44

Flaş Tabanlı SSDler (Flash-based SSDs)

Sabit disk sürücülerinin onlarca yıllık hakimiyetinin ardından, yeni bir çeşit kalıcı depolama aygıtı son zamanlarda dünyada önem kazandı. **Katı-hal depolama(solid-state storage)** olarak adlandırılan bu tip aygıtlarda, sabit sürücülerde olduğu gibi mekanik veya hareketli parçalar yoktur; bunun yerine, bellek ve işlemciler gibi transistörlerden yapılmışlardır. Bununla beraber, tipik bir rasgele erişimli belleğin(random access memory) (örn. DRAM) aksine, bu tür bir **katı-hal depolama cihazı(solid-state storage device)** (namı diğer, SSD) güç kaybına rağmen bilgileri korur ve bu nedenle verilerin kalıcı olarak depolanmasında kullanım için ideal bir adaydır.

Odaklanacağımız teknoloji, 1980'lerde [M+14] Fujio Masuoka tarafından oluşturulan **flaş(flash)** (daha spesifik olarak **NAND tabanlı flaş**) olarak biliniyor. Göreceğimiz gibi, flaşın bazı benzersiz özellikleri vardır. Örneğin, belirli bir yığına (yani bir **flaş sayfaya(flaş page)**) yazmak için, önce oldukça zahmetli olabilecek daha büyük bir yığın (yani bir **flaş blok(flash block)**) silmeniz gerekir. Ayrıca bir sayfaya çok sık yazmak sayfanın **yıpranmasına(wear out)** neden olur. Bu iki özellik, flaş tabanlı bir SSD'nin yapım aşamasını zorlu bir hale getirir:

KİLİT NOKTASI: FLAŞ TABANLI SSD NASIL YAPILIR

Flaş-tabanlı bir SSD'yi nasıl yaparız? Silmenin zahmetli doğasıyla nasıl başa çıkarız? Tekrarlı üzerine yazmanın cihazı yıpratacağı göz önüne alındığında, uzun süre dayanan bir cihaz nasıl yaparız? Teknolojideki ilerleme yarışı duracak mı? Veya şaşırmaktan vaz mı geçeceğiz?

44.1 Yalnızca Bir Biti Depolamak (Storing a Single Bit)

Flaş yongaları, bir veya daha fazla biti tek bir transistörde depolamak için tasarlanmıştır; transistör içinde hapsolan yük seviyesi ikili bir değere eşlenir. Bir **tek-seviyeli hücre(Single-Level Cell) (SLC)** flaşında, bir transistör içinde yalnızca tek bir bit depolanır (yani, 1 veya 0);bir **çok-seviyeli hücre(Multi-Level Cell) (MLC)** flaşı ile, iki bit farklı şarj seviyelerine kodlanır, örneğin 00, 01, 10 ve 11 düşük, biraz düşük, biraz yüksek ve yüksek seviyelerle temsil edilir. Hücre başına 3 biti kodlayan **üç-seviyeli hücre(Triple-Level Cell) (TLC)** flaşı bile vardır. Genel olarak, SLC yongaları daha yüksek performans sağlar ve daha pahalıdır.

İPUCU: TERMİNOLOJİYE DİKKAT EDİN

Daha önce birçok kez kullandığımız bazı terimlerin (bloklar, sayfalar(blocks, pages)) flash bağlamında kullanıldığını, ancak öncekinden biraz farklı şekillerde kullanıldığını fark etmişsinizdir. Yeni terimler hayatınızı zorlaştırmak için yaratılmadı (gerçi tam da bunu yapıyor olabilirler), terminoloji kararlarının verildiği merkezi bir otorite olmadığı için ortaya çıkıyorlar. Sizin için bir blok(block) olan, bağlama bağlı olarak başka biri için bir sayfa(page) olabilir ve bunun tersi de geçerlidir. İşiniz basit: her alandaki uygun terimleri bilmek ve bunları, ilgili disiplinde bilgili insanların neden bahsettiğinizi anlayabileceği şekilde kullanmak. Tek çözümün basit ama bazen acı verici olduğu zamanlardan biri: hafızanızı kullanın.

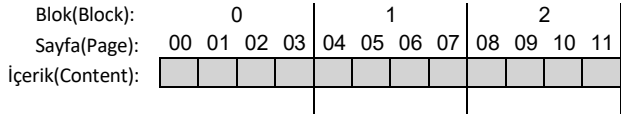
Elbette, bu tür bit düzeyinde depolamanın tam olarak nasıl çalıştığına dair cihaz fiziği düzeyinde pek çok ayrıntı var. Bu kitabın kapsamı dışında olsa da, konu hakkında daha fazlasını ilgili bölümde okuyabilirsiniz [J10].

44.2 Bitlerden Kümelere/Düzlemlere (From Bits to Banks/Planes)

Eski Yunan'da dedikleri gibi, tek bir biti (veya birkaçını) depolamak, bunu bir depolama sistemi yapmaz. Bu nedenle, flaş çipler, çok sayıda hücreden oluşan **kümeler(banks)** veya **düzlemler(planes)** halinde düzenlenir.

Bir kümeye iki farklı boyutlu birimde erişilir: tipik olarak 128 KB veya 256 KB boyutunda olan **bloklar(blocks)** (bazen **silme blokları(erase blocks)** olarak adlandırılır) ve birkaç KB boyutunda (örn. 4 KB) olan **sayfalar(pages)**. Her kümenin(bank) içinde çok sayıda blok vardır; her bloğun içinde çok sayıda sayfa vardır. Flaşı ele alırken, disklerde ve RAID'lerde bahsettiğimiz bloklardan ve sanal bellekte atıfta bulunduğumuz sayfalardan farklı olan bu yeni terminolojiyi hatırlamalısınız.

Şekil 44.1, bloklar ve sayfalar içeren bir flaş düzlemi örneğini göstermektedir; Bu basit örnekte, her biri dört sayfa içeren üç blok vardır. Bloklar ve sayfalar arasında neden ayrım yaptığımızı aşağıda göreceğiz; Bu ayrımın okuma ve yazma gibi flaş işlemleri ve hatta cihazın genel performansı için kritik olduğu barizdir. Öğreneceğiniz en önemli (ve tuhaf) şey, bir blok içindeki bir sayfaya yazmak için önce tüm bloğu silmeniz gerektiğidir; bu ince detay, flash tabanlı bir SSD oluşturmayı ilginç ve değerli bir zorluk haline getiriyor ve bu durum bölümün ikinci yarısının konusu.



Figür 44.1: **Basit Bir Flaş Çipi: Bloklar İçindeki Sayfalar**

44.3 Basit Flaş İşlemleri (Basic Flash Operations)

Bu flaş işlemi göz önüne alındığında, bir flaş çipinin desteklediği üç düşük seviyeli(makine diline yakın) işlem vardır. **Okuma(read)** komutu flaştan bir sayfa okumak için kullanılır; **sil(erase)** ve **program** yazmak için art arda kullanılır. Detaylar:

- **Okuma(bir sayfayı(page)):** Flaş çipin bir istemcisi, sadece okuma komutunu ve cihaza uygun sayfa numarasını belirterek herhangi bir sayfayı (örn. 2KB veya 4KB) okuyabilir. Bu işlem tipik olarak oldukça hızlıdır, cihazdaki konumdan bağımsız olarak ve (az ya da çok) önceki isteğin konumundan bağımsız olarak (bir diskten oldukça farklı olarak) 10larca mikrosaniye kadar. Herhangi bir yere aynı hızla erişebilmek, cihazın **rastgele erişimli(random access)** bir cihaz olduğu anlamına gelir.
- **Silme(bir bloğu(block)):** Bir sayfaya flaş ile yazmadan önce, aygıtın doğası gereği önce sayfanın içinde bulunduğu bloğun tamamını **silmeniz(erase)** gerekir. Silme, daha da önemlisi, bloğun içeriğini yok eder (her biti 1 değerine ayarlayarak); bu nedenle, silme işlemini gerçekleştirmeden önce blokta ilgilendiğiniz herhangi bir verinin başka bir yere (belleğe veya başka bir flash bloğa) kopyalandığından emin olmalısınız. Silme komutu oldukça zahmetlidir ve tamamlanması birkaç milisaniye sürer. Bittiğinde, tüm blok sıfırlanır ve her sayfa programlanmaya hazırdır.
- **Program (bir sayfayı):** Bir blok silindikten sonra, bir sayfadaki 1'lerin bazılarını 0'lara değiştirmek ve bir sayfanın istenen içeriğini flaşa yazmak için program komutu kullanılabilir. Bir sayfayı programlamak, bir bloğu silmekten daha az zahmetlidir, ancak bir sayfayı okumaktan daha zahmetlidir, genellikle modern flash çiplerde yaklaşık 100lerce mikrosaniye sürer.

Flaş yongalarını ele alırsak, her sayfanın kendisiyle ilişkili bir duruma sahip olduğunu görürüz. Sayfalar başlangıçta GEÇERSİZ(INVALID) durumundadır. Bir sayfanın içinde bulunduğu bloğu silerek, sayfanın (ve o blok içindeki tüm sayfaların) durumunu, bloktaki her sayfanın içeriğini sıfırlayan ama aynı zamanda (önemlisi) onları programlanabilir kılan SİLİNMİŞ(ERASED) olarak ayarlarsınız. Bir sayfayı programladığınızda, durumu GEÇERLİ(VALID) olarak değişir, yani içeriği ayarlanmıştır ve okunabilir. Okumalar sayfaların durumlarını(states) etkilemez (yine de yalnızca programlanmış sayfalardan okumanız gerekir). Bir sayfa programlandıktan sonra içeriğini değiştirmenin tek yolu, sayfanın içinde bulunduğu bloğun tamamını silmektir. 4 sayfalık bir blok içinde çeşitli silme ve program işlemlerinden sonra durum geçişine bir örnek:

		iiii	*Başlangıç: bloktaki sayfalar geçersiz* (i)
Sil() - Erase()	→	EEEE	*Blok içindeki sayfaların durumu silindi olarak değiştirildi* (E)
Program(0)	→	VEEE	*page 0'ı programla ve durumunu(state) geçerli(valid) yap (V)*
Program(0)	→	**hata(error)**	*Bir kere programlandıktan sonar tekrar programlanamaz*
Program(1)	→	VVEE	*page 1'i programla*
Sil() - Erase()	→	EEEE	*İçerikler silindi; sayfalar programlanabilir*

Detaylı Bir Örnek

Yazma süreci (yani silme ve programlama) çok sıra dışı olduğundan, mantıklı olduğundan emin olmak için ayrıntılı bir örnek üzerinden gidelim. Bu örnekte, 4 sayfalık bir blok içinde aşağıdaki dört adet 8 bitlik sayfaya sahip olduğumuzu hayal edin (her ikisi de gerçekçi olmayan küçük boyutlardadır, ancak bu örnekte kullanışlıdır); her sayfa GEÇERLİ(VALID)dir çünkü her biri önceden programlanmıştır.

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
GEÇERLİ VALID	GEÇERLİ VALID	GEÇERLİ VALID	GEÇERLİ VALID

Şimdi page 0'a yazmak istediğimizi ve onu yeni içeriklerle doldurduğumuzu varsayalım. Herhangi bir sayfaya yazmak için önce tüm bloğu silmeliyiz. Bunu yaptığımızı varsayarsak bloğun durumu bu şekilde olur:

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
SİLİNDİ ERASED	SİLİNDİ ERASED	SİLİNDİ ERASED	SİLİNDİ ERASED

Güzel! Şimdi devam edip sayfa 0'ı örneğin 00000011 içeriği ile programlayabilir, eski sayfa 0'ın (00011000 içeriğini) üzerine yazabiliriz. Bunu yaptıktan sonar bloğumuz bu şekilde görünür:

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
GEÇERLİ VALID	SİLİNDİ ERASED	SİLİNDİ ERASED	SİLİNDİ ERASED

Şimdi haberler kötü: page 1,2 ve 3'ün önceki içerikleri kaybolmuş oldu! Bu nedenle, bir blok içindeki herhangi bir sayfanın üzerine yazmadan önce, önemsediğimiz herhangi bir veriyi başka bir konuma (örneğin, bellek veya flash üzerinde başka bir yere) taşımamız gerekir. Silme işleminin doğası, yakında öğreneceğimiz gibi, flash tabanlı SSD'leri nasıl tasarladığımız üzerinde güçlü bir etkiye sahip olacaktır.

