39

Dosyalar Ve Dizinler

Şimdiye kadar iki temel işletim sistemi soyutlamasının geliştirildiğini gördük: işlem, yani CPU nun sanallaştırılması, ve adres alanı, yani belleğin sanallaştırılması. Kısacası bu iki soyutlama, programın sanki kendi işlemcisi (veya işlemcileri) varmış gibi; sanki kendi hafızası varmış gibi, kendi özel,izole dünyasındaymış gibi çalışmasına izin verir. Sonuç olarak bu yanılsama, sistemin programlanmasını çok daha kolay hale getirir ve bu nedenle günümüzde yalnızca masaüstü biligsayarlarda ve sunucularda değil, cep telefonları ve benzeri tüm programlanabilir platformlarda giderek yaygınlaşmaktadır.

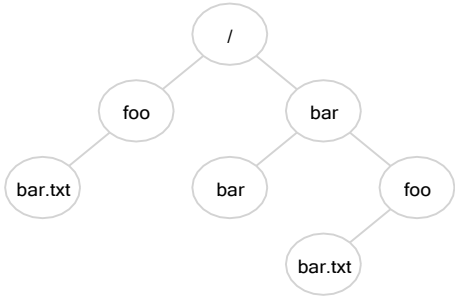
Bu bölümde sanallaştırmaya bir kritik parça daha ekliyoruz. puzzle: **kalıcı depolama (persistent storage)**. Kalıcı depolama cihazı, yani bir klasik olan **sabit disk sürücüsü (hard disk drive)** ya da daha modern olan **katı hâl sürücüsü (solid-state storage)**, bir bilgiyi sonsuza dek saklar.(ya da en azından uzunca bir süre). Elektrik kesildiğinde içeriği kaybolan belleğin aksine kalıcı depolama birimi bütün içeriği olduğu gibi tutar. Bu yüzden işletim sisteminin, kullanıcıların gerçekten önemsediği verilerini tuttuğu bu cihaza ekstra bir önem göstermesi gerekir.

En Önemli Soruya Gelelim: Kalıcı Bir Cihaz Nasıl Yönetilir
İşletim sistemi kalıcı bir cihazı nasıl yönetmeli? API ler nelerdir?
Bu uygulamaların en önemli noktaları nelerdir?

Sonraki birkaç bölümde, performansı ve güvenilirliği artırma yöntemlerine odaklanarak, kalıcı verileri yönetmek için kritik teknikleri keşfedeceğiz. UNIX dosya sistemiyle etkileşim kurarken göreceğiniz arayüzler olan APIye genel bir bakışla başlıyoruz.

39.1 Dosyalar ve Dizinler

Depolamanın sanallaştırılmasında zaman içinde iki temel soyutlama gelişmiştir. Bunlardan ilki **dosya (file)**. Dosya, her birini okuyabileceğiniz veya yazabileceğiniz doğrusal bir bayt dizisidir. Her dosya bir çeşit **alt düzey ad (low-level name)** sahiptir. Bu isim(göreceğimiz üzere) çoğunlukla kullanıcının farkında olmadığı, genellikle bir çeşit sayıdır.



Şekil 39.1: **Bir Dizin Ağacı Örneği**

Tarihsel nedenlerden dolayı, bir dosyanın alt düzey adı genellikle onun **düğüm numarası(inode number)** olarak adlandırılır.Önümüzdeki bölümlerde düğüm numaraları hakkında daha fazlasını öğreneceğiz; şimdilik, her dosyanın kendisiyle ilişkilendirilmiş bir düğüm numarası olduğunu varsayalım.

Çoğu sistemde, işletim sistemi dosyanın yapısı hakkında pek bir şey bilmez (örneğin, resim mi, metin dosyası mı yoksa C kodu mu); bunun yerine, dosya sisteminin sorumluluğu, bu tür verileri kalıcı olarak diskte depolamak ve verileri tekrar talep ettiğinizde, ilk etapta oraya koyduğunuz şeyi aldığınızdan emin olmaktır. Bunu yapmak göründüğü kadar basit değil!

