39

Interlude: Dosyalar ve Dizinler

Şimdiye kadar iki temel işletim sistemi soyutlamasının gelişimini gördük: CPU'nun sanallaştırması olan süreç ve sanallaştırma olan adres alanı hafızanın. Birlikte, bu iki soyutlama, bir programın kendi özel, yalıtılmış dünyasındaymış gibi çalışmasına izin verir; sanki kendi işlemcisine (veya işlemcilerine) sahipmiş gibi; kendi hafızasına sahiptir. Bu yanılsama, sistemi programlamayı çok daha kolay hale getirir ve bu nedenle bugün masaüstü bilgisayarlarda ve sunucularda değil , giderek artan bir şekilde tüm programlanabilir platformlarda yaygındır . cep telefonları ve benzerleri.

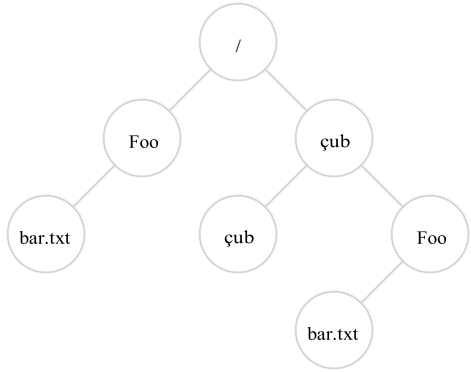
Bu bölümde, sanallaştırma bulmacasına bir kritik parça daha ekliyoruz: kalıcı depolama(persistent storage). Klasik sabit disk sürücüsü(hard disk drive)veya daha modern bir katı hal (solid state) depolama aygıtı gibi kalıcı bir depolama aygıtı (storage device), bilgileri kalıcı olarak (veya en azından uzun bir süre depolar)). Güç kaybı olduğunda içeriği kaybolan belleğin aksine, kalıcı bir depolama aygıtı bu tür verileri sağlam tutar. Bu nedenle, işletim sistemi böyle bir cihazla ekstra özen göstermelidir: burası kullanıcıların gerçekten önemsedikleri verileri sakladıkları yerdir.

ÖZELLİK: KALICI BİR CİHAZ NASIL YÖNETİLİR
İşletim sistemi kalıcı bir cihazı nasıl yönetmelidir?
API'ler nelerdir?
Implementatio n'nin önemli yönleri nelerdir?

Bu nedenle, önümüzdeki birkaç bölümde, performansı ve güvenilirliği artırma yöntemlerine odaklanarak kalıcı verileri yönetmek için kritik teknikleri keşfedeceğiz. Bununla birlikte, API'ye genel bir bakışla başlıyoruz: bir UNIX dosya sistemiyle etkileşim kurarken görmeyi beklediğiniz arayüzler.

39.1 Dosyalar Ve Dizinler

Depolamanın sanallaştırmasında zaman içinde iki temel soyutlama gelişmiştir. Birincisi **dosyadır(file)**. Dosya, her birini okuyabileceğiniz veya yazabileceğiniz doğrusal bir bayt dizisidir. Her dosyanın bir tür düşük düzeyli adı vardır, genellikle bir dizi **düşük düzeyli ad (low-level name);** Çoğu zaman, kullanıcı farkında değildir



Şekil 39.1: **Örnek Dizin Ağacı**

bu isim (göreceğimiz gibi). Tarihsel nedenlerden dolayı, bir dosyanın düşük seviyeli adı genellikle **dosya numarası (inode number)** olarak adlandırılır. Gelecek bölümlerde inode'lar hakkında çok daha fazla şey öğreneceğiz; şimdilik, her dosyanın kendisiyle ilişkili bir inode numarası olduğunu varsayalım .

Çoğu sistemde, işletim sistemi dosyanın yapısı hakkında fazla bir şey bilmez (örneğin, bir resim, bir metin dosyası veya C kodu olup olmadığı); bunun yerine, dosya sisteminin sorumluluğu, bu tür verileri diskte kalıcı olarak depolamak ve verileri tekrar talep ettiğinizde , ilk etapta oraya koyduğun şey. Bunu yapmak göründüğü kadar basit değil!

İkinci soyutlama , bir dizinin soyutlamasıdır. Bir dizin, bir dosya gibi, düşük seviyeli bir ada (yani, bir inode numarasına) sahiptir, ancak içeriği oldukça spesifiktir: bir listesini içerir (bize okunabilir ad, düşük seviyeli ad) çiftler. Örneğin, düşük düzeyli adı "10" olan bir dosya olduğunu ve kullanıcı tarafından okunabilen "foo" adıyla anıldığını varsayalım. Bu nedenle "foo"nun bulunduğu dizin, kullanıcı tarafından okunabilir adı düşük düzeyli adla eşleyen bir girişe ("foo", "10") sahip olacaktır. Bir dizindeki her girdi , dosyalara veya diğer dizinlere başvurur. Dizinleri diğer dizinlerin içine yerleştirerek, kullanıcılar altında tüm dosyaların bulunduğu rasgele bir **dizin ağacı (directory tree)** (veya **dizin hiyerarşisi (directory hierarchy)**) oluşturabilirler. ve dizinler saklanır.

Dizin hiyerarşisi bir **kök dizinde (root directory**) başlar (UNIX tabanlı sistemlerde, kök dizin basitçe / olarak adlandırılır) ve adlandırmak için bir tür **ayırıcı (separator)** kullanır İstenilen dosya veya dizin adlandırılana kadar sonraki **alt dizinler (sub-directories)**. Örneğin, bir kullanıcı / kök dizininde bir dizin foo oluşturduysa ve ardından `foo` dizininde bir dosya çubuğu oluşturduysa.txt, dosya,bu durumda /foo/bar.txt olacak **mutlak yol**

adına(absolute pathname) göre. Daha karmaşık bir dizin ağacı için Şekil 39.1'e bakın; Örnekteki geçerli dizinler `/`, `/foo`, `/bar`, `/bar/bar`, `/bar/foo` ve geçerli dosyalar `/foo/bar.txt` ve `/bar/foo` /bar.txt dizinleridir.

İPUCU:ADLANDIRMAYI DİKKATLİCE DÜŞÜNÜN

Adlandırma, bilgisayar sistemlerinin önemli bir yönüdür [SK09]. UNIX sistemlerinde, aklınıza gelebilecek hemen hemen her şey dosya sistemi üzerinden adlandırılır. Sadece dosyaların, cihazların, boruların ve hatta işlemlerin ötesinde [K84], eski bir dosya sistemine benzeyen şeyde bulunabilir. Adlandırmanın bu tekdüzeliği, sistemin kavramsal modelini kolaylaştırır ve sistemi daha basit ve daha modüler hale getirir. Bu nedenle, bir sistem veya arayüz oluştururken, hangi isimleri kullandığınızı dikkatlice düşünün.

Dizinler ve dosyalar, dosya sistemi ağacında farklı konumlarda bulundukları sürece aynı ada sahip olabilir (örneğin, çubuk adında iki dosya vardır.txt şekil, /foo/bar.txt ve /bar/foo/bar.txt).

Bu örnekteki dosya adının genellikle iki bölümden oluştuğunu da fark edebilirsiniz: çubuk ve txt, nokta ile ayrılmış. İlk bölüm rastgele bir name'dir, oysa dosya adının ikinci kısmı genellikle dosyanın **türünü (type),** örneğin C kodu olup olmadığını belirtmek için kullanılır (ör., .c) veya bir görüntü (ör. .jpg) ya da bir müzik dosyası (ör. .mp3). Bununla birlikte, bu genellikle sadece bir kuraldır: main.c adlı bir dosyada bulunan verilerin gerçekten C kaynak kodu olduğuna dair bir zorlama yoktur.

Böylece, dosya sistemi tarafından sağlanan harika bir şey görebiliriz: ilgilendiğimiz tüm dosyaları **adlandırmanın (convention)** uygun bir yolu . İsimler sistemlerde önemlidir, çünkü herhangi bir kaynağa erişmenin ilk adımı onu adlandırabilmektir. UNIX sistemlerinde, dosya sistemi böylece disk, USB bellek, CD-ROM, diğer birçok cihaz ve aslında diğer birçok şey üzerindeki dosyalara erişmek için birleşik bir yol sağlar. , tümü tek dizin ağacının altında bulunur.

39.2 Dosya Sistemi Arabirimi

Şimdi dosya sistemi arayüzünü daha ayrıntılı olarak tartışalım. Dosya oluşturma, bunlara erişme ve silme ile ilgili temel bilgilerle başlayacağız . Bunun basit olduğunu düşünebilirsiniz, ancak yol boyunca `unlink()` olarak bilinen dosyaları kaldırmak için kullanılan gizemli çağrıyı keşfedeceğiz. Umarım, bu bölümün sonunda , bu gizem sizin için o kadar gizemli olmayacak!

39.3 Dosya Oluşturma

En temel işlemlerle başlayacağız: dosya oluşturma. Bu, açık sistem çağrısı ile gerçekleştirilebilir; `open()` öğesini çağırıp OCREAT bayrağını geçirerek, bir

program yeni bir dosya oluşturabilir. Geçerli çalışma dizininde "foo" adlı bir dosya oluşturmak için bazı örnek kodlar aşağıda verilmiştir:

```
int fd = open("foo", O_CREAT| O_WRONLY| O_TRUNC,S_IRUSR|
S_IWUSR);
```

BİR KENARA: CREAT() SİSTEM ÇAĞRISI

Dosya oluşturmanın eski yolu, creat()'ı aşağıdaki gibi çağırmaktır:

```
// option: add second flag to set permissions
int fd = creat("foo");
```

`Creat()` öğesini şu bayraklarla `open()` olarak düşünebilirsiniz: OCREAT | OWRONLY | OTRUNC. Çünkü `open()` bir dosya oluşturabilir, `creat()` kullanımı biraz gözden düşmüştür (aslında, sadece bir kütüphane çağrısı olarak uygulanabilir). `open());` ancak, UNIX lore'da özel bir yere sahiptir. Özellikle , Ken Thompson'a UNIX'i yeniden tasarlarken neyi farklı yaptığı sorulduğunda, `"creat'ı` bir e ile hecelerdim."

`open()` yordamı birkaç farklı bayrak alır. Bu örnekte, ikinci parametre yoksa dosyayı `(OCREAT)` oluşturur, dosyanın yalnızca `(OWRONLY)` yazılabilmesini sağlar ve , dosya zaten varsa, sıfır baytlık bir boyuta keser ve böylece var olan içeriği `(OTRUNC)` kaldırır. Üçüncü parametre izinleri belirtir, bu durumda dosyayı sahibi tarafından okunabilir ve yazılabilir hale getirir.

