

Veri Bütünlüğü ve Koruma

Şu ana kadar incelediğimiz dosya sistemlerinde bulunan temel ilerlemelerin ötesinde, bazı özellikler 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 sağlamalıdır?

Bu genel alan veri bütünlüğü(**data integrity**)veya veri koruma(**data protection**) olarak adlandırılır. Bu nedenle, sisteminize yerleştirdiğiniz verilerin sistem size geri döndüğünde aynı olduğundan emin olmak için kullanılan teknikleri inceleyeceğiz.

KRİTİK NOKTA: VERİ BÜTÜNLÜĞÜ NASIL SAĞLANIR

Sistemlerin verilerin depolanması sırasında korunması nasıl sağlanmalıdır? Gerekli teknikler nelerdir? Bu teknikler ne kadar verimli olabilir ve alan ve zaman açısından ne kadar maliyetli olabilir?

45.1 Disk Hatası Modları

RAID bölümünde öğrendiğiniz gibi, diskler mükemmel değildir ve nadiren çalışmaz. İlk RAID sistemlerinde, hata modeli oldukça basitti: ya disk tamamen çalışıyor ya da tamamen çalışmıyor ve böyle bir hata algılanması basittir. Bu disk hatasının fail-stop modeli, RAID'ın inşasını biraz daha kolaylaştırır.

Modern disklerin sergilediği diğer tüm arıza modları hakkında öğrenmediğiniz şey. Spesifik olarak, Bairavasundaram ve ark. ayrıntılı olarak incelendiğinde [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

	Ucuz	Maliyetli
LSEs	9.40%	1.40%
Bozulmalar	0.50%	0.05%

Figür 45.1: LSE'lerin Sıklığı ve Blok Ydsuzluğu

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) yüzeye dokunursa, yüzeye zarar verebilir ve bitleri okunamaz hale getirebilir. Kozmik ışınlar da yanlış içeriklere yol açarak bitleri çevirebilir. Neyse ki, disk içi hata düzeltme kodları(**error correcting codes**) (ECC), sürücü tarafından bir bloktaki disk üzerindeki bitlerin iyi olup olmadığını belirlemek ve bazı durumlarda bunları düzeltmek için kullanılır; iyi değilse ve sürücüde hatayı düzeltmek için yeterli bilgi yoksa, bunları okumak için bir istek gönderildiğinde disk bir hata döndürür.

Bir disk bloğunun, diskin kendisi tarafından algılanamayacak şekilde bozulduğu durumlar da vardır. Örneğin, buggy disk üretici yazılımı bir bloğu yanlış konuma yazabilir; böyle bir durumda, disk ECC'si blok içeriğinin iyi olduğunu belirtir, ancak müşterinin bakış açısına göre, daha sonra erişildiğinde yanlış blok döndürülür. Benzer şekilde, bir blok hatalı bir veri yolu üzerinden ana bilgisayardan diske aktarıldığında bozulabilir; ortaya çıkan bozuk veriler diskte saklanır, ancak müşterinin istediği bu değildir. Bu tür arızalar özellikle sinsidir çünkü sessiz arızalardır; 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 (yazıldığı halde okunan bir kısmi hata) döndürebilir ve arada bir sadece yanlış verileri döndürebilir. (yazıldığı halde okunmayan 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 milyondan fazla disk sürücüsü) en az bir LSE veya blok bozulması sergileyen sürücülerin yüzdesini göstermektedir. Şekil, sonuçları "ucuz" sürücüler (genellikle SATA sürücüler) ve "pahalı" sürücüler (genellikle SCSI veya Fiber Kanal) olarak alt gruplara ayırır. Gördüğümüz 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 oluşturması muhtemeldir.
- 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.
- Disk temizleme yararlıdır (çoğu LSE bu şekilde bulunur)

Bozulma ile 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 düştürmek istiyorsanız hem LSE'leri tespit edip hem de bozulmayı engellemek için makineler eklemelisiniz.

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 olarak, ikisi arasında daha kolay olanı, yani gizli sektör hatalarını ele alalım.

KRİTİK NOKTA: VERİ BÜTÜNLÜĞÜ NASIL SAĞLANIR

*Bir depolama sistemi gizli sektör hatalarını nasıl ele almalıdır? Ne kadar
Bu tür bir kısmi arızanın üstesinden gelmek için fazladan makineye ihtiyaç var mı?*

Gizli sektör hatalarının (tanım gereği) kolaylıkla tespit edilebildiği 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 duyarak ve eksik değerleri 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 olduğ gibi, her şerit için iki parite bloğunu korumanın daha maliyetli olması nedeniyle bir maliyeti vardır; ancak, NetApp **WAFL** dosya sisteminin günlük yapıllı yapıya geçtiği durumda bu maliyeti azaltır [HLM94]. Kılan maliyet, ikinci eşlik bloğu için fazladan bir disk biçimindeki alandır.

45.3 Bozulmayı 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?

KRİTİK NOKTA: BOZULMAYA RAĞMEN VERİ BÜTÜNLÜĞÜ NASIL KORUNUR?

Bu tür arızaların sessiz doğası göz önüne alındığında, bir depolama sistemi ne yapabilir? Yolsuzluğun ne zaman ortaya çıktığını tespit etmek için? Hangi tekniklere ihtiyaç var? Nasıl verimli bir şekilde uygulanabilir?