İkinci soyutlama, bir **dizin(Directory)** soyutlamasıdır. Bir dosya gibi bir dizinin de düşük seviyeli bir adı (yani bir düğüm numarası) vardır, ancak içeriği oldukça belirgindir: (kullanıcı tarafından okunabilen ad, düşük seviyeli ad) çiftlerinin bir listesini içerir.Örneğin, alt düzey adı "10" olan bir dosya olduğunu ve kullanıcı tarafından okunabilen "foo" adıyla anıldığını varsayalım.Dolayısıyla, "foo"nun bulunduğu dizinde, kullanıcı tarafından okunabilen alt düzey adla eşleşen bir giriş ("foo", "10") olacaktır.Bir dizindeki her giriş, dosyalara veya diğer dizinlere atıfta bulunur. Dizinleri diğer dizinlerin içine yerleştirerek, kullanıcılar, altında tüm dosya ve dizinlerin depolandığı keyfi bir **dizin ağacı(directory tree) (veya dizin hiyerarşisi(directory hierarchy))** oluşturabilirler.

Dizin hiyerarşisi bir **kök dizinde(root directory**) başlar (UNIX tabanlı sistemlerde, kök dizin basitçe / olarak ifade edilir) ve istenen dosya veya dizin isimlendirilene kadar sonraki **alt dizinleri(sub directories)** adlandırmak için bir tür **ayırıcı(separator)** kullanır. Örneğin, bir kullanıcı / kök dizininde bir foo dizini oluşturduysa ve ardından foo dizininde bir bar.txt dosyası oluşturduysa, dosyaya **mutlak yol(absolute pathname)** adıyla başvurabiliriz, bu durumda bu, `/foo/bar.txt` olacaktır. Daha karmaşık bir dizin ağacı için bkz. Şekil 39.1; örnekteki geçerli dizinler `/`, `/foo`, `/bar`, `/bar/bar`, `/bar/foo` ve geçerli dosyalar `/foo/bar.txt` ve `/bar/foo/bar.txt`.

İPUCU: İSİM VERME KONUSUNDA DİKKATLİCE DÜŞÜNÜN
Adlandırma, bilgisayar sistemlerinin [SK09] önemli bir yönüdür. UNIX sistemlerinde, aklınıza gelebilecek hemen hemen her şey dosya sistemi aracılığıyla adlandırılır. Yalnızca dosyaların ötesinde, aygıtlar, kanallar ve hatta işlemler [K84], düz bir eski dosya sistemi gibi görünen şeylerde bulunabilir. Adlandırmadaki bu tekdüzelik, sistemin kavramsal modelini kolaylaştırır ve sistemi daha basit ve daha modüler hale getirir. Bu nedenle, bir sistem veya arayüz oluştururken kullandığınız adları dikkatlice düşünün.

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

Ayrıca, bu örnekteki dosya adının genellikle iki bölümden oluştuğunu da fark edebilirsiniz: `bar` ve `txt`, noktayla ayrılmış. İlk bölüm rastgele bir addır, oysa dosya adının ikinci bölümü genellikle dosyanın türünü belirtmek için kullanılır, örneğin C kodu mu (ör. .c) veya bir görüntü mü (ör. . `jpg`) veya bir müzik dosyası (örn. .`mp3`). Ancak, bu genellikle yalnızca bir kuraldır: `main.c` adlı bir dosyada bulunan verilerin gerçekten de C kaynak kodu olduğuna dair genellikle bir yaptırım yoktur.

Böylece, dosya sistemi tarafından sağlanan harika bir şeyi görebiliriz: ilgilendiğimiz tüm dosyaları **adlandırmanın** uygun bir yolu. Herhangi bir kaynağa erişmenin ilk adımı onu adlandırabilmek olduğundan, adlar sistemlerde önemlidir. UNIX sistemlerinde, dosya sistemi disk, USB bellek, CD-ROM, diğer pek çok aygıt ve aslında pek çok başka şey üzerindeki dosyalara tek bir dizin ağacı altında erişmek için birleşik bir yol sağlar.

