45

Veri Bütünlüğü ve Koruma

Şimdiye kadar incelediğimiz dosya sistemlerinde bulunan temel ilerlemelerin ötesinde, bir dizi özellik üzerinde çalışmaya değer.

Bu bölümde, bir kez daha güvenilirliğe odaklanıyoruz (daha önce RAID bölümünde depolama sistemi güvenilirliğini incelemiştik). Özellikle, modern depolama cihazlarının güvenilmez doğası göz önüne alındığında, bir dosya sistemi veya depolama sistemi verilerin güvenli olmasını nasıl sağlamalı?

Bu genel alana veri bütünlüğü(**data integrity)** veya veri koruma (**data protection**) denir. Bu nedenle, şimdi depolama sisteminize koyduğunuz verilerin, depolama sistemi size geri döndüğünde aynı olmasını sağlamak için kullanılan teknikleri inceleyeceğiz

CRUX: VERİ BÜTÜNLÜĞÜNÜ NASIL SAĞLANIR

Sistemler, depolamaya yazılan verilerin korunmasını nasıl sağlamalıdır? Hangi teknikler gereklidir? Bu tür teknikler, hem düşük alan hem de zaman giderleriyle nasıl verimli hale

45.1 DİSK HATASI MODLARI

RAID ile ilgili bölümde öğrendiğiniz gibi, diskler mükemmel değildir ve (bazen) arızalanabilir. İlk RAID sistemlerinde, arıza modeli oldukça basitti: ya tüm disk çalışıyor ya da tamamen arızalanıyor ve böyle bir arızanın tespiti çok kolay . Bu arıza durdurma (**fail-stop)** disk arızası modeli, RAID oluşturmayı nispeten basit hale getirir [S90].

Modern disklerin sergilediği diğer tüm arıza modları hakkında öğrenmediğiniz şey. Spesifik olarak, Bairavasundaram ve ark. ayrıntılı olarak incelendi [B+07, B+08], modern diskler bazen çoğunlukla çalışıyor gibi görünecek, ancak bir veya daha fazla bloğa başarıyla erişmede sorun yaşayacak. Spesifik olarak, iki tür tek blok hatası yaygındır ve dikkate alınmaya değerdir: gizli sektör hataları(**latent-sector errors)** (LSE'ler) ve blok bozulması (**block corruption).** Şimdi her birini daha ayrıntılı olarak tartışacağız.

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

Şekil 45.1: LSE Sıklığı ve Blok Bozulması(**Frequency Of LSEs And Block Corruption)**

LSE'ler, bir disk sektörü (veya sektör grubu) bir şekilde hasar gördüğünde ortaya çıkar. Örneğin, disk kafası herhangi bir nedenle (normal çalışma sırasında olmaması gereken bir kafa çarpması (**head crash)**) yüzeye dokunursa, yüzeye zarar verebilir ve bitleri okunamaz hale getirebilir. Neyse ki, disk içi hata düzeltme kodları (ECC), sürücü tarafından bir bloktaki disk üzerindeki bitlerin iyi olup olmadığını belirlemek ve bazı durumlarda bunları düzeltmek için kullanılır; iyi değillerse ve sürücüde hatayı düzeltmek için yeterli bilgi yoksa, bunları okumak için bir istek gönderildiğinde disk bir hata döndürür.

Ayrıca, bir disk bloğunun bozulmayacak şekilde bozulduğu(**corrupt)** durumlar da vardır. diskin kendisi tarafından algılanabilir. Örneğin, buggy disk üretici yazılımı bir bloğu yanlış konuma yazabilir; böyle bir durumda, disk ECC'si blok içeriğinin iyi olduğunu gösterir, ancak müşterinin bakış açısına göre, daha sonra erişildiğinde yanlış blok döndürülür. .Benzer şekilde, hatalı bir veri yolu üzerinden ana bilgisayardan diske aktarılan bir blok bozulabilir; ortaya çıkan bozuk veriler diskte saklanır, ancak müşterinin istediği bu değildir. Bu tür arızalar özellikle sinsidir çünkü sessiz arızalardır(**silent faults)**; disk, hatalı verileri döndürürken sorunla ilgili herhangi bir belirti vermez.

Prabhakaran ve ark. disk arızasının bu daha modern görüşünü arıza-kısmi disk arızası modeli [P+05] olarak tanımlar. Bu görüşe göre, diskler yine de tamamen başarısız olabilir (geleneksel arıza durdurma modelinde olduğu gibi); bununla birlikte, diskler görünüşte çalışıyor olabilir ve bir veya daha fazla bloğa erişilemez hale gelebilir (yani, LSE'ler) veya yanlış içerikleri tutabilir (yani, bozulma) Bu nedenle, görünüşte çalışan bir diske erişirken, arada bir belirli bir bloğu okumaya veya yazmaya çalışırken bir hata (sessiz olmayan bir kısmi hata) döndürebilir ve arada bir sadece yanlış verileri döndürebilir. (sessiz bir kısmi hata).

Bu tür hataların her ikisi de biraz nadirdir, ancak ne kadar nadirdir? Şekil 45.1, iki Bairavasundaram çalışmasından [B+07,B+08] bazı bulguları özetlemektedir.

Şekil, çalışma boyunca (yaklaşık 3 yıl,
1,5 milyon disk sürücüsü). Şekil, sonuçları "ucuz" sürücüler (genellikle SATA sürücüler) ve "pahalı" sürücüler (genellikle SCSI veya Fiber Kanal) olarak alt gruplara ayırır. Gördüğünüz gibi, daha iyi diskler satın almak her iki sorun türünün sıklığını azaltırken (yaklaşık bir büyüklük sırasına göre), yine de depolama sisteminizde bunları nasıl ele alacağınızı dikkatlice düşünmeniz gerekecek kadar sık meydana geliyor

LSE'ler hakkında bazı ek bulgular şunlardır

• Birden fazla LSE'ye sahip maliyetli disklerin, daha ucuz diskler kadar ek hatalar geliştirmesi olasıdır.
• Çoğu sürücü için yıllık hata oranı ikinci yılda artar
• LSE'lerin sayısı disk boyutuyla birlikte artar
• LSE içeren çoğu diskte 50'den az
• LSE içeren disklerin ek LSE geliştirme olasılığı daha yüksektir
• Önemli miktarda uzamsal ve zamansal yerellik mevcuttur.

Yolsuzlukla ilgili bazı bulgular:

• Disk temizleme yararlıdır (çoğu LSE bu şekilde bulunmuştur) Yolsuzlukla ilgili bazı bulgular:
• Bozulma olasılığı, aynı sürücü sınıfındaki farklı sürücü modellerinde büyük farklılıklar gösterir.
• Yaş etkileri modeller arasında farklıdır
• İş yükü ve disk boyutunun yolsuzluk üzerinde çok az etkisi vardır
• Bozuk disklerin çoğunda yalnızca birkaç bozulma vardır
• Bozulma, bir disk içinde veya RAID'deki diskler arasında bağımsız değildir
• Mekansal yerellik ve bazı zamansal yerellik vardır.
• LSE'ler ile zayıf bir korelasyon vardır.

Bu başarısızlıklar hakkında daha fazla bilgi edinmek için orijinal belgeleri [B+07,B+08] okumalısınız. Ancak umarım ana nokta açık olmalıdır: gerçekten güvenilir bir depolama sistemi oluşturmak istiyorsanız, hem LSE'leri tespit edip bunlardan kurtarma hem de bozulmayı engelleme makinelerini dahil etmelisiniz.

45.2 Gizli Sektör Hatalarını Ele Alma

Bu iki yeni kısmi disk arızası modu göz önüne alındığında, şimdi onlar hakkında ne yapabileceğimizi görmeye çalışmalıyız. İlk önce ikisinin daha kolayını, yani gizli sektör hatalarını ele alalım.