ÖZET

Özetlemek gerekirse, bir sayfayı okumak kolaydır: sadece sayfayı oku. Flash çipler bunu oldukça iyi ve hızlı bir şekilde yapıyor; performans açısından, mekanik arama ve dönme süresi nedeniyle yavaş olan modern disk sürücülerinin rastgele okuma performansını büyük ölçüde aşma potansiyeli sunarlar.

Bir sayfa yazmak daha zordur; önce bloğun tamamı silinmeli (önce ilgilendiğimiz herhangi bir veriyi başka bir konuma taşımaya özen göstererek) ve ardından istenen sayfa programlanmalıdır. Bu sadece zahmetli olmakla kalmaz, aynı zamanda bu program/silme döngüsünün sık sık tekrarlanması flaş yongaların sahip olduğu en büyük güvenilirlik sorununa yol açabilir: **aşınma(wear out)**. Flaş teknolojisi ile bir depolama sistemi tasarlarken yazma performansı ve güvenilirlik ana odak noktasıdır.

Device (Cihaz)	Read Okuma (μs)	Program (μs)	Erase Silme (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figür 44.2: **Ham Flaş Performans Özellikleri**

44.4 Flaş Performansı ve Güvenilirliği (Flash Performance And Reliability)

Ham flaş yongalardan bir depolama aygıtı oluşturmakla ilgilendiğimiz için, temel performans karakteristiklerini anlamak faydalı olacaktır. Şekil 44.2, popüler basında bulunan bazı sayıların kabaca bir özetini sunar. [V12]. Burada yazar, sırasıyla hücre başına 1, 2 ve 3 bit bilgi depolayan SLC, MLC ve TLC flaş boyunca okumaların, programların ve silmelerin temel işlem gecikmesini sunar.

Tablodan da görebileceğimiz gibi, okuma gecikmeleri oldukça iyidir ve tamamlanması yalnızca 10larca mikrosaniye sürer. Program gecikmesi daha yüksek ve daha değişkendir, SLC için 200 mikrosaniye kadar düşüktür, ancak her hücreye daha fazla bit yükledikçe daha yükselir; iyi bir yazma performansı elde etmek için paralel olarak birden fazla flaş yongası kullanmanız gerekecektir. Son olarak, silme işlemleri oldukça zahmetlidir ve tipik olarak birkaç milisaniye sürer. Bu sorunla başa çıkmak, modern flaş depolama tasarımının merkezinde yer alır.

Şimdi flaş çiplerin güvenilirliğini ele alalım. Çok çeşitli nedenlerle (sürücü kafasının kayıt yüzeyiyle gerçekten temas ettiği korkunç ve oldukça fiziksel **kafa çarpması(head crash)** dahil) arızalanabilen mekanik disklerin aksine, flaş çipleri saf silikondur ve bu anlamda endişelenilecek daha az güvenilirlik sorunu vardır. Birincil endişe **aşınmadır(wear out)**; bir flaş bloğu silinip programlandığında, yavaş yavaş biraz fazladan yük birikir. Zamanla, bu ekstra yük biriktikçe, 0 ile 1 arasında ayrım yapmak giderek daha zor hale gelir. Ayrım yapmanın imkansız hale geldiği noktada blok kullanılamaz hale gelir.

Bir bloğun tipik ömrü şu anda iyi bilinmemektedir. Üreticiler, MLC tabanlı blokları 10.000 P/E (Program/Silme) döngü ömrüne sahip olarak derecelendirir; yani, her blok arızalanmadan önce 10.000 kez silinebilir ve programlanabilir. SLC tabanlı yongalar, transistör başına yalnızca tek bir bit depoladıkları için, genellikle 100.000 P/E döngüsü olmak üzere daha uzun bir ömürle derecelendirilir. Bununla birlikte, son araştırmalar, yaşam sürelerinin beklenenden çok daha uzun olduğunu göstermiştir. [BD10].

Flaş çiplerdeki diğer bir güvenilirlik sorunu, **bozulma(disturbance)** olarak bilinir. Bir flaş içinde belirli bir sayfaya erişirken, komşu sayfalarda bazı bitlerin ters çevrilmesi mümkündür; bu tür bit çevirmeleri, sırasıyla sayfanın okunmasına veya programlanmasına bağlı olarak **okuma bozulmaları(read disturbs)** veya **programlama bozulmaları(program disturbs)** olarak bilinir.

İPUCU: GERİYE DÖNÜK UYUMLULUĞUN ÖNEMİ

Geriye dönük uyumluluk, katmanlı sistemlerde her zaman bir endişe kaynağıdır. İki sistem arasında istikrarlı bir arayüz tanımlayarak, sürekli birlikte çalışabilirliği sağlarken arayüzün her iki tarafında yenilik sağlar. Bu tür bir yaklaşım birçok alanda oldukça başarılı olmuştur: işletim sistemleri, uygulamalar için nispeten kararlı API'lere sahiptir, diskler, dosya sistemlerine aynı blok tabanlı arabirimi sağlar ve IP ağ yığınındaki her katman, yukarıdaki katmana sabit, değişmeyen bir arabirim sağlar.

Şaşırtıcı olmayan bir şekilde, bir nesilde tanımlanan arayüzler bir sonraki nesilde uygun olmayabileceğinden, bu tür katılığın bir dezavantajı olabilir. Bazı durumlarda, tüm sistemi tamamen yeniden tasarlamayı düşünmek faydalı olabilir. Sun ZFS dosya sisteminde [B07] mükemmel bir örnek bulunur; ZFS'nin üreticileri, dosya sistemleri ve RAID etkileşimini yeniden değerlendirerek daha etkili bir entegre bütün tasarladılar (ve sonra gerçekleştirdiler).

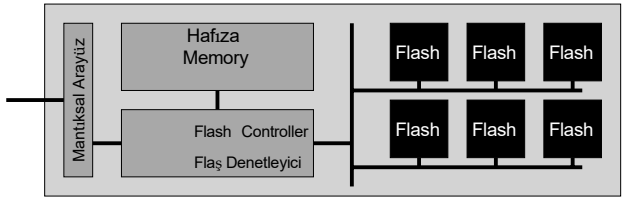
44.5 HAM FLAŞTAN FLAŞ TABANLI SSDlere (From Raw Flash to Flash-Based SSDs)

Flaş yongaları hakkındaki temel anlayışımız göz önüne alındığında, şimdi bir sonraki görevimizle karşı karşıyayız: temel bir flaş yonga setini tipik bir depolama aygıtı gibi görünen bir şeye nasıl dönüştüreceğiz? Standart depolama arabirimi, bir blok adresi verildiğinde 512 bayt (veya daha büyük) boyutunda blokların (sektörlerin) okunup yazılabileceği, blok tabanlı basit bir arabirimdir. Flaş tabanlı SSD'nin görevi, içindeki ham flaş yongalarının üzerinde bu standart blok arabirimini sağlamaktır.

SSD'yi incelediğimizde, belirli sayıda flaş yongasından oluşur (kalıcı depolama için). Bir SSD ayrıca bir miktar geçici (yani kalıcı olmayan) bellek (ör. SRAM) içerir; bu tür bir bellek, verileri önbelleğe alma ve arabelleğe almanın yanı sıra aşağıda öğreneceğimiz tabloları eşleştirmek için kullanışlıdır. Son olarak, bir SSD, cihazın çalışmasını düzenlemek için kontrol mantığı içerir. Detaylar için Agrawal [A+08]'e bakın; basitleştirilmiş blok diyagramı Figure 44.3'de görülmektedir(sayfa 7).

Bu kontrol mantığının temel işlevlerinden biri, istemci okumalarını ve yazmalarını gerçekleştirmek ve bunları gerektiği gibi dahili flaş işlemlerine dönüştürmektir. **Flaş çeviri katmanı(flash translation layer)** veya **FTL** tam olarak bu işlevi sağlar. FTL, mantıksal bloklarda (cihaz arayüzünü oluşturan) okuma ve yazma isteklerini alır ve bunları temeldeki fiziksel bloklarda ve fiziksel sayfalarda (gerçek flaş cihazını oluşturan) düşük seviyeli okuma, silme ve programlama komutlarına dönüştürür. FTL, mükemmel performans ve yüksek güvenilirlik sağlama hedefiyle bu görevi yerine getirmelidir.

Göreceğimiz gibi mükemmel performans, tekniklerin bir kombinasyonu ile gerçekleştirilebilir. Kilit noktalardan biri, paralel olarak birden fazla flaş yongası kullanmak olacaktır; Bu tekniği daha fazla tartışmayacak olsak da, tüm modern SSD'lerin daha yüksek performans elde etmek için dahili olarak birden çok yonga kullandığını söylemekle yetinelim. Diğer bir performans hedefi, FTL tarafından flaş yongalara verilen toplam yazma trafiğinin (bayt cinsinden) istemci tarafından SSD'ye verilen toplam yazma trafiğine (bayt cinsinden)



Figür 44.3: **Bir Flaş Tabanlı SSD: Mantıksal Diyagram**

bölünmesi olarak tanımlanan yazma amplifikasyonunu azaltmak olacaktır.

Birkaç farklı yaklaşımın kombinasyonu ile yüksek güvenilirlik elde edilecektir. Yukarıda bahsedildiği gibi ana endişelerden biri **aşınmadır(wear out)**. Tek bir blok çok sık silinir ve programlanırsa kullanılamaz hale gelir; sonuç olarak, FTL, aygıtın tüm bloklarının kabaca aynı anda aşınmasını sağlayarak, flaş blokları boyunca yazma işlemlerini mümkün olduğu kadar eşit bir şekilde yaymaya çalışmalıdır; bunu yapmaya **aşınma dengeleme(wear leveling)** denir ve herhangi bir modern FTL'nin önemli bir parçasıdır.

Diğer bir güvenilirlik endişesi, program bozulmasıdır. Bu bozulmaları en aza indirmek için, FTL'ler genellikle silinmiş bir blok içindeki sayfaları düşük sayfadan yüksek sayfaya doğru programlayacaktır. Bu sıralı programlama yaklaşımı, bozulmaları en aza indirir ve yaygın olarak kullanılır.

44.6 FTL ORGANIZASYONU: KÖTÜ BIR YAKLAŞIM (FTL Organization: A Bad Approach)

Bir FTL'nin en basit organizasyonu, **doğrudan haritalanmış(direct mapped)** dediğimiz bir şey olacaktır. Bu yaklaşımda, mantıksal sayfa N'ye okuma, doğrudan fiziksel sayfa N'nin okumasına eşlenir. Mantıksal sayfa N'ye yazmak daha karmaşıktır; FTL önce N sayfasının içerdiği bloğun tamamını okumalıdır; daha sonra bloğu silmek zorundadır; son olarak, FTL hem eski hem de yeni sayfaları programlar.

Tahmin edebileceğiniz gibi, doğrudan eşlemeli FTL'nin hem performans hem de güvenilirlik açısından birçok sorunu vardır. Performans sorunları her yazmada ortaya çıkar: aygıtın tüm bloğu okuması (zahmetli), silmesi (oldukça zahmetli) ve ardından programlaması (zahmetli) gerekir. Sonuç, ciddi yazma artışı (bir bloktaki sayfa sayısıyla orantılı) ve sonuç olarak, mekanik aramaları ve dönme gecikmeleri ile tipik sabit sürücülerden bile daha yavaş olan korkunç yazma performansıdır.

Daha da kötüsü, bu yaklaşımın güvenilirliğidir. Dosya sistemi meta verilerinin veya kullanıcı dosyası verilerinin üzerine art arda yazılırsa, aynı blok defalarca silinir ve programlanır, blok hızla eskir ve potansiyel olarak veri kaybedilir. Doğrudan haritalanmış yaklaşım, istemci iş yüküne yıpranma üzerinde çok fazla kontrol sağlar; iş yükü, yazma yükünü mantıksal bloklarına eşit şekilde yaymazsa, popüler verileri içeren temel fiziksel bloklar hızla aşınır. Hem güvenilirlik hem de performans nedenleriyle, doğrudan haritalanmış bir FTL kötü bir fikirdir.

