Log-structured File Systems

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

• Memory sizes were growing (Bellek boyutları büyüyordu)

Bellek büyüdükçe, daha fazla veri bellekte önbelleğe alınabilir. Daha fazla veri önbelleğe alındıkça, disk trafiği okumalar hizmete gireceğinden, giderek artan bir şekilde yazmalardan oluşacaktır. önbellek. Bu nedenle, dosya sistemi performansı büyük ölçüde yazma performansı tarafından belirlenir.

• There was a large and growing gap between random I/O performance and sequential I/O performance (Rastgele G/Ç performansları arasında büyük ve büyüyen bir boşluk vardı.) creases roughly 50%-100% every year; seek and rotational delay costs decrease much more slowly, maybe at 5%-10% per year [P98]. Thus, if one is able to use disks in a sequential manner, one gets a huge performance advantage, which grows over time.

• Existing file systems perform poorly on many common workloads (Mevcut dosya sistemleri, birçok yaygın iş yükünde düşük performans gösterir)

Örneğin, FFS [MJLF84] çok sayıda yazma işlemi gerçekleştirir.

bir blok boyutunda yeni bir dosya oluşturmak için: biri yeni bir inode için, biri

inode bit eşlemini, biri dizin veri bloğuna güncelleyin.

dosya içinde, biri güncellemek için inode dizinine, biri yeni verilere

yeni dosyanın dışında bir blok ve veri bit eşlemine bir

veri bloğunu tahsis edilmiş olarak işaretleyin. Böylece, FFS yerleştirilse de

aynı blok grubu içindeki bu blokların tümü, FFS'ye neden olur

birçok kısa arama ve müteakip dönme gecikmeleri ve dolayısıyla performans, sıralı tepe bant genişliğinin çok gerisinde kalacaktır.

• File systems were not RAID-aware (Dosya sistemleri RAID uyumlu değildi)

Örneğin, RAID-4 ve RAID 5, small-write problem (küçük yazma sorunu) burada tek bir mantıksal yazma

bloğu 4 fiziksel I/O'nun gerçekleşmesine neden olur. Mevcut dosya sistemleri

bu en kötü RAID yazma davranışından kaçınmaya çalışmayın.

İdeal bir dosya sistemi bu nedenle yazma performansına odaklanır ve

diskin sıralı bant genişliğinden yararlanmak için. Ayrıca,

yalnızca veri yazmakla kalmayan, aynı zamanda

diskteki meta veri yapılarını sık sık güncelleyin. Sonunda, işe yarayacak

hem RAID'lerde hem de tek disklerde.

Rosenblum ve Ousterhout'un tanıttığı yeni dosya sistemi türü, LFS Günlük Yapılı Dosya Sisteminin kısaltması. Diske yazarken, LFS önce tüm güncellemeleri (meta veriler dahil!) bir iç bellek segmentinde arabelleğe alır; Segment dolduğunda diske tek seferde yazılır.

diskin kullanılmayan bir bölümüne uzun, sıralı aktarım, yani LFS hiçbir zaman

mevcut verilerin üzerine yazar, bunun yerine her zaman segmentleri boş konumlara yazar. Segmentler büyük olduğu için disk verimli bir şekilde kullanılır ve dosya sisteminin performansı doruk noktasına yaklaşır.

Sırayla Diske Yazma

Böylece ilk zorluğumuzla karşı karşıyayız: tüm güncellemeleri nasıl

dosya sistemi durumunu diske bir dizi sıralı yazma işlemine mi dönüştürüyor? Anlamak

bu daha iyi, basit bir örnek kullanalım. Bir veri bloğu yazdığımızı düşünün.

D bir dosyaya. Veri bloğunun diske yazılması aşağıdakilerle sonuçlanabilir

D'nin A0 disk adresinde yazılı olduğu disk düzeni

Ancak, bir kullanıcı bir veri bloğu yazdığında, elde edilen sadece veri değildir.

diske yazılır; güncellenmesi gereken başka meta veriler de vardır.

Bu durumda dosyanın inode'unu (I) da diske yazalım ve

veri bloğu D'yi işaret edin. Diske yazıldığında, veri bloğu ve inode

şuna benzer bir şey olurdu (inode'un şu kadar büyük göründüğüne dikkat edin:

genellikle böyle olmayan veri bloğu; çoğu sistemde, veri blokları

4 KB boyutundadır, oysa bir inode çok daha küçüktür, yaklaşık 128 bayttır)

43.2 Sıralı ve Etkili Yazma

Ne yazık ki, sırayla diske yazmak (tek başına) yeterli değildir.

verimli yazma garantisi. Örneğin, bir single yazdığımızı hayal edin.

blok A adresine, T zamanında. Sonra biraz bekleyip şu adrese yazıyoruz:

A + 1 adresindeki disk (sıralı sırada sonraki blok adresi),

ancak T + δ zamanında. Birinci ve ikinci yazma arasında, ne yazık ki,

disk döndü; ikinci yazıyı yayınladığınızda, böylece bekleyecektir

işlenmeden önce bir rotasyonun çoğu için (özellikle, eğer rotasyon

Dönme işlemi zaman alır, disk işleme başlamadan önce Dönme − δ bekleyecektir

disk yüzeyine ikinci yazma). Ve böylece umarım görebilirsin

diske sırayla yazmanın başarmak için yeterli olmadığını

en yüksek performans; bunun yerine, çok sayıda bitişik düzenleme yapmalısınız.

iyi yazma elde etmek için sürücüye yazar (veya bir büyük yazma)

verim.

Bu amaca ulaşmak için LFS, yazma olarak bilinen eski bir teknik kullanır.

tamponlama1

. Diske yazmadan önce, LFS güncellemeleri takip eder.

hafıza; yeterli sayıda güncelleme aldığında, yazar

hepsini birden diske atarak diskin verimli kullanılmasını sağlar.

LFS'nin bir defada yazdığı büyük güncelleme yığınına şu ad verilir:

bir segmentin adı. Bu terim bilgisayarda aşırı kullanılmasına rağmen

sistemler, burada sadece LFS'nin gruplamak için kullandığı büyük bir yığın anlamına gelir

yazar. Böylece, diske yazarken, LFS güncellemeleri bir bellek içi arabelleğe alır.

segment ve ardından segmentin hepsini bir kerede diske yazar. Olduğu sürece

segment yeterince büyükse, bu yazma işlemleri verimli olacaktır.

Burada, LFS'nin iki küme güncellemesini küçük bir dosyaya tamponladığı bir örnek verilmiştir.

bölüm; gerçek segmentler daha büyüktür (birkaç MB). İlk güncelleme

1

Gerçekten de, muhtemelen birçok kişi tarafından icat edildiğinden, bu fikir için iyi bir alıntı bulmak zordur.

ve bilgi işlem tarihinde çok erken. Yazma arabelleğe almanın faydalarına ilişkin bir çalışma için,

bkz. Solworth ve Orji [SO90]; potansiyel zararlarını öğrenmek için bkz. Mogul [M94].