CRUX: GİZLİ SEKTÖR HATALARI NASIL ELE ALINIR

Bir depolama sistemi gizli sektör hatalarını nasıl ele almalıdır? Bu tür bir kısmi arızanın üstesinden gelmek için ne kadar fazladan makineye ihtiyaç vardır?

Görünüşe göre, gizli sektör hatalarının (tanım gereği) kolayca tespit edilebildikleri için ele alınması oldukça basittir. Bir depolama sistemi bir bloğa erişmeye çalıştığında ve disk bir hata döndürdüğünde, depolama sistemi doğru verileri döndürmek için sahip olduğu artıklık mekanizmasını kullanmalıdır. Örneğin, yansıtılmış bir RAID'de, sistem alternatif kopyaya erişmelidir; pariteye dayalı bir RAID-4 veya RAID-5 sisteminde, sistem, parite grubundaki diğer bloklardan bloğu yeniden oluşturmalıdır. Böylece, LSE'ler gibi kolayca tespit edilen problemler, standart fazlalık mekanizmaları yoluyla kolayca düzeltilir.

LSE'lerin artan yaygınlığı, yıllar içinde RAID tasarımlarını etkilemiştir. RAID-4/5 sistemlerinde özellikle ilginç bir sorun, hem tam disk arızaları hem de LSE'ler art arda meydana geldiğinde ortaya çıkar. Spesifik olarak, bir diskin tamamı arızalandığında, RAID, eşlik grubundaki diğer tüm diskleri okuyarak ve eksik değerleri **reconstruct** (yeniden hesaplayarak) diski yeniden oluşturmaya (örneğin, etkin bir yedekte) çalışır. Yeniden oluşturma sırasında diğer disklerden herhangi birinde bir LSE ile karşılaşılırsa, bir sorunumuz vardır: yeniden yapılandırma başarıyla tamamlanamaz.

Bu sorunla mücadele etmek için bazı sistemler fazladan fazlalık derecesi ekler. Örneğin, NetApp'ın RAID-DP'si bir [C+04] yerine iki eşlik diskine eşdeğerdir. Yeniden oluşturma sırasında bir LSE keşfedildiğinde, ekstra eşlik, eksik bloğun yeniden yapılandırılmasına yardımcı olur. Her zaman olduğu gibi, her şerit için iki parite bloğunu korumanın daha maliyetli olması nedeniyle bir maliyeti vardır; ancak, NetApp WAFL dosya sisteminin günlük yapılı yapısı çoğu durumda bu maliyeti azaltır [HLM94]. Kalan maliyet, ikinci eşlik bloğu için fazladan bir disk biçimindeki alandır

45.3 Yolsuzluğu Tespit Etme: Sağlama Toplamı

Şimdi daha zorlu bir sorun olan veri bozulması yoluyla sessiz başarısızlık sorununu çözelim. Bozulma ortaya çıktığında ve disklerin kötü veri döndürmesine yol açtığında, kullanıcıların hatalı veri almasını nasıl önleyebiliriz?

CRUX: BOZULMAYA RAĞMEN VERİ BÜTÜNLÜĞÜ NASIL KORUNUR?

Bu tür arızaların sessiz doğası göz önüne alındığında, bir depolama sistemi yolsuzluk ortaya çıktığında bunu tespit etmek için ne yapabilir? Hangi tekniklere ihtiyaç var? Bunları verimli bir şekilde nasıl uygulayabiliriz?

Gizli sektör hatalarının aksine, yolsuzluğun tespiti önemli bir sorundur. Bir müşteri bir bloğun kötüye gittiğini nasıl anlayabilir? Belirli bir bloğun kötü olduğu bir kez bilindiğinde, kurtarma öncekiyle aynıdır: bloğun başka bir kopyasına sahip olmanız gerekir (ve umarım bozuk değildir!). Bu nedenle, burada tespit tekniklerine odaklanıyoruz.

Modern depolama sistemleri tarafından veri bütünlüğünü korumak için kullanılan birincil mekanizma, sağlama toplamı olarak adlandırılır. Bir **checksum** (sağlama toplamı), girdi olarak bir veri yığınını (diyelim ki 4 KB'lik bir blok) alan ve söz konusu veriler üzerinde bir işlev hesaplayarak veri içeriğinin (örneğin 4 veya 8 bayt) küçük bir özetini üreten bir işlevin sonucudur. Bu özet, sağlama toplamı olarak adlandırılır. Böyle bir hesaplamanın amacı, sağlama toplamını verilerle birlikte depolayarak ve daha sonra verinin mevcut sağlama toplamının orijinal depolama değeriyle eşleştiğini daha sonraki erişimde onaylayarak bir sistemin verilerin bir şekilde bozulup bozulmadığını veya değiştirilip değiştirilmediğini algılamasını

İPUCU: ÜCRETSİZ ÖĞLE YEMEĞİ YOKTUR

Bedava Öğle Yemeği Diye Bir Şey Yoktur veya kısaca TNSTAAFL, görünüşte bedava bir şey alırken aslında bunun için muhtemelen bir miktar bedel ödediğinizi ima eden eski bir Amerikan deyimidir. Yemek yiyenlerin, müşterileri çekmek umuduyla onlara bedava öğle yemeği reklamı yaptıkları eski günlerden geliyor; ancak içeri girdiğinizde, "bedava" öğle yemeğini elde etmek için bir veya daha fazla alkollü içecek satın almanız gerektiğini fark ettiniz. Tabii ki, bu aslında bir sorun olmayabilir, özellikle de alkolik olmaya hevesliyseniz (veya tipik bir lisans öğrencisiyseniz).

Genel Sağlama Toplamı İşlevleri

Sağlama toplamlarını hesaplamak için bir dizi farklı işlev kullanılır ve güç (yani, veri bütünlüğünü korumada ne kadar iyi oldukları) ve hız (yani, ne kadar hızlı hesaplanabilecekleri) bakımından farklılık gösterir. Burada sistemlerde yaygın olan bir değiş tokuş ortaya çıkar: genellikle, ne kadar çok koruma alırsanız, o kadar pahalı olur. ücretsiz öğle yemeği diye bir şey yoktur.

Bazı kullanımların özel veya (XOR) tabanlı basit bir sağlama toplamı işlevi. XOR tabanlı sağlama toplamlarında, sağlama toplamı, sağlama toplamı yapılan veri bloğunun her bir parçasının XOR'lanmasıyla hesaplanır, böylece tüm bloğun XOR'unu temsil eden tek bir değer üretilir.

Bunu daha somut hale getirmek için, 16 baytlık bir blok üzerinde 4 baytlık bir sağlama toplamı hesapladığımızı hayal edin (bu blok, gerçekten bir disk sektörü veya bloğu olamayacak kadar küçüktür, ancak örnek teşkil edecektir). Onaltılı 16 veri baytı şöyle görünür:

```
365e c4cd ba14 8a92 ecef 2c3a 40be f666
```

Onları ikili olarak görüntülersek, aşağıdakileri elde ederiz:

```
0011 0110 0101 1110    1100 0100 1100 1101
1011 1010 0001 0100    1000 1010 1001 0010
1110 1100 1110 1111    0010 1100 0011 1010
0100 0000 1011 1110    1111 0110 0110 0110
```

Verileri satır başına 4 baytlık gruplar halinde sıraladığımız için, ortaya çıkan sağlama toplamının ne olacağını görmek kolaydır: son sağlama toplamı değerini elde etmek için her sütun üzerinde bir XOR gerçekleştirin:

```
0010 0000 0001 1011    1001 0100 0000 0011
```

Sonuç, onaltılık olarak 0x201b9403'tür.

XOR makul bir sağlama toplamıdır ancak sınırlamaları vardır. Örneğin, her sağlama toplamı biriminde aynı konumdaki iki bit değişirse, sağlama toplamı bozulmayı algılamaz. Bu nedenle, insanlar diğer sağlama toplamı fonksiyonlarını araştırmışlardır.