39.2 Dosya Sistemi Arayüzü

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

39.3 Dosya Oluşturma

En temel işlemlerle başlayacağız: bir dosya oluşturmak. Bu, açık sistem çağrısı ile gerçekleştirilebilir; `open()` öğesini çağırarak ve ona `O_CREAT` bayrağını ileterek, bir program yeni bir dosya oluşturabilir. Geçerli çalışma dizininde "foo" adlı bir dosya oluşturmak için bazı örnek kodlar:

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

C R E A T() SİSTEM ÇAĞRISI

Bir dosya oluşturmanın eski yolu, aşağıdaki gibi `creat()` işlevini çağırmaktır:

```
// seçenek: izinleri ayarlamak için ikinci bayrak ekleyin
int fd = creat("foo");                                    -
```

`creat()`'i aşağıdaki bayraklarla open() olarak düşünebilirsiniz: `O_CREAT | O_WRONLY |O_TRUNC. open()` bir dosya oluşturabildiğinden, `creat()`'nin kullanımı bir şekilde gözden düşmüştür (aslında, sadece `open()`'a bir kütüphane çağrısı olarak uygulanabilir); ancak UNIX bilgisinde özel bir yeri vardır. Özellikle, Ken Thompson'a UNIX'i yeniden tasarlıyor olsaydı neyi farklı yapacağı sorulduğunda, "creat'ı e ile hecelerdim" yanıtını verdi.

`open()` rutini bir dizi farklı bayrak alır.Bu örnekte, ikinci parametre dosya yoksa `(O_CREAT)` oluşturur, dosyanın yalnızca `(O_WRONLY)` üzerine yazılabilmesini sağlar ve dosya zaten varsa, dosyanın boyutunu sıfır bayta küçülterek mevcut içeriği kaldırır (`O_TRUNC`) **.**Üçüncü parametre izinleri belirtir, bu durumda dosyayı sahibi tarafından okunabilir ve yazılabilir hale getirir.

`open()` işlevinin önemli bir yönü, döndürdüğü şeydir: bir **dosya tanımlayıcı(file descriptor)**.Bir dosya tanımlayıcı yalnızca bir tamsayıdır, işlem başına özeldir ve UNIX sistemlerinde dosyalara erişmek için kullanılır; bu nedenle, bir dosya açıldığında, izniniz olduğunu varsayarak dosyayı okumak veya yazmak için dosya tanımlayıcıyı kullanırsınız.Bu şekilde, dosya tanımlayıcı size belirli işlemleri gerçekleştirme gücü verme **kapasitesine** sahip olan opak bir tanıtıcıdır. Bir dosya tanımlayıcıyı düşünmenin başka bir yolu, dosya türündeki bir nesneye yönelik bir işaretçi olarak düşünmektir; böyle bir nesneye sahip olduğunuzda, dosyaya erişmek için `read()` ve `write()` gibi diğer "metodları" çağırabilirsiniz (bunu nasıl yapacağımızı aşağıda göreceğiz).

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

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

Basit bir dizi (maksimum `NOFILE` açık dosyalarıyla), işlem bazında hangi dosyaların açıldığını izler. Dizinin her girişi, aslında, okunan veya yazılan dosya hakkındaki bilgileri izlemek için kullanılacak olan bir yapı dosyasına yönelik yalnızca bir işaretçidir; Bunu aşağıda daha ayrıntılı olarak tartışacağız.

TIP: STRACE Kullanımı (Ve Benzer Araçlar)

`Strace` aracı, programların neler yaptığını görmenin harika bir yolunu sunar. Çalıştırarak, bir programın hangi sistem çağrılarını yaptığını izleyebilir, bağımsız değişkenleri ve dönüş kodlarını görebilir ve genel olarak neler olup bittiğine dair çok iyi bir fikir edinebilirsiniz.