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

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

İPUCU: ÜCRETSİZ ÖĞLE YEMEĞİ YOKTUR

Bedava Öğle Yemeği Ya da kısaca TNSTAAFL Diye Bir Şey Yoktur
 görüntüğe göre elde ettiğinizi ima eden eski bir Amerikan deyimini ücretsiz bir şey, gerçekte muhtemelen bunun için bir miktar maliyet ödüyorsunuzBT. Yemek yiyenlerin bedava öğle yemeği reklamı yaptığı eski günlerden geliyor.onları kendine çekmeyi uman müşteriler için; sadece içeri girdiğinde, yaptın mı"ücretsiz" öğle yemeğini elde etmek için bir veya daha fazla satın almanız gerektiğini fark edin.alkollü içecekler. Tabii ki, bu aslında bir sorun olmayabilir, özellikle alkolik olmaya hevesliyseniz (veya tipik bir lisans öğrencisiyseniz).

Genel Sağlama Toplamı İşlevleri

Sağlama toplamalarını 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. Sistemlerde yaygın olan bir takas burada 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 toplamalarında, 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 üzerinden 4 baytlık bir sağlama toplamı hesapladığımızı hayal edin (bu blok, gerçekten bir disk sektörü veya bloğu olamayacak kadar küçük, ancak örnek teşkil edecek). 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: nihai 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 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 şu şekilde tanımlanır: $s1 = (s1 + d1) \text{ mod } 255$ (di hepsi üzerinden hesaplanmıştır); s2 ise: $s2 = (s2 + s1) \text{ mod } 255$ (yine tüm di'nin üzerinde) [F04]. Fletcher sağlama toplamı neredeyse CRC kadar güçlüdür (aşağıya bakın), tü mtek bitlik, çift bitlik hatalar ve birçok patlama hatası [F04].

Yaygın olarak kullanılan son bir sağlama toplamı, döngüsel artıklık denetimi (CRC) olarak bilinir. Bir veri bloğu D üzerinden sağlama toplamını hesaplamak istediğinizi varsayalım. Tek yapmanız gereken, D'yi büyük bir ikili sayıymış gibi ele almak (sonuçta yalnızca bir bit dizisidir) ve üzerinde anlaşılan bir değere (k) bölmek. Bu bölümün geri kalanı, CRC'nin değeridir. Görünen o ki, 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 toplamı olmadığı açık olmalıdır: aynı olmayan içeriklere sahip iki veri bloğunun aynı sağlama toplamalarına sahip olması mümkündür, buna çarpışma denir. Bu gerçek sezgisel olmalıdır: Sonuçta, bir sağlama toplamı hesaplamak, büyük bir şeyi (örneğin, 4 KB) alıyor ve çok daha küçük (örneğin, 4 veya 8 bayt) bir özet üretiyor. İyi bir sağlama toplamı işlevi seçerken, hesaplaması kolay kalırken çarpışma olasılığını en aza indiren bir işlev bulmaya çalışıyoruz.

Sağlama Düzeni(Checksum Layout)

Artık bir sağlama toplamını nasıl hesaplayacağınızı biraz anladığınıza göre, şimdi bir depolama sisteminde sağlama toplamalarının nasıl kullanılacağını analiz edelim. Ele almamız gereken ilk soru, sağlama toplamının düzenidir, yani sağlama toplamaları 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 adlandırılır. Böylece, sağlama toplamaları olmadan disk düzeni şöyle görünür:

D0	D1	D2	D3	D4	D5	D6
----	----	----	----	----	----	----

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

C[D0]	D0	C[D1]	D1	C[D2]	D2	C[D3]	D3	C[D4]	D4
-------	----	-------	----	-------	----	-------	----	-------	----

Sağlama toplamaları 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 sistemi sağlama toplamalarını 512 baytlık bloklar halinde depolamanın bir yolunu bulmalıdır. Böyle bir olasılık aşağıdaki gibidir:

C[D0]	C[D1]	C[D2]	C[D3]	C[D4]	D0	D1	D2	D3	D4
-------	-------	-------	-------	-------	----	----	----	----	----

Bu şemada, n sağlama toplamaları 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 toplamalarını kullanma

Bir sağlama toplamı düzenine karar verildikten sonra artık sağlama toplamalarının nasıl kullanılacağını gerçekten anlamaya başlayabiliriz. Bir D bloğunu okurken, istemci (yani dosya sistemi veya depolama denetleyicisi) sağlama toplamını da disk Cs(D)'den okur, buna depolanan sağlama toplamı adını veririz (dolayısıyla Cs alt simgesi). İstemci daha sonra, hesaplanan sağlama toplamı Cc(D) olarak adlandırdığımız, alınan D bloğu üzerinden sağlama toplamını hesaplar. Bu noktada müşteri, saklanan ve hesaplanan sağlama toplamalarını karşılaştırır; eşitlerse (yani, $Cs(D) == Cc(D)$), veriler muhtemelen bozulmamıştır ve bu nedenle kullanıcıya güvenli bir şekilde iade edilebilir. Eşleşmezlerse (yani, $Cs(D) \neq Cc(D)$), bu verilerin depolandığı andan itibaren değiştiğini gösterir (çünkü depolanan sağlama toplamı, verilerin o andaki değerini yansıtır). Bu durumda, sağlama toplamamızın tespit etmemize yardımcı olduğu bir bozulmamızdır.