43.3 Ne Kadar Tamponlanacak? Bu, şu soruyu gündeme getiriyor: LFS'nin kaç güncelleme yapması gerekiyor?

diske yazmadan önce tampon? Cevap, elbette, diske bağlıdır.

kendisi, özellikle konumlandırma ek yükünün ne kadar yüksek olduğu

aktarım hızı; benzer bir analiz için FFS bölümüne bakın.

Örneğin, her yazmadan önce konumlandırmanın (yani döndürme ve tura arama) kabaca T konumu saniyeler sürdüğünü varsayalım. Daha fazla varsayın

disk aktarım hızının Rpeak MB/s olduğunu. LFS tamponu ne kadar olmalıdır?

böyle bir disk üzerinde çalışırken yazmadan önce?

Bunu düşünmenin yolu, her yazdığınızda, bir

konumlandırma maliyetinin sabit ek yükü. Böylece, ne kadar var

bu maliyeti amortize etmek için yazmak? Ne kadar çok yazarsan o kadar iyi

(açıkça) ve en yüksek bant genişliğine ulaşmaya ne kadar yaklaşırsanız.

Somut bir cevap almak için D MB yazdığımızı varsayalım.

Bu veri yığınını yazma zamanı (Twrite), konumlandırma zamanıdır.

Tpozisyon artı transfer zamanı D (

D

Rpeak

), veya:

Twrite = Konum +

D

Rpeak

(43.1)

Ve böylece etkili yazma oranı (Reffici), ki bu sadece

yazılan veri miktarının, onu yazmak için harcanan toplam süreye bölümü şu şekildedir:

etkili =

D

yaz

=

D

Pozisyon +

D

Rpeak

. (43.2)

İlgilendiğimiz şey, efektif oranı (Reffekt) kapatmaktır.

en yüksek orana. Spesifik olarak, efektif oranın bir kesir olmasını istiyoruz

0 < F < 1 (tipik bir F 0,9 veya %90 olabilir)

tepe oranı). Matematiksel formda bu, Refect = istediğimiz anlamına gelir.

F × Rtepe.

Bu noktada, D için çözebiliriz:

etkili =

D

Pozisyon +

D

Rpeak

= F × Rtepe (43.3)

D = F × Rpeak × (Tpozisyonu +

D

Rpeak

) (43.4)

D = (F × Rtepe × Tkonumu) + (F × Rtepe ×

D

Rpeak

) (43.5)

D =

F

1 - F

× Rpeak × Tpozisyon (43.6)

Konumlandırma süresi 10 mil lisaniye ve tepe aktarım hızı 100 MB/sn olan bir diskle bir örnek yapalım; tepe noktasının %90'ı kadar etkili bir bant genişliği istediğimizi varsayalım (F = 0.9). Bu durumda D =

0.9

0,1 ×

100 MB/s × 0,01 saniye = 9 MB. görmek için bazı farklı değerler deneyin

tepe bant genişliğine yaklaşmak için ne kadar ara belleğe almamız gerektiği. Nasıl

Zirvenin %95'ine ulaşmak için çok şey gerekiyor? %99?

43.4 Problem: Düğümleri Bulma

LFS'de bir inode'u nasıl bulduğumuzu anlamak için, nasıl olduğunu kısaca gözden geçirelim.

tipik bir UNIX dosya sisteminde bir inode bulmak için. Tipik bir dosya sisteminde, örneğin

FFS veya hatta eski UNIX dosya sistemi olarak, inode'ları bulmak kolaydır, çünkü

bir dizi halinde düzenlenirler ve sabit konumlarda diske yerleştirilirler.

Örneğin, eski UNIX dosya sistemi tüm düğümleri diskin sabit bir bölümünde tutar. Böylece, bir inode numarası ve başlangıç adresi verildiğinde,

belirli bir inode bulmak için, tam disk adresini basitçe hesaplayabilirsiniz.

inode numarasını bir inode boyutuyla çarpmak ve bunu eklemek

diskteki dizinin başlangıç adresine; verilen dizi tabanlı indeksleme

inode numarası, hızlı ve basittir.

FFS'de bir inode numarası verilen bir inode bulmak sadece biraz daha fazladır.

karmaşık, çünkü FFS inode tablosunu parçalara ve yerlere böler

her bir silindir grubu içinde bir düğüm grubu. Böylece, kişi nasıl olduğunu bilmeli

büyük her inode öbeği ve her birinin başlangıç adresi. Bundan sonra,

hesaplamalar benzer ve aynı zamanda kolaydır.

LFS'de hayat daha zordur. Neden? Niye? Eh, dağıtmayı başardık

tüm disk boyunca düğümler! Daha da kötüsü, asla yerinde yazmayız ve

böylece bir inode'un en son sürümü (yani bizim istediğimiz) hareket etmeye devam eder. x

43.5 Dolaylı Çözüm: İnode Haritası:

Bunu düzeltmek için, LFS tasarımcıları bir düzeyde dolaylı

adı verilen bir veri yapısı aracılığıyla inode numaraları ve inode'lar arasında

inode haritası (imap). imap, inode numarası alan bir yapıdır.

girdi olarak ve en son sürümünün disk adresini üretir.

dosya numarası. Böylece, genellikle basit bir şekilde uygulanacağını hayal edebilirsiniz.

dizi, giriş başına 4 bayt (bir disk işaretçisi). Herhangi bir inode yazıldığında

diske, imap yeni konumu ile güncellenir.

43.6 Kontrol Noktası Bölgesi:

Zeki okuyucu (bu sizsiniz, değil mi?) bir sorun fark etmiş olabilir

burada. İnode haritasını nasıl bulacağız, artık parçaları da şimdi

diske yayıldı mı? Sonunda sihir yoktur: dosya sistemi

dosya aramaya başlamak için diskte bazı sabit ve bilinen konumlara sahip olmak.

LFS, bunun için diskte kontrol noktası bölgesi (CR) olarak bilinen böyle sabit bir yere sahiptir. Kontrol noktası bölgesi, inode haritasının en son parçalarına ve dolayısıyla inode haritasına yönelik işaretçiler içerir (yani bunların reklamları).

parçalar önce CR okunarak bulunabilir. Kontrol noktası bölgesine dikkat edin

yalnızca periyodik olarak güncellenir (örneğin her 30 saniyede bir) ve bu nedenle performans olumsuz etkilenmez. Böylece, disk düzeninin genel yapısı

bir kontrol noktası bölgesi içerir (ode haritasının en son parçalarına işaret eder); inode harita parçalarının her biri, inod e'ların adreslerini içerir; the

düğümler, tıpkı tipik UNIX dosya sistemleri gibi dosyalara (ve dizinlere) işaret eder.

İşte kontrol noktası bölgesinin bir örneği (bunun tüm yol boyunca olduğunu unutmayın)

diskin başlangıcı, 0 adresinde) ve tek bir imap yığın, inode,

ve veri bloğu. Gerçek bir dosya sisteminin elbette çok daha büyük bir

CR (aslında, daha sonra anlayacağımız gibi, iki tane olacaktır), birçok

imap parçaları ve tabii ki daha birçok inode, veri bloğu vb.