Araç aynı zamanda oldukça faydalı olabilecek bazı argümanları da alır. Örneğin, `-f` forklanmış çocukları da takip eder; `-t` her aramada günün saatini bildirir; `-e trace=open,close,read,write` yalnızca bu sistem çağrılarına yapılan çağrıları izler ve diğerlerini yok sayar. Başka birçok bayrak var; kılavuz sayfalarını okuyun ve bu harika aracı nasıl kullanacağınızı öğrenin.

39.4 Dosyaları Okuma ve Yazma

Bazı dosyalara sahip olduğumuzda, elbette onları okumak veya yazmak isteyebiliriz. Var olan bir dosyayı okuyarak başlayalım. Bir komut satırında yazıyor olsaydık, dosyanın içeriğini ekrana dökmek için `cat` programını kullanabilirdik.

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

Bu kod parçacığında, programın çıktısını, içinde "hello" kelimesini içeren foo dosyasına yönlendiriyoruz.Daha sonra dosyanın içeriğini görmek için cat kullanırız. Peki `cat` programı `foo` dosyasına nasıl erişiyor?

Bunu bulmak için, bir program tarafından yapılan sistem çağrılarını izlemek için inanılmaz kullanışlı bir araç kullanacağız.Linux'ta aracın adı strace'dir; diğer sistemlerde benzer araçlar bulunur (Mac'te dtruss'a veya bazı eski UNIX türevlerinde truss'a bakın).`Strace`'in yaptığı, bir program çalışırken yaptığı her sistem çağrısını izlemek ve izlemeyi görmeniz için ekrana dökmektir.İşte `cat`'ın ne yaptığını anlamak için `strace` kullanımına 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`'ın yaptığı ilk şey dosyayı okumak için açmaktır. Bununla ilgili not etmemiz gereken birkaç şey; ilk olarak, `O_RDONLY` bayrağıyla belirtildiği gibi, dosyanın yalnızca okumak (yazmak için değil) için açıldığı; ikincisi, 64-bit ofsetin kullanılması `(O_LARGEFILE)`; üçüncüsü, `open()` çağrısı başarılı olur ve 3 değerine sahip bir dosya tanımlayıcısı döndürür.

Neden ilk `open()` çağrısı beklediğiniz gibi 0 veya 1 değil de 3 döndürür?Görünen o ki, çalışan her işlemin zaten üç açık dosyası vardır, standart girdi (işlemin girdi almak için okuyabileceği), standart çıktı (işlemin ekrana bilgi dökmek için yazabileceği) ve standart hata (işlemin hata mesajları yazabileceği).Bunlar sırasıyla 0, 1 ve 2 dosya tanımlayıcıları ile temsil edilir. Bu nedenle, başka bir dosyayı ilk açtığınızda (yukarıda `cat`'ın yaptığı gibi), neredeyse kesinlikle dosya tanıtıcı 3 olacaktır.

Açma işlemi başarılı olduktan sonra `cat`, bir dosyadan bazı baytları tekrar tekrar okumak için `read()` sistem çağrısını kullanır.`read()` için ilk bağımsız değişken dosya tanımlayıcıdır, böylece dosya sistemine hangi dosyayı okuyacağını söyler; bir işlemde aynı anda birden çok dosya açık olabilir ve bu nedenle 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 başarılı bir şekilde geri döner, burada okuduğu bayt sayısını döndürür "hello" kelimesindeki harfler için 5 ve satır sonu işaretçisi için bir tane olmak üzere 6).

Bu noktada, strace'in başka bir ilginç sonucunu görürsünüz: `write()` sistem çağrısına, dosya tanımlayıcı 1'e yapılan tek bir çağrı. Yukarıda da belirttiğimiz gibi, bu tanımlayıcı standart çıktı olarak bilinir ve bu nedenle yazmak için kullanılır. `cat` programı kullanılarak ekrana "hello” kelimesinin yazdırılması amaçlanmıştır.