44.7 GÜNLÜK YAPILI FTL(a log-Structured FTL)

Bu nedenlerden dolayı, günümüzde çoğu FTL, hem depolama aygıtlarında (şimdi göreceğimiz gibi) hem de bunların üzerindeki dosya sistemlerinde (**günlük yapılı dosya sistemleri(log-structured file systems**) bölümünde göreceğimiz gibi) **Günlük Yapılıdır(log structured)**. Mantıksal N bloğa yazma esnasında, cihaz, yazmayı şu anda yazılmakta olan bloktaki bir sonraki boş noktaya ekler; bu yazma stiline **günlükleme(logging)** diyoruz. N bloğunun sonraki okumalarına izin vermek için, cihaz bir **eşleme tablosu(mapping table)** tutar (belleğinde ve bir şekilde cihazda kalıcıdır); bu tablo, sistemdeki her mantıksal bloğun fiziksel adresini saklar.

Temel günlük tabanlı yaklaşımın nasıl çalıştığını anladığımızdan emin olmak için bir örnek üzerinden gidelim. İstemciye göre cihaz, 512 baytlık sektörleri (veya sektör gruplarını) okuyup yazabildiği tipik bir disk gibi görünür. Basit olması için, istemcinin 4 KB boyutunda parçalar okuduğunu veya yazdığını varsayalım. Ayrıca, SSD'nin her biri dört adet 4 KB'lık sayfaya bölünmüş çok sayıda 16 KB boyutunda blok içerdiğini varsayalım; bu parametreler gerçekçi değildir (flash bloklar genellikle daha fazla sayfadan oluşur), ancak öğretici amaçlarımıza oldukça iyi hizmet edecektir.

İstemcinin aşağıdaki işlem dizisini yayınladığını varsayalım:

- Write(100) with contents a1 (a1'in içeriğini 100'e yaz)
- Write(101) with contents a2 (a2'nin içeriğini 101'e yaz)
- Write(2000) with contents b1 (b1'in içeriğini 2000'e yaz)
- Write(2001) with contents b2 (b2'nin içeriğini 2001'e yaz)

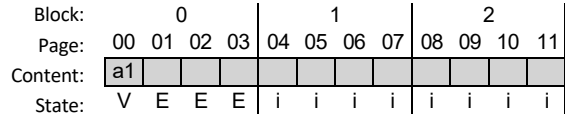
Bu **mantıksal blok adresleri(logical block addresses)** (ör. 100), SSD istemcisi (ör. bir dosya sistemi) tarafından bilgilerin nerede bulunduğunu hatırlamak için kullanılır. Dahili olarak, cihazın bu blok yazma işlemlerini ham donanım tarafından desteklenen silme ve program işlemlerine dönüştürmesi ve SSD'nin **fiziksel sayfasının(physical page)** verilerini depoladığı her bir mantıksal blok adresi için bir şekilde kaydetmesi gerekir. SSD'nin tüm bloklarının şu anda geçerli olmadığını ve herhangi bir sayfanın programlanabilmesi için silinmesi gerektiğini varsayalım. Burada, SSD'mizin ilk durumunu, tüm sayfaları GEÇERSİZ(INVALID) olarak işaretlenmiş olarak gösteriyoruz (i):

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	i	i	i	i	i	i	i	i	i	i	i	i

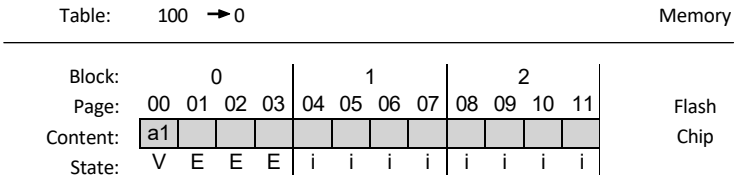
İlk yazma SSD tarafından alındığında (mantıksal blok 100'e), FTL bunu dört fiziksel sayfa içeren fiziksel blok 0'a yazmaya karar verir: 0, 1, 2 ve 3. Blok silinmediği için henüz ona yazamıyoruz; cihaz önce blok 0'a erase(sil) komutu vermelidir. Bunu yapmak aşağıdaki duruma yol açar:

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	E	E	E	E	i	i	i	i	i	i	i	i

Block 0(Blok 0) artık programlanmaya hazır. Çoğu SSD, sayfaları sırayla (yani düşükten yükseğe) yazar ve **program bozulmasıyla(program disturbance)** ilgili güvenilirlik sorunlarını azaltır. SSD daha sonra mantıksal blok 100'ün yazılmasını fiziksel sayfa 0'a yönlendirir:

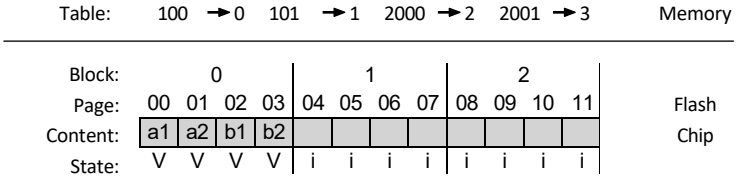


Peki ya istemci mantıksal blok 100'ü okumak isterse? Nerede olduğunu nasıl bulabilir? SSD, mantıksal blok 100'e verilen bir okumayı fiziksel sayfa 0'ın bir okumasına dönüştürmelidir. Bu tür işlevselliğe uyum sağlamak için FTL, mantıksal blok 100'ü fiziksel sayfa 0'a yazdığında, bu olguyu bir **bellek içi eşleme tablosuna(in-memory mapping table)** kaydeder. Bu eşleme tablosunun durumunu diyagramlarda da göreceğiz:



Artık istemci SSD'ye yazdığında ne olduğunu görebilirsiniz. SSD, genellikle bir sonraki boş sayfayı seçerek yazma için bir konum bulur; daha sonra o sayfayı bloğun içeriğiyle programlar ve mantıksal-fiziksel eşlemeyi eşleme tablosuna kaydeder. Sonraki okumalar, istemci tarafından sunulan mantıksal blok adresini verileri okumak için gereken fiziksel sayfa numarasına **çevirmek(translate)** için tabloyu kullanır.

Şimdi örnek yazma akışımızdaki diğer yazmaları inceleyelim: 101, 2000 ve 2001. Bu blokları yazdıktan sonra cihazın durumu şu şekildedir:



Günlük tabanlı yaklaşım, doğası gereği performansı artırır (yalnızca arada bir gerekli olan siler ve doğrudan eşlemeli yaklaşımın zahmetli okuma-değiştirme-yazma işleminden tamamen kaçınılır) ve güvenilirliği büyük ölçüde artırır. FTL artık yazma işlemlerini tüm sayfalara yayabilir, **aşınma dengelemesi(wear leveling)** denilen işlemi gerçekleştirebilir ve cihazın kullanım ömrünü uzatabilir; aşınma dengelemesini aşağıda daha ayrıntılı olarak tartışacağız.

NOT: FTL EŞLEME BİLGİSİ KALICILIĞI

Merak ediyor olabilirsiniz: Cihazın gücü kesilirse ne olur? Bellek içi eşleme tablosu kaybolur mu? Açıkçası, bu tür bilgiler gerçekten kaybolamaz çünkü aksi halde cihaz kalıcı bir depolama cihazı olarak işlev görmez. Bir SSD, eşleme/haritalama bilgilerini kurtarmak için bazı araçlara sahip olmalıdır.

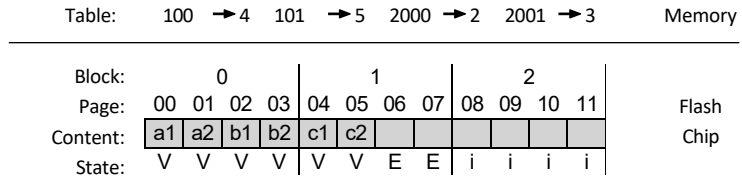
Yapılabilecek en basit şey, **bant dışı(out-of-band) (OOB)** alan adı verilen alana her sayfayla birlikte bazı haritalama/eşleme bilgilerini kaydetmektir. Cihaz gücü kesildiğinde ve yeniden başlatıldığında, OOB alanlarını tarayarak ve eşleme tablosunu bellekte yeniden oluşturarak eşleme tablosunu yeniden oluşturması gerekir. Bu temel yaklaşımın sorunları vardır; gerekli tüm eşleme bilgilerini bulmak için büyük bir SSD'yi taramak yavaştır. Bu sınırlamanın üstesinden gelmek için bazı üst düzey cihazlar, kurtarmayı hızlandırmak için daha karmaşık **günlük kaydı(logging)** ve kontrol **noktası(checkpointing)** teknikleri kullanır; kilitlenme tutarlılığı ve günlük yapılı dosya sistemleri hakkındaki bölümleri okuyarak günlük kaydı hakkında daha fazla bilgi edinin [AD14].

Ne yazık ki, günlük yapılandırmaya yönelik bu temel yaklaşımın bazı dezavantajları vardır. Birincisi, mantıksal blokların üzerine yazılması, çöp dediğimiz bir şeye, yani sürücüdeki eski veri sürümlerine ve yer kaplamasına yol açar. Cihazın, söz konusu blokları ve gelecekteki yazma işlemleri için boş alanı bulmak üzere periyodik olarak **çöp toplama(Garbage Collection) (GC)** gerçekleştirmesi gerekir; aşırı çöp toplama, aşırı yazmaya sebep olur ve performansı düşürür. İkincisi, bellek içi eşleme tablolarının yüksek maliyetidir; cihaz ne kadar büyükse, bu tür tablolar daha fazla belleğe ihtiyaç duyar. Şimdi sırayla her birini gözden geçirelim.

44.8 Çöp Toplama(Garbage Collection)

Bunun gibi günlük yapılı herhangi bir yaklaşımın ilk dezavantajı, çöpün yaratılmasıdır ve bu nedenle **çöp toplamanın(garbage collection)** (yani, ölü blok ıslahı) gerçekleştirilmesi gerekir. Bunu anlamak için devam eden örneğimizi kullanalım. Aygıta 100, 101, 2000 ve 2001 mantıksal bloklarının yazıldığını hatırlayın.

Şimdi, 100 ve 101 bloklarının c1 ve c2 içerikleriyle yeniden yazıldığını varsayalım. Yazmalar sonraki boş sayfalara yazılır (bu durumda fiziksel sayfalar 4 ve 5) ve eşleme tablosu buna göre güncellenir. Bu tür bir programlamayı mümkün kılmak için cihazın önce blok 1'i silmiş olması gerektiğini unutmayın:



Poblemi artık açıkça görüyoruz ki: fiziksel sayfalar 0 ve 1, GEÇERLİ(VALID) olarak işaretlenmiş olsalar da, içlerinde **çöp(garbage)** var, yani, blok 100 ve 101'in eski sürümleri. Aygıtın günlük-yapılı doğası nedeniyle, üzerine yazma işlemleri, yeni yazma işlemlerinin gerçekleşmesi için boş alan sağlamak üzere aygıtın geri kazanması gereken çöp blokları oluşturur.

Çöp bloklarını (**ölü bloklar(dead blocks)** olarak da adlandırılır) bulma ve bunları ileride kullanmak üzere geri alma işlemine **çöp toplama(garbage collection)** denir ve bu, herhangi bir modern SSD'nin önemli bir bileşenidir. Temel işlem basittir: bir veya daha fazla çöp sayfa içeren bir blok bulun, bu bloktaki canlı (çöp olmayan) sayfaları okuyun, bu canlı sayfaları günlüğe yazın ve (son olarak) tüm bloğu yazım işlemi için yeniden kullanın.