43.7 Diskten Dosya Okuma: Bir Reca: LFS'nin nasıl çalıştığını anladığınızdan emin olmak için şimdi gözden geçirelim

diskten bir dosya okumak için ne olması gerekir. elimizde hiçbir şey olmadığını varsayalım

başlamak için hafıza. Okumamız gereken ilk disk üzerindeki veri yapısı,

kontrol noktası bölgesi Kontrol noktası bölgesi, tüm inode haritasına işaretçiler (yani, disk address) içerir ve bu nedenle LFS, daha sonra ode haritasının tamamını okur ve onu bellekte önbelleğe alır. Bu noktadan sonra inode verildiğinde

Bir dosyanın numarası, LFS, imap'ta inode numarasından inode disk adres eşlemesine bakar ve dosyanın en son sürümünü okur.

dosya numarası. Dosyadan bir blok okumak için bu noktada LFS aynen devam eder.

doğrudan işaretçiler veya dolaylı işaretçiler kullanarak tipik bir UNIX dosya sistemi olarak

veya gerektiği gibi iki kat dolaylı işaretçiler. Yaygın durumda, LFS

okurken tipik bir dosya sistemiyle aynı sayıda G/Ç gerçekleştirir.

diskten dosya; tüm imap önbelleğe alınır ve bu nedenle LFS'nin yaptığı ekstra iş

okuma sırasında, imapta inode adresini aramaktır.

43.8 What About Directories? Buraya kadar sadece gazelleri ve veri bloklarını ele alarak tartışmamızı biraz basitleştirdik. Ancak, bir dosya sistemindeki bir dosyaya (örneğin,

/home/remzi/foo, en sevdiğimiz sahte dosya adlarından biri), bazı dizinlere de erişilmesi gerekiyor. Peki, LFS dizin verilerini nasıl depolar? Neyse ki, dizin yapısı temel olarak klasik UNIX ile aynıdır.

dosya sistemleri, burada bir dizin yalnızca (ad, inode numarası) koleksiyonudur

eşlemeler. Örneğin, diskte bir dosya oluştururken, LFS'nin her ikisinin de yazması gerekir.

yeni bir inode, bazı veriler, ayrıca dizin verileri ve onun inode'u

bu dosyaya bakın. LFS'nin bunu diskte sırayla yapacağını unutmayın.

(güncellemeleri bir süre arabelleğe aldıktan sonra). Böylece, bir foo dosyası oluşturmak

dizin, diskte aşağıdaki yeni yapılara yol açacaktır. İnode haritasının parçası, konumu için bilgileri içerir.

hem dizin dosyası dir hem de yeni oluşturulan dosya f. Böylece, ne zaman

foo dosyasına erişirken (f inode numaralı), önce

inode'un yerini bulmak için inode haritası (genellikle bellekte önbelleğe alınır)

dizin dizini (A3); daha sonra size veren inode dizini okursunuz

dizin verilerinin konumu (A2); bu veri bloğunu okumak size

(foo, k)'nin isim-inode-numarası eşlemesi. Daha sonra danışmak

inode numarası k (A1) inode konumunu bulmak için tekrar inode haritası ve son olarak

A0 adresinde istenen veri bloğunu okuyun.

LFS'de inode haritasının çözdüğü ciddi bir sorun daha var.

özyinelemeli güncelleme sorunu [Z+12] olarak bilinir. sorun ortaya çıkıyor

hiçbir zaman yerinde güncellenmeyen herhangi bir dosya sisteminde (LFS gibi), bunun yerine

güncellemeleri diskteki yeni konumlara taşır.

Spesifik olarak, bir inode her güncellendiğinde diskteki konumu değişir.

Dikkatli olmasaydık, bu aynı zamanda bir güncelleme gerektirecekti.

bu dosyaya işaret eden ve daha sonra zorunlu kılınan dizin

bu dizinin üst öğesinde bir değişiklik vb. dosyanın sonuna kadar

sistem ağacı.

LFS, inode haritasıyla bu sorunu akıllıca ortadan kaldırır. Rağmen

bir inode'un konumu değişebilir, değişiklik asla

dizinin kendisi; bunun yerine, dizin güncellenirken imap yapısı güncellenir.

aynı isimden numaraya eşlemeye sahiptir. Böylece dolaylı yoldan,

LFS, özyinelemeli güncelleme sorununu önler.

43.9 Yeni Bir Sorun: Çöp Toplama :

LFS ile ilgili başka bir sorun fark etmiş olabilirsiniz; yazmaya devam ediyor

bir dosyanın daha yeni sürümü, inode'u ve aslında tüm veriler dosyanın yeni bölümlerine

disk. Bu süreç, yazma işlemlerini verimli tutarken, LFS'nin ayrıldığını ima eder.

diskin her yerine dağılmış dosya yapılarının eski sürümleri

disk. Bu tür eski şeylere çöp diyoruz. LFS ile ilgili başka bir sorun fark etmiş olabilirsiniz; yazmaya devam ediyor

bir dosyanın daha yeni sürümü, inode'u ve aslında tüm veriler dosyanın yeni bölümlerine

disk. Bu süreç, yazma işlemlerini verimli tutarken, LFS'nin ayrıldığını ima eder.

diskin her yerine dağılmış dosya yapılarının eski sürümleri

disk. Bu tür eski şeylere çöp diyoruz. Diyagramda, hem inode hem de data bloğunun sahip olduğunu görebilirsiniz.

diskte iki versiyon, biri eski (soldaki) ve biri güncel ve

böylece yaşa (sağdaki). Bir verinin üzerine yazma gibi basit bir işlemle

blok, bir dizi yeni yapının LFS tarafından kalıcı olması gerekir, böylece

diskteki söz konusu blokların eski sürümleri.

Başka bir örnek olarak, orijinal k dosyasına bir blok eklediğimizi hayal edin. Bu durumda, inode'un yeni bir sürümü oluşturulur, ancak

eski veri bloğu hala inode tarafından işaret ediliyor. Böylece, hala canlı ve çok Öyleyse, inode'ların, veri bloklarının bu eski sürümleriyle ne yapmalıyız?

ve benzeri? Biri bu eski sürümleri etrafta tutabilir ve izin verebilir

kullanıcıların eski dosya sürümlerini geri yüklemelerini sağlar (örneğin, yanlışlıkla

bir dosyanın üzerine yazın veya silin, bunu yapmak oldukça kullanışlı olabilir); böyle bir dosya

sistem bir versiyonlama dosya sistemi olarak bilinir, çünkü

bir dosyanın farklı sürümleri.