Ancak doğrudan `write()`'ı çağırıyor mu? Belki (eğer yüksek oranda optimize edilmişse).Ancak değilse, `cat`'in yapabileceği şey kitaplık yordamını `printf()`; olarak çağırmak olabilir. dahili olarak, `printf()` kendisine iletilen tüm biçimlendirme ayrıntılarını hesaplar ve sonunda sonuçları ekrana yazdırmak için standart çıktıya yazar.`cat` programı daha sonra dosyadan daha fazlasını okumaya çalışır, ancak dosyada hiç bayt kalmadığından, `read()` 0 döndürür ve program bunun tüm dosyayı okuduğu anlamına geldiğini bilir. Böylece program, "`foo`" dosyasıyla yapıldığını belirtmek için `close()`'u çağırır ve karşılık gelen dosya tanımlayıcısını iletir. Dosya böylece kapatılır ve okunması tamamlanır.

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

Veri Yapısı — Açık Dosya Tablosu
Her işlem, her biri sistem çapında **açık dosya tablosundaki(open file table)** bir girişi ifade eden bir dizi dosya tanımlayıcı tutar.Bu tablodaki her giriş, tanımlayıcının hangi temel dosyaya atıfta bulunduğunu, geçerli ofseti ve dosyanın okunabilir veya yazılabilir olup olmadığı gibi diğer ilgili ayrıntıları izler.

39.5 Okuma Ve Yazma, Fakat Sırayla Değil

Şimdiye kadar dosyaların nasıl okunup yazılacağını tartıştık, ancak tüm erişimler **sıralıydı**; yani bir dosyayı ya baştan sona okuduk ya da bir dosyayı baştan sona yazdık.Ancak bazen, bir dosya içinde belirli bir ofseti okuyabilmek veya yazabilmek yararlıdır; örneğin, bir metin belgesi üzerinde bir dizin oluşturursanız ve bunu belirli bir kelimeyi aramak için kullanırsanız, sonunda belgedeki bazı **rastgele** ofsetlerden okuma yapabilirsiniz. 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 tanımlayıcı). İkinci bağımsız değişken, **dosya ofsetini** dosya içinde belirli bir konuma konumlandıran `offset`tir. Tarihsel nedenlerle nereden geldiği olarak adlandırılan üçüncü bağımsız değişken, aramanın tam olarak nasıl gerçekleştirildiğini belirler. Kılavuz sayfasından:

```
Eğer whence SEEK SET ise, ofset ofset baytlarına ayarlanır
Eğer whence SEEK_CUR ise, ofset geçerli konumuna artı
  ofset baytlarına ayarlanır.
Eğer whence SEEK_END ise, ofset dosya boyutu artı ofset
  baytlarına ayarlanır.
```

Bu açıklamadan da anlayabileceğiniz gibi, bir işlem açılan her dosya için, işletim sistemi bir sonraki okuma veya yazma işleminin dosyadan okumaya veya dosyaya yazmaya başlayacağını belirleyen "geçerli" bir ofseti izler.Bu nedenle, açık bir dosyanın soyutlamasının bir kısmı, iki yoldan biriyle güncellenen bir geçerli ofseti olmasıdır.Birincisi, N baytlık bir okuma veya yazma gerçekleştiğinde, geçerli ofsete N eklenir; bu nedenle her okuma veya yazma *dolaylı olarak* ofseti günceller.İkincisi, yukarıda belirtildiği gibi ofseti değiştiren `lseek` ile *açık bir şekilde* gerçekleştirilir. Tahmin etmiş olabileceğiniz gibi ofset, daha önce gördüğümüz *yapı dosyasında, struct proc*'tan referans alınarak tutulur. İşte yapının (basitleştirilmiş) bir xv6 tanımı:

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

LSEEK()I Çağırmak Bir Disk Araması Yapmaz

Kötü adlandırılmış bir sistem çağrısı olan `lseek()`, diskleri ve üzerlerindeki 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 baş 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'sunun kendi kendine gerçekleşmesine neden olmaz.