Bir bozulma 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

muhtemelen bir hata döndürmek için cevap verir. Her iki durumda da, yolsuzluk tespitinin sihirli bir değnek olmadığını 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 bulunur.

İlk başarısızlık modu, yanlış yönlendirilmiş yazma olarak adlandırılır. Bu, verileri yanlış konum dışında 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:

KRİTİK NOKTA: YANLIŞ YÖNLENDİRİLMİŞ YAZIMLAR NASIL ELE ALINIR
Bir depolama sistemi veya disk denetleyicisi yanlış yönlendirilmiş yazmaları nasıl algılamalıdır? Sağlama toplamından hangi ek özellikler gerekir?

Cevap, şaşırtıcı olmayan bir şekilde basit: her bir sağlama toplamına biraz daha fazla bilgi ekleyin. Bu durumda, bir fiziksel 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 bu 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 1	C[D0]	disk=1	block=0	D0	C[D1]	disk=1	block=1	D1	C[D2]	disk=1	block=2	D2
Disk 0	C[D0]	disk=0	block=0	D0	C[D1]	disk=0	block=1	D1	C[D2]	disk=0	block=2	D2

Disk üstü biçiminden, diskte artık makul miktarda fazlalık olduğunu görebilirsiniz: her blok için, disk numarası her blokta tekrarlanır ve söz konusu bloğun ofseti de bloğun yanında tutulur. . Gereksiz bilgilerin varlığı yine de şaşırtıcı olmamalıdır; fazlalık, hata saptamanın (bu durumda) ve düzeltmenin (diğerlerinde) anahtarıdır. Kusursuz disklerde kesinlikle gerekli olmasa da, biraz fazladan bilgi, ortaya çıkmaları durumunda sorunlu durumların tespit edilmesine yardımcı olmada uzun bir yol kat edebilir.

45.6 Son Bir Sorun: Kayıp Yazmalar

Ne yazık ki, yanlış yönlendirilmiş yazmalar 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 bu 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 toplamaları 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:

KRİTİK NOKTA: KAYIP YAZILAR NASIL İŞLENİR

Bir depolama sistemi veya disk denetleyicisi kayıp yazmaları nasıl algılamalıdır? Sağlama toplamından hangi ek özellikler gerekir?

[K+08]'e yardımcı olabilecek bir dizi olası çözüm var. 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. Ancak bu yaklaşım oldukça yavaştır ve bir yazmayı tamamlamak için gereken G/Ç sayısını iki katına çıkarır.

Bazı sistemler, kayıpyazmaları algılamak için sistemin başka bir yerine bir sağlama toplamı ekler. Örneğin, Sun'ın **Zettabyte Dosya Sistemi** (ZFS), her dosya sistemi inode'unda bir sağlama toplamı ve bir dosyaya dahil olan 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 Scrubbing

Tüm bu tartışma göz önüne alındığında, merak ediyor olabilirsiniz: Bu sağlama toplamaları gerçekte ne zaman kontrol ediliyor? Elbette, uygulamalar tarafından verilere erişildiğinde bir miktar kontrol gerçekleşir.

ancak çoğu veriye nadiren erişilir ve bu nedenle denetlenmeden kalır. Kontrol edilmeyen veriler, güvenilir bir depolama sistemi için sorunludur, çünkü bit çürümesi sonunda belirli bir veri parçasının tüm kopyalarını etkileyebilir.

Bu sorunu çözmek için, birçok sistem çeşitli biçimlerde [K+08] disk temizlemeyi kullanır. Disk sistemi, sistemin her bloğunu periyodik olarak okuyarak ve sağlama toplamalarının hala geçerli olup olmadığını kontrol ederek, belirli bir veri ögesinin tüm kopyalarının bozulma olasılığını azaltabilir. Tipik sistemler, taramaları gecelik veya haftalık olarak planlar.

45.8 Sağlama Toplama Genel Giderleri

Bitirmeden önce, veri koruması için sağlama toplamalarını kullanmanın bazı genel giderlerini tartışacağız. Bilgisayar sistemlerinde yaygın olduğu gibi, iki farklı genel gider türü vardır: uzay ve zaman.

Alan giderleri iki biçimde gelir. İlki diskin (veya başka bir depolama ortamının) kendisinde; saklanan her sağlama toplamı, diskte artık kullanıcı verileri için kullanılmayan yer kaplar. Tipik bir oran, %0,19 disk alanı ek yükü için 4 KB veri bloğu başına 8 baytlık bir sağlama toplamı olabilir.

İkinci tip alan yükü, sistemin belleğinde gelir. Verilere erişirken, artık bellekte verilerin kendisi kadar sağlama toplamaları için de yer olmalıdır. Ancak, sistem yalnızca sağlama toplamını kontrol eder ve bir kez yaptıktan sonra atarsa, bu ek yük kısa ömürlüdür ve çok da endişe verici değildir. Yalnızca sağlama toplamaları bellekte tutulursa (bellek bozulmasına karşı ek bir koruma düzeyi için [Z+13]), bu küçük ek yük gözlemlenebilir olacaktır.