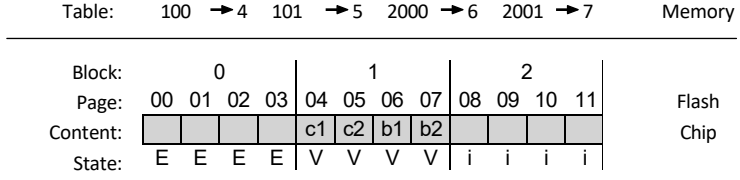
Şimdi bir örnekle açıklayalım. Cihaz, yukarıdaki blok 0 içindeki ölü sayfaları yeniden kullanmak istediğine karar verir. Blok 0, iki ölü bloğa (sayfa 0 ve 1) ve iki canlı bloğa (sırasıyla 2000 ve 2001 bloklarını içeren sayfa 2 ve 3) sahiptir. Bunu yapmak için, cihaz:

- Read live data (pages 2 and 3) from block 0 >> (• Blok 0'dan canlı verileri (sayfa 2 ve 3) okur)
- Write live data to end of the log >> (• Günlüğün sonuna güncel veriyi yazar)
- Erase block 0 (freeing it for later usage) >> (• Blok 0'ı siler (daha sonra kullanmak üzere serbest bırakır))

Çöp toplayıcının çalışması için, SSD'nin her sayfanın canlı mı yoksa ölü mü olduğunu belirlemesini sağlamak için her blokta yeterli bilgi olmalıdır. Bu amaca ulaşmanın doğal bir yolu, her blok içinde bir yerde, her sayfada hangi mantıksal blokların saklandığına ilişkin bilgiyi depolamaktır. Cihaz daha sonra, blok içindeki her sayfanın canlı verileri içerip içermediğini belirlemek için eşleme tablosunu kullanabilir.

Yukarıdaki örneğimizden (çöp toplama gerçekleşmeden önce), blok 0, 100, 101, 2000, 2001 mantıksal bloklarını tuttu. Eşleme tablosunu kontrol ederek (çöp toplamadan önce 100->4, 101->5, 2000->2, 2001->3'ü içerir), cihaz, SSD bloğu içindeki sayfaların her birinin canlı tutulup tutulmadığını kolayca belirleyebilir. Örneğin, 2000 ve 2001 hala harita tarafından açıkça işaret edilmektedir ve bu nedenle çöp toplama için adaydır; 100 ve 101 çöp toplama için aday değildir.

Örneğimizde bu çöp toplama işlemi tamamlandığında cihazın durumu şu şekildedir:



Gördüğünüz gibi, çöp toplama zahmetli olabilir ve canlı verilerin okunmasını ve yeniden yazılmasını gerektirebilir. Islah için ideal aday, yalnızca ölü sayfalardan oluşan bir bloktur; bu durumda blok, zahmetli veri geçişi olmadan hemen silinebilir ve yeni veriler için kullanılabilir.

NOT: TRIM İSİMLİ YENİ DEPOLAMA API'Sİ

Sabit diskleri düşündüğümüzde, genellikle onları okumak ve yazmak için en temel arabirim olarak düşünürüz: oku ve yaz (ayrıca genellikle yazma işlemlerinin gerçekten devam etmesini sağlayan bir tür **önbellek temizleme(cache flush)** komutu da vardır, ancak bazen bunu atlarız. basitlik için). Günlük yapılı SSD'lerde ve gerçekten de mantıksaldan fiziksele blokların esnek ve değişen bir eşlemesini tutan herhangi bir aygıtta, **kırpma(Trim)** işlemi olarak bilinen yeni bir arabirim kullanışlıdır.

Kırpma işlemi bir adres (ve muhtemelen bir uzunluk) alır ve cihaza adres (ve uzunluk) tarafından belirtilen blok(lar)ın silindiğini bildirir; bu nedenle cihazın artık verilen adres aralığı hakkında herhangi bir bilgiyi izlemesi gerekmez. Standart bir sabit sürücü için, kırpma özellikle kullanışlı değildir, çünkü sürücü blok adreslerinin belirli plaka, yol ve sektör(ler)e statik bir eşlemesine sahiptir. Bununla birlikte, günlük yapılı bir SSD için, artık bir bloğa ihtiyaç olmadığını bilmek son derece yararlıdır, çünkü SSD daha sonra bu bilgiyi FTL'den kaldırabilir ve daha sonra çöp toplama sırasında fiziksel alanı geri alabilir.

Bazen arayüzü ve uygulamayı ayrı varlıklar olarak düşünsek de, bu durumda arayüzü uygulamanın şekillendirdiğini görüyoruz. Karmaşık eşlemelerde, artık hangi bloklara ihtiyaç duyulmadığına dair bilgi, daha etkili bir uygulama sağlar.

GC maliyetlerini azaltmak için, bazı SSD'ler cihazı **overprovision(aşırı yer ayırma)** sağlar [A+08]; ekstra flaş kapasitesi eklenerek, temizlik geciktirilebilir ve arka plana itilebilir, belki de cihazın daha az meşgul olduğu bir zamanda yapılabilir. Daha fazla kapasite eklemek, temizlik için kullanılabilecek dahili bant genişliğini de artırır ve böylece istemci için algılanan bant genişliğine zarar vermez. Pek çok modern sürücü, bu şekilde aşırı tedarik sağlar; bu, mükemmel bir genel performans elde etmenin anahtarlarından biridir.

44.9 Eşleştirme Tablosu Boyutu(Mapping Table Size)

Günlük yapılandırmanın ikinci maliyeti, cihazın her 4 KB'lık sayfası için bir giriş içeren son derece büyük eşleme tabloları potansiyelidir. Örneğin, büyük bir 1 TB SSD ile, 4 KB sayfa başına tek bir 4 baytlık giriş, cihazın yalnızca bu eşlemeler için ihtiyaç duyduğu 1 GB belleğe neden olur! Bu nedenle, bu **sayfa düzeyinde(page-level)** FTL şeması pratik değildir.

Block-Based Mapping

Haritalama/eşleme maliyetlerini düşürmeye yönelik bir yaklaşım, sayfa başına değil, yalnızca cihazın bloğu başına bir işaretçi tutmak ve haritalama bilgisi miktarını [size blok/size page] faktörü kadar azaltmaktır. Bu **blok seviyesindeki(block-level)** FTL,

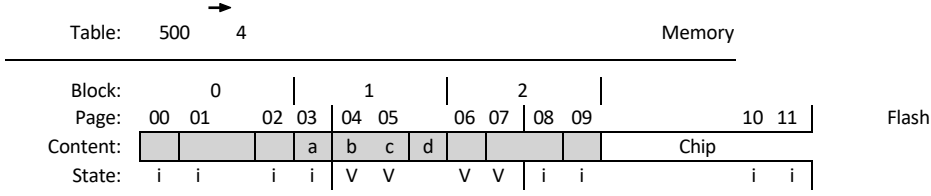
bir sanal bellek sisteminde daha büyük sayfa boyutlarına sahip olmaya benzer; bu durumda, VPN için daha az bit kullanırsınız ve her sanal adreste daha büyük bir kaymaya sahip olursunuz.

Ne yazık ki, günlük tabanlı bir FTL içinde blok tabanlı bir eşleme kullanmak, performans nedenleriyle pek iyi çalışmıyor. En büyük sorun, "küçük bir yazma" meydana geldiğinde ortaya çıkar (yani, fiziksel bir bloğun boyutundan daha küçük olan). Bu durumda, FTL eski bloktan büyük miktarda canlı veri okumalı ve bunu (küçük yazmadan gelen verilerle birlikte) yeni bir bloka kopyalamalıdır. Bu veri kopyalama, yazma işlemini büyük ölçüde artırır ve dolayısıyla performansı düşürür.

Bu konuyu daha net hale getirmek için bir örneğe bakalım. İstemcinin daha önce 2000, 2001, 2002 ve 2003 mantıksal bloklarını (a, b, c, d içerikleriyle birlikte) yazdığını ve bunların 4, 5, 6 ve 7. fiziksel sayfalarda 1. fiziksel blok içinde bulunduğunu varsayın. Sayfa başına eşlemelerde, çeviri tablosunun bu mantıksal bloklar için dört eşleme kaydetmesi gerekir: 2000→4, 2001→5, 2002→6, 2003→7.

Bunun yerine blok düzeyinde eşleme kullanırsak, FTL'nin tüm bu veriler için yalnızca tek bir adres çevirisini kaydetmesi gerekir. Bununla birlikte, adres eşleme, önceki örneklerimizden biraz farklıdır. Spesifik olarak, aygıtın mantıksal adres alanını, flash içindeki fiziksel blokların boyutu olan parçalara bölünmüş olarak düşünüyoruz. Böylece, mantıksal blok adresi iki kısımdan oluşur: yığın numarası ve ofset. Her fiziksel bloğa dört mantıksal bloğun sığdığını varsaydığımız için, mantıksal adreslerin ofset kısmı 2 bit gerektirir; kalan (en önemli) bitler yığın(chunk) numarasını oluşturur.

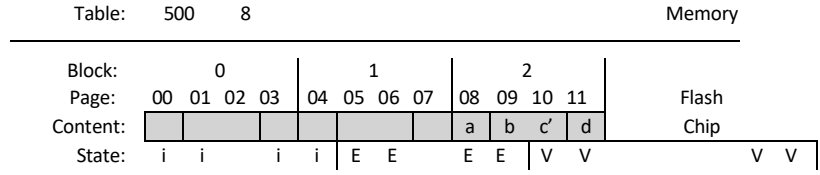
2000, 2001, 2002 ve 2003 mantıksal bloklarının tümü aynı yığın numarasına (500) ve farklı ofsetlere (sırasıyla 0, 1, 2 ve 3) sahiptir. Bu nedenle, blok düzeyinde bir eşlemeyle, bu şemada gösterildiği gibi, 500 yığınını blok 1'e eşleyen FTL kayıtları (fiziksel sayfa 4'ten başlayarak):



Blok tabanlı bir FTL'de okumak kolaydır. İlk olarak, FTL, adresten en üstteki bitleri alarak istemci tarafından sunulan mantıksal blok adresinden yığın numarasını çıkarır. Ardından FTL, tablodaki parça numarasından fiziksel sayfa eşlemesine bakar. Son olarak, FTL, mantıksal adresten ofseti bloğun fiziksel adresine ekleyerek istenen flash sayfasının adresini hesaplar.

Örneğin, istemci mantıksal adres 2002'ye bir okuma verirse, cihaz mantıksal yığın numarasını (500) çıkarır, eşleme tablosunda çeviriye bakar (4'ü bulur) ve mantıksal adresten (2) ofseti şu değere ekler: çeviri(4).Ortaya çıkan fiziksel sayfa adresi(6),

verilerin bulunduğu yerdir; FTL daha sonra bu fiziksel adrese okuma yapabilir ve istenen verileri elde edebilir (c). Peki ya istemci mantıksal blok 2002'ye yazarsa (c'nin içeriğiyle)? Bu durumda, FTL'nin 2000, 2001 ve 2003'ü okuması ve ardından dört mantıksal bloğun tümünü yeni bir konuma yazması ve eşleme tablosunu buna göre güncellemesi gerekir. Blok 1 (verilerin bulunduğu yer) daha sonra burada gösterildiği gibi silinebilir ve yeniden kullanılabilir.



Bu örnekten de görebileceğiniz gibi, blok düzeyinde eşlemeler, çeviriler için gereken bellek miktarını büyük ölçüde azaltırken, yazma işlemleri cihazın fiziksel blok boyutundan küçük olduğunda önemli performans sorunlarına neden olur; gerçek fiziksel bloklar 256 KB veya daha büyük olabileceğinden, bu tür yazma işlemlerinin oldukça sık gerçekleşmesi muhtemeldir. Bu nedenle daha iyi bir çözüme ihtiyaç vardır. Bunun, bölümün size bu çözümün ne olduğunu söylediğimiz kısmı olduğunu hissedebiliyor musunuz? Daha da iyisi, okumaya devam etmeden önce kendiniz çözebilir misiniz?