`open()'` in önemli bir yönü ne döndürdüğüdür: bir **dosya tanımlayıcısı(file descriptor).** Dosya tanımlayıcısı yalnızca bir tamsayıdır, işlem başına özeldir ve UNIX sistemlerinde dosyalara erişmek için kullanılır; Bu nedenle, bir dosya açıldıktan sonra, dosyayı okumak veya yazmak için dosya tanımlayıcısını kullanırsınız, bunu yapma izniniz olan birözet. Bu şekilde, bir dosya tanımlayıcısı bir **yetenektir (capability)** [L84], yani size belirli işlemleri gerçekleştirme gücü veren opak bir tanıtıcıdır.Dosya tanımlayıcısını düşünmenin başka bir yolu da dosya türündeki bir nesnenin işaretçisi olmaktır; Böyle bir nesneye sahip olduğunuzda, dosyaya erişmek için `read()` ve `write()` gibi diğer "yöntemleri " çağırabilirsiniz (şunu göreceğiz). aşağıda nasıl yapılır).

Yukarıda belirtildiği gibi, dosya tanımlayıcıları işletim sistemi tarafından işlem başına yönetilir.Bu, bir tür basit yapının (örneğin, bir dizi) UNIX sistemlerindeki proc yapısında tutulduğu anlamına gelir. İşte xv6 çekirdeğinden [CK + 08] ilgili parça:

```
struct proc {
  ...
  struct file *ofile[NOFILE]; // Open files
  ...
```

```
};
```

Basit bir dizi (maksimum NOFILE açık dosya içeren), işlem başına hangi dosyaların açıldığını izler. Dizinin her girişi aslında okunan veya yazılan dosya hakkındaki bilgileri izlemek için kullanılacak bir yapı dosyasının işaretçisidir; bunu aşağıda daha ayrıntılı olarak tartışacağız.

İPUCU: STRACE KULLANIMI (VE BENZER ARAÇLAR)

`strace` aracı, hangi programların uygun olduğunu görmek için harika bir yol sağlar. Çalıştırarak, bir programın hangi sistem çağrılarını yaptığını izleyebilir , argümanları görebilir ve kodları döndürebilir ve genellikle neler olup bittiği hakkında çok iyi bir fikir edinebilirsiniz . .

Araç ayrıca oldukça yararlı olabilecek bazı argümanlar alır. Örneğin, `-f` çatallı çocukları da takip eder;`-t` her aramada günün saatini bildirir; `-e trace=open,close,read,write` only traces çağrıları bu sistem çağrılarına çağrı yapar ve diğerlerini yok sayar. Başka birçok bayrak var; adam sayfalarını okuyun ve bu harika aracı nasıl kullanacağınızı öğrenin.

39.4 Dosyaları Okuma ve Yazma

Bazı dosyalarımız olduğunda, elbette onları okumak veya yazmak isteyebiliriz. Mevcut bir dosyayı okuyarak başlayalım. Bir command satırında yazıyor olsaydık, dosyanın içeriğini ekrana dökmek için program `cat` kullanabilirdik.

```
prompt> echo hello > foo
prompt> cat foo hello
prompt>
```

Bu kod parçacığında, program `echo` çıktısını , içinde "hello" kelimesini içeren `foo` dosyasına yönlendiririz. Daha sonra dosyanın içeriğini görmek için `cat` kullanıyoruz. Ancak kedi programı `foo` dosyasına nasıl erişir?

Bunu öğrenmek için, bir program tarafından yapılan sistem çağrılarını izlemek için inanılmaz derecede kullanışlı bir araç kullanacağız . Linux'ta, araç called **strace'tir**; diğer sistemler de benzer araçlara sahiptir **(Mac'te dtruss** veya bazı eski UNIX varyantlarında **truss** bölümüne bakınız). Strace'in yaptığı şey, çalışırken bir program tarafından yapılan her sistem çağrısını izlemek ve izlemeyi görmeniz için ekrana dökmektir.

İşte kedinin ne yaptığını anlamak için iz sürmemize bir örnek (okunabilirlik için bazı çağrılar kaldırıldı):

```
prompt> strace cat foo

...
```

```
        open("foo", O_RDONLY|O_LARGEFILE)       = 3
        read(3, "hello\n", 4096)                = 6
        write(1, "hello\n", 6)                  = 6
hello
        read(3, "", 4096)                       = 0
        close(3)                                = 0
...
prompt>
```

Cat'in yaptığı ilk şey, dosyayı okumak için açmaktır. Bu konuda dikkat etmemiz gereken birkaç nokta; ilk olarak, dosyanın `ORDONLY` bayrağında belirtildiği gibi yalnızca okuma için (yazma için değil) açılması; ikincisi, 64 bit uzaklığın kullanılması (`OLARGEFILE`); üçüncüsü, `open()` çağrısının başarılı olması ve 3 değerine sahip bir dosya tanımlayıcısı döndürmesi.

`Open()` için yapılan ilk çağrı neden beklediğiniz gibi 0 veya belki de 1 değil de 3 döndürüyor? Görünüşe göre, çalışan her işlemin zaten üç dosyası açık, standart giriş (işlemin giriş almak için okuyabileceği), standart çıktı (hangi işlem, bilgileri ekrana dökmek için yazabilir) ve standart hata (işlemin hata mesajları yazabileceği). Bunlar sırasıyla file descriptors 0, 1 ve 2 ile temsil edilir. Bu nedenle, başka bir dosyayı ilk açtığınızda (yukarıda kedinin yaptığı gibi), neredeyse kesinlikle dosya tanımlayıcısı 3 olacaktır.

Açma işlemi başarılı olduktan sonra `cat`, bir dosyadan bazı baytları tekrar tekrar okumak için `read()` sistem çağrısını kullanır. Okunacak ilk argument dosya tanımlayıcısıdır, böylece dosya sistemine hangi dosyayı okuyacağını söyler; Bir işlem elbette birden fazla dosyayı açabilir. bir kerede ve böylece tanımlayıcı, işletim sisteminin belirli bir okumanın hangi dosyaya atıfta bulunduğunu bilmesini sağlar. İkinci bağımsız değişken, `read()` sonucunun yerleştirileceği bir ara belleğe işaret eder; yukarıdaki sistem çağrısı izlemesinde, strace bu noktadaki okumanın sonuçlarını gösterir ("hello"). Üçüncü bağımsız değişken, bu durumda 4 KB olan arabelleğin boyutudur. `read()` çağrısı da başarıyla döner, burada okuduğu bayt sayısını döndürür ("hello" kelimesindeki harfler için 5 ve satır sonu işaretçisi için bir tane olmak üzere 6).

Bu noktada, izlemenin başka bir ilginç sonucunu görürsünüz: `write()` sistem çağrısına, dosya tanımlayıcısı 1'e yapılan tek bir çağrı. Yukarıda belirttiğimiz gibi, bu tanımlayıcı standart çıktı olarak bilinir ve bu nedenle "merhaba" kelimesini ekrana program olarak yazmak için kullanılır . kedinin yapması gerekiyor. Ama doğrudan `write()` olarak mı adlandırılıyor? Belki (yüksek oranda optimize edilmişse). Ama değilse, kedinin yapabileceği şey

kütüphane rutinini `printf()` olarak adlandırmaktır; dahili olarak, `printf()` kendisine iletilen tüm biçimlendirme ayrıntılarını bulur ve sonunda sonuçları ekrana yazdırmak için standart çıktıya yazar .

Cat programı daha sonra dosyadan daha fazla okumaya çalışır, ancak dosyada bayt kalmadığından, `read()` 0 değerini döndürür ve program bunun şu anlama geldiğini bilir. dosyanın tamamını okumuştur . Bu nedenle, program ilgili dosya tanımlayıcısına geçerek "foo " dosyasıyla yapıldığını belirtmek için `close()` öğesini çağırır. Böylece dosya kapatılır ve böylece okunması tamamlanır.

Dosya yazmak, benzer bir dizi adımla gerçekleştirilir. İlk olarak, yazma için bir dosya açılır, ardından `write`() sistem çağrısı çağrılır, daha büyük dosyalar için muhtemelen tekrar tekrar ve sonra `close()`. Bir dosyaya, belki de kendi yazdığınız bir programa ait veya dd yardımcı programını izleyerek (örneğin: `dd if=foo of=bar`.) yazmaları izlemek için strace kullanın.

BİR KENARA: VERİ YAPISI — AÇIK DOSYA TABLOSU

Her işlem, her biri sistem genelinde **açık dosya tablosundaki (open file table)** bir girişe başvuran dosya tanımlayıcılarının bir ar rayını tutar. Bu tablodaki her giriş, tanımlayıcının hangi temel dosyaya başvurduğunu , geçerli uzaklığı ve dosyanın okunabilir olup olmadığı gibi diğer ilgili ayrıntıları izler. veya yazılabilir.

39.5 Okuma ve Yazma, Ancak Sıralı Olarak Değil

Şimdiye kadar, dosyaların nasıl okunacağını ve yazılacağını tartıştık, ancak tüm erişim **sıralı (sequential)** olmuştur; yani, ya bir dosyayı baştan sona okuduk ya da baştan sona bir dosya yazdık.

Bununla birlikte, bazen, bir dosya içindeki belirli bir ofseti okuyabilmek veya yazabilmek yararlıdır; örneğin, bir metin belgesi üzerinde bir dizin oluşturur ve bunu belirli bir sözcüğü aramak için kullanırsanız, **rastgele(random)** bir şeyden okumaya başlayabilirsiniz. belge içindeki ofsetler. Bunu yapmak için `lseek()` sistem çağrısını kullanacağız. İşte fonksiyon prototipi:

```
off_t lseek(int fildes, off_t offset, int whence);
```

İlk bağımsız değişken tanıdıktır (bir dosya tanımlayıcısı). İkinci bağımsız değişken, **dosya ofsetini (file offset)** dosya içindeki belirli bir konuma konumlandıran ofsettir. Tarihsel nedenlerden dolayı çağrılan üçüncü argüman, arayışın tam olarak nasıl gerçekleştirildiğini belirler. Man sayfasından:

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
  location plus offset bytes.
```

```
If whence is SEEK_END, the offset is set to the size of
  the file plus offset bytes.
```

Bu açıklamadan da anlayabileceğiniz gibi, bir işlem açılan her dosya için işletim sistemi,bir sonraki okuma veya yazmanın nerede başlayacağını belirleyen "geçerli" bir ofseti izler. dosyadan okuma veya dosya içine yazma. Bu nedenle, açık bir dosyanın soyutlanmasının bir kısmı, iki yoldan biriyle güncelleştirilen geçerli bir uzaklığa sahip olmasıdır. Birincisi, N baytlık bir yeniden reklam veya yazma gerçekleştiğinde , geçerli ofsete N eklenir ; böylece her okuma veya yazma dolaylı olarak ofseti günceller. İkincisi açıkça lseek'tir, bu da yukarıda belirtildiği gibi ofseti değiştirir.

Ofset, tahmin edebileceğiniz gibi, `struct proc` 'tan referans alındığı gibi, daha önce gördüğümüz `struct file`'da tutulur. İşte yapının (basitleştirilmiş) bir xv6 tanımı:

```
struct file {
  int ref; char
  readable; char
  writable; struct
  inode *ip; uint
  off;
};
```

BİR KENARA : LSEEK() ÇAĞIRMAK DİSK ARAMA YAPMAZ

Kötü adlandırılmış sistem çağrısı `lseek`(), diskleri ve bunların üzerindeki dosya sistemlerinin nasıl çalıştığını anlamaya çalışan birçok öğrencinin kafasını karıştırır. İkisini karıştırmayın! `lseek`() çağrısı, belirli bir işlem için bir sonraki okumanın veya yazmanın başlayacağı ofseti izleyen işletim sistemi belleğindeki bir değişkeni değiştirir. Bir disk araması, diske verilen bir okuma veya yazma, son okuma veya yazma ile aynı yolda olmadığında gerçekleşir ve bu nedenle bir kafa hareketi gerektirir. Bunu daha da kafa karıştırıcı hale getiren şey, bir dosyanın rasgele kısımlarından/bölümlerine okumak veya bu kısımlara yazmak ve sonra bu rasgele kısımlara okumak/yazmak için `lseek`() işlevinin çağrılmasının gerçekten de daha fazla disk aramasına yol açacağı gerçeğidir. Bu nedenle,`lseek`() öğesinin çağrılması, yaklaşan bir okuma veya yazma işleminde aramaya yol açabilir, ancak kesinlikle herhangi bir disk I/O'sinin kendi kendine gerçekleşmesine neden olmaz.

Yapıda görebileceğiniz gibi,işletim sistemi bunu,açılan dosyanın okunabilir veya yazılabilir (veya her ikisi) olup olmadığını, hangi temel dosyaya atıfta bulunduğunu (`struct inode` pointer `ip`) ve geçerli ofset (kapalı) ile işaret eder. Aşağıda daha ayrıntılı olarak tartışacağımız bir referans sayısı `(ref)` da vardır.

Bu dosya yapıları, sistemde o anda açık olan tüm dosyaları temsil eder; birlikte, bazen **açık dosya tablosu (open file table)** olarak adlandırılırlar. xv6 kernel, burada gösterildiği gibi , bunları giriş başına bir kilit ile bir dizi olarak da tutar:

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

Bunu birkaç örnekle biraz daha netleştirelim. İlk olarak, 30 0 bayt boyutunda bir dosyayı açan ve her seferinde 100 bayt okuyarak `read()` sistem çağrısını tekrar tekrar çağırarak okuyan bir işlemi izleyelim. Her sistem çağrısı tarafından döndürülen değerler ve bu dosya erişimi için açık dosya tablosundaki geçerli ofsetin değeri ile birlikte ilgili sistem çağrılarının bir izi aşağıdadır:

Sistem Çağrıları	Return Code	Current Offset
`fd = open("file", ORDONLY);`	3	0
`read(fd, buffer,100);`	100	100
`read(fd, buffer, 100);`	100	200
`read (fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close (fd);`	0	-

İzden not edilmesi gereken birkaç ilgi çekici öğe var. İlk olarak, dosya açıldığında mevcut ofsetin nasıl sıfıra sıfırlandığını görebilirsiniz. Ardından, işlem tarafından her `read()` ile nasıl artırıldığını görebilirsiniz; bu, bir işlemin dosyanın bir sonraki parçasını almak için `read()`'i çağırmaya devam etmesini kolaylaştırır. Son olarak, dosyanın sonundan sonra yapılan bir `read()` girişiminin nasıl sıfır döndürdüğünü ve böylece sürece dosyayı bütünüyle okuduğunu gösterdiğini görebilirsiniz.

İkincisi, aynı dosyayı iki kez açan ve her birine bir okuma yayınlayan bir işlemi izleyelim.

Bu

Sistem Çağrıları	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", ORDONLY);`	3	0	–
`fd2 = open("file", ORDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

örnekte, iki dosya tanımlayıcı tahsis edilmiştir (3 ve 4) ve her biri açık dosya tablosundaki farklı bir girişi ifade eder (bu örnekte, tablo başlığında gösterildiği gibi 10 ve 11 girişleri; OFT, Açık Dosya Tablosu anlamına gelir). Neler olduğunu izlerseniz, her geçerli ofsetin bağımsız olarak nasıl güncellendiğini görebilirsiniz.

Son bir örnekte, bir işlem, okumadan önce geçerli ofseti yeniden konumlandırmak için `lseek()'i` kullanır; bu durumda, yalnızca tek bir açık dosya tablosu girişi gereklidir (ilk örnekte olduğu gibi).

System Calls	Return Kod	Curent Ofset
`fd = open("file", ORDONLY);`	3	0
`lseek(fd, 200, SEEKSET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

Burada, `lseek()` çağrısı önce geçerli ofseti 200 olarak ayarlar. `read()` sonraki 50 baytı okur ve geçerli uzaklığı buna göre güncelleştirir.

39.6 Paylaşılan Dosya Tablosu Girişleri: `fork()` Ve `dup()`

Çoğu durumda (yukarıda gösterilen örneklerde olduğu gibi), dosya tanıtıcısının açık dosya tablosundaki bir girişle eşlenmesi bire bir eşlemedir. Örneğin, bir işlem çalıştığında, bir dosyayı açmaya, okumaya ve ardından kapatmaya karar verebilir; bu örnekte, dosyanın açık dosya tablosunda benzersiz bir girişi olacaktır. Aynı anda başka bir işlem aynı dosyayı okusa bile, açık dosya tablosunda her birinin kendi girişi olacaktır. Bu sayede her mantıksal

```
int main(int argc, char *argv[]) { int fd =
open("file.txt", O_RDONLY); assert(fd >= 0); int rc =
fork();
        if (rc == 0) { rc = lseek(fd,
            10, SEEK_SET);
            printf("child: offset %d\n", rc);
        } else if (rc > 0) {
            (void) wait(NULL);
            printf("parent: offset %d\n",
                   (int) lseek(fd, 0, SEEK_CUR));
        } return
        0;
    }
```

Şekil 39.2: **Paylaşılan Üst/Alt Dosya Tablosu Girişleri (fork-seek.c)**

Bir dosyanın okunması veya yazılması bağımsızdır ve verilen dosyaya erişirken her birinin kendi geçerli uzaklığı vardır.

Ancak, açık dosya tablosundaki bir girişin paylaşıldığı birkaç ilginç durum vardır. Bu durumlardan biri, benzer bir işlem `fork()` ile bir alt süreç oluşturduğunda ortaya çıkar. Şekil 39.2'de, bir ebeveynin bir alt öğe oluşturduğu ve ardından tamamlanmasını beklediği küçük bir kod parçacığı gösterilmektedir. Child `lseek()` öğesine yapılan bir çağrı aracılığıyla geçerli ofseti ayarlar ve ardından çıkar. Son olarak parent, child'ı bekledikten sonra, mevcut ofseti kontrol eder ve değerini yazdırır.

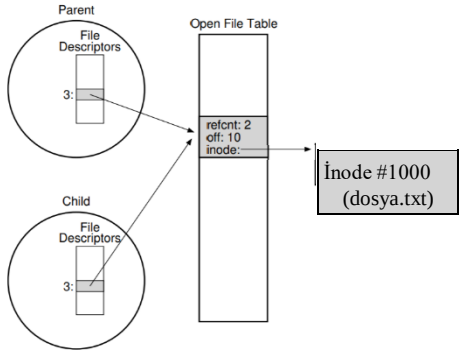
Bu programı çalıştırdığımızda aşağıdaki çıktıyı görüyoruz:

```
prompt> ./fork-seek
child: offset 10
parent: offset 10

prompt>
```

Şekil 39.3, her işlemin özel tanımlayıcı dizisini, paylaşılan açık dosya tablosu girişini ve ondan temeldeki dosya sistemi düğümüne yapılan referansı birbirine bağlayan ilişkileri gösterir. Sonunda burada **referans sayısını (reference count)** kullandığımızı unutmayın. Bir dosya tablosu girişi paylaşıldığında, referans sayısı artar; yalnızca her iki işlem de dosyayı kapattığında (veya çıktığında) giriş kaldırılacaktır.

Açık dosya tablosu girişlerini ebeveyn ve çocuk arasında paylaşmak bazen yararlıdır. Örneğin, bir görev üzerinde iş birliği içinde çalışan birkaç işlem oluşturursanız, herhangi bir ekstra koordinasyon olmadan aynı çıktı dosyasına yazabilirler. `fork()` çağrıldığında süreçler tarafından paylaşılanlar hakkında daha fazla bilgi için lütfen kılavuz sayfalarına bakın.



Şekil 39.3: **Açık Dosya Tablosu girişini paylaşma işlemleri**

Bir başka ilginç ve belki de daha kullanışlı paylaşım durumu, **dup()** sistem çağrısı (ve cousins, **dup2()** ve **dup3**()) ile gerçekleşir.

`dup()` çağrısı, bir işlemin var olan bir tanımlayıcıyla aynı temel açık dosyaya başvuran yeni bir dosya tanımlayıcısı oluşturmasına olanak tanır. Şekil 39.4'te `dup()'` un nasıl kullanılabileceğini gösteren küçük bir kod parçacığı gösterilmektedir.

`dup()` çağrısı (ve özellikle `dup2()`), bir UNIX kabuğu yazarken ve çıktı yeniden yönlendirme gibi işlemler gerçekleştirirken kullanışlıdır; biraz zaman geçirin ve nedenini düşünün! Ve şimdi, bunu bana neden kabuk projesini yaparken söylemediler diye düşünüyorsunuz. Ah, işletim sistemleriyle ilgili inanılmaz bir kitapta bile her şeyi doğru sırada alamazsınız. Üzgünüz!