Diğer bir temel sağlama toplamı işlevi toplamadır. Bu yaklaşımın hızlı olma avantajı vardır; hesaplamak, taşmayı göz ardı ederek, verilerin her bir parçası üzerinde 2'ye tümleyen toplama işlemi gerçekleştirmeyi gerektirir. Verilerdeki birçok değişikliği algılayabilir, ancak örneğin veriler kaydırıldığında iyi değildir.

Biraz daha karmaşık bir algoritma, mucit John G. Fletcher [F82] adına (tahmin edebileceğiniz gibi) **Fletcher check- sum (** sağlama toplamı) olarak bilinir. Hesaplaması oldukça basittir ve iki kontrol baytı olan s1 ve s2'nin hesaplanmasını içerir. Spesifik olarak, bir D bloğunun d1 ... dn baytlarından oluştuğunu varsayalım; s1 aşağıdaki gibi tanımlanır: s1 = (s1 + di) mod 255 (tüm di üzerinden hesaplanır); s2 ise: s2 = (s2 + s1) mod 255 (yine tüm di'nin üzerinde) [F04]. Fletcher sağlama toplamı neredeyse CRC kadar güçlüdür (aşağıya bakın), tüm tek bitlik, çift bitlik hataları ve birçok patlama hatasını [F04] tespit eder.

Yaygın olarak kullanılan son bir sağlama toplamı, **cyclic redundancy check** (döngüsel artıklık denetimi)(CRC) olarak bilinir. Bir veri üzerinden sağlama toplamını hesaplamak istediğinizi varsayalım.

blok D. Tek yapmanız gereken, D'yi büyük bir ikili sayıymış gibi ele almak (sonuçta sadece bir bit dizisidir) ve üzerinde anlaşılan bir değere (k) bölmek. Bu bölümün geri kalanı, CRC'nin değeridir. Görünüşe göre, bu ikili modulo işlemi oldukça verimli bir şekilde uygulanabilir ve bu nedenle CRC'nin ağ oluşturmadaki popülaritesi de artar. Daha fazla ayrıntı için başka bir yere bakın [M13]

Kullanılan yöntem ne olursa olsun, mükemmel bir sağlama toplamının olmadığı açık olmalıdır: aynı olmayan içeriklere sahip iki veri bloğunun aynı sağlama toplamlarına sahip olması mümkündür, buna çarpışma denir. Bu gerçek sezgisel olmalıdır: Ne de olsa, bir sağlama toplamı hesaplamak, büyük bir şeyi (örneğin, 4 KB) almak ve çok daha küçük (örneğin, 4 veya 8 bayt) bir özet üretmektir. İyi bir sağlama toplamı işlevi seçerken, hesaplaması kolay kalırken çarpışma olasılığını en aza indiren bir işlev bulmaya çalışıyoruz.

Sağlama Düzeni

Artık bir sağlama toplamını nasıl hesaplayacağınızı biraz anladığınıza göre, şimdi bir depolama sisteminde sağlama toplamlarının nasıl kullanılacağını analiz edelim. Ele almamız gereken ilk soru, sağlama toplamının düzenidir, yani sağlama toplamları diskte nasıl saklanmalıdır?

En temel yaklaşım, her disk sektörü (veya bloğu) ile bir sağlama toplamını basitçe saklar. Bir D veri bloğu verildiğinde, bu verinin sağlama toplamını C(D) olarak adlandıralım. Böylece, sağlama toplamları olmadan disk düzeni şöyle görünür:

D0	D1	D2	D3	D4	D5	D6

Sağlama toplamları ile düzen, her blok için tek bir sağlama toplamı ekler:

Sağlama toplamları genellikle küçük olduğundan (örneğin, 8 bayt) ve diskler yalnızca sektör boyutunda parçalar (512 bayt) veya bunların katları halinde yazabildiğinden, ortaya çıkan sorunlardan biri yukarıdaki düzenin nasıl elde edileceğidir. Sürücü üreticileri tarafından kullanılan bir çözüm, sürücüyü 520 baytlık sektörlerle biçimlendirmektir; sağlama toplamını depolamak için sektör başına fazladan 8 bayt kullanılabilir

Böyle bir işlevselliğe sahip olmayan disklerde, dosya sisteminin sağlama toplamlarını 512 baytlık bloklar halinde depolamanın bir yolunu bulması gerekir. Böyle bir olasılık aşağıdaki gibidir:

Bu şemada, n sağlama toplamları bir sektörde birlikte saklanır, ardından n veri bloğu gelir, ardından sonraki n blok için başka bir sağlama toplamı sektörü gelir ve bu böyle devam eder. Bu yaklaşım, tüm disklerde çalışma avantajına sahiptir, ancak daha az verimli olabilir; örneğin dosya sistemi D1 bloğunun üzerine yazmak istiyorsa, C(D1) içeren sağlama toplamı sektörünü okumalı, içindeki C(D1)'i güncellemeli ve ardından sağlama toplamı sektörünü ve yeni veri bloğu D1'i yazmalıdır (böylece , bir okuma ve iki yazma). Daha önceki yaklaşım (sektör başına bir sağlama toplamı) yalnızca tek bir yazma gerçekleştirir.

45.4 Sağlama toplamlarını kullanma

Bir sağlama toplamı düzenine karar verildikten sonra artık sağlama toplamlarının nasıl kullanılacağını gerçekten anlamaya başlayabiliriz. Bir D bloğunu okurken, müşteri (yani dosya sistemi veya depolama denetleyicisi) sağlama toplamını Cs(D) diskinden de okur, buna **stored checksum (**depolanan sağlama toplamı)(dolayısıyla Cs alt simgesi) adını veririz. İstemci daha sonra, **computed checksum** (hesaplanan sağlama toplamı) Cc(D) olarak adlandırdığımız, alınan D bloğu üzerinden sağlama toplamını hesaplar. Bu noktada müşteri, saklanan ve hesaplanan sağlama toplamlarını karşılaştırır; eşitlerse (yani, Cs(D)
== Cc(D), veriler muhtemelen bozulmamıştır ve bu nedenle kullanıcıya güvenli bir şekilde iade edilebilir. Eşleşmezlerse (yani, Cs(D) != Cc(D)), bu, verilerin saklandıkları zamandan bu yana değiştiği anlamına gelir (çünkü saklanan sağlama toplamı, verilerin o andaki değerini yansıtır). Bu durumda, sağlama toplamımızın tespit etmemize yardımcı olduğu bir yolsuzluğumuz var.
Bir yolsuzluk göz önüne alındığında, doğal soru, bu konuda ne yapmalıyız? Depolama sisteminde fazladan bir kopya varsa, cevap kolaydır: onun yerine onu kullanmayı deneyin. Depolama sisteminde böyle bir kopya yoksa, olası yanıt

bir hata döndürmek için. Her iki durumda da, yolsuzluk tespitinin sihirli bir değnek olmadığının farkına varın; bozulmamış verileri almanın başka bir yolu yoksa, şansınız kalmaz.

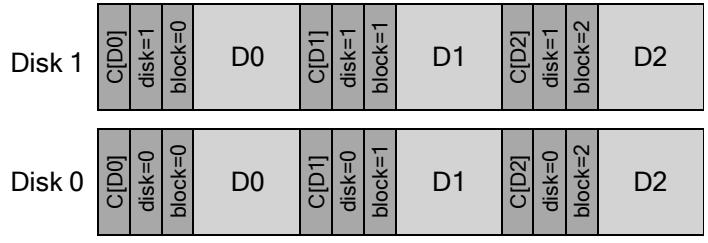
45.5 Yeni Bir Sorun: Yanlış Yönlendirilmiş Yazmalar

Yukarıda açıklanan temel şema, bozuk blokların genel durumunda iyi çalışır. Bununla birlikte, modern disklerde farklı çözümler gerektiren birkaç sıra dışı arıza modu vardır.

