böylece, bir veri eklerken bir blok k dosyasına, LFS aslında yeni veri bloğunu, inode'unu ve bir parça inode haritasının bir parçası aşşağıdaki şekildeki gibi diske yazılır:



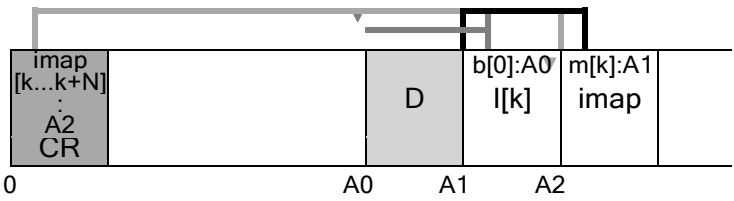
Bu resimde, imap olarak işaretlenmiş blokta saklanan imap dizisi parçası, LFS'ye k inodunun A1 disk adresinde olduğunu söyler; bu inode ise LFS'ye D veri bloğunun A0 adresinde olduğunu söyler.

43.6 Çözümü Tamamlama: Kontrol Noktası Bölgesi

Akıllı okuyucu (bu sensin, değil mi?) burada bir sorun fark etmiş olabilirsin. İnode haritasını nasıl bulabiliriz, şimdi onun parçaları da artık diske dağılmış durumda mı? Sonunda, sihir yoktur: dosya sistemi, bir dosya aramaya başlamak için diskte sabit ve bilinen bir konuma sahip olmalıdır.

LFS, bunun için diskte **check point region** (kontrol noktası bölgesi) **(CR**) olarak bilinen sabit bir yere sahiptir. Kontrol noktası bölgesi, inode haritasının en son parçalarına (yani adreslerine) işaretçiler içerir ve bu nedenle inode 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. Bu nedenle, disk üzerindeki düzenin genel yapısı bir kontrol noktası bölgesi içerir (inode haritasının en son parçalarını işaret eder); inode harita parçalarının her biri inode'ların adreslerini içerir; inode'lar tıpkı tipik UNIX dosya sistemleri gibi dosyalara (ve dizinlere) işaret eder.

İşte kontrol noktası bölgesinin bir örneği (tamamen diskin başında, 0 adresinde olduğuna dikkat edin) ve tek bir imap öbeği, inode ve veri bloğu. Gerçek bir dosya sistemi elbette çok daha büyük bir CR'ye sahip olacaktır (aslında, daha sonra anlayacağımız gibi iki tane olacaktır), birçok imap parçası ve elbette daha birçok inode, veri bloğu vb.



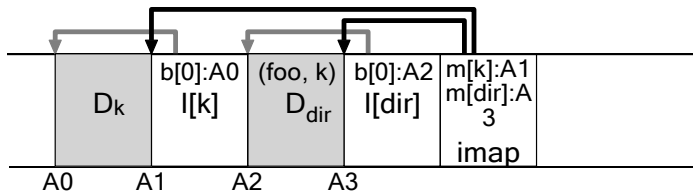
43.7 Diskten Dosya Okuma: Özet

LFS'nin nasıl çalıştığını anladığınızdan emin olmak için , şimdi bir dosyayı diskten okumak için ne yapılması gerektiğini inceleyelim. Başlamak için hafızamızda hiçbir şey olmadığını varsayalım. Okumamız gereken ilk disk içi veri yapısı kontrol noktası bölgesidir. Kontrol noktası bölgesi, tüm inode haritasına işaretçiler (yani, disk adresleri) içerir ve bu nedenle LFS daha sonra tüm inode haritasında okur ve onu bellekte önbelleğe alır. Bu noktadan sonra, bir dosyanın inode numarası verildiğinde, LFS, imap'teki inode-disk-adres eşlemesine inode numarasını arar ve inode'un en son sürümünde okur. Dosyadan bir bloğu okuyun, bu noktada, LFS, gerektiğinde doğrudan işaretçiler veya dolaylı işaretçiler veya iki kat dolaylı işaretçiler kullanarak tam olarak tipik bir UNIX dosya sistemi gibi ilerler. Genel durumda, LFS diskten bir dosya okurken tipik bir dosya sistemiyle aynı sayıda G/Ç gerçekleştirmelidir; tüm imap önbelleğe alınır ve bu nedenle LFS'nin bir okuma sırasında yaptığı ekstra iş, inode'un adresini imap'te aramaktır.

43.8 Peki ya dizinler?

Şimdiye kadar, tartışmamızı yalnızca inodeları ve veri bloklarını göz önünde bulundurarak biraz basitleştirdik. Ancak, bir dosya sistemindeki bir dosyaya erişmek için (örneğin, /home/remzi/foo, en sevdiğimiz sahte dosya adlarından biri), bazı yönlendirmelere de erişilmelidir. Peki LFS dizin verilerini nasıl saklar?

Neyse ki, dizin yapısı temel olarak klasik UNIX dosya sistemleriyle aynıdır, çünkü bir dizin sadece (ad, inode numarası) eşlemelerinin bir koleksiyonudur. Örneğin, diskte bir dosya oluştururken, LFS hem yeni bir inode, hem de bazı veriler ve bu dosyaya başvuran dizin verileri ve inode'u yazmalıdır. LFS'nin bunu diskte sırayla yapacağını unutmayın (güncelleştirmeleri bir süre arabelleğe aldıktan sonra). Bu nedenle, bir dizinde bir foo dosyası oluşturmak, diskte aşşağıdaki yeni yapıların oluşmasına yol açacaktır:



İnode haritasının parçası, hem dizin dosyasının hem de yeni oluşturulan *f* dosyasının konumu için bilgileri içerir. Bu nedenle, foo dosyasına erişirken (inode numarası *k* ile), dizin *dir* inode'unun (*A*3) konumunu bulmak için önce inode haritasına (genellikle bellekte önbelleğe alınmış) bakarsınız; Daha sonra size dizin verilerinin konumunu veren inode dizinini okursunuz (*A*2); Bu veri bloğunu okumak size (foo, *k*). Daha sonra inode numarası *k'nin* (A1) yerini bulmak için inode haritasına tekrar danışırsınız ve son olarak *A*0 adresinde istediğiniz veri bloğunu okursunuz.