Ancak, LFS bunun yerine bir dosyanın yalnızca en son canlı sürümünü tutar; böylece

(arka planda), LFS'nin bu eski ölü sürümleri periyodik olarak bulması gerekir

dosya verilerinin, düğümlerin ve diğer yapıların temizlenmesi ve temizlenmesi; temizlik gerekir

böylece sonraki yazmalarda kullanılmak üzere diskteki blokları tekrar serbest bırakın. Not

temizleme işleminin bir tür çöp toplama, bir teknik olduğunu

Programlar için kullanılmayan belleği otomatik olarak serbest bırakan programlama dillerinde ortaya çıkan.

Mekanizma kadar önemli olan segmentleri daha önce tartışmıştık.

bu, LFS'de diske büyük yazma işlemlerine olanak tanır. Görünüşe göre, onlar da oldukça

etkili temizliğin ayrılmaz bir parçasıdır. LFS olsaydı ne olacağını hayal edin temizleyici basitçe geçti ve tek veri bloklarını, düğümleri vb. serbest bıraktı,

temizlik sırasında. Sonuç: bir miktar boş deliğe sahip bir dosya sistemi

diskte ayrılan alan arasında karışık. Yazma performansı düşer

LFS geniş bir bitişik bölge bulamayacağı için önemli ölçüde

sıralı ve yüksek performansla diske yazmak için.

Bunun yerine, LFS temizleyici segment bazında çalışır, böylece

sonraki yazma için büyük alan parçalarını temizlemek. Temel temizleme işlemi aşağıdaki gibi çalışır. Periyodik olarak, LFS temizleyici bir

eski (kısmen kullanılmış) segment sayısı, hangi blokların kullanılacağını belirler

bu segmentler içinde yaşayın ve ardından yeni bir segment seti yazın

eskileri yazmak için serbest bırakan sadece içlerindeki canlı bloklarla.

Spesifik olarak, temizleyicinin M mevcut segmentte okumasını, içeriklerini N yeni segmentte (burada N < M) sıkıştırmasını ve ardından yazmasını bekliyoruz.

yeni konumlarda diske N segmenti. Eski M segmentleri daha sonra

serbest bırakılır ve sonraki yazma işlemleri için dosya sistemi tarafından kullanılabilir.

Ancak şimdi iki sorunumuz kaldı. Birincisi mekanizmadır:

LFS, bir segment içindeki hangi blokların aktif olduğunu ve hangilerinin aktif olduğunu nasıl anlayabilir?

ölü? İkincisi, politikadır: temizleyici ne sıklıkta çalışmalı ve hangisi

segmentleri temizlemek için seçmeli mi?

43.10 Blok Canlılığının Belirlenmesi: Önce mekanizmayı ele alıyoruz. Bir disk segmenti S içinde bir veri bloğu D verildiğinde, LFS, D'nin canlı olup olmadığını belirleyebilmelidir. Yapmak

bu nedenle LFS, her birini tanımlayan her bir bölüme biraz fazladan bilgi ekler.

engellemek. Spesifik olarak LFS, her D veri bloğu için inode numarasını içerir.

(hangi dosyaya ait) ve ofseti (bu dosyanın hangi bloğudur). Bu

bilgi bilinen segmentin başındaki bir yapıda kayıtlıdır.

segment özet bloğu olarak.

Bu bilgi göz önüne alındığında, olup olmadığını belirlemek basittir.

blok canlı veya ölü. A adresindeki diskte bulunan bir D bloğu için, bakın

segment özet bloğunda ve inode numarası N'yi ve ofseti bulun

T. Ardından, N'nin nerede yaşadığını bulmak için imap'e bakın ve N'yi diskten okuyun

(belki zaten hafızadadır, ki bu daha da iyidir). Son olarak, kullanarak

ofset T, nerede olduğunu görmek için inode'a (veya bazı dolaylı bloğa) bakın.

inode, bu dosyanın Tth bloğunun diskte olduğunu düşünüyor. Tam olarak diske işaret ediyorsa

A adresi, LFS, D bloğunun canlı olduğu sonucuna varabilir. Herhangi bir yere işaret ediyorsa

aksi takdirde LFS, D'nin kullanımda olmadığı (yani ölü olduğu) sonucuna varabilir ve böylece şunu bilir:

bu sürüme artık gerek kalmadığını. Bunun sözde kod özeti

süreç burada gösterilmektedir:

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
```

```
if (inode[T] == A)
// block D is alive
else
 // block D is garbage
```

İşte segmentin içinde bulunduğu mekanizmayı gösteren bir diyagram

özet bloğu (SS olarak işaretlenmiştir), A0 adresindeki veri bloğunun

aslında 0 uzaklığında k dosyasının bir parçasıdır. k için imap'i kontrol ederek şunları yapabilirsiniz:

inode'u bulun ve gerçekten o konumu işaret ettiğini görün.

LFS'nin belirleme sürecini yapmak için kullandığı bazı kısayollar vardır.

canlılık daha verimli. Örneğin, bir dosya kesildiğinde veya silindiğinde,

LFS sürüm numarasını artırır ve yeni sürüm numarasını

imap. Diskteki segmentte sürüm numarasını da kaydederek,

LFS, diskteki sürüm numarasını imap'teki bir sürüm numarasıyla karşılaştırarak yukarıda açıklanan daha uzun kontrolü kısa devre yapabilir, böylece

ekstra okumalardan kaçınmak.

43.11 Bir Politika Sorusu: Hangi Bloklar, Ne Zaman Temizlenmeli?

Yukarıda açıklanan mekanizmanın yanı sıra, LFS bir dizi

hem ne zaman temizleneceğini hem de hangi blokların değerli olduğunu belirleyen politikalar

temizlik. Ne zaman temizleneceğini belirlemek daha kolaydır; ya periyodik olarak, boşta kaldığınızda ya da disk dolu olduğu için mecbur kaldığınızda.

Hangi blokların temizleneceğini belirlemek daha zordur ve

birçok araştırma makalesinin konusu. Orijinal LFS kağıdında [RO91],

yazarlar sıcağı ve soğuğu ayırmaya çalışan bir yaklaşımı anlatıyor

bölüm. Sıcak segment, içeriğin sık sık sunulduğu segmenttir.

fazla yazılmış; bu nedenle böyle bir segment için en iyi politika uzun süre beklemektir.

Temizlemeden önceki süre, daha fazla blok üzerine yazıldığı için

(yeni segmentlerde) ve böylece kullanım için serbest bırakılıyor. Bunun aksine, soğuk bir segment birkaç ölü bloğa sahip olabilir, ancak içeriğinin geri kalanı nispeten

kararlı. Böylece yazarlar, soğuk segmentlerin temizlenmesi gerektiği sonucuna varmışlardır.

er ve sıcak segmentler sonra ve tam olarak yapan bir buluşsal yöntem geliştirin

o. Ancak, çoğu politikada olduğu gibi, bu yalnızca bir yaklaşımdır ve

tanım "en iyi" yaklaşım değildir; sonraki yaklaşımlar nasıl yapılacağını gösterir

daha iyi [MR+97].

43.12 Kilitlenme Kurtarma ve Günlük?

Son bir sorun: LFS çalışırken sistem çökerse ne olur?

diske yazmak? Günlük tutma ile ilgili önceki bölümde hatırlayabileceğiniz gibi, güncellemeler sırasındaki çökmeler dosya sistemleri için zordur ve bu nedenle bazı LFS'nin de dikkate alması gereken bir şey.

Normal çalışma sırasında, LFS arabellekleri bir segmentte yazar ve ardından

(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 segmentini işaret eder ve her segment

yazılacak bir sonraki bölüme işaret eder. LFS ayrıca periyodik olarak günceller.

kontrol noktası bölgesi Bunlardan herhangi biri sırasında çökmeler açıkça meydana gelebilir

işlemler (segmente yaz, CR'ye yaz). Peki LFS nasıl işliyor?

bu yapılara yazma sırasında çöküyor?

Önce ikinci durumu ele alalım. Hazır yanıt güncellemesinin gerçekleşmesini sağlamak için

atomik olarak, LFS aslında diskin her iki ucunda bir tane olmak üzere iki CR tutar ve

dönüşümlü olarak onlara yazar. LFS aynı zamanda dikkatli bir protokol uygular.

inode haritasına ve diğer bilgilere en son işaretçiler ile CR'nin güncellenmesi; özellikle, önce bir başlık (zaman damgasıyla birlikte) yazar, ardından

CR'nin gövdesi ve son olarak son bir blok (ayrıca bir zaman damgasıyla). Eğer

CR güncellemesi sırasında sistem çökerse, LFS bunu bir

tutarsız zaman damgası çifti. LFS her zaman en çok kullanmayı seçecektir

Tutarlı zaman damgalarına ve dolayısıyla tutarlı güncellemeye sahip olan son hazır yanıt

CR elde edilir.

Şimdi ilk durumu ele alalım. Çünkü LFS CR'yi her 30'da bir yazar.

saniye kadar, dosya sisteminin son tutarlı anlık görüntüsü oldukça

eskimiş. Böylece, yeniden başlatmanın ardından, LFS yalnızca içindekileri okuyarak kolayca kurtarılabilir.

kontrol noktası bölgesi, işaret ettiği imap parçaları ve sonraki dosyalar ve

dizinler; ancak güncellemelerin son birkaç saniyesi kaybolabilir.

Bunu geliştirmek için LFS, bu segmentlerin çoğunu yeniden oluşturmaya çalışır.

veritabanı topluluğunda ileri alma olarak bilinen bir teknik aracılığıyla.

Temel fikir, son kontrol noktası bölgesinden başlamak, sonunu bulmaktır.

günlük (CR'de bulunan) ve ardından okumak için bunu kullanın

sonraki bölümler ve içinde herhangi bir geçerli güncelleme olup olmadığına bakın. eğer varsa

LFS, dosya sistemini buna göre günceller ve böylece çoğunu kurtarır.

son kontrol noktasından bu yana yazılan veriler ve meta veriler. Rosenblum'a bakın

ayrıntılar için ödüllü tez [R92].

43.13 Özet: LFS, diski güncellemek için yeni bir yaklaşım sunar. Dosyaları yerlere fazla yazmak yerine, LFS her zaman kullanılmayan bir bölüme yazar.

disk ve daha sonra temizleme yoluyla bu eski alanı geri kazanır. Veritabanı sistemlerinde gölge sayfalama [L77] olarak adlandırılan bu yaklaşım ve

dosya sistemi konuşması bazen yazma sırasında kopya olarak adlandırılır ve LFS tüm güncellemeleri bir bellek içi segmen tte toplayabildiğinden oldukça verimli yazmaya olanak tanır

ve sonra bunları sırayla birlikte yazın.

Bu yaklaşımın dezavantajı, çöp üretmesidir; eski kopyalar

verilerin çoğu disk boyunca dağılmıştır ve eğer biri geri almak isterse

Böyle bir alan sonraki kullanım için, eski segmentler periyodik olarak temizlenmelidir. Temizlik, LFS'deki birçok tartışmanın odak noktası haline geldi ve temizlik maliyetleri [SS+95] hakkındaki endişeler, belki de LFS'nin üzerindeki ilk etkisini sınırladı.

alan. Ancak, Ne tApp'ın WAFL [HLM94], Sun'ın ZFS [B07] ve Linux btrfs [M07] gibi bazı modern ticari dosya sistemleri,

diske yazmaya yönelik benzer bir yazma üzerine kopyalama yaklaşımı ve dolayısıyla LFS'nin entelektüel mirası bu modern dosya sistemlerinde yaşamaya devam ediyor. Özellikle,

WAFL, temizlik sorunlarını bir özelliğe dönüştürerek ortadan kaldırdı; ile

dosya sisteminin eski sürümlerini anlık görüntüler aracılığıyla sağlayan kullanıcılar, mevcut dosyaları yanlışlıkla sildiklerinde eski dosyalara erişebilirler.

ceviri bitmistir.

soru : 1. ./lfs.py -n 3'ü çalıştırın, belki çekirdeği (-s) değiştirerek. Nihai dosya sistemini oluşturmak için hangi komutların çalıştırıldığını bulabilir misiniz? içindekiler? Bu komutların hangi sırayla verildiğini söyleyebilir misiniz?

Son olarak, finaldeki her bloğun canlılığını belirleyebilir misiniz?
dosya sistemi durumu? Hangi komutların çalıştırıldığını göstermek için -o
kullanın ve
-c son dosya sistemi durumunun canlılığını göstermek için. Ne kadar
sayısını artırdıkça görev sizin için zorlaşıyor
verilen komutlar (yani, -n 3'ü -n 5 olarak değiştirin)

```
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 ] ?    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 -- -- -- -- -- -- -- -- -- -- -- -- --
```

yorum: block 0,8,9,11,12,13,14 aktiftir.

2. soru : Yukarıdakileri acı verici buluyorsanız, kendinize biraz yardım edebilirsiniz.

her belirli komutun neden olduğu güncelleme kümesini gösterir. Yapmak ./lfs.py -n 3 -i komutunu çalıştırın. Şimdi anlamanın daha kolay olup olmadığına bakın

her komutun ne olması gerektiği. Rastgele çekirdeği şu şekilde değiştirin: yorumlamak için farklı komutlar alın (örneğin, -s 1, -s 2, -s 3, vb.).

yorum: -i komudu kesinlikle işleri kolaylaştırıyor komutları anlamamız için

3. soru ; Hangi güncellemelerin yapıldığını anlama yeteneğinizi daha fazla test etmek için
disk her komutla, aşağıdakini çalıştırın: ./lfs.py -o -F -s
100 (ve belki birkaç başka rastgele tohum). Bu sadece bir gösterir
komut kümesidir ve size dosyanın son durumunu GÖSTERMEZ
sistem. Dosya sisteminin son durumunun ne olduğu hakkında akıl yürütebilir misiniz?
olmalıdır?