```
int main(int argc, char *argv[]) {
    int fd = open("README", O_RDONLY);
    assert(fd >= 0); int fd2 =
    dup(fd);
    // now fd and fd2 can be used interchangeably
    return 0;
}
```

Şekil 39.4: **dup**() **ile** Paylaşılan **Dosya Tablosu Girişi (dup.c)**

39.7 `fsync()` ile Hemen Yazma

Çoğu zaman bir program `write`() çağrısı yaptığında, sadece dosya sistemine şunu söyler: lütfen bu verileri gelecekte bir noktada kalıcı depolamaya yazın. Dosya sistemi, performans nedenleriyle, bu tür yazmaları bir süre (örneğin 5 saniye veya 30 saniye) bellekte **arabelleğe (buffer)** alır; daha sonraki bir zamanda `write(s)` fiilen depolama aygıtına verilecektir. Çağıran uygulamanın perspektifinden bakıldığında, yazmalar hızlı olarak tamamlanıyor gibi görünmektedir ve yalnızca nadir durumlarda (örneğin, `write()` çağrısından sonra ancak diske yazma işleminden önce makinenin çökmesi) veriler kaybolacaktır.

Bununla birlikte, bazı uygulamalar bu nihai garantiden daha fazlasını gerektirir. Örneğin, bir veri tabanı yönetim sisteminde (DBMS), doğru bir kurtarma protokolünün geliştirilmesi, zaman zaman diske yazmayı zorlama yeteneğini gerektirir.

Bu tür uygulamaları desteklemek için, çoğu dosya sistemi bazı ek denetim API'lerini sağlar. UNIX dünyasında, birpplications için ulaştığı `fsync (int fd)` olarak bilinir. Bir işlemin belirli bir dosya oluşturması için `fsync()` öğesini çağırdığında, dosya sistemi tüm **kirli(dirty)** (yani henüz yazılmamış) verilerini diske koyarak zor yanıt verir. Belirtilen dosya koruyucusu tarafından başvurulan dosya. `fsync()` yordamı, tüm bu yazma işlemlerini tamamladıktan sonra geri döner.

İşte `fsync()`'in nasıl kullanılacağına dair basit bir örnek. Kod, `foo` dosyasını açar, ona tek bir veri öbeği yazar ve ardından yazmaların hemen diske zorlanmasını sağlamak için `fsync()`öğesini çağırır.`fsync()` geri döndüğünde, uygulama verilerin kalıcı olduğunu bilerek güvenli bir şekilde devam edebilir (`fsync()` doğru şekilde uygulanmışsa, yani).

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
assert(fd > -1); int rc =
write(fd, buffer, size);
assert(rc == size); rc =
fsync(fd); assert(rc == 0);
```

İlginçtir ki, bu dizi bekleyebileceğiniz her şeyi garanti etmez; Bazı durumlarda, `foo` dosyasını içeren dizini de `fsyn)` yapmanız gerekir. Bu adımın eklenmesi, yalnızca dosyanın kendisinin diskte olmasını sağlamakla kalmaz, aynı zamanda yeni oluşturulmuşsa dosyanın da kalıcı olarak dizinin bir parçası

olmasını sağlar. Şaşırtıcı olmayan bir şekilde, bu tür ayrıntılar genellikle göz ardı edilir ve birçok uygulama düzeyinde hataya yol açar [P + 13, P + 14].

39.8 Dosyaları Yeniden Adlandırma

Bir dosyaya sahip olduğumuzda, bazen bir dosyaya farklı bir ad verebilmek yararlıdır. Komut satırına yazarken, bu mv komutuyla gerçekleştirilir; Bu örnekte, file foo çubuk olarak yeniden adlandırılır:

ATARAFI: MMAP() VE PERSISTENT MEMORY
(Konuk Kenara: Terence Kelly)

Bellek haritalama (Memory mapping), dosyalardaki kalıcı verilere erişmenin alternatif bir yoludur. mmap() sistem çağrısı, bir dosyadaki bayt uzaklıkları ile çağırma işlemindeki sanal adresler arasında bir yazışma oluşturur; birincisine **destek dosyası(backing file),** ikincisine **ise bellek içi görüntü(in-memory image)** denir. İşlem daha sonra CPU talimatlarını (yani, yükler ve depolar) kullanarak bellek içi görüntüye yedekleme dosyasına erişebilir.

Dosya destekli bellek eşlemeleri, dosyaların kalıcılığını belleğin erişim semantikleriyle birleştirerek, **kalıcı bellek (persistent memory)** adı verilen bir yazılım soyutlamasını destekler. Kalıcı bellek programlama stili, bellek ve depolama için farklı veri formatları arasındaki çeviriyi ortadan kaldırarak uygulamaları kolaylaştırabilir [K19].

```
p = mmap(NULL, file_size, PROT_READ|PROT_WRITE,
         MAP_SHARED, fd, 0);
assert(p != MAP_FAILED);
for (int i = 1; i < argc;
i++)
  if (strcmp(argv[i], "pop") == 0) //
    pop if (p->n > 0) // stack not
    empty
      printf("%d\n", p->stack[--p->n]);
  } else { // push if (sizeof(pstack_t) + (1 +
    p->n) * sizeof(int)
        <= file_size) // stack not
  full p->stack[p->n++] =
  atoi(argv[i]); }
```

`pstack.c` pstack.c programı (yukarıda bir snippet ile birlikte OSTEP kodu github deposuna dahil edilmiştir), `ps.img` dosyasında hayata sıfırlardan oluşan bir çanta olarak başlayan, örneğin komut satırında `truncate` veya `dd` yardımcı programı Dosya, yığının boyutunun bir sayısını ve yığın içeriğini tutan bir tamsayılar dizisini içerir. Destek dosyasını `mmap()` –işledikten sonra, bellek içi görüntüye C işaretçilerini kullanarak yığına erişebiliriz, örneğin, `p->n` yığındaki öğelerin sayısına erişir ve `p->stack` dizisini yığınlar. Yığın kalıcı olduğu için, `pstack'in` bir çağrısıyla aktarılan veriler bir sonraki tarafından açılabilir.

Örneğin, itme ve itmenin atanması arasındaki bir çökme, kalıcı yığınımızı tutarsız bir durumda bırakabilir. Uygulamalar, kalıcı belleği hataya göre atomik olarak güncelleyen mekanizmalar kullanarak bu tür hasarları önler [K20].

```
prompt> mv foo bar
```

İzlemeyi kullanarak, `mv'nin` tam olarak iki argüman alan `rename(char *old, char *new)` sistem çağrısını kullandığını görebiliriz: dosya `(old)` ve yeni ad `(new)`.

`rename()` çağrısı tarafından sağlanan ilginç bir garanti, sistem çökmeleriyle ilgili olarak (genellikle) **atomik (atomic)** bir çağrı olarak uygulanmasıdır; yeniden adlandırma sırasında sistem çökerse, dosya eski ad veya yeni ad olarak adlandırılır ve arada garip bir durum ortaya çıkamaz. Bu nedenle, `rename()`, dosya durumuna atomik güncelleme gerektiren belirli uygulama türlerini desteklemek için kritiktir.

Burada biraz daha spesifik olalım . Bir dosya düzenleyicisi (örneğin, emacs) kullandığınızı ve bir dosyanın ortasına bir çizgi eklediğinizi düşünün. Dosyanın adı, örneğin foo.txt. Düzenleyicinin , yeni dosyanın orijinal içeriğine ve eklenen satıra sahip olmasını garanti etmek için dosyayı güncelleme şekli aşağıdaki gibidir (basitlik için hata denetimini göz ardı ederek)):

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
           S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of
file fsync(fd); close(fd); rename("foo.txt.tmp",
"foo.txt");
```

Editörün bu örnekte yaptığı şey basittir: dosyanın yeni sürümünü geçici bir ad altında yazın `(foo.txt.tmp)`, `fsync()` ile diske zorlayın tıklatın ve uygulama yeni dosya meta verilerinin ve içeriğinin diskte olduğundan emin olduğunda, geçici dosyayı özgün dosyanın adıyla yeniden adlandırın. Bu son adım, yeni dosyayı atomik olarak değiştirirken, aynı zamanda dosyanın eski sürümünü eşzamanlı olarak siler ve böylece atomik bir dosya güncellemesi elde edilir.

39.9 Dosyalar Hakkında Bilgi Alma

Dosya erişiminin ötesinde, dosya sisteminin depoladığı her dosya hakkında makul miktarda bilgi tutmasını bekliyoruz. Dosyalarla ilgili bu tür verileri genellikle **meta veri (metadata)** olarak adlandırırız. Belirli bir dosyanın meta verilerini görmek için `stat()` veya `fstat()` sistem çağrılarını kullanabiliriz. Bu çağrılar, bir dosyaya bir yol adı (veya dosya tanımlayıcı) alır ve Şekil 39.5'te görüldüğü gibi bir istatistik yapısını doldurur.

Her dosya hakkında, boyutu (bayt olarak), düşük düzey adı (yani inode numarası), bazı sahiplik bilgileri ve dosyaya ne zaman erişildiği veya değiştirildiği hakkında bazı bilgiler dahil olmak üzere pek çok bilginin tutulduğunu görebilirsiniz. , Diğer şeylerin yanı sıra. Bu bilgiyi görmek için stat komut satırı aracını kullanabilirsiniz. Bu örnekte, önce bir dosya (dosya adı verilir) oluştururuz

ve ardından dosya hakkında bazı şeyler öğrenmek için stat komut satırı aracını kullanırız.

```
struct stat {

  dev_t     st_dev;     // ID of device containing file
  ino_t     st_ino;     // inode number
  mode_t    st_mode;    // protection
  nlink_t st_nlink; // number of hard links uid_t
      st_uid;      // user ID of owner gid_t st_gid;
      // group ID of owner dev_tst_rdev;     //
  device ID (if special file) off_t    st_size;
      // total size, in bytes
  blksize_t st_blksize; // blocksize for filesystem I/O
  blkcnt_t st_blocks; // number of blocks allocated
  time_t    st_atime; // time of last access
  time_t    st_mtime; // time of last modification
  time_t    st_ctime; // time of last status change
};
```

Şekil 39.5: **İstatistik yapısı.**

İşte Linux'taki çıktı :

```
prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6   Blocks: 8    IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084    Links: 1
Access: (0640/-rw-r-----) Uid: (30686/remzi)
  Gid: (30686/remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Her dosya sistemi genellikle bu tür bilgileri **inode1(dosya numarası)** adı verilen bir yapıda tutar. Dosya sistemi uygulaması hakkında konuşurken düğümler hakkında daha çok şey öğreniyor olacağız. Şimdilik, bir inode'u dosya sistemi tarafından tutulan ve içinde yukarıda gördüğümüz gibi bilgileri içeren kalıcı bir veri yapısı olarak düşünmelisiniz. Tüm düğümler diskte bulunur; etkin olanların bir kopyası, erişimi hızlandırmak için genellikle bellekte önbelleğe alınır.

[1] Bazı dosya sistemleri bu yapıları benzer, ancak biraz farklı adlar olarak adlandırır, örneğin dnodlar; Ancak temel fikir benzerdir.

39.10 Dosyaları Kaldırma

Bu noktada, sıralı veya sırasız olarak dosya oluşturmayı ve bunlara erişmeyi biliyoruz. Ancak dosyaları nasıl silersiniz? UNIX kullandıysanız, muhtemelen bildiğinizi düşünürsünüz: sadece `rm` programını çalıştırın. Ancak `rm` bir dosyayı kaldırmak için hangi sistem çağrısını kullanır? Öğrenmek için eski dostumuz strace'i tekrar kullanalım. İşte o sinir bozucu dosyayı kaldırıyoruz:

`foo`:

```
prompt> strace rm foo
...
unlink("foo")                           = 0
...
```

İzlenen çıktıdan bir grup ilgisiz kırıntıyı kaldırdık ve geriye gizemli bir şekilde adlandırılan `unlink()` sistem çağrısına tek bir çağrı bıraktık. Gördüğünüz gibi, `unlink()` sadece kaldırılacak dosyanın adını alır ve başarı ile sıfır döndürür. Ancak bu bizi büyük bir bilmeceye götürüyor: bu sistem çağrısı neden `unlink` olarak adlandırılıyor? Neden sadece kaldırmıyor veya silmiyorsunuz? Bu bilmecenin cevabını anlamak için önce dosyaları değil, dizinleri de anlamamız gerekir.