Alan genel giderleri küçük olsa da, sağlama toplamının neden olduğu zaman ek yükleri oldukça belirgin olabilir. En azından CPU, hem veri depolandığında (depolanan sağlama toplamının değerini belirlemek için) hem de veriye erişildiğinde (sağlama toplamını yeniden hesaplamak ve depolanan sağlama toplamıyla karşılaştırmak için) her blok üzerinden sağlama toplamını hesaplamalıdır. Sağlama toplamaları (ağ yığınları dahil) kullanan birçok sistem tarafından kullanılan, CPU ek yüklerini azaltmaya yönelik bir yaklaşım, veri kopyalama ve sağlama toplamını tek bir kolaylaştırılmış etkinlikte birleştirmektir; kopya her şekilde gerekli olduğundan (örneğin, verileri çekirdek sayfası ön belleğinden bir kullanıcı arabelleğine kopyalamak için), birleştirilmiş kopyalama/sağlama toplamı oldukça etkili olabilir.

CPU genel giderlerinin ötesinde, bazı sağlama toplamı şemaları, özellikle sağlama toplamaları verilerden farklı bir şekilde depolandığında (dolayısıyla bunlara erişmek için fazladan G/Ç'ler gerektiğinde) ve arka plan temizleme için gereken herhangi bir ekstra G/Ç için fazladan G/Ç ek yüklerine neden olabilir. İlki tasarım gereği azaltılabilir; ikincisi ayarlanabilir ve böylece etkisi, belki de bu tür temizleme faaliyetinin ne zaman gerçekleştiğini kontrol ederek sınırlandırılabilir. Çoğu (tümü değil!) üretken işinin yattığı gecenin ortası, bu tür temizleme faaliyetini gerçekleştirmek ve depolama sisteminin sağlamlığını artırmak için iyi bir zaman olabilir.

45.9 Özet

Sağlama toplamı uygulamasına ve kullanımına odaklanarak modern depolama sistemlerinde veri korumayı tartıştık. Farklı sağlama toplamaları, farklı hata türlerine karşı koruma sağlar; depolama cihazları geliştikçe, şüphesiz yeni arıza modları ortaya çıkacaktır. Belki de böyle bir değişiklik, araştırma topluluğu ve endüstrisini bu temel yaklaşımlardan bazılarını yeniden gözden geçirmeye veya tamamen yeni yaklaşımlar icat etmeye zorlayacaktır. Zaman gösterecek. Ya da olmayacak. Zaman bu şekilde komik.

Kaynakça

- [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. Gracanac, 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 Transactions 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, others named 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 is the 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. Bairavasundaram, 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 in current 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...*

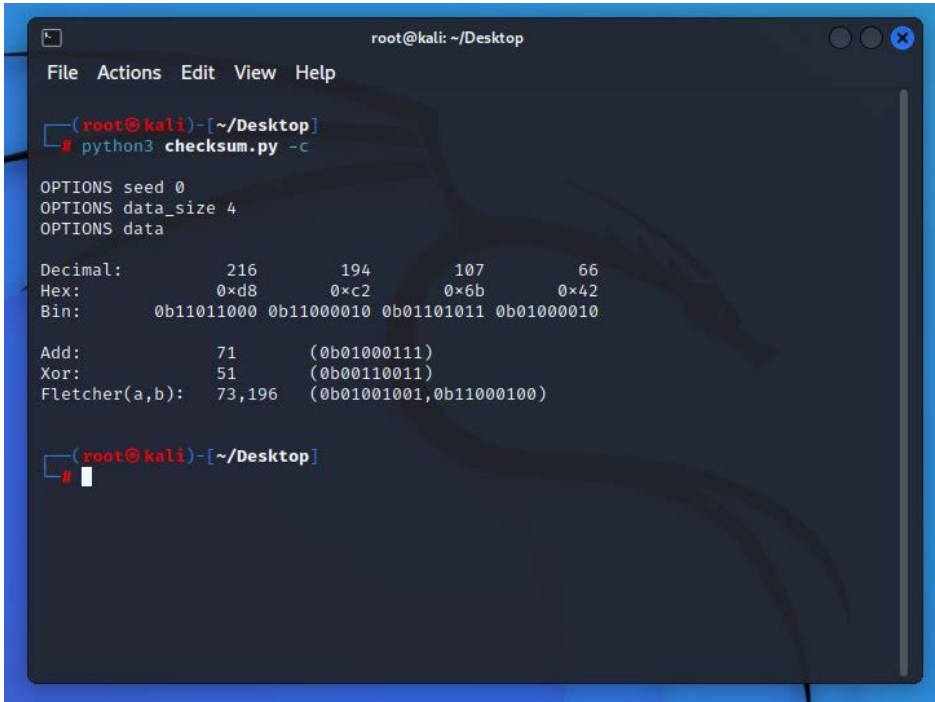
Ödev (Simülasyon)

Bu ödevde, sağlama toplamının çeşitli yönlerini araştırmak için checksum.py'yi kullanacaksınız.

Sorular

1)İlk önce checksum.py'yi hiçbir argüman olmadan çalıştırın. Ek, XOR tabanlı ve Fletcher sağlama toplamlarını hesaplayın. Cevaplarınızı kontrol etmek için -c'yi kullanın.