LFS'de inode haritasının çözdüğü **recursive update problem** (özyinelemeli güncelleme sorunu) [Z + 12] olarak bilinen başka bir ciddi sorun daha var. Sorun, hiçbir zaman yerinde güncelleştirilmeyen (LFS gibi) herhangi bir dosya sisteminde ortaya çıkar, bunun yerine güncelleştirmeleri diskteki yeni konumlara taşır.

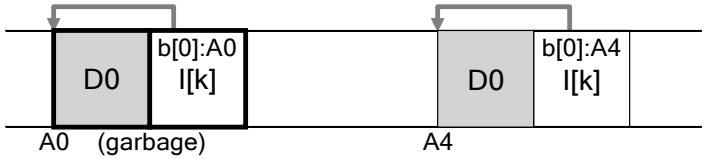
Spesifik olarak, bir inode her güncellendiğinde, diskteki konumu değişir. Dikkatli olmasaydık, bu aynı zamanda bu dosyaya işaret eden dizinde bir güncelleme yapılmasını da gerektirir. Bu daha sonra o dizinin üst öğesinde bir değişikliği zorunlu kılacaktı ve dosya sistemi ağacına kadar bu böyle devam edecekti.

LFS, inode haritası ile bu sorunu akıllıca önler. Bir inode'un konumu değişebilse de, değişiklik hiçbir zaman dizinin kendisine yansıtılmaz; bunun yerine, dizin aynı addan inode numarası eşleşmesine sahipken imap yapısı güncelleştirilir. Böylece, dolaylı olarak, LFS özyinelemeli güncelleme sorununu önler.

43.9 Yeni Bir Sorun: Çöp Toplama

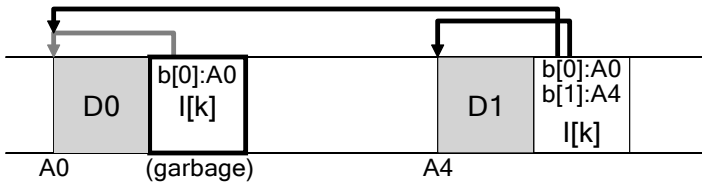
LFS ile ilgili başka bir sorun fark etmiş olabilirsiniz; bir dosyanın en son sürümünü (inode ve verileri dahil) diskteki yeni konumlara tekrar tekrar yazar. Bu işlem, yazmaları verimli tutarken, LFS'nin dosya yapılarının eski sürümlerini disk boyunca dağınık bıraktığı anlamına gelir. Biz (oldukça törensizce) bu eski versiyonlara **garbage** (çöp) diyoruz.

Örneğin, tek bir veri bloğu *D*0'a işaret eden k inode numarası ile başvurulan mevcut bir dosyaya sahip olduğumuz durumu hayal edelim. Şimdi bu bloğu hem yeni bir inode hem de yeni bir veri bloğu oluşturarak güncelliyoruz. LFS'nin ortaya çıkan disk içi düzeni şöyle görünecektir (basitlik için imap'i ve diğer yapıları atladığımızı unutmayın; yeni inode'a işaret etmek için yeni bir imap parçasının da diske yazılması gerekir):



Diyagramda, hem inode hem de veri bloğunun diskte iki sürümü olduğunu, birinin eski (soldaki) ve bir akımın ve dolayısıyla **live** (canlı) (sağdaki) olduğunu görebilirsiniz. Bir veri bloğunu (mantıksal olarak) güncellemenin basit bir eylemiyle, LFS tarafından bir dizi yeni yapı kalıcı hale getirilmeli, böylece söz konusu blokların eski sürümleri diskte bırakılmalıdır.

Başka bir örnek olarak, ojinal k dosyasına bir blok eklediğimizi hayal edin. Bu durumda, inode'un yeni bir sürümü oluşturulur, ancak eski veri bloğu hala inode tarafından işaret edilir. Bu nedenle, hala canlı ve mevcut dosya sisteminin bir parçası:



Peki inode'ların, veri bloklarının ve benzerlerinin bu eski sürümleriyle ne yapmalıyız? Bu eski sürümleri etrafta tutabilir ve kullanıcıların eski dosya sürümlerini geri yüklemelerine izin verebilir (örneğin, yanlışlıkla bir dosyanın üzerine yazdıklarında veya sildiklerinde, bunu yapmak oldukça kullanışlı olabilir); Böyle bir dosya sistemi, bir dosyanın farklı sürümlerini takip ettiği için **versioning file system** olarak bilinir.

Ancak, LFS bunun yerine bir dosyanın yalnızca en son canlı sürümünü tutar; bu nedenle (arka planda), LFS, dosya verilerinin, inode'ların ve diğer yapıların bu eski ölü sürümlerini periyodik olarak bulmalı ve **clean**(temizlemelidir); temizlik gerekir

böylece diskteki blokları sonraki yazmalarda kullanılmak üzere tekrar serbest bırakın. Temizleme işleminin, programlar için kullanılmayan bilgileri otomatik olarak serbest bırakan programlama dillerinde ortaya çıkan bir teknik olan bir **garbage collection**(çöp toplama) biçimi olduğunu unutmayın.

LFS'de diske büyük yazma işlemlerine olanak sağlayan mekanizmalar kadar önemli olan segmentleri daha önce tartışmıştık. Görünüşe göre, etkili temizliğin de oldukça ayrılmaz bir parçası. LFS temizleyici temizlik sırasında tek veri bloklarını, inode'ları vb. Basitçe geçip serbest bıraksaydı ne olacağını hayal edin. Sonuç: diskte ayrılan alan arasında karıştırılmış bir miktar boş delik bulunan bir dosya sistemi. LFS, diske sırayla ve yüksek performansla yazmak için büyük bir bitişik bölge bulamayacağından yazma performansı önemli ölçüde düşecektir.

Bunun yerine, LFS temizleyici segment bazında çalışır, böylece sonraki yazılar için büyük alan parçaları temizler. Temel temizleme işlemi aşağıdaki gibi çalışır. Periyodik olarak, LFS temizleyici bir dizi eski (kısmen kullanılan) segmenti okur, bu segmentlerde hangi blokların aktif olduğunu belirler ve ardından sadece canlı blokları içlerinde bulunduran yeni bir segment seti yazar ve eskilerini serbest bırakır. Özellikle, temizleyicinin M mevcut segmentlerini okumasını, içeriklerini N yeni segmentlerine (burada N < *M*) sıkıştırmasını ve ardından *N* segmentlerini yeni konumlardaki diske yazmasını bekliyoruz. Eski *M* segmentleri daha sonra serbest bırakılır ve sonraki yazma işlemleri için dosya sistemi tarafından kullanılabilir.