39.11 Dizin Oluşturma

Dosyaların ötesinde, dizinle ilgili bir dizi sistem çağrısı, dizin oluşturmanıza, okumanıza ve silmenize olanak tanır. Bir dizine asla doğrudan yazamayacağınızı unutmayın. Dizinin biçimi, dosya sistemi meta verileri olarak kabul edildiğinden, dosya sistemi, dizin verilerinin bütünlüğünden kendisinin sorumlu olduğunu düşünür; bu nedenle, bir dizini yalnızca örneğin içinde dosyalar, dizinler veya başka nesne türleri oluşturarak dolaylı olarak güncelleyebilirsiniz. Bu şekilde dosya sistemi, dizin içeriğinin beklendiği gibi olmasını sağlar.

Bir dizin oluşturmak için tek bir sistem çağrısı olan `mkdir()` kullanılabilir. İsimsiz `mkdir` programı, böyle bir dizin oluşturmak için kullanılabilir. Şimdi `mkdir` adlı basit bir dizin oluşturmak için programı çalıştırdığımızda ne olduğuna bir göz atalım.

`foo`:

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                      = 0
...
prompt>
```

Böyle bir dizin oluşturulduğunda, minimum düzeyde içeriğe sahip olmasına rağmen "boş" olarak kabul edilir. Spesifik olarak, boş bir dizinde iki giriş bulunur: kendisine atıfta bulunan bir giriş ve ebeveynine başvuran bir giriş. İlki "." `(dot)` ve ikincisi ".." `(dot-dot)` dizini olarak anılır. Bu dizinleri programa bir işaret `(-a)` ileterek görebilirsiniz.`Is` :

```
prompt> ls -a
./ ../ prompt>
ls -al total 8
drwxr-x--- 2 remzi remzi    6 Apr 30 16:17 ./ drwxr-
x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

İPUCU: GÜÇLÜ KOMUTLARA KARŞI DİKKATLİ OLUN

`rm` programı bize güçlü komutların harika bir örneğini ve bazen çok fazla gücün ne kadar kötü bir şey olabileceğini gösterir. Örneğin, bir grup dosyayı aynı anda kaldırmak için aşağıdakine benzer bir şey yazabilirsiniz:

```
prompt> rm *
```

Burada * geçerli dizindeki tüm dosyalarla eşleşir. Ancak bazen dizinleri ve aslında tüm içeriklerini de silmek istersiniz.Bunu,`rm '-ye` her dizine özyinelemeli olarak inmesini ve içeriğini de kaldırmasını söyleyerek yapabilirsiniz:

```
prompt> rm -rf *
```

Bu küçük karakter dizisinde başınız derde girdiği zaman, yanlışlıkla bir dosya sisteminin kök dizininden komut verdiğinizde, böylece her dosya ve dizini buradan kaldırmış olursunuz. Hata!

Bu nedenle, güçlü komutların iki ucu keskin kılıcını hatırlayın; size az sayıda tuş vuruşuyla çok iş yapma yeteneği verirken, aynı zamanda hızlı ve kolay bir şekilde büyük zarar verebilirler.

39.12 Okuma Dizinleri

Artık bir dizin oluşturduğumuza göre, bir tane de okumak isteyebiliriz. Aslında `ls` programının yaptığı da tam olarak budur.ls gibi kendi küçük aracımızı yazalım ve nasıl yapıldığını görelim.

Bir dizini bir dosyaymış gibi açmak yerine, bunun yerine yeni bir dizi çağrı kullanırız. Aşağıda, bir dizinin içeriğini yazdıran örnek bir program bulunmaktadır. Program, işi yapmak için `opendir()`, `readdir()` ve `closedir()` olmak üzere üç çağrı kullanır ve arayüzün ne kadar basit olduğunu görebilirsiniz; her seferinde bir dizin girişini okumak ve dizindeki her dosyanın adını ve inode numarasını yazdırmak için basit bir döngü kullanıyoruz.

```
int main(int argc, char *argv[]) {
DIR *dp = opendir("."); assert(dp
!= NULL); struct dirent *d;
    while ((d = readdir(dp)) != NULL) { printf("%lu
        %s\n", (unsigned long) d->d_ino, d-
        >d_name);
    }
    closedir(dp);
    return 0;
}
```

Aşağıdaki bildirim, struct dirent veri yapısındaki her dizin girişinde bulunan bilgileri gösterir:

```
struct dirent {
  char           d_name[256]; // filename
  ino_t          d_ino;       // inode number
  off_t          d_off;       // offset to the next dirent
  unsigned short d_reclen;    // length of this record
  unsigned char d_type; };    // type of file
```

Dizinler bilgi açısından hafif olduğundan (temel olarak, yalnızca birkaç diğer ayrıntıyla birlikte adı inode numarasıyla eşlemek), bir program her dosya hakkında daha fazla bilgi almak için `stat()` işlevini çağırmak isteyebilir, örneğin uzunluğu veya uzunluğu gibi. diğer detaylı bilgiler. Gerçekten de, `-l` bayrağını ilettiğinizde `ls'nin` yaptığı tam olarak budur; Kendiniz görmek için `ls'de` bu bayrakla ve bayraksız strace deneyin.

39.13 Dizinleri Silme

Son olarak, `rmdir()` (aynı adı taşıyan `rmdir` programı tarafından kullanılan) çağrısı olan bir dizini silebilirsiniz. Bununla birlikte, dosya silmenin aksine, dizinleri kaldırmak daha tehlikelidir, çünkü tek bir komutla büyük miktarda veriyi silebilirsiniz. Bu nedenle, `rmdir()` dizinin boş olması şartına sahiptir (yani, yalnızca "." ve ".." girişleri) silinmeden önce. Boş olmayan bir dizini silmeye çalışırsanız, `rmdir()` çağrısı başarısız olur.

39.14 Sabit Bağlantılar

Şimdi, dosya sistemi ağacına giriş yapmanın yeni bir yolunu anlayarak, bir dosyayı kaldırmanın neden `unlink(` aracılığıyla gerçekleştirildiğinin gizemine geri dönüyoruz. `link()` olarak bilinen bir sistem çağrısı aracılığıyla. `link()` sistem çağrısı iki bağımsız değişken alır: eski bir yol adı ve yeni bir

argüman; Yeni bir dosya adını eskisine "bağladığınızda", aslında aynı dosyaya başvurmak için başka bir yol oluşturursunuz. Bu örnekte gördüğümüz gibi, ln komut satırı programı bunu yapmak için kullanılır:

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

Burada içinde "hello" kelimesi olan bir dosya oluşturduk ve `file2` dosyasını çağırdık. Daha sonra `ln` programını kullanarak bu dosyaya sabit bir bağlantı oluştururuz. Bundan sonra ya `file'yi` ya da file2'yi açarak inceleyebiliriz.

`link()'in` çalışma şekli, bağlantıyı oluşturduğunuz dizinde başka bir ad oluşturması ve bunu *aynı* inode numarasına (yani, düşük düzeyli adı) orijinal dosyanın adı. Dosya hiçbir şekilde kopyalanmaz; bunun yerine, artık her ikisi de aynı dosyaya başvuran iki insan tarafından okunabilir adınız (`file` ve `file2`) vardır. Bunu, her dosyanın inode numarasını yazdırarak dizinin kendisinde bile görebiliriz:

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

`-i` bayrağını `ls'ye` ileterek, her dosyanın inode numarasını (dosya adının yanı sıra) yazdırır. Ve böylece, bağlantının gerçekte ne yaptığını görebilirsiniz: sadece aynı inode numarasına (bu örnekte 67158084) yeni bir referans yapın.

Şimdiye kadar `unlink` ()'in neden `unlink()` olarak adlandırıldığını görmeye başlıyor olabilirsiniz. Bir dosya oluşturduğunuzda, gerçekten iki şey yapıyorsunuz demektir. İlk olarak, dosya hakkında hemen hemen tüm ilgili bilgileri izleyecek bir yapı (inode) yapıyorsunuz, bloğunun diskte olduğu boyutunu, ve benzerleri. İkincisi, insan tarafından okunabilen bir adı bu dosyaya bağlıyor ve bu bağlantıyı bir dizine koyuyorsunuz.

Bir dosyaya, dosya sistemine sabit bir bağlantı oluşturduktan sonra, orijinal dosya adı `(file)` ile yeni oluşturulan dosya adı `(file2)` arasında bir fark yoktur; Aslında, her ikisi de sadece inode numarası 67158084 bulunan dosya hakkındaki temel meta verilere bağlantılardır.

2 Bu kitabın yazarlarının ne kadar yaratıcı olduğuna bir kez daha dikkat edin. Bir de "Cat" (gerçek hikaye) adında bir kedimiz vardı. Ancak öldü ve artık "Hammy" adında bir hamsterımız var. Güncelleme: Hammy de artık öldü. Evcil hayvan cesetleri birikiyor.

Bu nedenle, bir dosyayı dosya sisteminden kaldırmak için `unlink()` diyoruz. Yukarıdaki örnekte, örneğin dosya adlı dosyayı kaldırabilir ve yine de dosyaya zorluk çekmeden erişebiliriz:

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

. Bunun çalışmamasının nedeni, dosya sisteminin dosya bağlantısını kaldırdığında, inode numarası içindeki bir **referans sayısını (referance count)** kontrol etmesidir. Bu başvuru sayısı (bazen **bağlantı sayısı (link count)** olarak adlandırılır), dosya sisteminin bu belirli düğüme kaç farklı dosya adının bağlandığını izlemesine olanak tanır. `unlink()` çağrıldığında, insan tarafından okunabilir ad (silinmekte olan dosya) ile verilen inode numarası arasındaki "bağlantıyı" kaldırır ve referans sayısını azaltır; yalnızca referans sayısı sıfıra ulaştığında, dosya sistemi inode'u ve ilgili veri bloklarını serbest bırakır ve böylece dosyayı gerçekten "siler".

Elbette `stat()` kullanarak bir dosyanın referans sayısını görebilirsiniz. Bir dosyaya sabit bağlantılar oluşturup sildiğimizde ne olduğunu görelim. Bu örnekte, aynı dosyaya üç bağlantı oluşturup bunları sileceğiz. Bağlantı sayısını izleyin!