```
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
```

4. soru : Şimdi hangi dosyaların ve dizinlerin yayında olduğunu belirleyip belirleyemeyeceğinize bakın. bir dizi dosya ve dizin işleminden sonra. tt ./lfs.py'yi çalıştırın
-n 20 -s 1 ve ardından son dosya sistemi durumunu inceleyin. Yapabilir misin
hangi yol adlarının geçerli olduğunu anladınız mı? tt ./lfs.py -n 20'yi çalıştırın
-s 1 -c -v sonuçlarını görmek için. Cevaplarınızın olup olmadığını görmek için -o ile çalıştırın. Rastgele komutlar dizisi verildiğinde eşleştirin. Daha fazla sorun elde etmek için farklı rastgele tohumlar kullanın

gr

```
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 ] ?    n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0n0
[  15 ] ?    y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1y1
[  16 ] ?    p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2p2
[  17 ] ?    l3l3l3l3l3l3l3l3l3l3l3l3l3l3l3l3
```

```
[  17 ] ?    l3l3l3l3l3l3l3l3l3l3l3l3l3l3l3
[  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 ] ?    v1v1v1v1v1v1v1v1v1v1v1v1v1v1v1v1
[  36 ] ?    x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2
[  37 ] ?    t3t3t3t3t3t3t3t3t3t3t3t3t3t3t3t3
[  38 ] ?    v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4v4
[  39 ] ?    n5n5n5n5n5n5n5n5n5n5n5n5n5n5n5n5
[  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 ] ?    n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6n6
[  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
```

```
[  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
[  67 ] ?    chunk(imap): 52 53 40 66 -- -- -- -- -- -- -- -- -- -- --
[  68 ] ?    u0u0u0u0u0u0u0u0u0u0u0u0u0u0u0u0
[  69 ] ?    v1v1v1v1v1v1v1v1v1v1v1v1v1v1v1v1
[  70 ] ?    g2g2g2g2g2g2g2g2g2g2g2g2g2g2g2g2
[  71 ] ?    v3v3v3v3v3v3v3v3v3v3v3v3v3v3v3v3
[  72 ] ?    r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4r4
[  73 ] ?    c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5
[  74 ] ?    type:reg size:8 refs:1 ptrs: 34 68 69 70 71 72 73 20
[  75 ] ?    chunk(imap): 52 53 74 66 -- -- -- -- -- -- -- -- -- -- --
[  76 ] ?    a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0
[  77 ] ?    a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1
[  78 ] ?    t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2t2
[  79 ] ?    g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3
[  80 ] ?    type:reg size:8 refs:1 ptrs: 34 68 69 70 76 77 78 79
[  81 ] ?    chunk(imap): 52 53 80 66 -- -- -- -- -- -- -- -- -- -- --
[  82 ] ?    [.,0] [..,0] [ln7,4] [lt0,2] [oy3,1] [af4,3] -- --
[  83 ] ?    [.,4] [..,0] -- -- -- -- -- --
[  84 ] ?    type:dir size:1 refs:3 ptrs: 82 -- -- -- -- -- -- --
[  85 ] ?    type:dir size:1 refs:2 ptrs: 83 -- -- -- -- -- -- --
[  86 ] ?    chunk(imap): 84 53 80 66 85 -- -- -- -- -- -- -- -- -- -- --
[  87 ] ?    type:reg size:8 refs:1 ptrs: -- 42 43 44 45 46 47 48
[  88 ] ?    chunk(imap): 84 87 80 66 85 -- -- -- -- -- -- -- -- -- -- --
[  89 ] ?    [.,4] [..,0] [zp3,5] -- -- -- -- --
[  90 ] ?    type:dir size:1 refs:2 ptrs: 89 -- -- -- -- -- -- --
[  91 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  92 ] ?    chunk(imap): 84 87 80 66 90 91 -- -- -- -- -- -- -- -- -- --
[  93 ] ?    [.,4] [..,0] [zp3,5] [zu5,6] -- -- -- --
[  94 ] ?    type:dir size:1 refs:2 ptrs: 93 -- -- -- -- -- -- --
[  95 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  96 ] ?    chunk(imap): 84 87 80 66 94 91 95 -- -- -- -- -- -- -- -- --
[  97 ] ?    [.,0] [..,0] [ln7,4] [lt0,2] -- [af4,3] -- --
[  98 ] ?    type:dir size:1 refs:3 ptrs: 97 -- -- -- -- -- -- --
[  99 ] ?    chunk(imap): 98 -- 80 66 94 91 95 -- -- -- -- -- -- -- -- --
```

```
Live directories:  ['/ln7']
Live files:  ['/lt0', '/af4', '/ln7/zp3', '/ln7/zu5']
```

yorum: Aktif dosyaları ve dizinleri bulmak için son imap'a bakmak gerekir.

5. soru: Şimdi bazı özel komutlar verelim. Öncelikle bir dosya oluşturalım
ve tekrar tekrar yazın. Bunu yapmak için, size izin veren -L bayrağını kullanın.
yürütülecek belirli komutları belirtin. ”/foo” dosyasını oluşturalım
ve ona dört kez yaz:
-L c,/foo:w,/foo,0,1:w,/foo,1,1:w,/foo,2,1:w,/foo,3,1
-Ö. Son dosya sisteminin canlılığını belirleyip belirleyemeyeceğinize bakın.
belirtmek, bildirmek; cevaplarınızı kontrol etmek için -c kullanın?

```
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 ] live checkpoint: 19 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   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 ]      type:reg size:1 refs:1 ptrs: 8 -- -- -- -- -- -- --
[  10 ]      chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  11 ] live t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0t0
[  12 ]      type:reg size:2 refs:1 ptrs: 8 11 -- -- -- -- -- --
[  13 ]      chunk(imap): 5 12 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  14 ] live k0k0k0k0k0k0k0k0k0k0k0k0k0k0k0k0
[  15 ]      type:reg size:3 refs:1 ptrs: 8 11 14 -- -- -- -- --
[  16 ]      chunk(imap): 5 15 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  17 ] live g0g0g0g0g0g0g0g0g0g0g0g0g0g0g0g0
[  18 ] live type:reg size:4 refs:1 ptrs: 8 11 14 17 -- -- -- --
[  19 ] live chunk(imap): 5 18 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

yorum: Block 0,4,5,8,11,14,17,18,19 canlı ve yeniden sonuncu imap yardım ediyor.

6. soru: Şimdi aynı şeyi dört yerine tek bir yazma işlemiyle yapalım. ./lfs.py -o -L c,/foo:w,/foo,0,4 komutunu çalıştırın.
"/foo" dosyasını oluşturun ve tek bir yazma işlemiyle 4 blok yazın Canlılığı tekrar hesaplayın ve -c ile doğru olup olmadığınızı kontrol edin.