İlk başarısızlık modu, **misdirected write** (yanlış yönlendirilmiş yazma) olarak adlandırılır. Bu yanlış konum dışında, verileri diske doğru yazan disk ve RAID denetleyicilerinde ortaya çıkar. Tek diskli bir sistemde bu, diskin Dx bloğunu x'i adreslemek için (istendiği gibi) değil, y'yi adreslemek için yazdığı (böylece Dy'yi "bozduğu") anlamına gelir; ek olarak, bir çok diskli sistemde, denetleyici ayrıca Di,x'i disk i'nin x adresine değil, bunun yerine başka bir disk j'ye yazabilir. Böylece sorumuz:

> Yukarıda açıklanan temel şema, bozuk blokların genel durumunda iyi çalışır. Bununla birlikte, modern disklerde farklı çözümler gerektiren birkaç sıra dışı arıza modu vardır.

Cevap, şaşırtıcı olmayan bir şekilde basit: her bir sağlama toplamına biraz daha fazla bilgi ekleyin. Bu durumda, **physical identifier (**fiziksel bir tanımlayıcı) (fiziksel kimlik) eklemek oldukça faydalıdır. Örneğin, saklanan bilgiler şimdi sağlama toplamı C(D)'yi ve bloğun hem disk hem de sektör numaralarını içeriyorsa, istemcinin belirli bir yerel ayarda doğru bilginin bulunup bulunmadığını belirlemesi kolaydır. Spesifik olarak, müşteri disk 10'daki (D10,4) blok 4'ü okuyorsa, saklanan bilgiler aşağıda gösterildiği gibi o disk numarasını ve sektör ofsetini içermelidir. Bilgiler eşleşmezse, yanlış yönlendirilmiş bir yazma gerçekleşmiş ve artık bir bozulma algılanmıştır. İşte bu eklenen bilgilerin iki diskli bir sistemde nasıl görüneceğine dair bir örnek. Sağlama toplamları genellikle küçük (ör. 8 bayt) ve bloklar çok daha büyük (ör. 4 KB veya daha büyük) olduğundan, bu rakamın kendinden önceki diğerleri gibi ölçekli olmadığına dikkat edin



Disk işletiminden, diskte artık oldukça fazla yedeklilik olduğunu görün: Her blok için, disk numarası her blok içinde tekrarlanır ve söz konusu bloğun ofseti de bloğun bir yanında tutulur. Yine de gereksiz dosyaların varlığı sürpriz olmamalıdır; yedeklilik hata kontrolü (bu durumda) ve kurtarmanın (diğerlerinde) anahtarıdır. Mükemmel disklerle kesinlikle gerekli görmese de, biraz ekstra bilgi, ortaya çıkmaları durumunda rakip konumların tespit edilmesine yardımcı olmak için uzun bir yol kat edebilir.

45.6 One Last Problem: Lost Writes

Ne ki yazı, yanlış yönlendirilmiş yazılar ele alacağımız son sorun değil. Spesifik olarak, bazı modern depolama aygıtlarında kayıp yazma olarak bilinen bir sorun vardır; bu, aygıt üst katmana bir yazmanın tamamlandığını bildirdiğinde ortaya çıkar, ancak aslında hiçbir zaman devam etmez; bu nedenle geriye kalan, güncellenen yeni içerik yerine bloğun eski içeriğidir.

Buradaki bariz soru şudur: Yukarıdan sağlama toplamı stratejilerimizden herhangi biri (örneğin, temel sağlama toplamları veya fiziksel kimlik) kayıp yazmaları tespit etmeye yardımcı olur mu? Ne yazık ki cevap hayır: eski bloğun muhtemelen eşleşen bir sağlama toplamı vardır ve yukarıda kullanılan fiziksel kimlik (disk numarası ve blok ofseti) de doğru olacaktır. Böylece son sorunumuz:

> CRUX: KAYIP YAZILARLA NASIL BAŞA ÇIKILIR Bir depolama sistemi veya disk denetleyicisi kayıp yazmaları nasıl algılamalıdır? Sağlama toplamından hangi ek özellikler gereklidir?

[K+08] öğesine yardımcı olabilecek bir dizi olası çözüm vardır. Klasik bir yaklaşım [BS04], bir yazma doğrulaması veya yazmadan sonra okuma gerçekleştirmektir; Bir sistem, bir yazma işleminden sonra verileri hemen geri okuyarak, verilerin gerçekten disk yüzeyine ulaşmasını sağlayabilir. Bununla birlikte, bu yaklaşım oldukça yavaştır ve bir yazmayı tamamlamak için gereken I/OS sayısını iki katına çıkarır.

Bazı sistemler, kayıp yazmaları algılamak için sistemin başka bir yerine bir sağlama toplamı ekler. Örneğin, Sun'ın Zettabyte Dosya Sistemi (ZFS), her dosya sisteminde bir sağlama toplamı ve bir dosyaya dahil edilen her blok için dolaylı blok içerir. Bu nedenle, bir veri bloğuna yazma kaybolsa bile, inode içindeki sağlama toplamı eski verilerle eşleşmeyecektir. Ancak hem inode'a hem de verilere yazma işlemleri aynı anda kaybolursa, böyle bir şema başarısız olur, bu pek olası olmayan (ama ne yazık ki mümkün!) bir durumdur.

45.7 Fırçalama

Tüm bu tartışma göz önüne alındığında, merak ediyor olabilirsiniz: Bu sağlama toplamları gerçekte ne zaman kontrol ediliyor?

Elbette, uygulamalar tarafından verilere erişildiğinde bir miktar kontrol yapılır, ancak çoğu veriye nadiren erişilir ve bu nedenle kontrol edilmeden kalır. Bit çürümesi sonunda belirli bir veri parçasının tüm kopyalarını etkileyebileceğinden, denetlenmeyen veriler güvenilir bir depolama sistemi için sorunludur.

Bu sorunu çözmek için, birçok sistem çeşitli formların disk fırçalamasını kullanır [K + 08]. Disk sistemi, sistemin her bloğunu periyodik olarak okuyarak ve sağlama toplamlarının hala geçerli olup olmadığını kontrol ederek, belirli bir veri öğesinin tüm kopyalarının kopma olasılığını azaltabilir. Tipik sistemler taramaları gecelik veya haftalık olarak zamanlar.

45.8 Sağlama Genel Giderleri

Kapatmadan önce, şimdi veri koruması için sağlama toplamlarını kullanmanın bazı ek yüklerini tartışıyoruz. Bilgisayar sistemlerinde yaygın olduğu gibi iki farklı ek yük türü vardır: uzay ve zaman.

Uzay tepeleri iki şekilde gelir. Birincisi, diskin (veya başka bir depolama ortamının) kendisindedir; depolanan her sağlama toplamı diskte yer kaplar ve artık kullanıcı verileri için kullanılamaz. Tipik bir oran, disk üzerinde %0,19 ek yük için 4 KB veri bloğu başına 8 baytlık bir sağlama toplamı olabilir.

İkinci tür alan yükü, sistemin belleğinde gelir. Verilere erişirken, artık verilerin yanı sıra sağlama toplamları için de bellekte yer olmalıdır. Bununla birlikte, sistem sağlama toplamını kontrol eder ve daha sonra bir kez atarsa, bu ek yük kısa ömürlüdür ve çok fazla endişe verici değildir. Yalnızca sağlama toplamları bellekte tutulursa (bellek bozulmasına karşı ek bir koruma düzeyi için [Z+13]) bu küçük ek yük gözlemlenebilir.

Alan ek yükleri küçük olsa da, çek toplamanın neden olduğu zaman ek yükleri oldukça belirgin olabilir. En azından, CPU'nun hem veriler depolandığında (depolanan sağlama toplamının değerini belirlemek için) hem de erişildiğinde (sağlama toplamını yeniden hesaplamak ve depolanan sağlama toplamıyla karşılaştırmak için) her blok üzerindeki sağlama toplamını hesaplaması gerekir. Sağlama toplamları (ağ yığınları dahil) kullanan birçok sistem tarafından kullanılan CPU ek yüklerini azaltmanın bir avantajı, veri kopyalama ve sağlama toplamını tek bir kolaylaştırılmış etkinlikte birleştirmektir; kopyaya her şekilde ihtiyaç duyulduğundan (örneğin, verileri çekirdek sayfası önbelleğinden bir kullanıcı arabelleğine kopyalamak için), birleşik kopyalama / sağlama toplamı oldukça etkili olabilir.