Hibrit Haritalama/Eşleme(Hybrid Mapping)

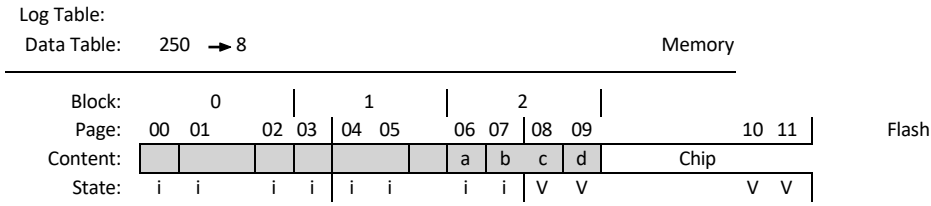
Esnek yazmayı mümkün kılmak ve aynı zamanda haritalama maliyetlerini azaltmak için birçok modern FTL, bir **hibrit haritalama(hybrid mapping)** tekniği kullanır. Bu yaklaşımla, FTL birkaç bloğu silinmiş halde tutar ve tüm yazma işlemlerini onlara yönlendirir; bunlara **günlük blokları(log blocks)** denir. FTL, saf blok tabanlı eşlemenin gerektirdiği tüm kopyalama işlemleri olmadan günlük bloğu içindeki herhangi bir konuma herhangi bir sayfayı yazabilmek istediğinden, bu günlük blokları için sayfa başına eşlemeleri tutar.

Bu nedenle, FTL mantıksal olarak belleğinde iki tür eşleme tablosuna(mapping table) sahiptir: günlük tablosu(log table) olarak adlandıracağımız sayfa başına küçük bir eşleme kümesi ve veri tablosunda(data table) daha büyük bir blok başına eşleme kümesi. Belirli bir mantıksal bloğu ararken, FTL önce günlük tablosuna başvurur; mantıksal bloğun konumu burada bulunmazsa, FTL konumunu bulmak için veri tablosuna başvuracak ve ardından istenen verilere erişecektir.

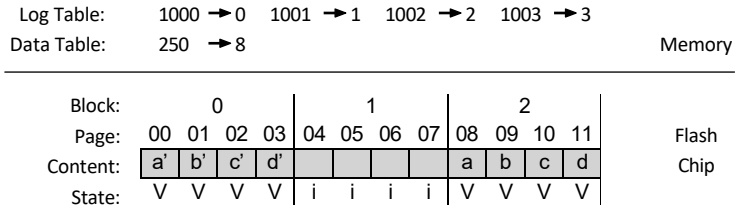
Hibrit eşleme(hybrid mapping) stratejisinin anahtarı, günlük bloklarının(log blocks) sayısını küçük tutmaktır. Günlük bloklarının sayısını azaltmak için, FTL'nin günlük bloklarını (sayfa başına bir işaretçisi olan) periyodik olarak incelemesi ve bunları yalnızca tek bir blok işaretçisi tarafından işaret edilebilecek bloklara dönüştürmesi gerekir. Bu geçiş, blok [KK+02] içeriğine bağlı olarak üç ana teknikten biri ile gerçekleştirilir.

Örneğin, FTL'nin daha önce 1000, 1001, 1002 ve 1003 mantıksal sayfalarını yazdığını ve bunları fiziksel blok2'ye (fiziksel sayfalar 8, 9, 10, 11) yerleştirdiğini varsayalım

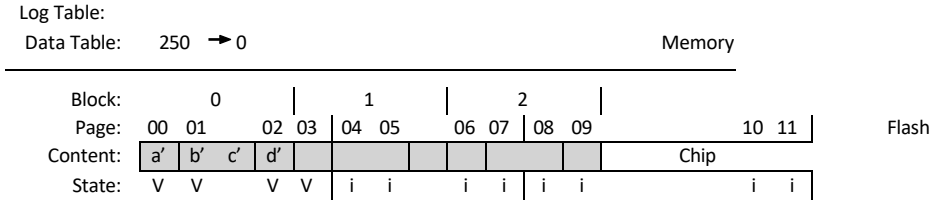
; 1000, 1001, 1002 ve 1003'e yazma içeriğinin sırasıyla a, b, c ve d olduğunu varsayalım.



Şimdi istemcinin bu blokların her birinin üzerine (a', b', c' ve d' verileriyle) tam olarak aynı sırada, şu anda mevcut olan günlük bloklarından birinde, örneğin fiziksel blok 0'da (fiziksel sayfalar 0) yazdığını varsayalım. , 1, 2 ve 3). Bu durumda, FTL şu duruma sahip olacaktır:

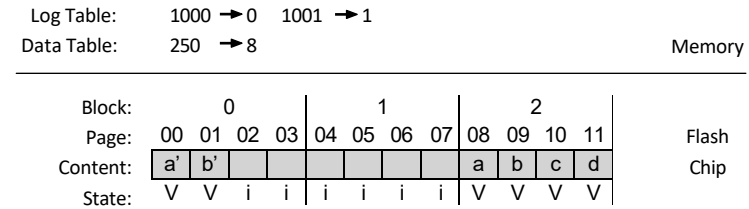


Bu bloklar tam olarak öncekiyle aynı şekilde yazıldığından, FTL, **anahtar birleştirme(switch merge)** olarak bilinen işlemi gerçekleştirebilir. Bu durumda, günlük bloğu (0) artık 0, 1, 2 ve 3 numaralı bloklar için depolama konumu haline gelir ve tek bir blok işaretçisi ile işaretlenir; eski blok (2) artık silinir ve log bloğu olarak kullanılır. Bu en iyi durumda, gereken tüm sayfa başına işaretçilerin yerini tek bir blok işaretçi alır.



Bu anahtar birleştirme(switch merge), hibrit bir FTL için en iyi durumdur. Ne yazık ki, bazen FTL o kadar da şanslı olmayabilir. Aynı başlangıç koşullarına sahip olduğumuz (fiziksel blok 2'de depolanan 1000 ... 1003 mantıksal blokları) olduğu, ancak daha sonra istemcinin 1000 ve 1001 mantıksal bloklarının üzerine yazdığı durumu hayal edin.

Sizce bu durumda ne olur? Başa çıkmak neden daha zorlaşır? *(sonraki sayfada sonuca bakmadan önce düşünün)*



Bu fiziksel bloğun diğer sayfalarını yeniden birleştirmek ve böylece onlara yalnızca tek bir blok işaretçisi ile başvurabilmek için FTL, **kısmi birleştirme(partial merge)** adı verilen işlemi gerçekleştirir. Bu işlemde, mantıksal bloklar 1002 ve 1003, fiziksel blok 2'den okunur ve ardından günlüğe eklenir. SSD'nin ortaya çıkan bu durumu, yukarıdaki anahtar birleştirme ile aynıdır; ancak bu durumda, FTL'nin hedeflerine ulaşmak için fazladan G/Ç(Input-Output) gerçekleştirmesi gerekiyordu, bu da yazma işlemlerini artırıyordu.

FTL'nin karşılaştığı son durum, **tam birleştirme(full merge)** olarak bilinir ve daha fazla çalışma gerektirir. Bu durumda, FTL'nin temizleme işlemini gerçekleştirmek için diğer birçok bloktan sayfaları bir araya getirmesi gerekir. Örneğin, 0, 4, 8 ve 12 mantıksal bloklarının log bloğu A'ya yazıldığını hayal edin. Bu günlük bloğunu(log block) blok eşlemeli(block-mapped) bir sayfaya(page) dönüştürmek için, FTL önce 0, 1, 2 ve 3 mantıksal bloklarını içeren bir veri bloğu oluşturmalıdır ve bu nedenle FTL başka bir yerden 1, 2 ve 3'ü okumalı ve sonra 0, 1, 2 ve 3'ü birlikte yazmalıdır. Daha sonra, birleştirme aynı şeyi mantıksal blok 4 için yapmalı, 5, 6 ve 7'yi bulmalı ve bunları tek bir fiziksel blokta uzlaştırmalıdır. Aynısı mantıksal bloklar 8 ve 12 için yapılmalıdır ve ardından (son olarak), log bloğu A serbest bırakılabilir. Sık sık yapılan tam birleştirmeler, şaşırtıcı olmadığı üzere, performansa ciddi şekilde zarar verebilir ve bu nedenle mümkün olduğunca kaçınılmalıdır. [GY+09].

Sayfa Eşleme ve Önbelleğe Alma (Page Mapping Plus Caching)

Yukarıdaki hibrit yaklaşımın karmaşıklığı göz önüne alındığında, sayfa eşlemeli FTL'lerin bellek yükünü azaltmak için daha basit yollar önerildi. Muhtemelen en basiti, FTL'nin yalnızca aktif kısımlarını bellekte önbelleğe almak, böylece gereken bellek miktarını azaltmaktır. [GY+09].

Bu yaklaşım işe yarayabilir. Örneğin, belirli bir iş yükü yalnızca küçük bir sayfa grubuna erişiyorsa, bu sayfaların çevirileri bellek içi FTL'de saklanır ve yüksek bellek maliyeti olmadan performans mükemmel olur. Tabii ki, bu yaklaşım da kötü performans gösterebilir. Bellek, gerekli çevirilerin **çalışma kümesini(working set)** içeremezse, her erişim, verilerin kendisine erişebilmeden önce eksik eşlemeyi getirmek için asgari olarak fazladan bir hızlı okuma gerektirecektir. Daha da kötüsü, yeni eşlemeye **yer açmak(evict)** için FTL'nin eski bir eşlemeyi çıkarması gerekebilir ve bu eşleme **kirliyse(dirty)** (yani, henüz kalıcı olarak flaşa yazılmamışsa), fazladan bir yazma işlemi de gerçekleşecektir. Ancak çoğu durumda, iş yükü yerel çalışacak ve bu önbelleğe alma yaklaşımı hem bellek ek yüklerini azaltacak hem de performansı yüksek tutacaktır.

44.10 Aşınma Dengeleme(Wear Leveling)

Son olarak, modern FTL'lerin uygulaması gereken ilgili bir arka plan etkinliği, yukarıda tanıtıldığı gibi **aşınma dengelemesidir(wear leveling)**. Temel fikir basittir: birden fazla silme/program döngüsü bir flaş bloğunu aşındıracağından, FTL bu işi cihazın tüm bloklarına eşit şekilde yaymak için elinden gelenin en iyisini yapmalıdır. Bu şekilde, birkaç "popüler" bloğun hızla kullanılamaz hale gelmesi yerine, tüm bloklar kabaca aynı anda yıpranacaktır.

Temel günlük yapılandırma yaklaşımı, yazma yükünü dağıtmak için iyi bir iş yapar ve çöp toplamada da yardımcı olur. Bununla birlikte, bazen bir blok, üzerine yazılmayan uzun ömürlü verilerle doldurulabilir; bu durumda, çöp toplama bloğu asla geri alamaz ve bu nedenle yazma yükünden adil payını almaz.

Bu sorunu çözmek için, FTL'nin bu tür bloklardan tüm canlı verileri periyodik olarak okuması ve başka bir yere yeniden yazması gerekir, böylece bloğu tekrar yazmaya uygun hale getirmiş olur. Bu aşınma seviyelendirme işlemi, SSD'nin yazma yükünü artırır ve böylece tüm blokların kabaca aynı oranda aşınmasını sağlamak için ekstra G/Ç(Input/Output) gerektiğinden performansı düşürür. Literatürde birçok farklı algoritma mevcuttur [A+08, M+14]; eğer ilgileniyorsanız daha fazlasını okuyun.

44.11 SSD Performansı ve Maliyeti(SSD Performance And Cost)

Bitirmeden önce, kalıcı depolama sistemlerinde nasıl kullanılacağını daha iyi anlamak için modern SSD'lerin performansını ve maliyetini inceleyelim. Her iki durumda da klasik sabit disk sürücüleri (HDD'ler) ile karşılaştıralım ve ikisi arasındaki en büyük farkları vurgulayalım.

Performans(Performance)

Sabit disk sürücülerinden farklı olarak, flash tabanlı SSD'lerin hiçbir mekanik bileşeni yoktur ve aslında "rastgele erişimli" aygıtlar olmaları bakımından birçok yönden DRAM'e daha benzerler. Disk sürücüleriyle karşılaştırıldığında performanstaki en büyük fark, rasgele okuma ve yazma işlemleri gerçekleştirilirken fark edilir; tipik bir disk sürücüsü saniyede yalnızca birkaç yüz rasgele G/Ç(I/O) gerçekleştirebilirken, SSD'ler çok daha iyisini yapabilir. Burada, SSD'lerin ne kadar daha iyi performans gösterdiğini görmek için modern SSD'lerden alınan bazı verilere bakalım; FTL'lerin ham yongaların performans sorunlarını ne kadar iyi gizlediğine özellikle dikkat edelim.