Yapıda görebileceğiniz gibi, işletim sistemi açılan dosyanın okunabilir veya yazılabilir (veya her ikisi) olup olmadığını , hangi temel dosyaya atıfta bulunduğunu (`struct inode` işaretçisi `ip` ile gösterildiği gibi) ve geçerli offseti`(off)` belirleyebilir.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 anılırlar. xv6 çekirdeği, burada gösterildiği gibi, giriş başına bir kilitle bunları da bir dizi olarak tutar:

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

Birkaç örnekle bunu biraz daha netleştirelim.İlk olarak, bir dosyayı (300 bayt boyutunda) açan ve her seferinde 100 bayt okuyan `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 ofset değeri ile birlikte ilgili sistem çağrılarının bir izi aşağıdadır:

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	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.

İkinci olarak, **aynı** dosyayı iki kez açan ve her birine bir okuma gönderen bir işlemi izleyelim.

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

Bu 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 Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

Burada, `lseek()` çağrısı önce mevcut ofseti `200`'e ayarlar. Ardından gelen `read()` sonraki 50 baytı okur ve mevcut ofseti buna göre günceller.

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

Çoğu durumda (yukarıda gösterilen örneklerde olduğu gibi), dosya tanımlayıcısının 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 girdisi olacaktır. Aynı anda başka bir işlem aynı dosyayı okusa bile, açık dosya tablosunda her birinin kendi girişi olacaktır. Bu sayede her şey 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`)**

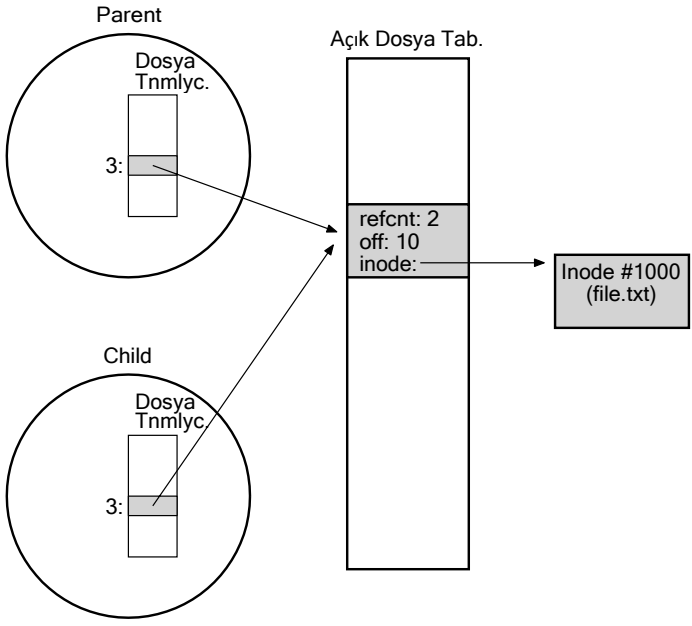
bir dosyanın okunması veya yazılması bağımsızdır ve verilen dosyaya erişirken her birinin kendi geçerli ofseti vardır.Ancak, açık dosya tablosundaki bir girişin *paylaşıldığı* birkaç ilginç durum vardır.Bu durumlardan biri, bir parent süreç `fork()` ile bir child süreç oluşturduğunda meydana gelir.Şekil 39.2, bir ebeveynin bir çocuk oluşturduğu ve ardından onun tamamlanmasını beklediği küçük bir kod parçacığını göstermektedir.Çocuk, bir `lseek()` çağrısı yoluyla geçerli ofseti ayarlar ve ardından çıkar. Son olarak ebeveyn, çocuğu 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** kullandığımızı unutmayın. Bir dosya tablosu girişi paylaşıldığında, referans sayısı artar; yalnızca her iki işlem de dosyayı kapattığında (veya çıktığında) giriş kaldırılır.