CPU ek yüklerinin ötesinde, bazı sağlama toplamı şemaları, özellikle sağlama toplamları verilerden ayrı olarak depolandığında (bu nedenle bunlara erişmek için fazladan I/O gerektirdiğinde) ve arka plan temizliği için gereken ekstra I/O için ekstra I/O ek yüklerine neden olabilir. İlki tasarım gereği azaltılabilir; ikincisi ayarlanabilir ve bu nedenle etkisi, belki de böyle bir fırçalama faaliyeti gerçekleştiğinde kontrol edilerek sınırlandırılabilir. Gecenin yarısı, üretken işçilerin çoğunun (hepsi değil!) yatağa gittiği zaman, bu tür fırçalama faaliyetlerini gerçekleştirmek ve depolama sisteminin sağlamlığını artırmak için iyi bir zaman olabilir.

45.9 Özet

Modern depolama sistemlerinde veri korumasını tartıştık, sağlama toplamı uygulamasına ve kullanımına odaklandık. Farklı sağlama toplamları farklı hata türlerine karşı koruma sağlar; Depolama cihazları geliştikçe, şüphesiz yeni arıza modları ortaya çıkacaktır. Belki de böyle bir değişiklik, yeniden arama topluluğunu ve endüstriyi bu temel yaklaşımlardan bazılarını yeniden gözden geçirmeye veya tamamen yeni yaklaşımlar icat etmeye zorlayacaktır. Zaman söyleyecek ya da söylemeyecek. Zaman bu şekilde komiktir.

References

[B+07] "An Analysis of Latent Sector Errors in Disk Drives" by L. Bairavasundaram, G. Goodson, S. Pasupathy, J. Schindler. SIGMETRICS '07, San Diego, CA. *The first paper to study latent sector errors in detail. The paper also won the Kenneth C. Sevcik Outstanding Student Paper award, named after a brilliant researcher and wonderful guy who passed away too soon. To show the OSTEP authors it was possible to move from the U.S. to Canada, Ken once sang us the Canadian national anthem, standing up in the middle of a restaurant to do so. We chose the U.S., but got this memory.*

[B+08] "An Analysis of Data Corruption in the Storage Stack" by Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '08, San Jose, CA, February 2008. *The first paper to truly study disk corruption in great detail, focusing on how often such corruption occurs over three years for over 1.5 million drives.*

[BS04] "Commercial Fault Tolerance: A Tale of Two Systems" by Wendy Bartlett, Lisa Spainhower. IEEE Transactions on Dependable and Secure Computing, Vol. 1:1, January 2004. *This classic in building fault tolerant systems is an excellent overview of the state of the art from both IBM and Tandem. Another must read for those interested in the area.*

[C+04] "Row-Diagonal Parity for Double Disk Failure Correction" by P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar. FAST '04, San Jose, CA, February 2004. *An early paper on how extra redundancy helps to solve the combined full-disk-failure/partial-disk-failure problem. Also a nice example of how to mix more theoretical work with practical.*

[F04] "Checksums and Error Control" by Peter M. Fenwick. Copy available online here: `http://www.ostep.org/Citations/checksums-03.pdf`. *A great simple tutorial on checksums, available to you for the amazing cost of free.*

[F82] "An Arithmetic Checksum for Serial Transmissions" by John G. Fletcher. IEEE Trans- actions on Communication, Vol. 30:1, January 1982. *Fletcher's original work on his eponymous checksum. He didn't call it the Fletcher checksum, rather he just didn't call it anything; later, othersnamed it after him. So don't blame old Fletch for this seeming act of braggadocio. This anecdote might remind you of Rubik; Rubik never called it "**Rubik's cube**"; rather, he just called it "my cube."*

[HLM94] "File System Design for an NFS File Server Appliance" by Dave Hitz, James Lau, Michael Malcolm. USENIX Spring '94. *The pioneering paper that describes the ideas and product at the heart of NetApp's core. Based on this system, NetApp has grown into a multi-billion dollar storage company. To learn more about NetApp, read Hitz's autobiography "How to Castrate a Bull" (which isthe actual title, no joking). And you thought you could avoid bull castration by going into CS.*

[K+08] "Parity Lost and Parity Regained" by Andrew Krioukov, Lakshmi N. Bairavasun- daram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '08, San Jose, CA, February 2008. *This work explores how different checksum schemes work (or don't work) in protecting data. We reveal a number of interesting flaws incurrent protection strategies.*

[M13] "Cyclic Redundancy Checks" by unknown. Available: `http://www.mathpages.com/home/kmath458.htm`. *A super clear and concise description of CRCs. The internet is full of information, as it turns out.*

[P+05] "IRON File Systems" by V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. Gunawi, A. Arpaci-Dusseau, R. Arpaci-Dusseau. SOSP '05, Brighton, England. *Our paper on how disks have partial failure modes, and a detailed study of how modern file systems react to such failures. As it turns out, rather poorly! We found numerous bugs, design flaws, and other oddities in this work. Some of this has fed back into the Linux community, thus improving file system reliability. You're welcome!*

[RO91] "Design and Implementation of the Log-structured File System" by Mendel Rosenblum and John Ousterhout. SOSP '91, Pacific Grove, CA, October 1991. *So cool we cite it again.*

[S90] "Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial" by Fred B. Schneider. ACM Surveys, Vol. 22, No. 4, December 1990. *How to build fault tolerant services. A must read for those building distributed systems.*

[Z+13] "Zettabyte Reliability with Flexible End-to-end Data Integrity" by Y. Zhang, D. Myers, A. Arpaci-Dusseau, R. Arpaci-Dusseau. MSST '13, Long Beach, California, May 2013. *How to add data protection to the page cache of a system. Out of space, otherwise we would write something...*

Simülasyon Ödevi

Bu ödevde checksum.py dosyası kullanılarak checksumların belirli alanlarıaraştırılacaktır

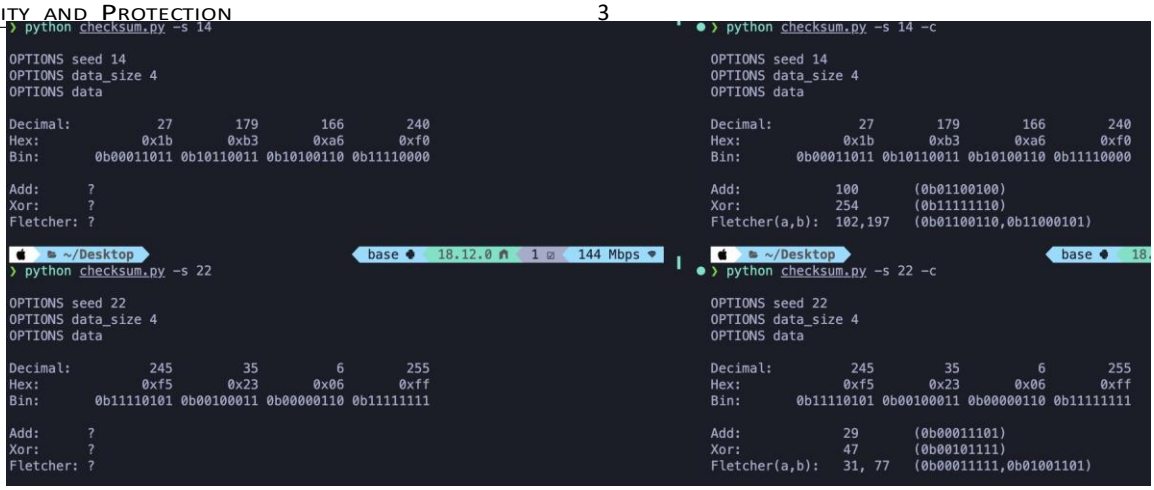
1. Önce cheksum.py dosyasını argüman olmadan çalıştırın. Additive, XOR-Bazlı ve Fletcher teknikleriyle checlsumları hesaplayın. -c bayrağını kullanarak cevaplarınızıkontrol edin.