Tablo 44.4, üç farklı SSD ve birinci sınıf bir sabit disk için bazı performans verilerini göstermektedir; veriler birkaç farklı çevrimiçi kaynaktan alınmıştır [S13, T15]. Soldaki iki sütun rastgele G/Ç performansını ve sağdaki iki sütun sıralı yazma/okuma performansını gösteriyor; ilk üç satır, üç farklı SSD'ye (Samsung, Seagate ve Intel'den) ilişkin verileri gösteriyor ve son satır, bir **sabit disk sürücüsünün(hard disk drive)** (veya **HDD**), bir Seagate ileri teknoloji diskinin performansını gösteriyor.

Tablodan birkaç ilginç gerçek öğrenebiliriz. Birincisi ve en dramatik olanı, SSD'ler ile sabit sürücü arasındaki rastgele G/Ç performansındaki farktır.

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure 44.4: **SSDs And Hard Drives: Performance Comparison**

Figür 44.4: **SSDler ve Hard Diskler arasında Performans Karşılaştırması**

SSD'ler rastgele G/Ç'lerde(I/O) onlarca hatta yüzlerce MB/s elde ederken, bu "yüksek performanslı" sabit sürücünün zirvesi sadece birkaç MB/sn'dir (aslında, 2 MB/sn'ye yuvarlanmıştır). İkincisi, sıralı performans açısından çok daha az fark olduğunu görebilirsiniz; SSD'ler daha iyi performans gösterse de, ihtiyacınız olan tek şey sıralı performanssa, bir sabit sürücü yine de iyi bir seçimdir. Üçüncüsü, SSD rasgele okuma performansının SSD rasgele yazma performansı kadar iyi olmadığını görebilirsiniz. Rastgele yazma performansının bu kadar beklenmedik şekilde iyi olmasının nedeni, birçok SSD'nin rastgele yazmaları sıralı yazmalara dönüştüren ve performansı iyileştiren günlük yapılı tasarımından kaynaklanmaktadır. Son olarak, SSD'ler sıralı ve rastgele G/Ç'ler arasında bir miktar performans farkı sergiledikleri için, sabit diskler için dosya sistemlerinin nasıl oluşturulacağına ilişkin sonraki bölümlerde öğreneceğimiz tekniklerin çoğu SSD'ler için hala geçerlidir; sıralı ve rasgele G/Ç'ler arasındaki farkın büyüklüğü daha küçük olsa da, rasgele G/Ç'leri azaltmak için dosya sistemlerinin nasıl tasarlanacağını dikkatlice düşünmek için yeterince boşluk vardır.

Maliyet(Cost)

Yukarıda gördüğümüz gibi, SSD'lerin performansı, sıralı G/Ç gerçekleştirirken bile modern sabit sürücüleri büyük ölçüde geride bırakıyor. Öyleyse SSD'ler neden tercih edilen depolama ortamı olarak sabit sürücülerin yerini tamamen almadı? Cevap basit: maliyet veya daha spesifik olarak kapasite birimi başına maliyet. Şu anda [A15], bir SSD'nin maliyeti 250 GB'lık bir sürücü için yaklaşık 150 USD'dir; böyle bir SSD, GB başına 60 sente mal oluyor. Tipik bir sabit disk, 1 TB depolama için yaklaşık 50 ABD dolarına mal olur, bu da GB başına 5 sente mal olduğu anlamına gelir. Bu iki depolama ortamı arasında hala 10 kattan fazla maliyet farkı var.

Bu performans ve maliyet farklılıkları, büyük ölçekli depolama sistemlerinin nasıl oluşturulacağını belirler. Asıl mesele performanssa, özellikle rastgele okuma performansı önemliyse, SSD'ler harika bir seçimdir. Öte yandan, büyük bir veri merkezi kuruyorsanız ve çok büyük miktarda bilgi depolamak istiyorsanız, büyük maliyet farkı sizi sabit disklere yönlendirecektir. Elbette hibrit bir yaklaşım mantıklı olabilir - bazı depolama sistemleri hem SSD'ler hem de sabit sürücülerle bir araya getiriliyor, daha popüler "sıcak" veriler için daha az sayıda SSD kullanılıyor ve yüksek performans sağlanırken geri kalan "soğuk" veriler depolanıyor. Maliyetten tasarruf etmek için sabit disklerdeki (daha az kullanılan) veriler. Fiyat farkı var olduğu sürece, sabit diskler tercih sebebidir.

44.12 Özet(Summary)

Flash tabanlı SSD'ler, dünya ekonomisine güç sağlayan veri merkezlerindeki dizüstü bilgisayarlarda, masaüstü bilgisayarlarda ve sunucularda yaygınlaşıyor. Bu nedenle onlar hakkında bir şeyler bilmelisiniz, değil mi?

İşte kötü haber: bu bölüm (bu kitaptaki pek çok bölüm gibi), teknolojinin durumunu anlamanın yalnızca ilk adımıdır. Ham teknoloji hakkında daha fazla bilgi edinebileceğiniz bazı yerler arasında, gerçek cihaz performansı (Chen ve diğerleri [CK+09] ve Grupp ve diğerleri [GC+09] tarafından yapılanlar gibi), FTL tasarımındaki sorunlar (çalışmalar dahil) yer alır. Agrawal ve diğerleri [A+08], Gupta ve diğerleri [GY+09], Huang ve diğerleri [H+14], Kim ve diğerleri [KK+02], Lee ve diğerleri [L+07] ve Zhang ve diğerleri [Z+12]) ve hatta flaştan oluşan dağıtılmış sistemler (Gordon [CG+09] ve CORFU [B+12] dahil). Ve tabiri caizse, bir SSD'den yüksek performans elde etmek için yapmanız gereken her şeye gerçekten iyi bir genel bakış, " unwritten contract " konulu bir makalede bulunabilir. [HK+17].

Sadece akademik makaleler okumayın; ayrıca popüler basındaki son gelişmeleri de okuyun (örneğin [V12]). Burada, Samsung'un performansı (SLC yazmaları hızlı bir şekilde arabelleğe alabilir) ve kapasiteyi (TLC hücre başına daha fazla bit depolayabilir) en üst düzeye çıkarmak için aynı SSD içinde hem TLC hem de SLC hücrelerini kullanması gibi daha pratik (ancak yine de yararlı) bilgiler öğreneceksiniz. Ve bu, dedikleri gibi, buzdağının sadece görünen kısmı. İşe koyulun ve "buzdağı" hakkında kendi başınıza daha fazla bilgi edinin, belki de Ma ve diğerlerinin mükemmel (ve yakın tarihli) anketinden başlayabilirsiniz. [M+14]. Yine de dikkatli olun; buzdağları en güçlü gemileri bile batırabilir [W15].

NOT: SSD'YLE ALAKALI ANAHTAR TERİMLER

- Bir **flaş çipi(flash chip)**, her biri **silme blokları(erase blocks)** (bazen sadece **bloklar(blocks)** olarak adlandırılır) halinde düzenlenen birçok bankadan oluşur. Her blok ayrıca belirli sayıda **sayfaya(page)** bölünmüştür.
- Bloklar büyüktür (128KB–2MB) ve nispeten küçük (1KB–8KB) birçok sayfa içerir.
- Flaştan okumak için adres ve uzunluk içeren bir okuma komutu verin; bu, bir itemcinin bir veya daha fazla sayfayı okumasına izin verir.
- Flaşa yazmak daha karmaşıktır. İlk olarak, istemci tüm bloğu **silmelidir(erase)** (bu, blok içindeki tüm bilgileri siler). Ardından, istemci her sayfayı tam olarak bir kez **programlayarak(program)** yazmayı tamamlayabilir.
- Belirli bir bloğa (veya blok aralığına) artık gerek kalmadığında bunu gerçekleştirmek için bir **kırpma(trim)** işlemi kullanılabilir. Flaş güvenilirliği çoğunlukla **aşınma(wear out)** ile belirlenir; bir blok çok sık silinir ve programlanırsa kullanılamaz hale gelir.
- Flaş tabanlı **katı hal depolama aygıtı (SSD) (solid-state storage device (SSD))**, blok tabanlı normal bir okuma/yazma diski gibi davranır; bir **flash çeviri katmanı(FTL) (flash translation layer (FTL))** kullanarak, bir istemciden gelen okuma ve yazma işlemlerini okumalara, silmelere ve temel flash yongalara programlamaya dönüştürür.
- FTL'lerin çoğu, silme/program döngülerini en aza indirerek yazma maliyetini azaltan **günlük yapılıdır(log-structured)**. Bir bellek içi çeviri katmanı, fiziksel ortam içinde mantıksal yazma işlemlerinin nerede yapıldığını izler.
- Günlük yapılı FTL'lerle ilgili temel sorunlardan biri, **aşırı yazmaya(write amplification)** yol açan **çöp toplama(garbage collection)** maliyetidir.
- Diğer bir sorun da oldukça büyüyebilen eşleme tablosunun boyutudur. **Hibrit eşleme(hybrid mapping)** kullanmak veya yalnızca FTL'nin aktif parçalarını **önbelleğe almak(caching)** olası çözümlerdir.
- Son bir sorun da **aşınma dengelemesidir(wear leveling)**; söz konusu blokların silme/program yükünden paylarını almalarını sağlamak için, FTL ara sıra çoğunlukla okunan bloklardan veri taşımalıdır.

Referanslar/Kaynakça (References)

[A+08] "Design Tradeoffs for SSD Performance" by N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy. USENIX '08, San Diego California, June 2008. *An excellent overview of what goes into SSD design.*

[AD14] "Operating Systems: Three Easy Pieces" by *Chapters: Crash Consistency: FSCK and Jour- naling and Log-Structured File Systems*. Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *A lot more detail here about how logging can be used in file systems; some of the same ideas can be applied inside devices too as need be.*

[A15] "Amazon Pricing Study" by Remzi Arpaci-Dusseau. February, 2015. *This is not an actual paper, but rather one of the authors going to Amazon and looking at current prices of hard drives and SSDs. You too can repeat this study, and see what the costs are today. Do it!*

[B+12] "CORFU: A Shared Log Design for Flash Clusters" by M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, J. D. Davis. NSDI '12, San Jose, California, April 2012. *A new way to think about designing a high-performance replicated log for clusters using Flash.*

[BD10] "Write Endurance in Flash Drives: Measurements and Analysis" by Simona Boboila, Peter Desnoyers. FAST '10, San Jose, California, February 2010. *A cool paper that reverse en- gineers flash-device lifetimes. Endurance sometimes far exceeds manufacturer predictions, by up to 100×.*

[B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Available here: http://www.ostep.org/Citations/zfs_last.pdf. *Was this the last word in file sys- tems? No, but maybe it's close.*

[CG+09] "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications" by Adrian M. Caulfield, Laura M. Grupp, Steven Swanson. ASPLOS '09, Wash- ington, D.C., March 2009. *Early research on assembling flash into larger-scale clusters; definitely worth a read.*

[CK+09] "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives" by Feng Chen, David A. Koufaty, and Xiaodong Zhang. SIGMET- RICS/Performance '09, Seattle, Washington, June 2009. *An excellent overview of SSD performance problems circa 2009 (though now a little dated).*

[G14] "The SSD Endurance Experiment" by Geoff Gasior. The Tech Report, September 19, 2014. Available: http://techreport.com/review/27062. *A nice set of simple experiments measuring performance of SSDs over time. There are many other similar studies; use google to find more.*

[GC+09] "Characterizing Flash Memory: Anomalies, Observations, and Applications" by L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf. IEEE MICRO '09, New York, New York, December 2009. *Another excellent characterization of flash performance.*

[GY+09] "DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings" by Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar. ASP- LOS '09, Washington, D.C., March 2009. *This paper gives an excellent overview of different strategies for cleaning within hybrid SSDs as well as a new scheme which saves mapping table space and improves performance under many workloads.*

[HK+17] "The Unwritten Contract of Solid State Drives" by Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. EuroSys '17, Belgrade, Serbia, April 2017. *Our own paper which lays out five rules clients should follow in order to get the best performance out of modern SSDs. The rules are request scale, locality, aligned sequentiality, grouping by death time, and uniform lifetime. Read the paper for details!*