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

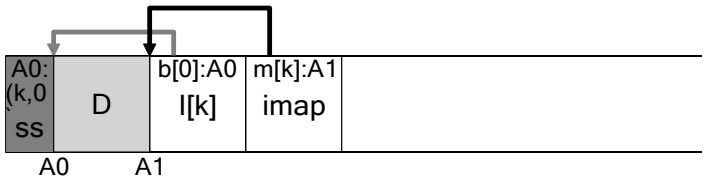
43.10 Blok Canlılığının Belirlenmesi

Önce mekanizmayı ele alıyoruz. Disk üzerindeki bir S segmenti içindeki bir veri bloğu *D* göz önüne alındığında, 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, inode numarasını (hangi dosyaya ait olduğu) ve ofsetini (dosyanın hangi bloğu olduğunu) içerir. Bu bilgiler, segmentin başındaki **segment summary block** (segment özet bloğu) 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 inode numarası *N'yi* ve ofset *T'yi* bulun. Ardından, N'nin nerede yaşadığını bulmak için imap'e bakın ve N'yi diskten okuyun (belki de zaten bellektedir, ki bu daha da iyidir). Son olarak, T ofsetini kullanarak inode'a (veya bazı dolaylı bloğa) bakın ve inode'un bu dosyanın Tth bloğunun diskte nerede olduğunu düşündüğünü görün. Tam olarak disk adresi A'ya işaret ediyorsa, LFS, D bloğunun canlı olduğu sonucuna varabilir. Başka bir yere işaret ederse, LFS, D'nin kullanımda olmadığı (yani öldüğünü) ve böylece bu sürümün artık gerekli olmadığını bildiği sonucuna varabilir. İşte bir sözde kod özeti:

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

Burada, segment özet bloğunun (*SS* işaretli) *A*0 adresindeki veri bloğunun aslında 0 ofsetindeki k dosyasının bir parçası olduğunu kaydettiği mekanizmayı gösteren bir diyagram bulunmaktadır. İmap'i *k* için kontrol ederek, inode'u bulabilir ve gerçekten o 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 artırır ve yeni sürüm numarasını imap'e kaydeder. LFS, sürüm numarasını disk üzerindeki segmente de kaydederek, disk üzerindeki sürüm numarasını imap'teki bir sürüm numarasıyla karşılaştırarak yukarıda açıklanan daha uzun kontrolü kısa devre yapabilir ve böylece fazladan okumalardan kaçınabilir.

43.11 Bir Politika Sorusu: Hangi Bloklar Ne Zaman Temizlenecek?

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

Hangi blokların temizleneceğini belirlemek daha zordur ve birçok araştırma makalesinin konusu olmuştur. Orijinal LFS makalesinde [RO91], yazarlar *sıcak* ve *soğuk* ayrımlarını ayırmaya çalışan bir yaklaşımı tanımlamaktadır. Sıcak bir segment, içeriğin sık sık üzerine yazıldığı bir bölümdür; Bu nedenle, böyle bir segment için en iyi politika, temizlemeden önce uzun süre beklemektir, çünkü giderek daha fazla blok üzerine yazılır (yeni segmentlerde) ve böylece kullanım için serbest bırakılır. Soğuk bir segment, genel olarak, birkaç ölü bloğa sahip olabilir, ancak içeriğinin geri kalanı nispeten kararlıdır. Bu nedenle, yazarlar soğuk segmentlerin er ya da geç temizlenmesi ve sıcak segmentlerin daha sonra temizlenmesi gerektiği sonucuna varırlar ve bunu yapan bir sezgisel yöntem geliştirirler. Bununla birlikte, çoğu politikada olduğu gibi, bu politika mükemmel değildir; Daha sonraki apache'ler nasıl daha iyi yapılacağını göstermektedir [MR + 97].

43.12 Kilitlenme Kurtarma ve Günlük

Son bir sorun: LFS diske yazarken sistem çökerse ne olur? Günlük tutma ile ilgili önceki bölümde hatırlayabileceğiniz gibi, güncellemeler sırasındaki çökmeler dosya sistemleri için zordur ve bu nedenle LFS'nin de göz önünde bulundurması gereken bazı şeylerdir.

Normal çalışma sırasında, LFS arabellekleri bir segmentte yazar ve sonra (segment dolduğunda veya belirli bir süre geçtiğinde) segmenti diske yazar. LFS, bu yazma işlemlerini bir günlükte düzenler, yani kontrol noktası bölgesi bir baş ve kuyruk segmentine işaret eder, her segment yazılacak bir sonraki segmente işaret eder. LFS ayrıca denetim noktası bölgesini düzenli aralıklarla güncelleştirir. Bu işlemlerden herhangi biri sırasında çökmeler açıkça meydana gelebilir (bir segmente yazın, CR'ye yazın). Peki LFS, bu yapılara yazma sırasında çökmeleri nasıl ele alıyor?

Önce ikinci vakayı ele alalım. CR güncelleştirmesinin 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 inode haritasına ve diğer bilgilere en son işaretçilerle güncellerken dikkatli bir protokol uygular; Özellikle, önce bir başlık (zaman damgası ile), sonra CR'nin gövdesi ve son olarak son bir tane blog (ayrıca bir zaman damgası ile) yazar. CR güncelleştirmesi sırasında sistem çökerse, LFS tutarsız bir zaman damgası çifti 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 vakayı ele alalım. LFS, CR'yi her 30 saniyede bir yazdığından, dosya sisteminin son tutarlı anlık görüntüsü oldukça eski olabilir. Böylece, yeniden başlattıktan sonra, LFS, kontrol noktası bölgesinde, p'nin bulunduğu imap parçalarını ve sonraki dosyaları ve dizinleri okuyarak kolayca kurtarabilir; ancak, güncellemelerin son birkaç saniyesi kaybolacaktır.

Bunu geliştirmek için LFS, veritabanı topluluğunda **roll forward** (ileri sarma) olarak bilinen bir teknikle bu segmentlerin çoğunu yeniden oluşturmaya çalışır. Temel fikir, son kontrol noktası bölgesi ile başlamak, günlüğün sonunu (CR'ye dahil olan) bulmak ve ardından sonraki segmentleri okumak ve içinde geçerli güncellemeler olup olmadığını görmek için bunu 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 bakınız [R92].