Bir dosyayı tek seferde yazmak arasındaki temel fark nedir?
burada yapıyoruz) her seferinde bir blok yapmaya mı (yukarıdaki gibi)? Ne
bu size güncellemeleri ana bellekte arabelleğe almanın önemini anlatıyor mu?
gerçek LFS'nin yaptığı gibi bellek?

```
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: 13 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   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 ] ?    t1t1t1t1t1t1t1t1t1t1t1t1t1t1t1t1
[  10 ] ?    k2k2k2k2k2k2k2k2k2k2k2k2k2k2k2k2
[  11 ] ?    g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3g3
[  12 ] ?    type:reg size:4 refs:1 ptrs: 8 9 10 11 -- -- -- --
[  13 ] ?    chunk(imap): 5 12 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

yorum: Daha fazla yazma işlemi üretmenin ynaı sıra, küçük yazma işlemleri daha fazla çöp üretti.

7.soru: Başka bir özel örnek yapalım. Önce şunu çalıştırın: ./lfs.py
-L c,/foo:w,/foo,0,1. Bu komut dizisi ne işe yarar? Şimdi ./lfs.py -L c,/foo:w,/foo,7,1 komutunu çalıştırın. bu ne anlama geliyor
komut dizisi yapar? İkisi nasıl farklı? ne söyleyebilirsin
bu iki komut kümesinden inode'daki boyut alanı hakkında?

```
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 ] ?    v0v0v0v0v0v0v0v0v0v0v0v0v0v0v0
[   9 ] ?    type:reg size:8 refs:1 ptrs: -- -- -- -- -- -- -- 8
[  10 ] ?    chunk(imap): 5 9 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

yorum : Bir block başta yazılır. bir block sonda yazılır. ilk büyüklük 1 ikincisi 8dir

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

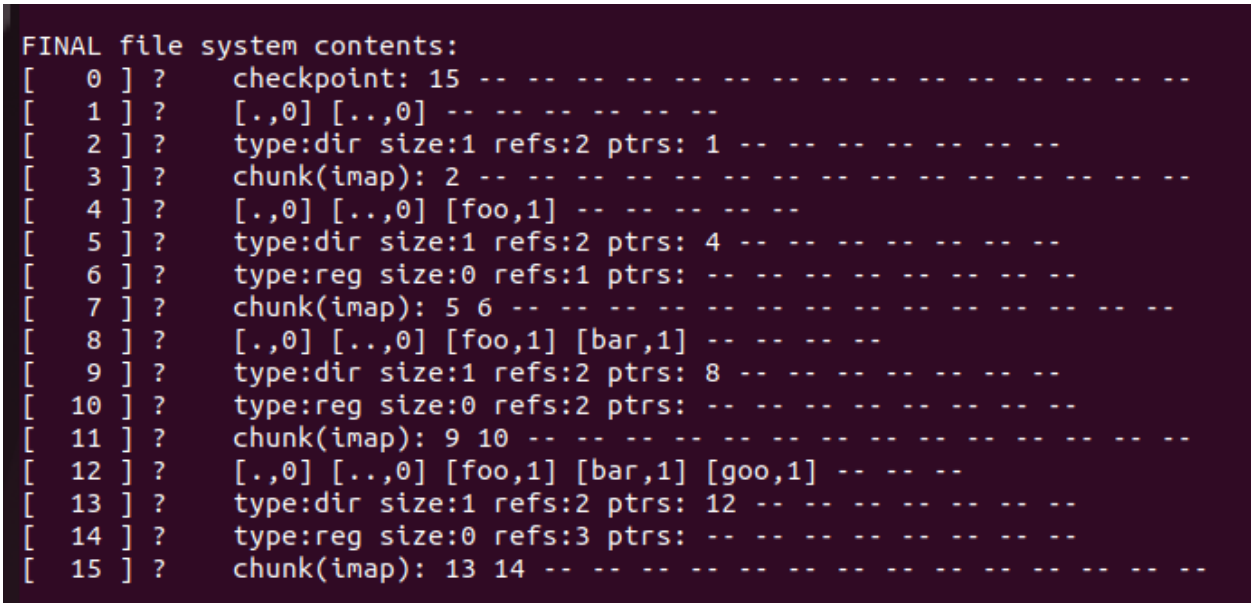
```
INITIAL file system contents:
[   0 ] live checkpoint: 3 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   1 ] live [.,0] [..,0] -- -- -- -- -- --
[   2 ] live type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] live chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

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 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

yorum: Oluşturulan dosyanın ilk başta bir veri bloğu yoktur, ancak dizinin hemen bir veri bloğu vardır ve ayrıca kök dizine bir referansı artırır.

Soru9: LFS simülatörü, sabit bağlantıları da destekler. Aşağıdakileri çalıştırın
nasıl çalıştıklarını incelemek için:
./lfs.py -L c,/foo:l,/foo,/bar:l,/foo,/goo -o -i.

Bir sabit bağlantı oluşturulduğunda hangi bloklar yazılır? Nasıl
bu sadece yeni bir dosya oluşturmaya benzer ve nasıl farklıdır? Nasıl
bağlantılar oluşturuldukça referans sayısı alanı değişiyor mu?

```
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

[   0 ] ?    checkpoint: 7 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[   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 -- -- -- -- -- -- -- -- -- -- -- -- --


link file   /foo /bar

[   0 ] ?    checkpoint: 11 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[   8 ] ?    [.,0] [..,0] [foo,1] [bar,1] -- -- -- --
[   9 ] ?    type:dir size:1 refs:2 ptrs: 8 -- -- -- -- -- -- --
[  10 ] ?    type:reg size:0 refs:2 ptrs: -- -- -- -- -- -- -- --
[  11 ] ?    chunk(imap): 9 10 -- -- -- -- -- -- -- -- -- -- -- -- -- --


link file   /foo /goo

[   0 ] ?    checkpoint: 15 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
...
[  12 ] ?    [.,0] [..,0] [foo,1] [bar,1] [goo,1] -- -- --
[  13 ] ?    type:dir size:1 refs:2 ptrs: 12 -- -- -- -- -- -- --
[  14 ] ?    type:reg size:0 refs:3 ptrs: -- -- -- -- -- -- -- --
[  15 ] ?    chunk(imap): 13 14 -- -- -- -- -- -- -- -- -- -- -- -- -- --
```



Yorum: Kontrol noktası bölgesi, üst dizin veri bloğu ve inode, imap.

Aynı: her ikisi de güncelleme kontrol noktası bölgesi, üst dizin veri bloğu ve inode, imap.

Farklı: yeni dosya oluştur, bunun için yeni bir inode oluşturur, sabit bağlantı oluştur yeni inode oluşturmaz.

Bağlantılı dosyanın referans sayısını bir artıracaktır.

`Soru10:` LFS birçok farklı politika kararı verir. biz keşfetmeyiz
birçoğu burada – belki geleceğe kalan bir şeyler – ama burada
keşfettiğimiz basit bir tanesi: inode numarasının seçimi. İlk önce koş
./lfs.py -p c100 -n 10 -o -as ücretsiz kullanmaya çalışan "sıralı" tahsis politikası ile olağan davranışı göstermek için

sıfıra en yakın inode numaraları. Ardından, "rastgele" bir politikaya geçin
./lfs.py -p c100 -n 10 -o -ar çalıştırarak (-p c100 bayrağı, rastgele işlemlerin yüzde 100'ünün dosya oluşturma olmasını sağlar).
Rastgele bir ilkenin sıralı bir ilkeye karşı disk üzerindeki farkları nelerdir?
politika sonucu? Bu, gerçek bir LFS'de inode numaralarını seçmenin önemi hakkında ne söylüyor?