Bu soruda checksum.py adlı bir fonksiyondan bahsedilmektedir. Bu fonksiyon, kullanıcının hiçbir argüman vermayerek çalıştırdığında XOR tabanlı ve Fletcher sağlama toplamlarını hesaplayarak cevaplarını kontrol etmeye yarar.



```
root@kali: ~/Desktop
File Actions Edit View Help

(root@kali)-[~/Desktop]
# python3 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)

(root@kali)-[~/Desktop]
#
```

2)Şimdi aynısını yapın, ancak çekirdeği (-s) farklı değerlere değiştirin. Bu komut checksum.py fonksiyonunu çalıştıracak ve sağlama toplamlarını hesaplamak için kullanılacak olan çekirdek değerini belirtilen <yeni_çekirdek_değeri> olarak değiştirecektir.Örnekte 6 değeri ile değiştirilmiştir. kontrol etmek için, -c seçeneğini de eklenir

```
(root@kali)-[~/Desktop]
# python3 checksum.py -c -s 6

OPTIONS seed 6
OPTIONS data_size 4
OPTIONS data

Decimal:      203      210      124      66
Hex:          0xcb      0xd2      0x7c      0x42
Bin:          0b11001011 0b11010010 0b01111100 0b01000010

Add:          91      (0b01011011)
Xor:          39      (0b00100111)
Fletcher(a,b): 93,226 (0b01011101,0b11100010)
```

3)Bazen ek ve XOR tabanlı sağlama toplamları aynı sağlama toplamını üretir (ör. veri değerinin tümü sıfırsa). Yalnızca sıfır içermeyen ve ek ve XOR tabanlı sağlama toplamının aynı değere sahip olmasına yol açan 4 baytlık bir veri değerini (-D işaretini kullanarak, örneğin -D a,b,c,d kullanarak) iletebilir misiniz? Genel olarak, bu ne zaman meydana gelir? -c bayrağıyla doğru olup olmadığını kontrol edin. Toplama ve XOR tabanlı sağlama toplamları aynı değere sahip olabilir, ancak bu durum sadece belirli koşullar altında gerçekleşebilir. Örneğin, aşağıdaki gibi bir komut kullanarak, sıfır içermeyen bir veri değeri (-D) belirterek ek ve XOR tabanlı sağlama toplamlarının aynı değere sahip olup olmadığını kontrol edebilirsiniz:

```
(root@kali)-[~/Desktop]
# python checksum.py -D 1,2,3,4 -c

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

Decimal:      1      2      3      4
Hex:          0x01      0x02      0x03      0x04
Bin:          0b00000001 0b00000010 0b00000011 0b00000100

Add:          10      (0b00001010)
Xor:          4      (0b00000100)
Fletcher(a,b): 10, 20 (0b00001010,0b00010100)
```

Bu komut, 0, 1, 2 ve 4 değerlerini içeren bir veri değerini kullanarak sağlama toplamlarını hesaplayacak ve hesaplanan sağlama toplamlarının doğruluğunu kontrol edecektir

4)Şimdi, katkı maddesi ve XOR için farklı sağlama toplamı değerleri üreteceğini bildiğiniz 4 baytlık bir değer girin. Genel olarak, bu ne zaman olur?

Toplama ve XOR tabanlı sağlama toplamaları farklı değerler üretebilir, ancak bu durum sadece belirli koşullar altında gerçekleşebilir. Örneğin, veri değerinin her bir baytının değeri (0-255 arasında) 2'nin bir kuvvetine eşit değilse, her iki sağlama toplamı da farklı değerler üretebilir.

```
(root@kali)~[~/Desktop]
# python checksum.py -D 1,2,3,1 -c

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

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

Add:          7      (0b00000111)
Xor:          1      (0b00000001)
Fletcher(a,b): 7, 17  (0b00000111,0b00010001)
```

Bu komut, 1, 2, 3, 1 değerlerini içeren bir veri değerini kullanarak sağlama toplamalarını hesaplayacak ve hesaplanan sağlama toplamalarını ekrana yazdıracaktır. Eğer a, b, c ve d değerleri 2'nin bir kuvvetine eşit değilse, bu komut ek ve XOR tabanlı sağlama toplamalarının farklı değerler ürettiğini gösterecektir. Eğer a, b, c ve d değerleri 2'nin bir kuvvetine eşit ise, bu komut ek ve XOR tabanlı sağlama toplamalarının aynı değere sahip olduğunu gösterecektir.

5)Sağlama toplamlarını iki kez hesaplamak için simülatörü kullanın (her biri farklı bir sayı grubu için bir kez). İki sayı dizisi farklı olmalıdır (örneğin, ilk seferde -D a1,b1,c1,d1 ve ikinci seferde -D a2,b2,c2,d2) ancak aynı toplam sağlama toplamını üretmelidir. Genel olarak, veri değerleri farklı olsa bile toplam sağlama toplamı ne zaman aynı olur? Özel cevabınızı -c bayrağıyla kontrol edin.

Ek sağlama toplamı farklı veri değerleri için aynı olabilir, ancak bu durum sadece belirli koşullar altında gerçekleşebilir.