43.13 Özet

LFS, diski güncelleştirmek için yeni bir yaklaşım sunar. P bağcıklarındaki dosyaların üzerine yazmak yerine, LFS her zaman diskin kullanılmayan bir bölümüne yazar ve daha sonra temizleme yoluyla bu eski alanı geri kazanır. Veritabanı sistemlerinde **shadow paging** (gölge sayfalama) [L77] ve dosya sistemi konuşmasında bazen yazma üzerine kopyala olarak adlandırılan bu **copy-on-write**, LFS'nin tüm güncellemeleri bellek içi bir segmentte toplayabildiği ve daha sonra bunları sırayla birlikte yazabildiği için oldukça verimli yazma sağlar.

İPUCU: KUSURLARI ERDEMLERE DÖNÜŞTÜRÜN

Sisteminizin temel bir kusuru olduğunda, bunu bir özelliğe veya yararlı bir şeye dönüştürüp dönüştüremeyeceğinize bakın. NetApp'in WAFL'si bunu eski dosya içerikleriyle yapar; Eski sürümleri kullanılabilir hale getirerek, WAFL artık çok sık temizlik konusunda endişelenmek zorunda kalmaz (sonunda arka planda eski sürümleri siler) ve böylece harika bir özellik sağlar ve LFS temizleme sorununun çoğunu tek bir harika bükülmede ortadan kaldırır. Sistemlerde bunun başka örnekleri var mı? Kuşkusuz, ama onları kendiniz düşünmek zorunda kalacaksınız, çünkü bu bölüm büyük harf "O" ile bitti. Üzerinde. Yapılmış. Mahvolmuş. Dışarıdayız. Barış!

LFS'nin ürettiği büyük yazılar, birçok farklı cihazda performans için mükemmeldir. Sabit sürücülerde, büyük yazmalar duruş süresinin en aza indirilmesini sağlar; RAID-4 ve RAID-5 gibi eşlik tabanlı RAID'lerde, küçük yazma probleminden tamamen kaçınırlar. Son zamanlarda yapılan araştırmalar, Flash tabanlı SSD'lerde [H + 17] yüksek performans için büyük G/Ç'lerin gerekli olduğunu bile göstermiştir; 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 tarafına dağılmıştır ve eğer biri daha sonraki kullanım için böyle bir alanı yeniden talep etmek istiyorsa, eski segmentleri periyodik olarak temizlemelidir. Temizlik, LFS'de birçok tartışmanın odağı haline geldi ve temizlik maliyetleri [SS + 95] ile ilgili endişeler belki de LFS'nin sahadaki ilk etkisini sınırladı. Bununla birlikte, NetApp'in **WAFL** [HLM94], Sun'ın **ZFS** [B07] ve Linux **btrfs** [R + 13] ve hatta modern **flash-based** (flash tabanlı) **SSD'ler** [AD14] dahil olmak üzere bazı modern ticari dosya sistemleri, diske yazma konusunda benzer bir yazma üzerine kopyalama yaklaşımını benimser ve dolayısıyla entelektüel miras LFS, bu modern dosya sistemlerinde yaşamaya devam ediyor. Özellikle, WAFL temizleme sorunlarını bir özelliğe dönüştürerek çözdü; Dosya sisteminin eski sürümlerini **(snapshots)** anlık görüntüler aracılığıyla sağlayarak, kullanıcılar mevcut dosyaları yanlışlıkla sildiklerinde eski dosyalara erişebilirler.

Referanslar

[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 sys- tems. 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.*

Ödev (Simülasyon)

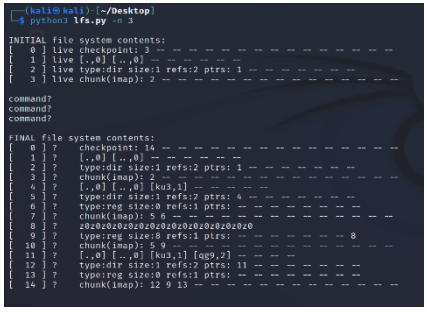
Bu bölümde, LFS tabanlı bir dosya sisteminin nasıl çalıştığını daha iyi anlamak için kullanabileceğiniz basit bir LFS simülatörü olan lfs.py tanıtılmaktadır. Simülatörün nasıl çalıştırılacağıyla ilgili ayrıntılar için README'yi okuyun.

SORULAR:

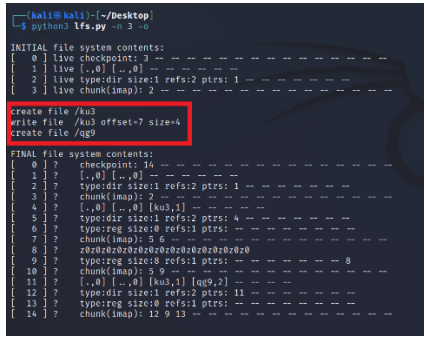
1. Çalıştır. /lfs.py -n 3, belki de çekirdek (-s) değiştirir. Son dosya sistemi içeriğini oluşturmak için hangi komutların çalıştırıldığını öğrenebilir misiniz? Bu komutların hangi sırayla verildiğini söyleyebilir misiniz? Son olarak, son dosya sistemi durumundaki her bloğun canlılığını belirleyebilir misiniz? Hangi komutların çalıştırıldığını göstermek için -o komutunu kullanın ve -c son dosya sistemi durumunun canlılığını göstermek için. Verilen komutların r sayısını artırdıkça (yani, -n 3'ü -n 5'e değiştirdikçe) görev sizin için ne kadar zorlaşıyor?

   SORU 1 CEVAP :

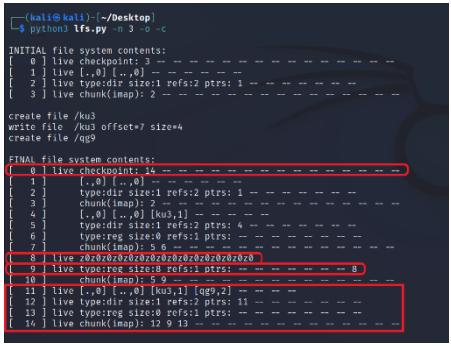
1. /lfs.py -n 3 çalıştırıldığındaki çıktı :



2. Aşşağıdaki görselde kırmızı ile işaretli kısımda hangi komutların çalıştırıldığı gösterilmektedir. Aynı zamanda hangi sıra ile verildiği görülmektedir. Bizden istenilenleri de -o parametresi ile bulmaktayız.