    a. Burada hiçbir bayrak verilmediği zaman sabit bir veri kullanılıyor seed ile. -cbayrağı checksum değerlerinin hesaplanması için kullanılıyor.

```
❯ python checksum.py

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data

Decimal:          216        194        107         66
Hex:             0xd8       0xc2       0x6b       0x42
Bin:       0b11011000 0b11000010 0b01101011 0b01000010

Add:       ?
Xor:       ?
Fletcher:  ?

 ~/Desktop
❯ python checksum.py -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data

Decimal:          216        194        107         66
Hex:             0xd8       0xc2       0x6b       0x42
Bin:       0b11011000 0b11000010 0b01101011 0b01000010

Add:              71       (0b01000111)
Xor:              51       (0b00110011)
Fletcher(a,b):    73,196   (0b01001001,0b11000100)
```

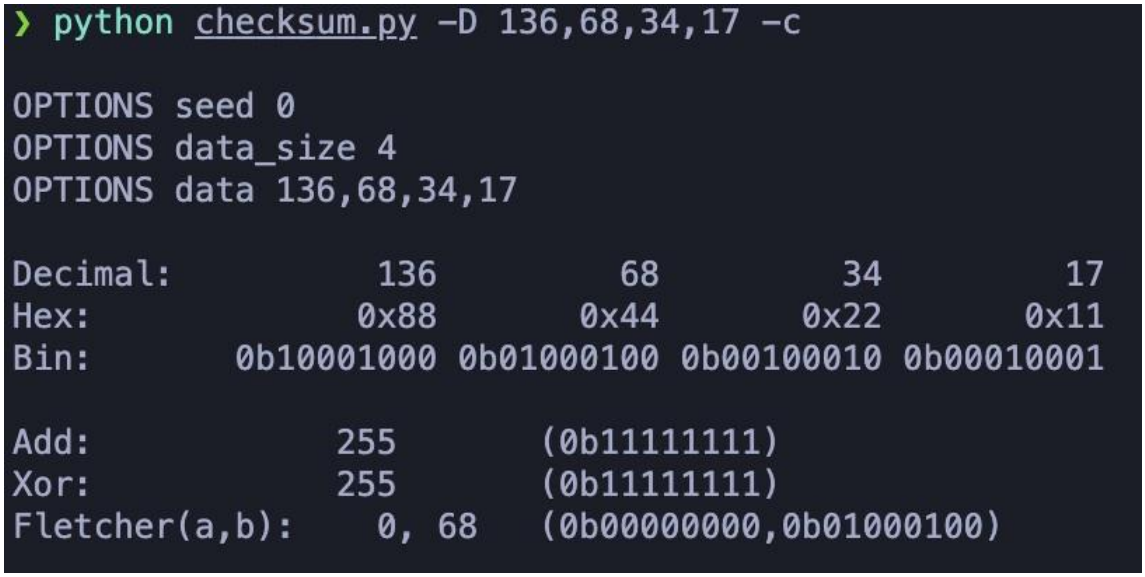
2. Daha sonrasında aynı şeyi tekrar yapın ancak bu sefer -s bayrağını kullanarakseed(rastgele değişken) değerini değiştirin.

   a. -s barağı rastgele değişken oluşturmak için kullanılan bir değeri atamaya yarıyor. Bu bize başka sistemlerde de birebir aynı sonuçları almamız için birdeğişken sağlamış oluyor



3. Bazen toplama ve XOR-bazlı checksumlar aynı checksum'ı üretir (örneğin, veri değerinin tümü sıfırsa). Sadece sıfır içermeyen ve ek ve XOR-bazlı checksum aynı değere sahip olmasına yol açan 4 baytlık bir veri değeri (-D bayrağını kullanarak, örneğin, -D a, b, c, d) iletebilir misiniz? Genel olarak, bu ne zaman meydana gelir? -c bayrağıyla doğru olup olmadığınızı kontrol edin.

   a. XOR değeri sadece 0-1 veya 1-0 değerlerinde meydana gelir. Additive için de değerlerin 1 olması için birisinin 0 diğerinin 1 olması lazım. Toplamda 255 olmaküzere sayıyı bu şekilde 4 parçaya bölebiliriz.



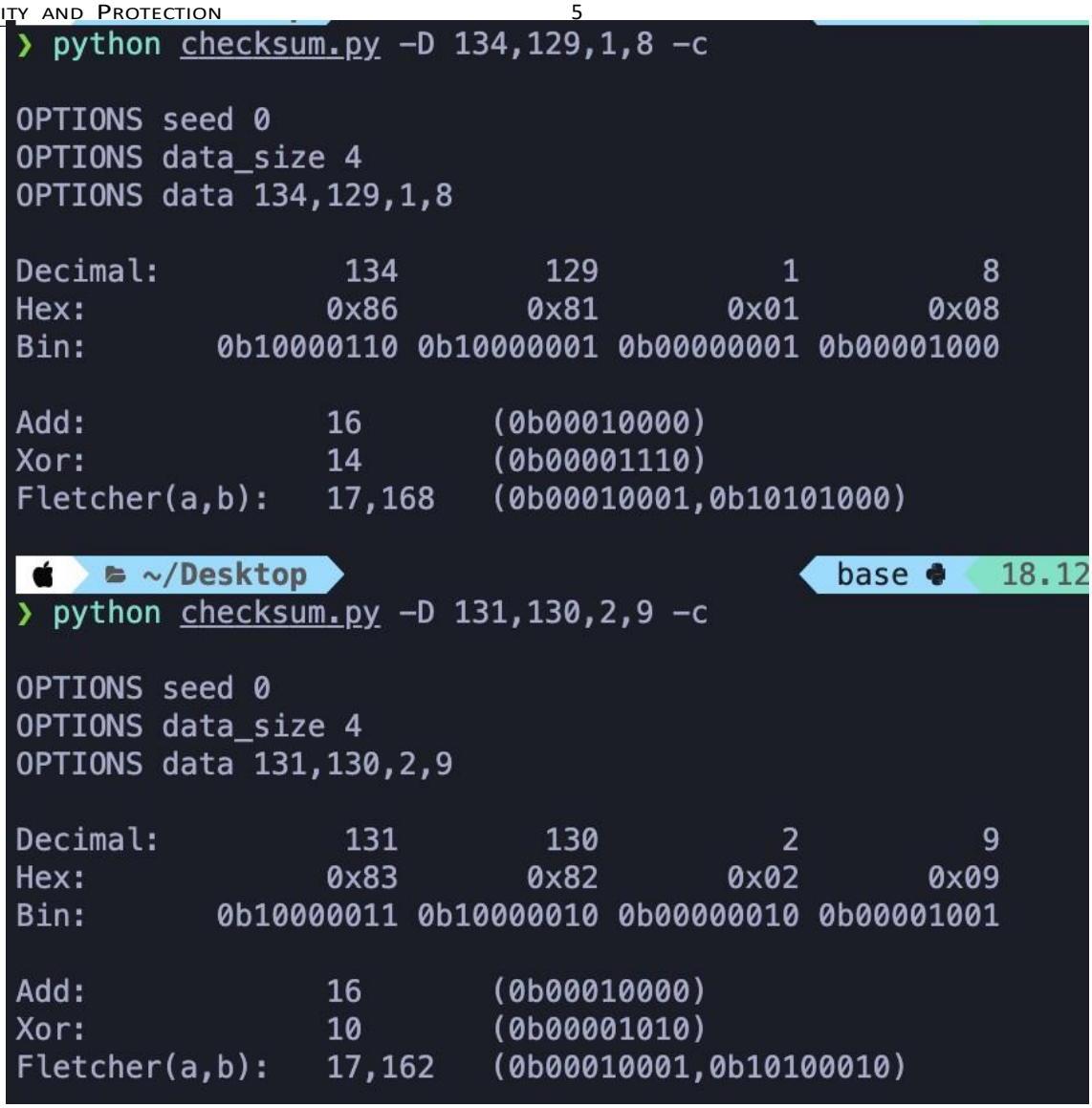
4. Şimdi, additive ve XOR için farklı checksum değerleri üreteceğini bildiğiniz 4 baytlıkbir değer girin. Genel olarak, bu ne zaman olur?