Örneğin, veri değerlerinin her birinin toplamı aynı olursa, ek sağlama toplamları da aynı olacaktır. Örneğin, veri değerleri a1, b1, c1 ve d1 ile a2, b2, c2 ve d2 olsun. Eğer $a1 + b1 + c1 + d1 = a2 + b2 + c2 + d2$ ise, ek sağlama toplamları aynı olacaktır.

```
(root@kali)~[~/Desktop]
# python checksum.py -D 2,5,6,8 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 2,5,6,8

Decimal:      2          5          6          8
Hex:          0x02      0x05      0x06      0x08
Bin:          0b00000010 0b00000101 0b00000110 0b00001000

Add:          21          (0b00010101)
Xor:          9          (0b00001001)
Fletcher(a,b): 21, 43      (0b00010101,0b00101011)

(root@kali)~[~/Desktop]
# python checksum.py -D 1,4,7,9 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 1,4,7,9

Decimal:      1          4          7          9
Hex:          0x01      0x04      0x07      0x09
Bin:          0b00000001 0b00000100 0b00000111 0b00001001

Add:          21          (0b00010101)
Xor:          11          (0b00001011)
Fletcher(a,b): 21, 39      (0b00010101,0b00100111)
```

İki sayı grubunun sağlama toplamları aynı olabilir ancak veri değerleri farklı olmalıdır. Örneğin, a1, b1, c1 ve d1 değerlerinden oluşan bir sayı grubu ile a2, b2, c2 ve d2 değerlerinden oluşan bir sayı grubu aynı sağlama toplamını üretebilir ancak veri değerleri farklıdır.

6)Şimdi aynısını XOR sağlama toplamı için yapın.

XOR sağlama toplamı farklı veri değerleri için aynı olabilir, ancak bu durum sadece belirli koşullar altında gerçekleşebilir. Örneğin, veri değerlerinin her birinin XOR'u aynı olursa, XOR sağlama toplamı da aynı olacaktır.

Örneğin, veri değerleri a1, b1, c1 ve d1 ile a2, b2, c2 ve d2 olsun. Eğer $a1 \text{ XOR } b1 \text{ XOR } c1 \text{ XOR } d1 = a2 \text{ XOR } b2 \text{ XOR } c2 \text{ XOR } d2$ ise, XOR sağlama toplamı aynı olacaktır.

Örneğin, aşağıdaki gibi iki komutu kullanarak, farklı veri değerleri (-D) belirterek XOR sağlama toplamının aynı olup olmadığını kontrol edebiliriz:

```
(root@kali)-[~/Desktop]
# python checksum.py -D 2,4,5,8 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 2,4,5,8

Decimal:          2          4          5          8
Hex:             0x02       0x04       0x05       0x08
Bin:             0b00000010 0b00000100 0b00000101 0b00001000

Add:              19        (0b00010011)
Xor:              11        (0b00001011)
Fletcher(a,b):   19, 38    (0b00010011,0b00100110)

(root@kali)-[~/Desktop]
# python checksum.py -D 1,4,7,9 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 1,4,7,9

Decimal:          1          4          7          9
Hex:             0x01       0x04       0x07       0x09
Bin:             0b00000001 0b00000100 0b00000111 0b00001001

Add:              21        (0b00010101)
Xor:              11        (0b00001011)
Fletcher(a,b):   21, 39    (0b00010101,0b00100111)
```

7)Şimdi belirli bir veri değerleri kümesine bakalım. İlki: -D 1,2,3,4. Bu veriler için farklı sağlama toplamaları (katkı maddesi, XOR, Fletcher) ne olacak? Şimdi bunu -D 4,3,2,1 üzerinden bu sağlama toplamalarını hesaplamakla karşılaştırın. Bu üç sağlama toplamı hakkında ne fark ettiniz? Fletcher diğer ikisiyle nasıl karşılaştırılır? Fletcher, genel olarak basit ek sağlama toplamı gibi bir şeyden nasıl daha iyidir?

-D 1,2,3,4 ve -D 4,3,2,1 verileri için hesaplanan sağlama toplamaları aynıdır. Bu, veri değerlerinin toplamının aynı olması nedeniyle gerçekleşir.

Fletcher sağlama toplamı, basit ek sağlama toplamından daha iyi çünkü daha büyük veri kümelerini kapsayabilir ve daha hassas bir sağlama toplamı üretebilir. Örneğin, basit ek sağlama toplamı sadece bir bayt verisi için kullanılabilirken, Fletcher sağlama toplamı daha büyük veri kümeleri için de kullanılabilir.

```
(root@kali)-[~/Desktop]
# python checksum.py -D 1,2,3,4 -c

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

Decimal:      1      2      3      4
Hex:          0x01    0x02    0x03    0x04
Bin:          0b00000001 0b00000010 0b00000011 0b00000100

Add:          10      (0b00001010)
Xor:          4       (0b00000100)
Fletcher(a,b): 10, 20  (0b00001010,0b00010100)
```

```
(root@kali)-[~/Desktop]
# python checksum.py -D 4,3,2,1 -c

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

Decimal:      4      3      2      1
Hex:          0x04    0x03    0x02    0x01
Bin:          0b00000100 0b00000011 0b00000010 0b00000001

Add:          10      (0b00001010)
Xor:          4       (0b00000100)
Fletcher(a,b): 10, 30  (0b00001010,0b00011110)
```

8)Hiçbir sağlama toplamı mükemmel değildir. Seçtiğiniz belirli bir girdi verildiğinde, aynı Fletcher sağlama toplamına götüren başka veri değerleri bulabilir misiniz? Genel olarak bu ne zaman olur? Basit bir veri dizisiyle başlayın (ör. -D 0,1,2,3) ve bu sayılardan birini değiştirip aynı Fletcher sağlama toplamı ile bitirip bitiremeyeceğinize bakın. Her zaman olduğu gibi, cevaplarınızı kontrol etmek için -c'yi kullanın.