Açık dosya tablosu girişlerini ebeveyn ve çocuk arasında paylaşmak bazen yararlıdır. Örneğin, bir görev üzerinde işbirliği 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 Paylaşan İşlemler**

Bir başka ilginç ve belki de daha yararlı paylaşım durumu **dup()** (ve kuzenleri **dup2()** ve **dup3**()) ile gerçekleşir.

dup() çağrısı, bir işlemin, mevcut bir tanımlayıcıyla aynı temeldeki açık dosyaya atıfta bulunan yeni bir dosya tanımlayıcısı oluşturmasına izin verir.Şekil 39.4 `dup()` işlevinin nasıl kullanılabileceğini gösteren küçük bir kod parçasını gösterir.

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

```
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() (dup.c) İle Paylaşılan Dosya Tablosu Girişi**

39.7 `fsync()` İle Hemen Yazma

Çoğu zaman bir program `write()` işlevini çağırdığında, dosya sistemine sadece şunu söyler: lütfen bu verileri gelecekte bir noktada kalıcı depolama alanına yazın. Dosya sistemi, performans nedenleriyle, bu tür yazmaları bir süre için (örneğin 5 saniye veya 30 saniye) bellekte tamponlaya caktır; daha sonraki bir zamanda, yazma(lar) depolama aygıtına gerçekten verilecektir. Çağıran uygulamanın bakış açısına göre, yazmalar hızlı bir şekilde tamamlanır ve yalnızca nadir durumlarda (örneğin, makine `write()` çağrısından sonra ancak diske yazmadan önce çökerse) veriler kaybolur.

Ancak, bazı uygulamalar bu eşit garantiden daha fazlasını gerektirir. Örneğin, bir veritabanı yönetim sisteminde (DBMS), doğru bir kurtarma protokolünün geliştirilmesi, zaman zaman diske yazmaya zorlama yeteneği gerektirir.

Bu tür uygulamaları desteklemek için çoğu dosya sistemi bazı ek kontrol API'leri sağlar. UNIX dünyasında, uygulamalara sağlanan arayüz `fsync(int fd)` olarak bilinir. Bir işlem belirli bir dosya tanımlayıcısı için `fsync()` işlevini çağırdığında, dosya sistemi belirtilen dosya tanımlayıcısı tarafından atıfta bulunulan dosya için tüm kirli (yani henüz yazılmamış) verileri diske zorlayarak yanıt verir. Tüm bu yazımlar tamamlandığında `fsync()` yordamı geri döner.

İşte `fsync()` işlevinin nasıl kullanılacağına dair basit bir örnek. Kod `foo` dosyasını açar, dosyaya tek bir veri yığını yazar ve ardından yazımların hemen diske zorlandığından emin olmak için `fsync()` işlevini çağırır. `fsync()` geri döndüğünde, uygulama verilerin kalıcı hale getirildiğini bilerek (`fsync()` doğru şekilde uygulandıysa) güvenli bir şekilde yoluna devam edebilir.

```
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);
```

İlginç bir şekilde, bu dizi beklediğiniz her şeyi garanti etmez; bazı durumlarda, foo dosyasını içeren dizini de fsync() yapmanız gerekir. Bu adımın eklenmesi yalnızca dosyanın kendisinin diskte olmasını değil, aynı zamanda yeni oluşturulmuşsa dosyanın da kalıcı olarak dizinin bir parçası olmasını sağlar. Şaşırtıcı olmayan bir şekilde, bu tür ayrıntılar genellikle gözden kaçar ve uygulama düzeyinde birçok hataya yol açar [P+13,P+14].

39.8 Dosyaların Yeniden Adlandırılması

Bir dosyaya sahip olduğumuzda, bazen dosyaya farklı bir isim verebilmek yararlı olabilir. Komut satırında yazarken, bu mv komutu ile gerçekleştirilir; bu örnekte,foo dosyasının adı bar olarak değiştirilmiştir.

MMAP() Ve Kalıcı Belllek
(Terence Kelly)

Bellek eşleme (Memory mapping), dosyalardaki kalıcı verilere erişmenin alternatif bir yoludur. **mmap()** sistem çağrısı, bir dosyadaki bayt ofsetleri ile arama sürecindeki sanal adresler arasında bir yazışma oluşturur; ilki **destek dosyası(backing file)** , ikincisi ise **bellek içi görüntüsü (in-memory image)** olarak adlandırılır. İşlem daha sonra bellek içi görüntüye CPU talimatlarını (yani yükler ve depolar) kullanarak destek dosyasına erişebilir.

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