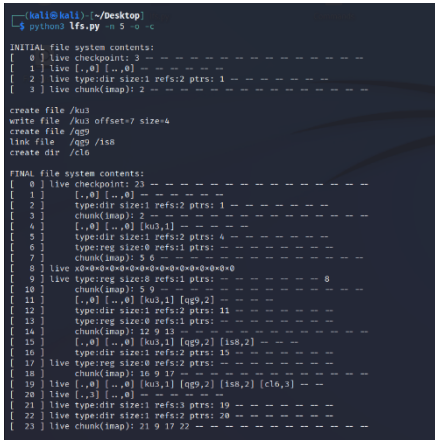


3. Her blogun canlılığı kırmızı alanlarla belirtilmiştir. Yani -c parametresi ile bulduğumuz live yazan kısımlardır.



4. Komutlarımızın r sayısını arttırdığımızdaki çıktılarımız. Görüldüğü üzere daha fazla işlem gerçekleşmektedir.

r sayısı = 5

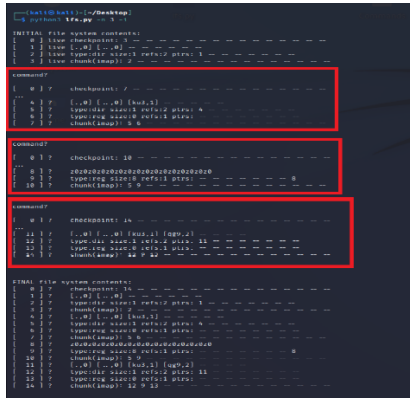


r sayısı = 9

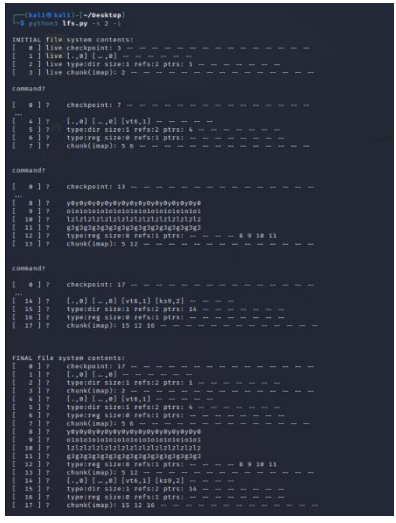
2. Yukarıdakileri acı verici bulursanız, her bir komutun neden olduğu güncellemeler kümesini göstererek kendinize biraz yardımcı olabilirsiniz. Bunu yapmak için, /lfs.py -n 3 -i komutunu çalıştırın. Şimdi her komutun ne olması gerektiğini anlamanın daha kolay olup olmadığına bakın. Yorumlanacak farklı komutlar almak için rastgele çekirdeği değiştirin (örneğin; -s 1, -s 2, -s 3, vb.).

SORU 2 CEVAP :

1. Her bir komutun neden olduğu güncellemeler kümesi aşşağıdaki gibidir. Bunu -I parametresi kullanarak görmekteyiz.



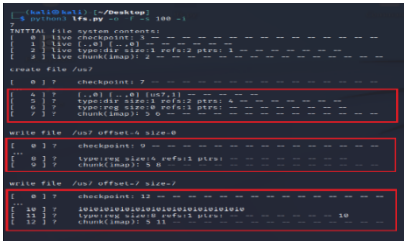
2. Yorumlanacak farklı komutlar almak için rastgele çekirdeği -s parametresi ile değiştirdiğimizde çıktımızdaki değişiklikleri görmektesiniz.



3. Her komut tarafından diskte hangi güncelleştirmelerin yapıldığını anlama yeteneğinizi daha fazla sınamak için aşağıdakileri çalıştırın: /lfs.py -o -F -s 100 (ve belki de birkaç rastgele tohum). Bu sadece bir dizi komutu gösterir ve size dosya sisteminin son durumunu göstermez. Dosya sisteminin son durumunun ne olması gerektiği hakkında bir fikir yürütebilir misiniz?

   SORU 3 CEVAP :

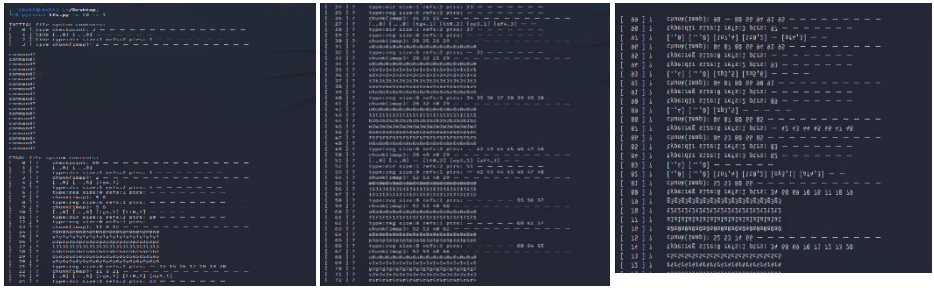
   1. lfs.py -o -F -s 100 çalıştırdığımızdaki çıktımız ve -i parametresi kullanarak dosya sisteminin son durumunun nasıl olması gerektiğni görmekteyiz :



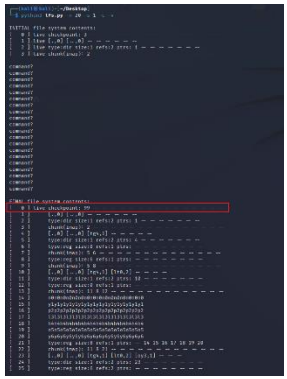
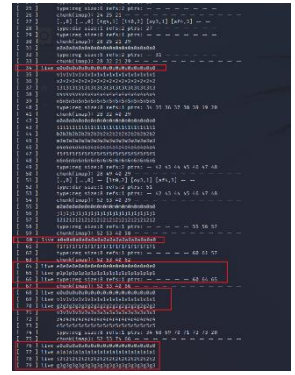
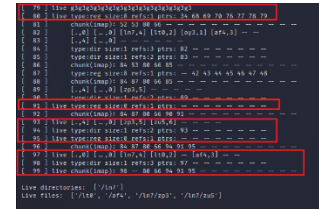
4. Şimdi bir dizi dosya ve dizin işleminden sonra hangi dosya ve dizinlerin canlı olduğunu belirleyip belirleyemeyeceğinize bakın ./lfs.py -n 20 -s 1 ve ardından son dosya sistemi durumunu inceleyin. Hangi yol adlarının geçerli olduğunu anlayabiliyor musunuz? Şu dosyayı çalıştırın ./lfs.py -n 20 -s 1 -c -v sonuçları görmek için. Rastgele komut serisi göz önüne alındığında yanıtlarınızın eşleşip eşleşmediğini görmek için -o ile çalıştırın. Daha fazla sorun yaşamak için farklı random tohumları kullanın.