a. Aynı pozisyonlarda 1 değeri olduğu zaman mümkündür.

```
❯ python checksum.py -D 134,129,1,8 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 134,129,1,8

Decimal:          134        129          1          8
Hex:             0x86       0x81       0x01       0x08
Bin:       0b10000110 0b10000001 0b00000001 0b00001000

Add:              16       (0b00010000)
Xor:              14       (0b00001110)
Fletcher(a,b):    17,168   (0b00010001,0b10101000)
```

5. Simülatörü kullanarak iki kez (her bir için farklı bir sayı seti için) checksumları hesaplayın. İki sayı dizisi farklı olmalıdır (örneğin, ilk kez -D a1, b1, c1, d1 ve ikinci kez -D a2, b2, c2, d2 ancak aynı ekleyici checksum üretmelidir. Genel olarak, farklı veri değerlerine rağmen aynı ekleyici checksum'a ne zaman erişilir? Cevaplarınızı -c bayrağı ile kontrol edin.

   a. Verilen data setindeki sayıların binary sonuçlarında aynı sıralarda 1 olmadığısürece ekleyici checksum değeri aynı olacaktır.



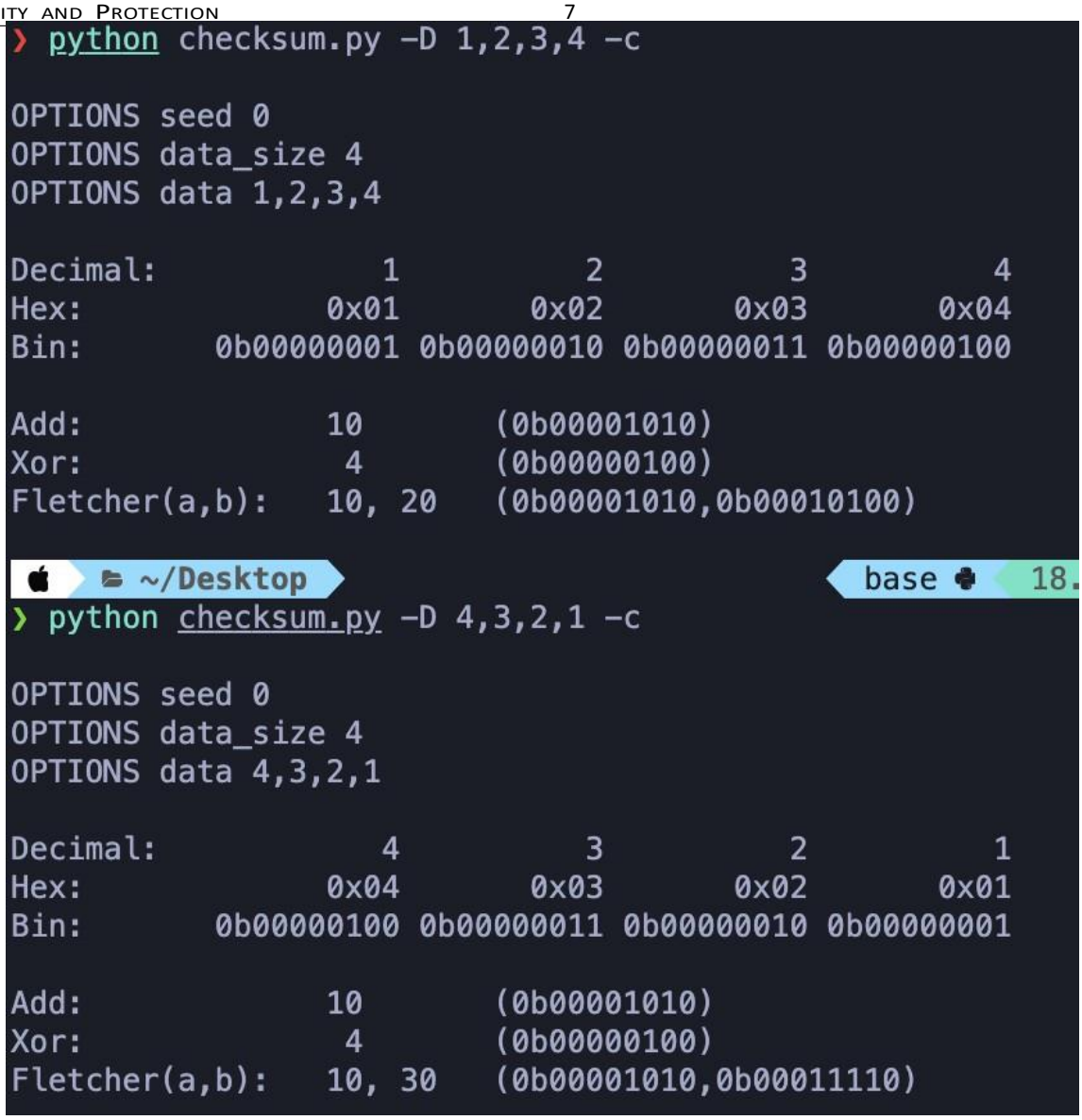
6. Şimdi aynısını XOR checksum için yapın.

   a. XOR için de benzer mantıkla gidilebilir. Burada önemli olan konu, verilen sayıların binary değerlerinin xorlanmış hallerini aynı oluşturmak.

```
❯ python checksum.py -D 88,145,37,168 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 88,145,37,168

Decimal:           88        145         37        168
Hex:             0x58       0x91       0x25       0xa8
Bin:       0b01011000 0b10010001 0b00100101 0b10101000

Add:              182       (0b10110110)
Xor:               68       (0b01000100)
Fletcher(a,b):  183,   9    (0b10110111,0b00001001)

 ~/Desktop                                    base  18.12.0
❯ python checksum.py -D 216,149,33,40 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 216,149,33,40

Decimal:          216        149         33         40
Hex:             0xd8       0x95       0x21       0x28
Bin:       0b11011000 0b10010101 0b00100001 0b00101000

Add:              182       (0b10110110)
Xor:               68       (0b01000100)
Fletcher(a,b):  183,142    (0b10110111,0b10001110)
```

7. Sırada belirli bir veri değerleri kümesine bakalım. İlk veri kümemiz: - D
   1, 2, 3, 4. Bu veri kümesi için checksumlar (ekleyici, XOR, Fletcher) ne olacak?
   Daha sonrasında bu sonucu -D 4, 3, 2, 1 veri seti üzerinde bu checksumları kıyasla.Bu üç checksum hakkında farklı olanlar nedir? Fletcher diğer ikisiyle nasıl kıyaslanabilir? Fletcher genel olarak neden daha basit bir checksum metodu olan ekleyiciden "daha iyidir"?

   a. Diğer checksum hesaplamalarına kıyasla fletcher tekniğinde sayıların sıralaması önemlidir.