[H+14] "An Aggressive Worn-out Flash Block Management Scheme To Alleviate SSD Perfor- mance Degradation" by Ping Huang, Guanying Wu, Xubin He, Weijun Xiao. EuroSys '14, 2014. *Recent work showing how to really get the most out of worn-out flash blocks; neat!*

[J10] "Failure Mechanisms and Models for Semiconductor Devices" by Unknown author. Re- port JEP122F, November 2010. Available on the internet at this exciting so-called web site: http://www.jedec.org/sites/default/files/docs/JEP122F.pdf. *A highly detailed discussion of what is going on at the device level and how such devices fail. Only for those not faint of heart. Or physicists. Or both.*

[KK+02] "A Space-Efficient Flash Translation Layer For Compact Flash Systems" by Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho. IEEE Transactions on Con- sumer Electronics, Volume 48, Number 2, May 2002. *One of the earliest proposals to suggest hybrid mappings.*

[L+07] "A Log Buffer-Based Flash Translation Layer by Using Fully-Associative Sector Trans- lation. " Sang-won Lee, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song. ACM Transactions on Embedded Computing Systems, Volume 6, Number 3, July 2007 *A terrific paper about how to build hybrid log/block mappings.*

[M+14] "A Survey of Address Translation Technologies for Flash Memories" by Dongzhe Ma, Jianhua Feng, Guoliang Li. ACM Computing Surveys, Volume 46, Number 3, January 2014. *Probably the best recent survey of flash and related technologies.*

[S13] "The Seagate 600 and 600 Pro SSD Review" by Anand Lal Shimpi. AnandTech, May 7, 2013. Available: http://www.anandtech.com/show/6935/seagate-600-ssd-review. *One of many SSD performance measurements available on the internet. Haven't heard of the internet? No problem. Just go to your web browser and type "internet" into the search tool. You'll be amazed at what you can learn.*

[T15] "Performance Charts Hard Drives" by Tom's Hardware. January 2015. Available here: http://www.tomshardware.com/charts/enterprise-hdd-charts. *Yet another site with performance data, this time focusing on hard drives.*

[V12] "Understanding TLC Flash" by Kristian Vatto. AnandTech, September, 2012. Available: http://www.anandtech.com/show/5067/understanding-tlc-nand. *A short descrip- tion about TLC flash and its characteristics.*

[W15] "List of Ships Sunk by Icebergs" by Many authors. Available at this location on the "web": http://en.wikipedia.org/wiki/Listofshipssunkbyicebergs. *Yes, there is a wikipedia page about ships sunk by icebergs. It is a really boring page and basically everyone knows the only ship the iceberg-sinking-mafia cares about is the Titanic.*

[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 research on a new idea to reduce mapping table space; the key is to re-use the pointers in the file system above to store locations of blocks, instead of adding another level of indirection.*

Ödev(Simulasyon) (Homework (Simulation))

Bu bölüm ssd.py dosyasını anlatıyor, ssd.py basit bir SSD simulasyonudur ve SSD'nin nasıl çalıştığını anlamaya yardımcı olur. README dosyasını okuyarak simulasyonun nasıl çalıştığı hakkında detaylara bakabilirsiniz. O uzun bir README dosyasıdır, bu yüzden bir bardak çay hazırlayın (gerektiği kadar kafein alın), okuma gözlükleriniz takın, kedinizin dizinizde kıvrılmasına izin verin ve işe koyulun ☺

Sorular(Questions)

1.Ödev, çoğunlukla "-T log" bayrağıyla simule edilen log yapılı(log structured) SSD'lere odaklanacaktır. Karşılaştırma için diğer SSD türlerini kullanacağız. Komutu şu bayraklarla çalıştırın: -T log -s 1 -n 10 -q. Hangi operasyonların gerçekleştiğini anlayabilir misiniz? Sorunuza cevap bulmak için -c komutunu kullanın (veya -q –c yerine sadece –C kullanın). Farklı rasgele iş yükleri oluşturmak için farklı -s değerleri kullanın.

```
eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$ python3 ssd.py -T log -s 1 -n 10 -q
ARG seed 1
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds True
ARG show_stats False
ARG compute False

FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live

cmd   0:: command(??) -> ??

FTL    12:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  u
Live  +

cmd   1:: command(??) -> ??

FTL    12:  0  32:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM
Live  ++

cmd   2:: read(32) -> ??

FTL    12:  0  32:  1
```

```
cmd   2:: read(32) -> ??

FTL    12:  0  32:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM
Live  ++

cmd   3:: command(??) -> ??

FTL    12:  0  32:  1  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0
Live  +++

cmd   4:: command(??) -> ??

FTL    12:  0  32:  1  36:  3  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  ++++

cmd 5:: command(??) -> ??

FTL    12:  0  32:  1  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  +++

cmd   6:: read(32) -> ??

FTL    12:  0  32:  1  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  +++

cmd 7:: command(??) -> ??

FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
```

```
cmd 7:: command(??) -> ??

FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  + +

cmd   8:: read(12) -> ??

FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  + +

cmd   9:: read(12) -> ??

FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  + +


eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$
```

Cevaplara bakmak için –C flag'i ekleyeceğiz

```
eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$ python3 ssd.py -T log -s 1 -n 10 -q -c
ARG seed 1
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds True
ARG show_stats False
ARG compute True

FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live

cmd   0:: write(12, u) -> success

FTL    12:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  u
Live  +

cmd   1:: write(32, M) -> success

FTL    12:  0  32:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM
Live  ++

cmd   2:: read(32) -> M

FTL    12:  0  32:  1
```

```
cmd   2:: read(32) -> M

FTL    12:  0  32:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM
Live  ++

cmd   3:: write(38, 0) -> success

FTL    12:  0  32:  1  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0
Live  +++

cmd   4:: write(36, e) -> success

FTL    12:  0  32:  1  36:  3  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  ++++

cmd   5:: trim(36) -> success

FTL    12:  0  32:  1  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  +++

cmd   6:: read(32) -> M

FTL    12:  0  32:  1  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  +++

cmd   7:: trim(32) -> success

FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
```

```
FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  + +

cmd   8:: read(12) -> u

FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  + +

cmd   9:: read(12) -> u

FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uM0e
Live  + +


eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$
```

-s parametresini değiştirerek başka bir örnek inceleyelim

```
eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$ python3 ssd.py -T log -s 2 -n 10 -q
ARG seed 2
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds True
ARG show_stats False
ARG compute False

FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live

cmd   0:: command(??) -> ??

FTL    36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F
Live  +

cmd   1:: command(??) -> ??

FTL    29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9
Live  ++

cmd   2:: command(??) -> ??

FTL    19:  2  29:  1  36:  0
```

```
cmd   2:: command(??) -> ??

FTL    19:  2  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9I
Live  +++

cmd 3:: command(??) -> ??

FTL    29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9I
Live  ++

cmd   4:: command(??) -> ??

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   5:: read(29) -> ??

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   6:: read(22) -> ??

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   7:: command(??) -> ??

FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
```

```
cmd   7:: command(??) -> ??

FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ige
Live  ++ ++

cmd   8:: read(36) -> ??

FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ige
Live  ++ ++

cmd   9:: command(??) -> ??

FTL    22:  3  28:  4  29:  1  36:  0  49:  5
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9IgeF
Live  ++ +++


eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$
```

-c flagini ekleyerek diğer örneğimizin cevabına bakalım

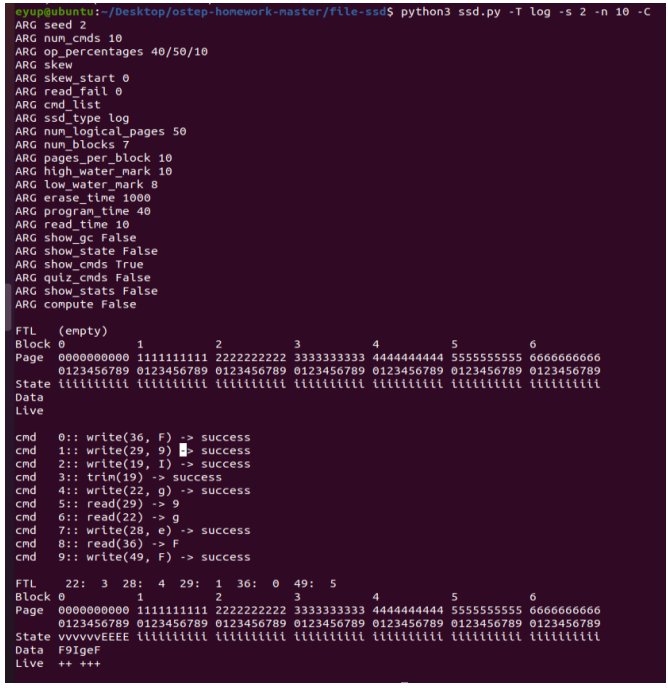
```
eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$ python3 ssd.py -T log -s 2 -n 10 -q -c
ARG seed 2
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds True
ARG show_stats False
ARG compute True

FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live

cmd   0:: write(36, F) -> success

FTL    36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F
Live  +

cmd   1:: write(29, 9) -> success

FTL    29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9
Live  ++

cmd   2:: write(19, I) -> success
```

```
cmd   2:: write(19, I) -> success

FTL    19:  2  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9I
Live  +++

cmd   3:: trim(19) -> success

FTL    29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9I
Live  ++

cmd   4:: write(22, g) -> success

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   5:: read(29) -> 9

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   6:: read(22) -> g

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   7:: write(28, e) -> success

FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
```

```
FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ige
Live  ++ ++

cmd   8:: read(36) -> F

FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ige
Live  ++ ++

cmd   9:: write(49, F) -> success

FTL    22:  3  28:  4  29:  1  36:  0  49:  5
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9IgeF
Live  ++ +++
```

2. Şimdi sadece komutları gözlemleyin ve Flash'ın ara durumlarını çözüp çözemeyeceğinize bakın. Komutları gözlemlemek için -T log -s 2 –n 10 –C bayraklarıyla çalıştırın. Şimdi, her komut arasında Flash'ın durumuna bakın; durumları göstermek ve haklı olup olmadığınızı görmek için -F'yi kullanın. Gelişmekte olan uzmanlığınızı test etmek için farklı rastgele değerler kullanın.

-T log -s 2 –n 10 –C flagleri eklenmiş komutumuzun çıktısı



-F flagi eklenmiş halinin çıktısı

```
eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$ python3 ssd.py -T log -s 2 -n 10 -C -F
ARG seed 2
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state True
ARG show_cmds True
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live

cmd   0:: write(36, F) -> success

FTL    36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F
Live  +

cmd   1:: write(29, 9) -> success

FTL    29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9
Live  ++

cmd   2:: write(19, I) -> success

FTL    19:  2  29:  1  36:  0
```

```
cmd   2:: write(19, I) -> success

FTL    19:  2  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9I
Live  +++

cmd   3:: trim(19) -> success

FTL    29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9I
Live  ++

cmd   4:: write(22, g) -> success

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   5:: read(29) -> 9

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   6:: read(22) -> g

FTL    22:  3  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ig
Live  ++ +

cmd   7:: write(28, e) -> success

FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
```

```
cmd   7:: write(28, e) -> success

FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ige
Live  ++ ++

cmd   8:: read(36) -> F

FTL    22:  3  28:  4  29:  1  36:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9Ige
Live  ++ ++

cmd   9:: write(49, F) -> success

FTL    22:  3  28:  4  29:  1  36:  0  49:  5
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  F9IgeF
Live  ++ +++


eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$
```

-s parametresini değiştirerek diğer örneği inceleyelim

```
eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$ python3 ssd.py -T log -s 3 -n 10 -C -F
ARG seed 3
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state True
ARG show_cmds True
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live

cmd   0:: write(3, 0) -> success

FTL     3:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0
Live  +

cmd   1:: write(12, e) -> success

FTL     3:  0  12:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0e
Live  ++

cmd   2:: trim(3) -> success

FTL    12:  1
```