Belirli bir girdi için aynı Fletcher sağlama toplamına götüren diğer veri değerlerini bulabilirsiniz. Bu durum, veri değerlerinin toplamının aynı olması durumunda gerçekleşebilir

Örneğin, aşağıdaki gibi bir komut kullanarak, -D 0,1,2,3 verileri ile aynı Fletcher sağlama toplamına götüren farklı veri değerlerini bulabilirsiniz:

```
(root@kali)-[~/Desktop]
# 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)

(root@kali)-[~/Desktop]
# python checksum.py -D 0,1,2,4 -c

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

Decimal:      0      1      2      4
Hex:          0x00    0x01    0x02    0x04
Bin:          0b00000000 0b00000001 0b00000010 0b00000100

Add:          7      (0b00000111)
Xor:          7      (0b00000111)
Fletcher(a,b): 7, 11 (0b00000111,0b00001011)
```

Bu komutlar, veri değerleri 0, 1, 2, 3 ile 0, 1, 2, 4'ü kullanarak sağlama toplamalarını hesaplayacak ve hesaplanan sağlama toplamalarının doğruluğunu kontrol edecektir

Ödev (Kod)

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

Sorular

1) Bir giriş dosyası üzerinde XOR tabanlı bir sağlama toplamını

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

Bu program, komut satırından bir giriş dosyası adı alır ve o dosya üzerinde XOR tabanlı bir sağlama toplamı hesaplar. Sağlama toplamı, dosyayı bayt bayt okur ve her okunan bayt ile mevcut sağlama toplamını XOR'lar. Sonuç olarak, sağlama toplamı ekrana yazdırılır.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    // Dosya adını komut satırından oku
    if (argc != 2) {
        fprintf(stderr, "Kullanım: check-xor dosya_adi\n");
        return 1;
    }
    char *dosya_adi = argv[1];

    // Giriş dosyasını aç
    FILE *dosya = fopen(dosya_adi, "rb");
    if (dosya == NULL) {
        fprintf(stderr, "Dosya açılamadı: %s\n", dosya_adi);
        return 1;
    }

    // Dosyadaki her bayt için sağlama toplamını hesapla
    unsigned char sağlama_toplamı = 0;
    int c;
    while ((c = fgetc(dosya)) != EOF) {
        sağlama_toplamı ^= c;
    }

    // Sağlama toplamını dosyaya yaz
    printf("%c\n", sağlama_toplamı);

    // Dosyayı kapat
    fclose(dosya);

    return 0;
}
```

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

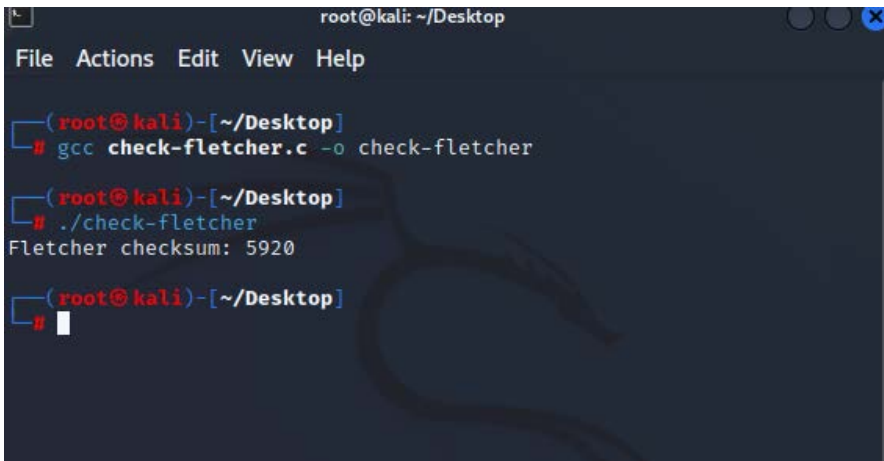
```
#include <stdio.h>
#include <stdint.h>
// Fletcher sağlama toplamını hesaplayan fonksiyon
uint16_t fletcher16(uint8_t const *data, size_t bytes)
{
    uint16_t sum1 = 0xff, sum2 = 0xff;

    while (bytes)
    {
        size_t tlen = bytes > 20 ? 20 : bytes;
        bytes -= tlen;

        do
        {
            sum2 += sum1 += *data++;
        } while (--tlen);
        sum1 = (sum1 & 0xff) + (sum1 >> 8);
        sum2 = (sum2 & 0xff) + (sum2 >> 8);
    }