   SORU 4 CEVAP :

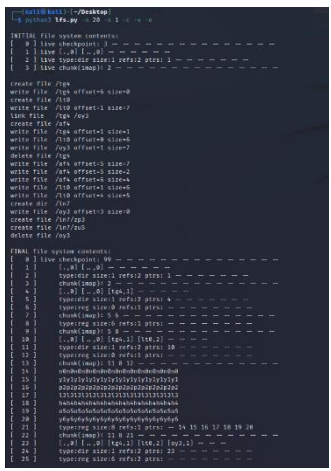
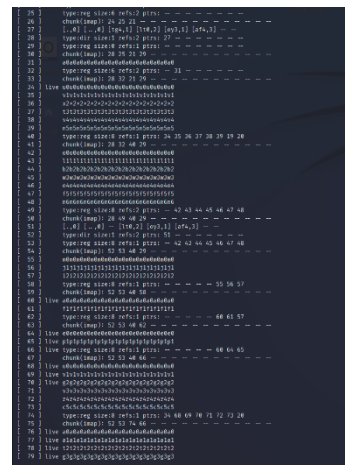
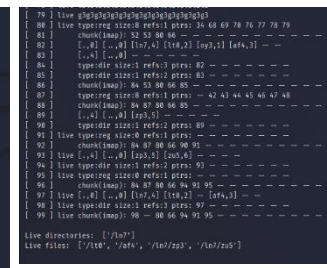
   1. ./lfs.py -n 20 -s 1 çalıştırdığımızda ekran görüntümüz aşşağıdaki gibidir son dosya sistemimizin ne kadar uzun olduğundan işlemlerin fazla olduğunu anlamaktayız. Yol adlarının geçerliliğini anlayabiliriz fark ettiyseniz son dosya sisteminde yapılan işlemlerin isimleri yazılmaktadır ve canlılığını imaplere bakarak anlayabiliriz imap asıl oluşturulan dosyayı işaret ediyorsa canlılık var demektir.



2. /lfs.py -n 20 -s 1 -c -v kodunu çalıştırdığımızda çıktımız aşşağıdaki gibidir ve kırmızı ile işaretli kısımların canlı bloglar olduğunu görmekteyiz live etiketi ile de etiketlendiklerini görmekteyiz. Kodumuzun en son iki satırında da yol adlarının geçerliliği hakkında bilgi edinmekteyiz.

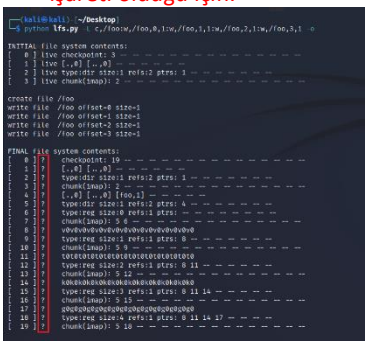
3. -o parametresi ile çalıştırıldığındaki kod çıktımız aşşağıdaki gibi olmaktadır. Bu bize yukarıdaki tahmin edebileceğimiz işlemleri Create file diye yazan kısımdan sonraki kısımlarda. göstermektedir.

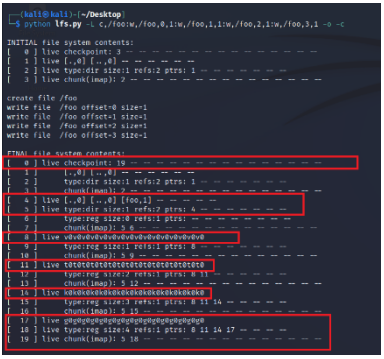
5. Şimdi bazı özel komutlar yayınlayalım. İlk olarak, bir dosya oluşturalım ve üzerine tekrar tekrar yazalım. Bunu yapmak için, yürütülecek belirli komutları belirtmenizi sağlayan -L bayrağını kullanın. "/foo" dosyasını oluşturalım ve dört kez yazalım: -L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:w,/foo,3,1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

SORU 5 CEVAP :

1. -L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:w,/foo,3,1 -o kodun çıktısı aşşağıda gösterilmektedir. Canlılığı belirleyemediğimizi görmektesiniz ? işareti olduğu için.



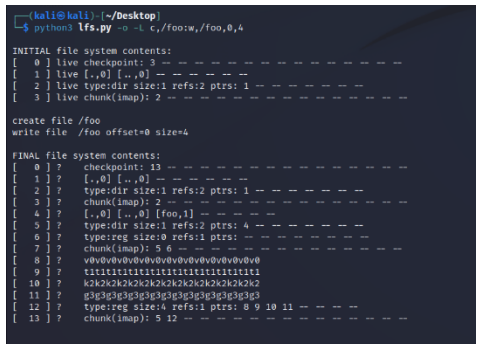
2. -c parametresi ile kullanıdığımızda canlılığı görebiliyoruz. Aynı zamanda canlılığı imape bakarak anlayabiliriz eğer imap oluşturulan dosyayı işaret ediyorsa canlılık var demektir.



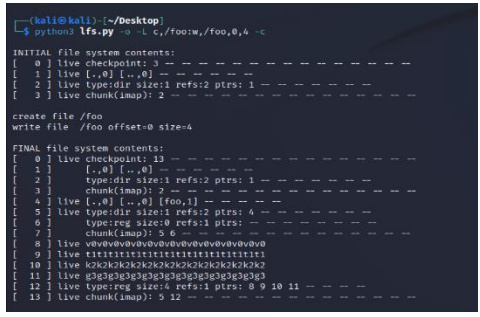
6. Şimdi, aynı şeyi yapalım, ancak dört yerin e tek bir yazma işlemi ile. Çalıştır. ./lfs.py -o -L c,/foo:w,/foo,0,4 "/foo" dosyasını oluşturmak Canlılığı tekrar hesaplayın ve -c ile doğru olup olmadığınızı kontrol edin. Bir dosyayı bir kerede yazmak (burada yaptığımız gibi) ile her seferinde bir blok yapmak (yukarıdaki gibi) arasındaki temel fark nedir? Bu, gerçek LFS'nin yaptığı gibi ana bellekteki güncellemeleri arabelleğe almanın önemi hakkında size ne söylüyor?