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084    Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084    Links: 2 ...
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084    Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> rm file2
prompt> stat file3
```

```
... Inode: 67158084    Links: 1 ...
prompt> rm file3
```

39.15 Sembolik Bağlantılar

Gerçekten yararlı olan başka bir bağlantı türü daha vardır ve buna **sembolik bağlantı(symbolic link)** veya bazen **yumuşak bağlantı (soft link)**denir. Sabit bağlantılar biraz sınırlıdır: bir dizine bir tane oluşturamazsınız (dizin ağacında bir döngü oluşturacağınızdan korktuğunuz için); diğer disk bölümlerindeki dosyalara sabit bağlantı veremezsiniz (çünkü inode numaraları s, dosya sistemleri arasında değil, yalnızca belirli bir dosya sistemi içinde benzersizdir); vb. Böylece, sembolik bağlantı adı verilen yeni bir bağlantı türü oluşturuldu [MJLF84]. Böyle bir bağlantı oluşturmak için, aynı `ln` programını, ancak `-s` bayrağıyla kullanabilirsiniz. İşte bir örnek:

```
prompt> echo hello >
file prompt> ln -s file
file2 prompt> cat file2
hello
```

Gördüğünüz gibi, esnek bağlantı oluşturmak hemen hemen aynı görünüyor ve orijinal dosyaya artık dosya adı dosyanın yanı sıra sembolik bağlantı adı `file2` yoluyla da erişilebilir.

Ancak, bu yüzeysel benzerliğin ötesinde, sembolik bağlar aslında sabit bağlantılardan oldukça farklıdır. İlk fark, sembolik bir bağın aslında farklı türde bir dosya olmasıdır. Normal dosya ve dizinlerden zaten bahsetmiştik; sembolik bağlar, dosya sisteminin bildiği üçüncü bir türdür. Sembolik bağlantıdaki bir istatistik, hepsini ortaya çıkarır:

```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...
```

`ls` çalıştırmak da bu gerçeği ortaya çıkarır. `ls` çıktısının uzun biçiminin ilk karakterine yakından bakarsanız, en soldaki sütundaki ilk karakterin normal dosyalar için `a-`, dizinler için `d` ve esnek bağlantılar için bir `l` olduğunu görebilirsiniz. Ayrıca sembolik bağın boyutunu (bu durumda 4 bayt) ve bağlantının neyi işaret ettiğini (file adlı dosya) da görebilirsiniz.

```
prompt> ls -al
```

```
drwxr-x--- 2 remzi remzi 29 May 3 19:10 ./ drwxr-x--- 27
remzi remzi 4096 May 3 15:14 ../ -rw-r----- 1 remzi
remzi  6 May 3 19:10 file lrwxrwxrwx 1 remzi remzi   4
May 3 19:10 file2 -> file
```

`File2`'nin 4 bayt olmasının nedeni, sembolik bir bağlantının oluşturulma şeklinin, bağlantılı dosyanın yol adını bağlantı dosyasının verileri olarak tutmaktır. Dosya adlı bir dosyaya bağladığımız için, `file2` bağlantı dosyamız küçüktür (4 bayt). Daha uzun bir yol adına bağlanırsak, bağlantı dosyamız daha büyük olur:

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi 6 May 3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May 3 19:17 file3 ->
                                        alongerfilename
```

Son olarak, sembolik bağlantıların oluşturulma şekli nedeniyle, **sarkan bir referans (dangling reference)**olarak bilinen şeyin olasılığını bırakırlar:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello prompt> rm
file prompt> cat
file2
cat: file2: No such file or directory
```

Bu örnekte görebileceğiniz gibi, sabit bağlantılardan oldukça farklı olarak, dosya adlı özgün dosyanın kaldırılması, bağlantının artık var olmayan bir yol adına işaret etmesine neden olur.

39.16 İzin Bitleri ve Erişim Kontrol Listeleri

Bir işlemin özeti iki merkezi sanallaştırma sağladı: CPU ve bellek. Bunların her biri, kendi özel CPU'suna ve kendi *özel* belleğine sahip olduğu yanılsamasını verdi; Gerçekte, altındaki işletim sistemi çeşitli teknikler kullandı. Sınırlı fiziksel kaynakları rakip kuruluşlar arasında güvenli ve emniyetli bir şekilde paylaşmak.

Dosya sistemi ayrıca bir diskin sanal bir görünümünü sunar ve bu bölümde açıklandığı gibi bir grup ham bloktan çok daha kullanıcı dostu dosya ve dizinlere

dönüştürür. Bununla birlikte, soyutlama, CPU ve belleğinkinden önemli ölçüde farklıdır, çünkü dosyalar farklı kullanıcılar ve işlemler arasında yaygın olarak paylaşılır ve (her zaman) özel. Bu nedenle, dosya sistemlerinde genellikle çeşitli derecelerde o f paylaşımını etkinleştirmek için daha kapsamlı bir mekanizmalar kümesi bulunur.

Bu tür mekanizmaların ilk şekli klasik UNIX **izin bitleridir (permission bits).**

Bir dosya foo.txt izinlerini görmek için şunu yazmanız yeterlidir:

```
prompt> ls -l foo.txt
-rw-r--r-- 1 remzi wheel 0 Aug 24 16:29 foo.txt
```

Bu çıktının sadece ilk kısmına, yani `-rw-r--r--`'ye dikkat edeceğiz. Buradaki ilk karakter sadece dosyanın türünü gösterir: `-` normal bir dosya için (ki bu foo.txt'dir), d bir dizin için, l sembolik bir bağlantı için vb.; bu (çoğunlukla) izinlerle ilgili değildir, bu yüzden şimdilik görmezden geleceğiz.

Sonraki dokuz karakterle (`rw-r--r--`) temsil edilen izin bitleriyle ilgileniyoruz. Bu bitler, her normal dosya, dizin ve diğer varlıklar için, kimlerin ve nasıl erişebileceğini kesin olarak belirler.

İzinler üç gruplamadan oluşur: dosyanın **sahibinin (owner)** dosyaya neler yapabileceği, bir **gruptaki (group)** birinin dosyaya neler yapabileceği ve son olarak, herhangi birinin (bazen başvurulan şey) diğer gibi) yapabilir. Sahibinin, group üyesinin veya başkalarının sahip olabileceği yetenekler, dosyayı okuma, yazma veya yürütme yeteneğini içerir.

Yukarıdaki örnekte, `ls` çıktısının ilk üç karakteri, dosyanın sahibi tarafından hem okunabilir hem de yazılabilir olduğunu (`rw-`) ve yalnızca grup çarkının üyeleri ve ayrıca sistemdeki herhangi biri tarafından okunabileceğini gösterir (`r-- ardından r--`).

Dosyanın sahibi, örneğin chmod komutunu kullanarak **(dosya modunu (file mode)** değiştirmek için) bu izinleri kolayca değiştirebilir. Sahibi dışındaki herkesin dosyaya erişme yeteneğini kaldırmak için şunu yazabilirsiniz:

```
prompt> chmod 600 foo.txt
```

KENARA: DOSYA İSTEMLERİ İÇİN SÜPER KULLANICI

Dosya sisteminin yönetilmesine yardımcı olmak için hangi kullanıcının ayrıcalıklı işlemler yapmasına izin verilir? Örneğin, etkin olmayan bir kullanıcının dosyalarının yer kazanmak için silinmesi gerekiyorsa, bunu yapma hakkına kim sahiptir?

Yerel dosya sistemlerinde, ortak varsayılan, ayrıcalıklardan bağımsız olarak tüm dosyalara erişebilen bir tür **süper kullanıcı(super user) (yani kök (root)** olmasıdır. AFS gibi dağıtılmış bir dosya sisteminde (erişim denetim listelerine sahip),`system:administrators` adlı bir grup, bunu yapmak için güvenilen kullanıcıları içerir. Her iki durumda da, bu güvenilir kullanıcılar doğal bir güvenlik riskini temsil eder; bir saldırgan bir şekilde böyle bir kullanıcının

kimliğine bürünebilirse, sistemdeki tüm bilgilere erişebilir ve böylece beklenen gizlilik ve koruma garantilerini ihlal edebilir.

Bu komut, sahibi için okunabilir bit (4) ve yazılabilir bit (2) etkinleştirir (VEYA bunları birlikte kullanmak yukarıdaki 6'yı verir), ancak grubu ve diğer izin bitlerini 0 ve 0 olarak ayarlayın , sırasıyla, böylece izinleri `rw-------` olarak ayarlar.

Yürütme biti özellikle ilginçtir. Normal dosyalar için, varlığı bir programın çalıştırılıp çalıştırılamayacağını belirler. Örneğin, `hello.csh` adlı basit bir kabuk betiğimiz varsa, aşağıdakileri yazarak çalıştırmak isteyebiliriz:

```
prompt> ./hello.csh
hello, from shell world.
```

Ancak, yürütme bitini bu dosya için düzgün şekilde ayarlamazsak, aşağıdakiler gerçekleşir:

```
prompt> chmod 600 hello.csh
prompt> ./hello.csh
./hello.csh: Permission denied.
```

Yönetmenler için, yürütme biti biraz farklı davranır. Özellikle, bir kullanıcının (veya grubun veya herkesin) verilen dizine dizinleri (yani cd) değiştirme gibi şeyler yapmasını sağlar ve yazılabilir bit ile birlikte, orada dosyalar oluşturun. Bu konuda daha fazla bilgi edinmenin en iyi yolu: kendiniz oynayın. Endişelenmeyin! muhtemelen) hiçbir şeyi çok kötü bir şekilde berbat etmeyeceksiniz.

İzin bitlerinin ötesinde, AFS olarak bilinen (sonraki bir bölümde ele alınacaktır) olarak bilinen dağıtılmış dosya sistemi gibi bazı dosya sistemleri daha karmaşık kontroller içerir. Örneğin AFS, bunu dizin başına bir **erişim kontrol listesi (access control list) (ACL)** şeklinde yapar. Erişim kontrol listeleri, belirli bir kaynağa tam olarak kimlerin erişebileceğini temsil etmenin daha genel ve güçlü bir yoludur. Bir dosya sisteminde bu, yukarıda açıklanan biraz sınırlı sahip/grup/herkes modeli izin bitlerinin aksine, bir kullanıcının bir dizi dosyayı kimin okuyup kimin okuyamayacağına dair çok özel bir liste oluşturmasını sağlar.

Örneğin, `fs listacl` komutuyla gösterildiği gibi, bir author'nin AFS hesabındaki özel bir dizinin erişim denetimleri şunlardır:

```
prompt> fs listacl private
Access list for private is
Normal rights:
```

```
system:administrators rlidwka
remzi rlidwka
```

Listeleme,hem sistem yöneticilerinin hem de remzi kullanıcısının bu dizindeki dosyaları arayabildiğini, ekleyebildiğini, silebildiğini ve yönetebildiğini ve ayrıca bu dosyaları okuyabildiğini, yazabildiğini ve kilitleyebildiğini göstermektedir.

Birinin (bu durumda, diğer yazar) bu dizine erişmesine izin vermek için, remzi kullanıcısı aşağıdaki komutu yazabilir.