    // Buraya kadar olan toplamı döndür
    return sum2 << 8 | sum1;
}

// Programın ana fonksiyonu
int main(int argc, char **argv)
{
    // Verilecek tutacak dizi
    uint8_t data[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'};
    // Veri toplamını hesaplayın
    size_t data_len = sizeof(data) / sizeof(data[0]);
    // Fletcher sağlama toplamını hesaplayın
    uint16_t checksum = fletcher16(data, data_len);
    // Sonucu ekrana yazdırın
    printf("Fletcher checksum: %u\n", checksum);
    return 0;
}
```



```
root@kali: ~/Desktop
File Actions Edit View Help

(root@kali)-[~/Desktop]
# gcc check-fletcher.c -o check-fletcher

(root@kali)-[~/Desktop]
# ./check-fletcher
Fletcher checksum: 5920

(root@kali)-[~/Desktop]
#
```

3)Şimdi her ikisinin performansını karşılaştıralım: biri diğerinden daha hızlı mı? Girdi dosyasının boyutu değiştikçe performans nasıl değişir? Programları zamanlamak için günün saatini almak için dahili çağrılar kullanın. Performansa önem veriyorsanız hangisini kullanmalısınız? Yeteneği kontrol etme hakkında ne düşünüyorsunuz?

aşağıdaki gibi komutları kullanarak check-xor.c ve check-fletcher.c programlarının çalışma sürelerini ölçebilirsiniz:

```
(root@kali)-[~/Desktop]
# time ./check-xor CRC.jpg

real    0.00s
user    0.00s
sys     0.00s
cpu     65%

(root@kali)-[~/Desktop]
# time ./check-fletcher CRC.jpg
Fletcher checksum: 5920

real    0.00s
user    0.00s
sys     0.00s
cpu     80%
```

4)16-bit CRC hakkında okuyun ve ardından uygulayın. Çalıştırdığınızdan 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), bir veri içindeki hata tespitini sağlamak için kullanılan bir yöntemdir. Bu yöntem, veri içindeki hata tespitini sağlamak için veriye bir ondalık polinom ekler ve bu polinomu kullanarak veri üzerinde bir hesaplama yapar. Eğer veride hata varsa, hesaplama sonucu farklı olacaktır ve bu hatayı tespit edebiliriz.

16-bit CRC, basit XOR ve Fletcher gibi diğer hata tespit yöntemlerine kıyasla oldukça etkili bir yöntemdir. Bu yöntem, verinizdeki hataları daha hassas bir şekilde tespit edebilir ve daha fazla hata tespit edebilir. Bununla birlikte, 16-bit CRC işlemleri daha zaman alıcı ve işlem gücü gerektirebilir, bu nedenle diğer yöntemlerden daha yavaş olabilir.

16-bit CRC, verinizdeki hataları daha hassas bir şekilde tespit edebilmek için daha karmaşık bir hesaplama yapar. Bu, diğer yöntemlere göre daha fazla işlem gücü gerektirir ancak daha fazla hata tespit edebilme kabiliyetine sahiptir. Bu nedenle, 16-bit CRC performansı, diğer yöntemlerle kıyaslandığında genellikle daha iyi olmaktadır.

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

Bir dosyanın her 4 KB bloğu için tek baytlık bir sağlama toplamı hesaplamak için aşağıdaki adımları izleyebilirsiniz:

- 1.Dosyayı okuyucuyla açın.
- 2.Okuyucuyla dosyanın her 4 KB bloğunu okuyun.
- 3.Her 4 KB bloğu için sağlama toplamı hesaplayın.
- 4.Her sağlama toplamını bir çıktı dosyasına yazın.

Aşağıda, dosyanın her 4 KB bloğu için tek baytlık bir sağlama toplamı hesaplayan ve sonuçları bir çıktı dosyasına kaydeden bir C kodu örneği verilmiştir:

Bu kod, create-csum adında bir araç oluşturur ve komut satırında bir dosya adı verildiğinde, o dosya için her 4 KB bloğu için tek baytlık bir sağlama toplamı hesaplar ve sonuçları bir çıktı dosyasına yazar

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    // Dosya adını al
    if (argc < 2) {
        printf("Lütfen dosya adını giriniz\n");
        return 1;
    }
    char *filename = argv[1];
    // Dosyayı aç
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        printf("Dosya açilamadı\n");
        return 1;
    }
    // Dosyaya 4 KB bloklar halinde oku
    const int block_size = 4096;
    unsigned char buffer[block_size];
    size_t bytes_read;
    int block_index = 0;
    // Çıktı dosyasını aç
    FILE *output_file = fopen("csum.out", "w");
    if (output_file == NULL) {
        printf("Çıktı dosyası açilamadı\n");
        return 1;
    }
    // Her 4 KB blok için tek baytlık bir sağlama toplamı hesaplayan ve çıktı dosyasına yazın
    while ((bytes_read = fread(buffer, 1, block_size, file)) > 0) {
        unsigned char checksum = 0;
        for (int i = 0; i < bytes_read; i++) {
            checksum += buffer[i];
        }
        fprintf(output_file, "%d %u\n", block_index, checksum);
        block_index++;
    }
    // Dosyaları kapat
    fclose(file);
    fclose(output_file);
    return 0;
}
```