SORU 6 CEVAP :

1. Tek bir yazma işlemi ile ./lfs.py -o -L c,/foo:w,/foo,0,4 "/foo" komutu çalıştırdığımızdaki ekran çıktımız aşşağıdadır. Aynı zamanda canlılığı imape bakarak anlayabiliyoruz imapimiz oluşan dosyayı işaret ediyorsa canlılık var demektir.



2. Canlılığı – c ile live etikete sahip bloglarda görmekteyiz.

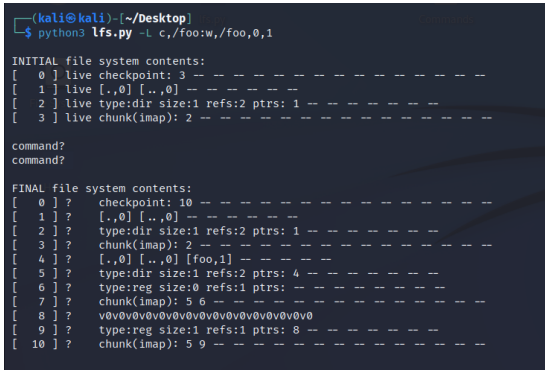


3. Sıralı yasmak verimlilik konusunda başarısı azdır. Tek seferde buffera alarak yasmak daha verimli ve hızlıdır. Kod çıktılarımıza baktığımızda ikisinin de sonucu aynı olduğunu görmekteyiz fakat çalışma verimlilikleri farklıdır.

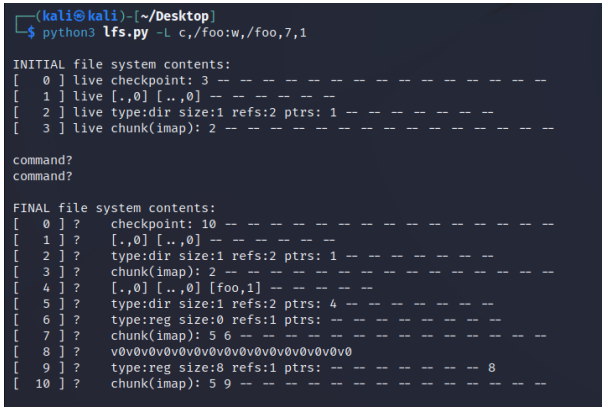
7. Başka bir özel örnek yapalım. İlk olarak, aşağıdakileri çalıştırın: ./lfs.py -L c,/foo:w,/foo,0,1. Bu komut kümesi ne işe yarar? Şimdi, çalıştırın ./lfs.py -L c,/foo:w,/foo,7,1. Bu komut kümesi ne işe yarar? İkisi arasındaki fark nedir? Bu iki komut kümesinden inode'daki boyut alanı hakkında ne söyleyebilirsiniz?

SORU 7 CEVAP :

1. lfs.py -L c,/foo:w,/foo,0,1 çalıştırıldığında ekran çıktısı aşşağıdaki gibidir. Bu işlem ilk olarak foo adında bir dosya oluşturur ve bir defa yazdırma işine yarar.



2. lfs.py -L c,/foo:w,/foo,7,1 komutu çalıştırıldığında kodun çıktısı aşşağıdaki gibidir. Bu işlem foo adında bir dosya oluşturur ve bir defa yazdırma işlemi gerçekleştirir.



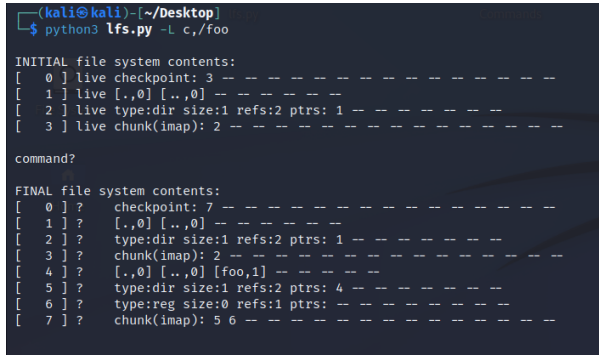
3. Bu iki komut arasındaki fark boyutlar ve foo,0.1'de 8'inci bloga yönlendirme vardır.

4. Foo,7.1 de boyut 8, foo,0.1 de ise boyutumuzun 1 olduğunu görmekteyiz.

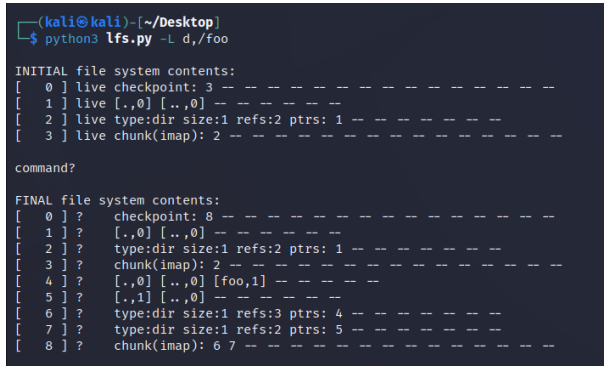
8. Şimdi açıkça dosya oluşturma ve dizin oluşturma konularına bakalım. Bir dosya ve ardından bir dizin oluşturmak için ./lfs.py -L c,/foo ve ./lfs.py -L d,/foo Simülasyonları çalıştırın. Bu koşular hakkında benzer olan nedir ve farklı olan nedir?

SORU 8 CEVAP :

1. ./lfs.py -L c,/foo komutu çıktımız aşşağıdadır.



2. ./lfs.py -L d,/foo komutu çıktımız aşşağıdadır.