```
prompt> fs setacl private/ andrea rl
```

`Remzi'nin` gizliliği var! Ama şimdi daha da önemli bir ders öğrendiniz: iyi bir evlilikte, dosya sisteminde bile sır olamaz3.

39.17 Dosya Sistemi Oluşturma ve Kurma

Şimdi dosyalara, dizinlere ve belirli özel bağlantı türlerine erişmek için temel arabirimleri gezdik. Ancak tartışmamız gereken bir konu daha var: birçok temel dosya sisteminden tam bir dizin ağacının nasıl birleştirileceği. Bu görev, önce dosya sistemleri yapmak ve ardından içeriklerini erişilebilir kılmak için bunları monte etmek yoluyla gerçekleştirilir.

Bir dosya sistemi oluşturmak için çoğu dosya sistemi, tam olarak bu görevi yerine getiren ve genellikle `mkfs` ("make fs" olarak telaffuz edilir) olarak adlandırılan bir araç sağlar. Buradaki fikir şu: araca girdi olarak bir aygıt (örneğin, bir disk bölümü, örneğin `/dev/sda1` gibi) ve bir dosya sistemi türü (örneğin, ext3) verin ve o, basitçe boş bir dosya sistemi yazar. bir kök dizini ile bu disk bölümüne. Ve `mkfs` dedi ki, bir dosya sistemi olsun!

Bununla birlikte, böyle bir dosya sistemi oluşturulduktan sonra, tek tip dosya sistemi ağacında erişilebilir hale getirilmesi gerekir. Bu görev, `mount` programı aracılığıyla gerçekleştirilir (bu, temeldeki sistemin gerçek işi yapması için `mount()` çağrısını yapar). Bağlamanın yaptığı şey, oldukça basit bir şekilde mevcut bir dizini hedef **bağlama noktası (mount point)** olarak almak ve esasen bu noktada dizin ağacına yeni bir dosya sistemi yapıştırmaktır.

Burada bir örnek yararlı olabilir. `/dev/sda1` aygıt bölümünde saklanan ve şu içeriklere sahip olan bağlantısız bir ext3 dosya sistemimiz olduğunu hayal edin: a ve b olmak üzere iki alt dizini içeren bir kök dizin, her biri sırayla `foo` adlı tek bir dosya tutar. Diyelim ki bu dosya sistemini `/home/users` bağlama noktasına bağlamak istiyoruz. Şöyle bir şey yazacağız:

İPUCU: TOCTTOU'YA KARŞI DİKKATLİ OLUN

1974'te McPhee, bilgisayar sistemlerinde bir sorun fark etti. Özellikle, McPhee şunları kaydetti: "... Bir geçerlilik denetimi ile bu geçerlilik denetimiyle bağlantılı işlem arasında bir zaman aralığı varsa, [ve,] çoklu görev yoluyla,

[3] 1996'dan beri mutlu bir şekilde evlendim, merak ediyorsanız. Biliyoruz, sen değildin.

geçerlilik denetimi değişkenleri kasıtlı olarak bu zaman aralığında değişti ve kontrol programı tarafından geçersiz bir işlem gerçekleştirilmesine neden oldu." Bugün buna **Kullanım Zamanına Kontrol Zamanı (Time Of Check To Time Of Use) (TOCTTOU)** sorunu diyoruz ve ne yazık ki, hala ortaya çıkabilir.

Bishop ve Dilger [BD96] tarafından açıklandığı gibi basit bir örnek, Bir kullanıcının daha güvenilir bir hizmeti nasıl kandırabileceğini ve böylece sorun yaratabileceğini gösterir. Örneğin, bir posta hizmetinin kök olarak çalıştığını (ve böylece bir sistemdeki tüm dosyalara erişme ayrıcalığına sahip olduğunu) düşünün. Bu hizmet, kullanıcının gelen kutusu dosyasına aşağıdaki gibi bir iletişim iletisi ekler. İlk olarak , dosya hakkında bilgi almak için `lstat()` öğesini çağırır, özellikle de dosya aslında hedef kullanıcıya ait normal bir dosya olduğundan ve dosya posta sunucusunun güncelleştirmemesi gereken başka bir dosya. Ardından, denetim başarılı olduktan sonra, sunucu dosyayı yeni iletiyle güncelleştirir.

Ne yazık ki, kontrol ve güncelleme arasındaki boşluk bir soruna yol açar: saldırgan (bu durumda, postayı alan kullanıcı ve bu nedenle izin veren kullanıcı) gelen kutusuna erişmek için) gelen kutusu dosyasını (`rename()`'ye yapılan bir çağrı aracılığıyla) `/etc/passwd` gibi hassas bir dosyaya işaret edecek şekilde değiştirir (kullanıcılar ve şifreleri hakkında bilgi tutar). Bu geçiş doğru zamanda gerçekleşirse (kontrol ve ccess arasında), sunucu hassas dosyayı posta. Saldırgan artık bir e-posta, ayrıcalıkta bir yükseltme göndererek hassas dosyaya yazabilir; saldırgan `/etc/passwd` dosyasını güncelleyerek root privileges ile bir hesap ekleyebilir ve böylece sistemin kontrolünü ele geçirebilir.

TOCTTOU problemine [T+08] basit ve harika çözümler yoktur. Bir yaklaşım, çalıştırmak için kök ayrıcalıklarına ihtiyaç duyan hizmetlerin sayısını azaltmaktır, bu da yardımcı olur. ONOFOLLOW bayrağı, hedef sembolik bir bağlantıysa `open()` başarısız olacak şekilde yapar, böylece söz konusu bağlantıları gerektiren saldırılardan kaçınır. **İşlemsel bir dosya sistemi (transactional file system)** [H + 18] kullanmak gibi daha radikal yaklaşımlar sorunu çözecektir, geniş dağıtımda çok fazla işlemsel dosya sistemi yoktur. Bu nedenle, olağan (topal) tavsiye: yüksek ayrıcalıklarla çalışan kod yazarken dikkatli olun!

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

Başarılı olursa, bağlama böylece bu yeni dosya sistemini kullanılabilir hale getirir. Ancak, yeni dosya sistemine nasıl erişildiğine dikkat edin. Kök dizinin içeriğine bakmak için `ls` şu şekilde kullanırız:

```
prompt> ls /home/users/
a b
```

Gördüğünüz gibi,`/home/users/` yol adı artık yeni bağlanan dizinin kökünü ifade eder. Benzer şekilde, a ve b dizinlerine `/home/users/a` ve `/home/users/b` yol adlarıyla erişebiliriz. Son olarak, foo adlı dosyalara `/home/users/a/ foo` ve `/home/users/b/foo` üzerinden erişilebilir. Ve böylece montajın güzelliği: bir dizi ayrı dosya sistemine sahip olmak yerine, mount tüm dosya sistemlerini tek bir ağaçta birleştirerek adlandırmayı tek tip hale getirir. ve kullanışlı.

Sisteminize neyin ve hangi noktalarda monte edildiğini görmek için, montaj programını çalıştırmanız yeterlidir. Şunun gibi bir şey görürsünüz:

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw) /dev/sda7
on /var/vice/cache type ext3 (rw) tmpfs on
/dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

Bu çılgın karışım, ext3 (standart bir disk tabanlı dosya sistemi), proc dosya sistemi (mevcut işlemler hakkındaki bilgilere erişmek için bir dosya sistemi), tmpfs (yalnızca geçici dosyalar için bir dosya sistemi) dahil olmak üzere çok sayıda farklı dosya sisteminin olduğunu gösterir.) ve AFS (dağıtılmış bir dosya sistemi) bu tek makinenin dosya sistemi ağacına yapıştırılmıştır.

39.18 Özet

UNIX sistemlerindeki (ve aslında herhangi bir sistemdeki) dosya sistemi arabirimi görünüşte oldukça ilkeldir, ancak bunda ustalaşmak istiyorsanız anlamanız gereken çok şey vardır. Elbette, onu (çok) kullanmaktan daha iyi bir şey yoktur. O yüzden lütfen öyle yapın! Tabii ki daha fazlasını okuyun; her zaman olduğu gibi, Stevens [SR05] başlangıç yeridir.

Bir KENARA: ANAHTAR DOSYA SİSTEM ŞARTLARI

- **Dosya (File),** oluşturulabilen, okunabilen, yazılabilen ve silinebilen bir bayt dizisidir. Benzersiz bir şekilde atıfta bulunan düşük seviyeli bir adı (yani bir sayı) vardır. Düşük seviyeli ada genellikle **i-numarası (i-number)** denir.
- **Dizin**, her biri insan tarafından okunabilir bir ad ve eşlendiği düşük düzeyli bir ad içeren bir kümeler koleksiyonudur. Her girdi başka bir dizine veya bir dosyaya başvurur. Her dizinin kendisi de düşük seviyeli bir ada (i-numarası) sahiptir. Bir dizinin her zaman iki özel girişi vardır:.giriş, kendisini ifade eder ve ..giriş, ana şirketine atıfta bulunur.
- Bir **dizin ağacı** veya **dizin hiyerarşisi,** tüm dosyaları ve dizinleri **kökünden** başlayarak büyük bir ağaç halinde düzenler.

- Bir dosyaya erişmek için, işlemin işletim sisteminden izin istemek üzere bir sistem çağrısı (genellikle `open()`) kullanması gerekir. İzin verilirse, işletim sistemi izinler ve çadırda izin verildiği gibi okuma veya yazma erişimi için kullanılabilecek bir **dosya tanımlayıcısı (file descriptor)** döndürür.
- Her dosya tanıtıcı, **açık dosya tablosundaki (open file table)** bir girişi ifade eden özel, işlem başına bir varlıktır. Buradaki giriş, bu erişimin hangi dosyaya atıfta bulunduğunu, dosyanın **geçerli ofsetini(current offset)** (yani, bir sonraki okuma veya yazmanın dosyanın hangi bölümüne erişeceğini) ve diğer ilgili bilgileri izler.
- `read()` ve `write()` çağrıları geçerli ofseti doğal olarak günceller; aksi takdirde, işlemler değerini değiştirmek için `lseek()` öğesini kullanabilir ve dosyanın farklı bölümlerine rasgele erişim sağlayabilir.
- Kalıcı medyada güncellemeleri zorlamak için, bir işlemin `fsync()` veya ilgili çağrıları kullanması gerekir. Ancak, yüksek performansı korurken bunu doğru bir şekilde yapmak zordur [P + 14], bu yüzden bunu yaparken dikkatlice düşünün.
- Dosya sisteminde insan tarafından okunabilen birden çok isme sahip olmak için aynı temel dosyaya atıfta bulunun, **sabit bağlantılar(hard links)** veya **sembolik bağlantılar (symbolic links)** kullanın. Her biri farklı koşullarda yararlıdır, bu nedenle kullanmadan önce güçlü ve zayıf yönlerini göz önünde bulundurun. Ve unutmayın, bir dosyayı silmek, onun dizin hiyerarşisinden son bir `unlink()` gerçekleştirmektir.
- Çoğu dosya sistemi, paylaşımı etkinleştirmek ve devre dışı bırakmak için mekanizmalara sahiptir. Bu tür denetimlerin temel bir biçimi **izin bitleri (premissions bit)** tarafından sağlanır; daha karmaşık **erişim kontrol listeleri (acces control list),** oluşum sırasında kimlerin erişebileceği ve manipüle edebileceği konusunda tam olarak daha kesin kontrol sağlar.

Referans

[BD96] "Checking for Race Conditions in File Accesses" by Matt Bishop, Michael Dilger. Computing Systems 9:2, 1996. *A great description of the TOCTTOU problem and its presence in file systems.*

[CK+08] "The xv6 Operating System" by Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. From: https://github.com/mit-pdos/xv6-public. *As mentioned before, a cool and simple Unix implementation. We have been using an older version (2012-01-30-1-g1c41342) and hence some examples in the book may not match the latest in the source.*

[H+18] "TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions" by Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, E. Witchel. USENIX ATC '18, June 2018. *The best paper at USENIX ATC '18, and a good recent place to start to learn about transactional file systems.*

[K19] "Persistent Memory Programming on Conventional Hardware" by Terence Kelly. ACM Queue, 17:4, July/August 2019. *A great overview of persistent memory programming; check it out!*

[K20] "Is Persistent Memory Persistent?" by Terence Kelly. Communications of the ACM, 63:9, September 2020. *An engaging article about how to test hardware failures in system on the cheaps; who knew breaking things could be so fun?*

[K84] "Processes as Files" by Tom J. Killian. USENIX, June 1984. *The paper that introduced the /proc file system, where each process can be treated as a file within a pseudo file system. A clever idea that you can still see in modern* UNIX *systems.*

[L84] "Capability-Based Computer Systems" by Henry M. Levy. Digital Press, 1984. Available: http://homes.cs.washington.edu/~levy/capabook. *An excellent overview of early capability-based systems.*

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM TOCS, 2:3, August 1984. *We'll talk about the Fast File System (FFS) explicitly later on. Here, we refer to it because of all the other random fun things it introduced, like long file names and symbolic links. Sometimes, when you are building a system to improve one thing, you improve a lot of other things along the way.*

[P+13] "Towards Efficient, Portable Application-Level Consistency" by Thanumalayan S. Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. ArpaciDusseau. HotDep '13, November 2013. *Our own work that shows how readily applications can make mistakes in committing data to disk; in particular, assumptions about the file system creep into applications and thus make the applications work correctly only if they are running on a specific file system.*

[P+14] "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications" by Thanumalayan S. Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. OSDI '14, Broomfield, Colorado, October 2014. *The full conference paper on this topic – with many more details and interesting tidbits than the first workshop paper above.*

[SK09] "Principles of Computer System Design" by Jerome H. Saltzer and M. Frans Kaashoek. Morgan-Kaufmann, 2009. *This tour de force of systems is a must-read for anybody interested in the field. It's how they teach systems at MIT. Read it once, and then read it a few more times to let it all soak in.*

[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens and Stephen A. Rago. Addison-Wesley, 2005. *We have probably referenced this book a few hundred thousand times. It is that useful to you, if you care to become an awesome systems programmer.*

[T+08] "Portably Solving File TOCTTOU Races with Hardness Amplification" by D. Tsafrir, T. Hertz, D. Wagner, D. Da Silva. FAST '08, San Jose, California, 2008. *Not the paper that introduced TOCTTOU, but a recent-ish and well-done description of the problem and a way to solve the problem in a portable manner.*

Ödev (Kod)

Bu ödevde, bölümde açıklanan API'lerin nasıl çalıştığını öğreneceğiz. Bunu yapmak için, çoğunlukla çeşitli UNIX yardımcı programlarına dayanan birkaç farklı program yazacaksınız.

Soru

1. Stat: Belirli bir dosya veya dizindeki `stat()` sistem çağrısını çağıran komut satırı program `stat'ının` kendi sürümünüzü yazın. Dosya boyutunu, ayrılan blokların nu mber'ini, referans (bağlantı) sayısını vb. Yazdırın. Dizindeki girişlerin sayısı değiştikçe dizinin bağlantı sayısı nedir? Kullanışlı arayüzler:
   stat(), doğal olarak.

   CEVAP1:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>


int main(int argc, char **argv){

  struct stat dirstat;
  stat(argv[1],&dirstat);

  printf("File to be analyzed: %s\n", (argv[1]));
  printf("File Size: %lld\n",dirstat.st_size);
  printf("Number of blocks allocated:
%lld\n",dirstat.st_blocks);
  printf("Reference link count:
%d\n",dirstat.st_nlink);
  printf("File Permissions: ");
  if (S_ISDIR(dirstat.st_mode)){
    printf("Directory - ");
  }
  if (dirstat.st_mode & S_IRUSR){
    printf("read permission, owner - ");
  }
  if (S_IWUSR & dirstat.st_mode){
    printf("write permission, owner - ");
  }
```

```
if (dirstat.st_mode & S_IXUSR){
  printf("execute/search permission, owner - ");
}
if (dirstat.st_mode & S_IRGRP){
  printf("read permission, group - ");
}
if (dirstat.st_mode & S_IWGRP){
  printf("write permission, group - ");
}
if (dirstat.st_mode & S_IXGRP){
  printf("execute/search permission, group - ");
}
if (dirstat.st_mode & S_IROTH){
  printf("read permission, others - ");
}
if (dirstat.st_mode & S_IWOTH){
  printf("write permission, others - ");
}
if (dirstat.st_mode & S_IXOTH){
  printf("execute/search permission, others - ");
}
if (dirstat.st_mode & S_ISUID){
  printf("set-user-ID on execution - ");
}
if (dirstat.st_mode & S_ISGID){
  printf("set-group-ID on execution - ");
}
if (dirstat.st_mode & S_ISVTX){
```

```
    printf("on directories, restricted deletion flag -
");
  }
  printf("\nFile inode: %llu\n",dirstat.st_ino);

}
```

2. **Dosyaları Listele:** Verilen dizindeki dosyaları listeleyen bir program yazın. Herhangi bir argüman olmadan çağrıldığında, program sadece dosya adlarını yazdırmalıdır. `-l` bayrağıyla çağrıldığında, program her dosya hakkında sahip, grup, izinler ve `stat()` sistem çağrısından elde edilen diğer bilgiler gibi bilgileri yazdırmalıdır. Program, örneğin `myls -l` dizini gibi okunacak dizin olan bir ek argüman almalıdır. Dizin verilmezse, program yalnızca geçerli çalışma dizinini kullanmalıdır. Yararlı arayüzler: `stat(),opendir(),readdir(), getcwd().`

3. **Kuyruk:** Bir dosyanın son birkaç satırını yazdıran bir program yazın . Program, dosyanın sonuna yaklaşmaya çalıştığı , bir veri bloğunda okuduğu ve daha sonra istenen satır sayısı; bu noktada, bu satırları dosyanın başından sonuna kadar yazdırmalıdır. Programı çağırmak için şunu yazmalısınız: mytail -n dosyası, burada n, yazdırılacak dosyanın sonundaki satır sayısıdır . Kullanışlı arayüzler: `stat(), lseek(), open(), read() , close().`

   CEVAP 3:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <dirent.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <stdio.h>



int main(int argc, char **argv){

  DIR *direct;
  direct = opendir(".");
  int n = 0, lines = 0, i;
  FILE *file;

  char c, *name;



  if (argc > 1){
    if (argv[1][0] == '-'){
      char *str;
      str = argv[1];
      str++;
      n = atoi(str);
    } else {
      name = argv[1];
      file = fopen(argv[1],"r");
    }
    if (argc > 2){
```

```
    if (argv[2][0] == '-'){
      char *str;
      str = argv[2];
      str++;
      n = atoi(str);
    } else {
      name = argv[2];
      file = fopen(argv[2],"r");
    }
  }
}

while (!feof(file)){
  c = fgetc(file);
  if(c == '\n'){
    lines++;
  }
}

fclose(file);
file = fopen(name, "r");

i=0;

while (i < lines-n){
  c = fgetc(file);
  if(c == '\n'){
```

```
      i++;
    }
  }

  while (1==1){
    c = fgetc(file);
    if (!feof(file)){
      printf("%c", c);
    } else {
      break;
    }
  }
  printf("\n");
  fclose(file);
}
```

4. **Özyinelemeli Arama:** Dosya sistemi ağacındaki her dosya ve dizinin adını, ağacın belirli bir noktasından başlayarak yazdıran bir program yazın. Örneğin, bağımsız değişkenler olmadan çalıştırıldığında , program geçerli çalışma diziniyle başlamalı ve içeriğini ve herhangi bir alt dizinler, vb., CWD' deki kökü olan tüm ağaç yazdırılana kadar. Tek bir bağımsız değişken (dizin adı) verilirse , bunun yerine ağacın kökü olarak bunu kullanın. Özyinelemeli aramamızı, güçlü bul komut satırı aracına benzer şekilde daha eğlenceli seçeneklerle hassaslaştırın. Kullanışlı arayüzler: bunu anlayın.

CEVAP 4:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <dirent.h>
```

```
#include <string.h>
#include <stdio.h>
#define MAX 10000

void explore(char *name){
  DIR *direct;

  direct = opendir(name);
  struct stat dirstat;
  char directories[MAX];
  struct dirent *strdir;


  strdir = readdir(direct);

  while (strdir != NULL){

    if  (strcmp(strdir->d_name,  ".")  !=  0  &&
strcmp(strdir->d_name, "..") != 0){
      printf("%s\n", strdir->d_name);
      stat(strdir->d_name,&dirstat);
      if (S_ISDIR(dirstat.st_mode)){
        strcpy(directories, name);
        strcat(directories, "/");
        strcat(directories, strdir->d_name);
        printf("%sdirect:\n", directories);
        explore(directories);
      }
    }
    strdir = readdir(direct);
  }
  closedir(direct);
  printf("%s\n", name);
}

int main(int argc, char **argv){


  DIR *direct;
  char *name;
  int count=0;

  if (argc > 1){
    name = argv[1];
  }  else {
```

```
    name = ".";
  }

 direct = opendir(name);
 struct stat dirstat;
 printf("Files: \n");

 char directories[MAX];
 struct dirent *strdir;


 strdir = readdir(direct);
 while (strdir != NULL){
  if (strcmp(strdir->d_name, ".") != 0 && strcmp(strdir->d_name, "..") !=
0){
    printf("%s\n", strdir->d_name);
    stat(strdir->d_name,&dirstat);
    if (S_ISDIR(dirstat.st_mode)){
      strcpy(directories, name);
      strcat(directories, "/");
      strcat(directories, strdir->d_name);
      printf("%sdirect:\n", directories);
      explore(directories);
    }
  }
  strdir = readdir(direct);
 }
 closedir(direct);
 printf("%s\n", name);


}
```