8. Hiçbir checksum mükemmel değildir. Seçtiğiniz bir veri listesine göre, aynı Fletcherchecksumuna yol açan diğer veri değerlerini bulabilir misiniz? Genel olarak bu ne zaman gerçekleşir? Basit bir veri dizisiyle başlayın (örn. -D 0,1, 2, 3) ve bu sayılardan birini değiştirerek aynı Fletcher checksumuna ulaşmayı deneyin. Her zamanki gibi, cevaplarınızı -c ile kontrol edin.

   a. Üst sınır 255 olduğu için, 3 yerine 258 kullanılabilir. bu aslında bir overflow çözümüdür.

```
❯ python checksum.py -D 0,1,2,3 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 0,1,2,3

Decimal:            0          1          2          3
Hex:             0x00       0x01       0x02       0x03
Bin:       0b00000000 0b00000001 0b00000010 0b00000011

Add:                6       (0b00000110)
Xor:                0       (0b00000000)
Fletcher(a,b):      6, 10   (0b00000110,0b00001010)

 ~/Desktop                                        base  18.
❯ python checksum.py -D 0,1,2,258 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 0,1,2,258

Decimal:            0          1          2        258
Hex:             0x00       0x01       0x02       0x102
Bin:       0b00000000 0b00000001 0b00000010 0b100000010

Add:                5       (0b00000101)
Xor:              257       (0b100000001)
Fletcher(a,b):      6, 10   (0b00000110,0b00001010)
```

Ev ödevi (kod)

Ödevin bu bölümünde, çeşitli sağlama toplamlarını uygulamak için kendi kodunuzun bir kısmını yazacaksınız.

Sorular

1. Bir giriş dosyası üzerinden XOR tabanlı bir sağlama toplamını hesaplayan ve sağlama toplamını çıktı olarak yazdıran kısa bir C programı (check-xor.c adı verilir) yazın. (Bir bayt) sağlama toplamını saklamak için 8 bitlik işaretsiz bir karakter kullanın. Beklendiği gibi çalışıp çalışmadığını görmek için bazı test dosyaları oluşturun.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
 if (argc != 2) {
  printf("Kullanım: %s <dosya_adi>\n", argv[0]);
  return 1;
 }

 FILE *input_file = fopen(argv[1], "r");
 if (input_file == NULL) {
  perror("Dosya açılamadı");
  return 1;
 }

 // Sağlama toplamını saklamak için 8 bitlik işaretsiz bir karakter
 unsigned char checksum = 0;

 // Dosyayı okumak için bir bayt oku
 int c;
 while ((c = fgetc(input_file)) != EOF) {
  checksum ^= c;
 }

 // Sağlama toplamını yazdır
 printf("Sağlama toplamı: %d\n", checksum);

 fclose(input_file);
 return 0;
```

   Bu program, komut satırında dosya adını verilen bir giriş dosyasındaki her

bir bayt için XOR tabanlı bir sağlama toplamı hesaplar. Bu sağlama toplamı, dosyanın her bir baytını tek tek XOR'layarak hesaplanır. Sağlama toplamını 8 bitlik işaretsiz bir karakter olarak saklar ve sonrasında sağlama toplamını çıktı olarak yazdırır. Program sona erer.

Program, dosya açılamadığında veya kullanım hatası olduğunda uygun bir hata mesajı yazdırır ve 1 değerini döndürür. Aksi takdirde, program 0 değerini döndürür.

2. Şimdi bir giriş dosyası üzerinden Fletcher sağlama toplamını hesaplayan kısa bir C programı (check-fletcher.c adı verilir) yazın. Çalışıp çalışmadığını görmek için programınızı bir kez daha test edin

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Kullanım: %s <dosya_adi>\n", argv[0]);
    return 1;
  }

  FILE *input_file = fopen(argv[1], "r");
  if (input_file == NULL) {
    perror("Dosya açılamadı");
    return 1;
  }

  // Sağlama toplamını saklamak için iki 16 bitlik işaretsiz sayı
  unsigned short checksum_low = 0;
  unsigned short checksum_high = 0;

  // Dosyayı okumak için bir bayt oku
  int c;
  while ((c = fgetc(input_file)) != EOF) {
    checksum_low += c;
    checksum_high += checksum_low;
  }

  // Sağlama toplamını yazdır
  printf("Sağlama toplamı: %d\n", (checksum_high << 16) | checksum_low);

  fclose(input_file);
  return 0;
}
```

Bu program, komut satırında dosya adını verilen bir giriş dosyasındaki her bir bayt için Fletcher sağlama toplamını hesaplar. Bu sağlama toplamı, dosyanın her bir baytını 16 bitlik işaretsiz sayılar olarak iki tane değişkende tutularak hesaplanır. İlk değişken, her bir bayt için toplamı tutar ve ikinci değişken, ilk değişkende tutulan toplamın toplamını tutar. Sağlama toplamı, ikinci değişkende tutulan toplamın üst 16 biti ile ilk değişkende tutulan toplamın alt 16 bitinin birleşiminden oluşur. Sağlama toplamını çıktı olarak yazdırır ve program sona erer.

Program, dosya açılamadığında veya kullanım hatası olduğunda uygun bir hata mesajı yazdırır ve 1 değerini döndürür. Aksi takdirde, program 0 değerini döndürür.

3. Şimdi her ikisinin performansını karşılaştırın: biri diğerinden daha mı hızlı? Girdi dosyasının boyutu değiştikçe performans nasıl değişir? Programlara zaman ayırmak için dahili aramaları kullanın. Performansa önem veriyorsanız hangisini kullanmalısınız? Yeteneği kontrol etme hakkında?

   Girdi dosyasının boyutu, programın performansını doğrudan etkileyebilir. Genel olarak, daha büyük girdi dosyaları için daha uzun süre gerekebilir ve programın daha fazla bellek ve işlem gücüne ihtiyacı olabilir. Dahili arama fonksiyonları, programın performansını artırmak için kullanılabilir. Dahili arama fonksiyonları, programın bellekte bulunan verileri daha hızlı bir şekilde tarayarak, arama süresini azaltabilir. Ancak, dahili arama fonksiyonlarının kullanımı programın kodunu daha karmaşık hale getirebilir, bu nedenle performansa önem veriyorsanız bu fonksiyonların kullanımını dikkatli bir şekilde değerlendirmeniz gerekebilir. Ayrıca, programın yeteneğini kontrol etmek için belirli testler yapılması ve bu testlerin sonuçlarına göre programın yeteneğini geliştirmek gerekebilir.

4. 16-bit CRC hakkında okuyun ve ardından uygulayın.
   Çalıştığından emin olmak için birkaç farklı giriş üzerinde test edin. Basit XOR ve Fletcher ile karşılaştırıldığında performansı nasıl? Kontrol etme kabiliyetine ne dersiniz?

   16-bit CRC (Cyclic Redundancy Check), verileri hataları tespit etmek ve düzeltmek için kullanılan bir yöntemdir. Bu yöntem, verileri kontrol ederken birkaç farklı matematiksel işlemi kullanır. Örneğin, bir XOR (Exclusive OR) işlemi veya Fletcher algoritması gibi.
   16-bit CRC, veriyi belirli bir şekilde parçalara ayırır ve bunları bir CRC kontrol toplamına ekler. Bu kontrol toplamı, verinin doğruluğunu kontrol etmek için kullanılır. Eğer veride bir hata bulunmuşsa, kontrol toplamı doğru değerlerle eşleşmeyecektir ve bu hatayı tespit edebiliriz.
   Basit XOR ve Fletcher ile karşılaştırıldığında, 16-bit CRC'nin performansı biraz daha iyi olabilir. Ancak, bu konuda net bir yargıda bulunmak için daha fazla araştırma yapmak gerekir.
   16-bit CRC'nin kontrol etme kabiliyetine gelince, bu yöntem genellikle verilerin doğruluğunu kontrol etmek için kullanılır. Ancak, hataları

düzeltme kabiliyetine sahip değildir. Yani, eğer veride bir hata bulunmuşsa, bu hatayı düzeltmek için başka bir yöntem kullanmak gerekir.

5. Şimdi bir dosyanın her 4 KB bloğu için tek baytlık bir sağlama toplamı hesaplayan ve sonuçları bir çıktı dosyasına (komut satırında belirtilen) kaydeden bir araç oluşturun. Bir dosyayı okuyan, her bloğun sağlama toplamlarını hesaplayan ve sonuçları başka bir dosyada saklanan sağlama toplamlarıyla karşılaştıran ilgili bir araç oluşturun. Bir sorun varsa program dosyanın bozuk olduğunu yazdırmalıdır. Dosyayı el ile bozarak programı test edin