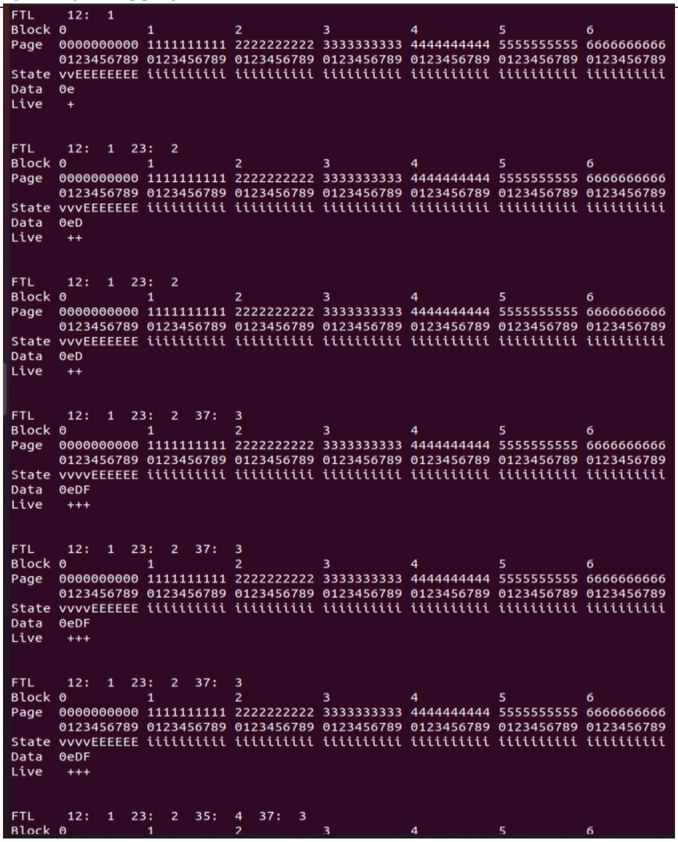
```
cmd   2:: trim(3) -> success

FTL    12:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0e
Live   +

cmd   3:: write(23, D) -> success

FTL    12:  1  23:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eD
Live   ++

cmd   4:: read(23) -> D

FTL    12:  1  23:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eD
Live   ++

cmd   5:: write(37, F) -> success

FTL    12:  1  23:  2  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDF
Live   +++

cmd   6:: read(23) -> D

FTL    12:  1  23:  2  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDF
Live   +++

cmd   7:: read(37) -> F

FTL    12:  1  23:  2  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
```

```
cmd   7:: read(37) -> F

FTL    12:  1  23:  2  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDF
Live   +++

cmd   8:: write(35, S) -> success

FTL    12:  1  23:  2  35:  4  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDFS
Live   ++++

cmd   9:: write(46, o) -> success

FTL    12:  1  23:  2  35:  4  37:  3  46:  5
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDFSo
Live   +++++


eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$
```

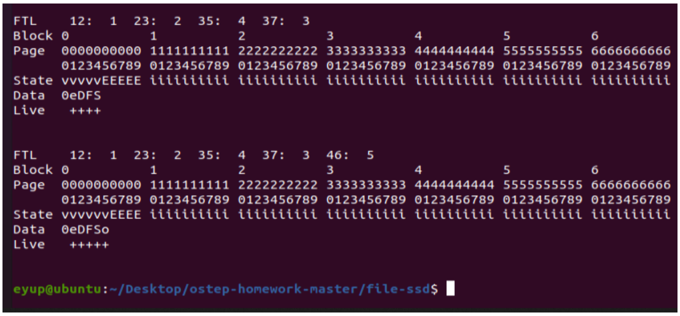
3. -r 20 bayrağını ekleyerek bu sorunu biraz daha ilginç hale getirelim. Bu, komutlarda ne gibi farklılıklara neden oldu? Sorularınıza cevap bulmak için tekrar -c komutunu kullanın.

-r 20 flagi eklenmiş çıktı

```
eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$ python3 ssd.py -T log -s 3 -n 10 -F -r 20
ARG seed 3
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 20
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state True
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live


FTL     3:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0
Live  +


FTL     3:  0  12:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0e
Live  ++


FTL    12:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvFFFFFFFF iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
```



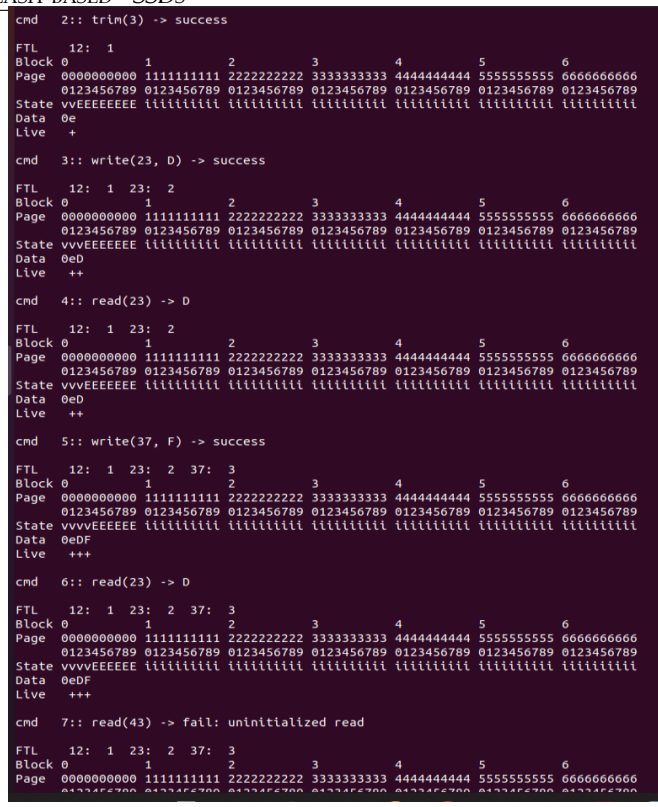
```
FTL    12:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0e
Live   +


FTL    12:  1  23:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eD
Live   ++


FTL    12:  1  23:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eD
Live   ++


FTL    12:  1  23:  2  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDF
Live   +++


FTL    12:  1  23:  2  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDF
Live   +++


FTL    12:  1  23:  2  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDF
Live   +++


FTL    12:  1  23:  2  35:  4  37:  3
Block 0          1          2          3          4          5          6
```

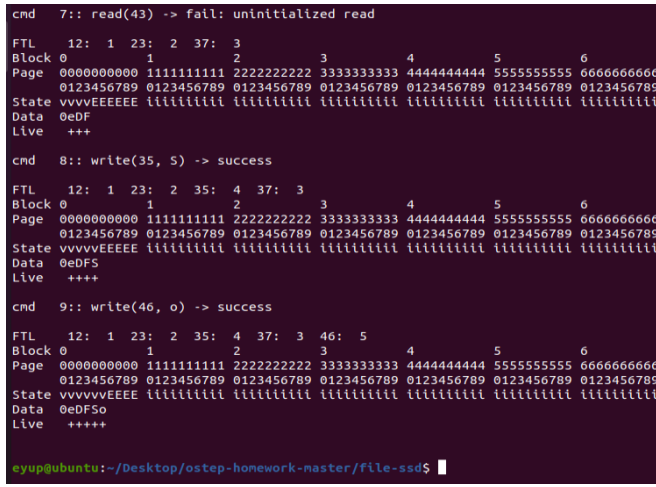


```
FTL    12:  1  23:  2  35:  4  37:  3
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDFS
Live   ++++


FTL    12:  1  23:  2  35:  4  37:  3  46:  5
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0eDFSo
Live   +++++

eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$
```

-C parametresi ile cevapları kontrol edelim

```
eyup@ubuntu:~/Desktop/ostep-homework-master/file-ssd$ python3 ssd.py -T log -s 3 -n 10 -C -F -r 20
ARG seed 3
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 20
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state True
ARG show_cmds True
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live

cmd   0:: write(3, 0) -> success

FTL     3:  0
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0
Live  +

cmd   1:: write(12, e) -> success

FTL     3:  0  12:  1
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  0e
Live  ++

cmd   2:: trim(3) -> success

FTL    12:  1
Block 0          1          2          3          4          5          6
```





4. Performans, silme, program ve okuma sayısına göre belirlenir (burada

kırpmaların serbest olduğunu varsayıyoruz). Aynı iş yükünü yukarıdaki gibi yeniden çalıştırın, ancak ara durumları göstermeden (ör. -T log -s 1 -n 10). Bu iş yükünün tamamlanmasının ne kadar süreceğini tahmin edebilir misiniz? (varsayılan silme süresi 1000 mikrosaniye, program süresi 40 ve okuma süresi 10'dur) Cevabınızı kontrol etmek için -S bayrağını kullanın. -E, -W, -R bayraklarıyla silme, programlama ve okuma sürelerini de değiştirebilirsiniz.

5. Şimdi, günlük yapılı yaklaşımın(log-structured approach) performansını ve (çok kötü) doğrudan yaklaşımı(direct approach) (-T günlüklü yerine -T doğrudan) karşılaştırın. İlk olarak, doğrudan yaklaşımın nasıl performans göstereceğini tahmin edin, ardından -S bayrağıyla cevabınızı kontrol edin. Genel olarak, günlük yapılı yaklaşım doğrudan yaklaşımdan ne kadar daha iyi performans gösterir?

6. Şimdi çöp toplayıcının davranışını inceleyelim. Bunu yapmak için, yüksek (-G) ve düşük (-g) filigranları uygun şekilde ayarlamalıyız. Öncelikle log yapılı SSD'ye herhangi bir çöp toplama olmadan daha büyük bir iş yükü yüklediğnizde ne olduğunu gözlemleyelim. Bunu yapmak için komutu -T log -n 1000 bayraklarıyla çalıştırın (yüksek filigran varsayılanı 10'dur, dolayısıyla GC bu yapılandırmada çalışmaz). Ne olacağını düşünüyorsun? Görmek için -C ve belki –F komutunu kullan.

7. Çöp toplayıcıyı etkinleştirmek için daha düşük değerler kullanın. Yüksek filigran (-G N), sisteme N blok kullanıldıktan sonra toplamaya başlamasını söyler; düşük filigran (-G M), kullanımda olan yalnızca M blok olduğunda sisteme toplamayı durdurmasını söyler. Çalışan bir sistem için hangi filigran değerlerinin işe yarayacağını düşünüyorsunuz? Komutları ve ara cihaz durumlarını göstermek için -C ve -F tuşlarını kullanın ve görün.

8. Bir diğer yararlı bayrak, toplayıcının çalıştığında ne yaptığını gösteren -J'dir. Hem komutları hem de GC(garbage collector) davranışını görmek için -T log -n 1000 -C -J bayraklarıyla çalıştırın. GC hakkında neyi farkettiniz? GC'nin nihai etkisi elbette performanstır. Nihai istatistiklere bakmak için -S'yi kullanın; çöp toplama(garbage collection) nedeniyle kaç tane fazladan okuma ve yazma oluyor? Bunu ideal SSD (-T ideal) ile karşılaştırın; Flash'ın doğası gereği ne kadar fazladan okuma, yazma ve silme var? Log yapılı yaklaşımı doğrudan yaklaşımla da karşılaştırın;log yapılı yaklaşım hangi açıdan (silme, okuma, programlama) üstündür?

9. Keşfedilecek son bir husus, **iş yükü çarpıklığıdır(workload skew)**. İş yükü değişikliklerine çarpıklık eklemek, mantıksal blok alanının daha küçük bir kısmına daha fazla yazmanın gerçekleşmesini sağlayacak şekilde yazmaları değiştirir. Örneğin, -K 80/20 ile çalıştırmak, yazma işlemlerinin %80'inin blokların %20'sine gitmesini sağlar. Bazı farklı çarpıklık değerleri seçin ve çarpıklığı anlamak için önce -T direct'i ve ardından günlük yapılı bir cihaz üzerindeki etkiyi görmek için -T log'u kullanarak rastgele seçilmiş birçok işlemi (ör. -n 1000) gerçekleştirin. Ne olmasını umarsınız? Keşfedilecek diğer bir küçük çarpıklık kontrolü -k 100'dür; çarpık bir iş yüküne bu bayrağı ekleyerek, ilk 100 yazma çarpık olmaz. Fikir, önce çok fazla veri oluşturmak, ancak daha sonra yalnızca bir kısmını güncellemektir. Bunun bir çöp toplayıcı üzerinde ne gibi bir etkisi olabilir?