```
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

FINAL file system contents:
[   0 ] ?    checkpoint: 52 38 -- -- -- 23 -- 13 53 -- -- 48 8 -- 33 --
[   1 ] ?    [.,0] [..,0] -- -- -- -- -- --
[   2 ] ?    type:dir size:1 refs:2 ptrs: 1 -- -- -- -- -- -- --
[   3 ] ?    chunk(imap): 2 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   4 ] ?    [.,0] [..,0] [kg5,205] -- -- -- -- --
[   5 ] ?    type:dir size:1 refs:2 ptrs: 4 -- -- -- -- -- -- --
[   6 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[   7 ] ?    chunk(imap): 5 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[   8 ] ?    chunk(imap): -- -- -- -- -- -- -- -- -- -- -- -- -- -- 6 -- --
[   9 ] ?    [.,0] [..,0] [kg5,205] [hm5,114] -- -- -- --
[  10 ] ?    type:dir size:1 refs:2 ptrs: 9 -- -- -- -- -- -- --
[  11 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  12 ] ?    chunk(imap): 10 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  13 ] ?    chunk(imap): -- -- 11 -- -- -- -- -- -- -- -- -- -- -- -- --
[  14 ] ?    [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] -- -- --
[  15 ] ?    type:dir size:1 refs:2 ptrs: 14 -- -- -- -- -- -- --
[  16 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  17 ] ?    chunk(imap): 15 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  18 ] ?    chunk(imap): -- -- -- -- 16 -- -- -- -- -- -- -- -- -- -- --
[  19 ] ?    [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] [zv9,81] -- --
[  20 ] ?    type:dir size:1 refs:2 ptrs: 19 -- -- -- -- -- -- --
[  21 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  22 ] ?    chunk(imap): 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  23 ] ?    chunk(imap): -- 21 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  24 ] ?    [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] [zv9,81] [xr4,130] --
[  25 ] ?    type:dir size:1 refs:2 ptrs: 24 -- -- -- -- -- -- --
[  26 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  27 ] ?    chunk(imap): 25 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  28 ] ?    chunk(imap): -- -- 26 -- -- -- -- -- -- -- -- -- -- -- -- --
[  29 ] ?    [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] [zv9,81] [xr4,130] [px9,238]
[  30 ] ?    type:dir size:1 refs:2 ptrs: 29 -- -- -- -- -- -- --
```

```
[  18 ] ?    chunk(imap): -- -- -- -- 16 -- -- -- -- -- -- -- -- -- -- --
[  19 ] ?    [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] [zv9,81] -- --
[  20 ] ?    type:dir size:1 refs:2 ptrs: 19 -- -- -- -- -- -- --
[  21 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  22 ] ?    chunk(imap): 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  23 ] ?    chunk(imap): -- 21 -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  24 ] ?    [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] [zv9,81] [xr4,130] --
[  25 ] ?    type:dir size:1 refs:2 ptrs: 24 -- -- -- -- -- -- --
[  26 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  27 ] ?    chunk(imap): 25 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  28 ] ?    chunk(imap): -- -- 26 -- -- -- -- -- -- -- -- -- -- -- -- --
[  29 ] ?    [.,0] [..,0] [kg5,205] [hm5,114] [ht6,20] [zv9,81] [xr4,130] [px9,238]
[  30 ] ?    type:dir size:1 refs:2 ptrs: 29 -- -- -- -- -- -- --
[  31 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  32 ] ?    chunk(imap): 30 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  33 ] ?    chunk(imap): -- -- -- -- -- -- -- -- -- -- -- -- -- -- 31 --
[  34 ] ?    [gu5,27] -- -- -- -- -- -- --
[  35 ] ?    type:dir size:2 refs:2 ptrs: 29 34 -- -- -- -- -- --
[  36 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  37 ] ?    chunk(imap): 35 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  38 ] ?    chunk(imap): -- -- -- -- 16 -- -- -- -- -- -- 36 -- -- -- --
[  39 ] ?    [gu5,27] [kv6,141] -- -- -- -- -- --
[  40 ] ?    type:dir size:2 refs:2 ptrs: 29 39 -- -- -- -- -- --
[  41 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  42 ] ?    chunk(imap): 40 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  43 ] ?    chunk(imap): -- -- 26 -- -- -- -- -- -- -- -- -- -- 41 -- --
[  44 ] ?    [gu5,27] [kv6,141] [wg3,180] -- -- -- -- --
[  45 ] ?    type:dir size:2 refs:2 ptrs: 29 44 -- -- -- -- -- --
[  46 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  47 ] ?    chunk(imap): 45 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  48 ] ?    chunk(imap): -- -- -- -- 46 -- -- -- -- -- -- -- -- -- -- --
[  49 ] ?    [gu5,27] [kv6,141] [wg3,180] [og9,140] -- -- -- --
[  50 ] ?    type:dir size:2 refs:2 ptrs: 29 49 -- -- -- -- -- --
[  51 ] ?    type:reg size:0 refs:1 ptrs: -- -- -- -- -- -- -- --
[  52 ] ?    chunk(imap): 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
[  53 ] ?    chunk(imap): -- -- 26 -- -- -- -- -- -- -- -- -- 51 41 -- --
```

Yorum: Yeni dosyaların inode'ları rastgele seçilir, bu nedenle inode haritası kaos içindedir.

`Soru 11`: Varsaydığımız son bir şey de, LFS simülatörünün her güncellemeden sonra kontrol noktası bölgesini her zaman güncellediğidir. gerçek LFS, durum böyle değil: uzun süre önlemek için periyodik olarak güncellenir.

arar. Bazı işlemleri ve dosya sisteminin ara ve son durumlarını görmek için ./lfs.py -N -i -o -s 1000'i çalıştırın.
kontrol noktası bölgesi diske alınmaya zorlanmaz. eğer ne olurdu
kontrol noktası bölgesi asla güncellenmez mi? Periyodik olarak güncellenirse ne olur? Dosya sistemini nasıl kurtaracağınızı çözebilir misiniz? günlükte ileri giderek son durum?

```
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 -- -- -- -- -- -- -- -- -- -- --
```

Yorum: Kontrol noktası bölgesi hiç güncellenmezse tüm işlemlerin bir anlamı olmayacaktır.

Periyodik olarak güncellenmesi iyi olmalıdır.

En son günlük, en son imapa sahiptir, bunu kontrol noktası bölgesini güncellemek için kullanın.

Ogrenci no: 2020123094

Isim: mostafa zanaty

Odev bitmistir.