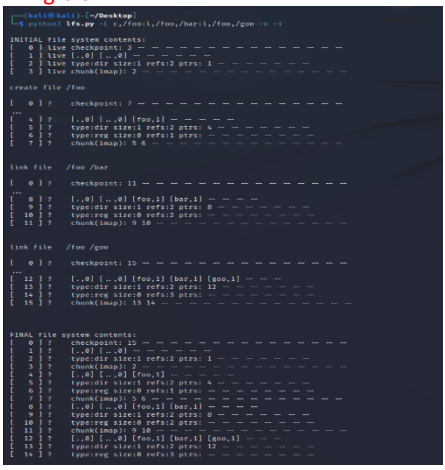


3. Bu iki komut arasında benzer olan şey live olan blokları göstermemektedir. İkiside foo adında bir dosya oluşturmaktadır. Son dosya işleminde gördüğümüz gibi farklılıkları ise boyutları farklıdır ve farklı yönlendirmeler ile oluşturulmuştur.

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. Sabit bir bağlantı oluşturulduğunda hangi bloklar yazılır? Bu sadece yeni bir dosya oluşturmaya nasıl benzer ve nasıl farklıdır? Bağlantılar oluşturuldukça referans sayısı alanı nasıl değişir?

   SORU 9 CEVAP :

   1. ./lfs.py -L c,/foo:l,/foo,/bar:l,/foo,/goo -o -i komutu çıktısı aşşağıdaki gibidir.

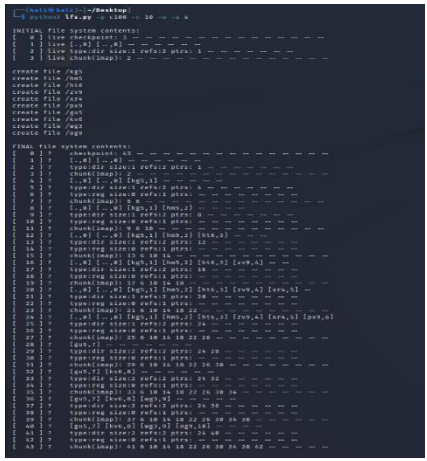


   2. Bağlantılar oluştukça referans sayısı alanı sürekli 4 artarak oluşmuştur. Dosya oluşturmayla aradaki fark bu kısımda görüldüğü üzere iki tane type olmasıdır. Uygulama bittiğinde foo bar ve goo'nun yazıldığını görmekteyiz.

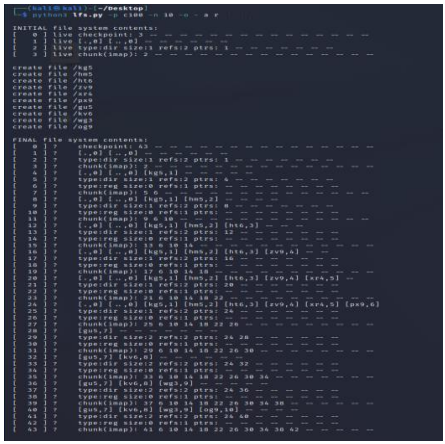
10. LFS birçok farklı politika kararı alır. Çoğunu burada keşfetmiyoruz belki de gelecek için kalan bir şey ama işte keşfettiğimiz basit bir şey: inode sayısının seçimi. İlk olarak, ./lfs.py -p c100 -n 10 -o -a s sıfıra yakın serbest inode numaralarını kullanmaya çalışan "sıralı" tahsis politikası ile olağan davranışı göstermek için. Ardından, ./lfs.py -p c100 -n 10 -o - a r komutunu çalıştırarak "rastgele" bir ilkeye geçin (-p c100 bayrağı, rasgele işlemlerin yüzde 100'ünün dosya oluşturma işlemi olmasını sağlar). Rastgele bir ilke ile sıralı ilke arasında hangi disk içi farklılıklar neden olur? Bu, gerçek bir LFS'de inode sayılarını seçmenin önemi hakkında ne söylüyor?

SORU 10 CEVAP :

1. ./lfs.py -p c100 -n 10 -o -a s komutu çıktısı aşşağıdaki gibidir.



2. ./lfs.py -p c100 -n 10 -o - a r komutu çıktısı aşşağıdaki gibidir.

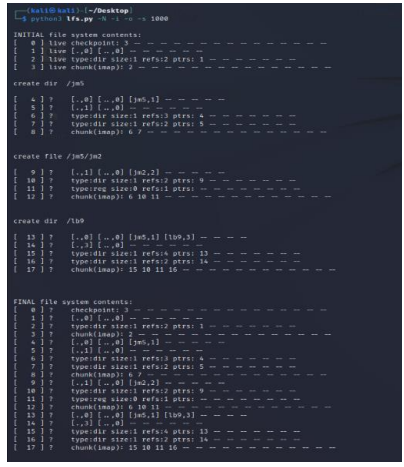


3. Bir sürücünün yüzeyine daha fazla bit paketlendikçe, söz konusu bitlere erişirken bant genişliği artar. Bununla birlikte, arama ve rotasyon gecikme maliyetleri yavaş yavaş azalmıştır; Ucuz ve küçük motorların plakaları daha hızlı döndürmesini veya disk kolunu daha hızlı hareket ettirmesini sağlamak zordur. Böylece, diskleri sıralı bir şekilde kullanabiliyorsanız, aramalara ve rotasyonlara neden olan yaklaşımlara göre büyük bir performans avantajı elde edersiniz.

11. Varsaydığımız son bir şey, LFS simülatörünün her güncellemeden sonra kontrol noktası bölgesini güncellemesidir. Gerçek LFS'de durum böyle değil: uzun arayışlardan kaçınmak için periyodik olarak güncellenir. Çalıştır ./lfs.py -N -i -o -s 1000 bazı işlemleri ve denetim noktası bölgesi diske zorlanmadığında dosya sisteminin ara ve son durumlarını görmek için. Kontrol noktası bölgesi hiçbir zaman güncellenmezse ne olur? Ya periodik olarak güncellenirse? Günlükte ileri sarılarak dosya sistemini en son durumuna nasıl kurtaracağınızı bulabilir misiniz?

SORU 11 CEVAP:

1. /lfs.py -N -i -o -s 1000 kodunun çıktısı aşşağıdaki gibidir.



2. Güncelleme olmazsa imap'in sabit konumu arasında daha fazla disk arayışı olur. Periyodik olarak güncelleme olursa çalışmada aksaklık olmaz performans kayıplarına çözüm olur.