```
1  p = mmap(NULL, file_size, PROT_READ|PROT_WRITE,
2           MAP_SHARED, fd, 0);
3  assert(p != MAP_FAILED);
4  for (int i = 1; i < argc; i++)
5    if (strcmp(argv[i], "pop") == 0) // pop
6      if (p->n > 0)  // stack not empty
7        printf("%d\n", p->stack[--p->n]);
8    } else {  // push
9      if (sizeof(pstack_t) + (1 + p->n) * sizeof(int)
10         <= file_size)  // stack not full
11       p->stack[p->n++] = atoi(argv[i]);
12   }
```

`pstack.c` programı (yukarıda bir snippet ile birlikte OSTEP kodu github deposuna dahil edilmiştir), `ps.img` dosyasında hayata sıfırlardan oluşan bir çanta olarak başlayan kalıcı bir yığın depolar; örneğin, `truncate` veya `dd` yardımcı programı aracılığıyla komut satırında oluşturulur. 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` tamsayı dizisini yığınlar .Yığın kalıcı olduğu için, pstack'in bir çağrılması ile `push`lanan veriler bir sonraki çağrı tarafından `pop` edilebilir.

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

```
prompt> mv foo bar
```

Strace kullanarak, mv'nin rename(char *old, char *new) sistem çağrısını kullandığını görebiliriz.Tam olarak iki parametre alır: dosyanın orijinal adı (old) ve yeni adı (yeni).

rename() çağrısı tarafından sağlanan ilginç bir güvence, sistem çökmelerine göre (genellikle) **atomik(atomic)** bir çağrı olarak uygulanmasıdır; yeniden adlandırma sırasında sistem çökerse, dosya ya eski adla ya da yeni adla adlandırılır ve arada garip bir durum ortaya çıkmaz. Bu nedenle rename(), dosya durumunun atomik olarak güncellenmesini gerektiren belirli türdeki uygulamaları desteklemek için kritik öneme sahiptir.

Burada biraz daha spesifik olalım. Bir dosya düzenleyici (örneğin emacs) kullandığınızı ve bir dosyanın ortasına bir satır eklediğinizi düşünün. Örnek için dosyanın adı foo.txt olsun. Düzenleyicinin, yeni dosyanın orijinal içeriğe ve eklenen satıra sahip olduğunu garanti etmek için dosyayı güncelleme şekli aşağıdaki gibidir (basitlik için hata kontrolünü göz ardı ederek):

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // dosyanın yeni sürümünü yaz
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Bu örnekte editörün yaptığı şey basittir: dosyanın yeni sürümünü geçici bir adla (foo.txt.tmp) yazmak, fsync() ile diske zorlamak ve ardından uygulama yeni dosya meta verilerinin ve içeriğinin diskte olduğundan emin olduğunda geçici dosyayı orijinal dosyanın adıyla yeniden adlandırmak. Bu son adım atomik olarak yeni dosyayı yerine yerleştirirken aynı anda dosyanın eski sürümünü 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 bekleriz. Dosyalar hakkındaki bu tür verilere genellikle **üst veri** (**metadata)** adını veririz. Belirli bir dosyanın meta verilerini görmek için stat() veya fstat() sistem çağrılarını kullanabiliriz. Bu çağrılar bir dosyanın yol adını (veya dosya tanımlayıcısını) alır ve Şekil 39.5'te görüldüğü gibi bir stat yapısı doldurur. Her dosya hakkında, boyutu (bayt cinsinden), düşük seviyeli adı(low level name) (yani inode numarası), bazı sahiplik bilgileri ve diğer şeylerin yanı sıra dosyaya ne zaman erişildiği veya değiştirildiği hakkında bazı bilgiler de dahil olmak üzere çok sayıda bilgi tutulduğunu görebilirsiniz. Bu bilgileri görmek için komut satırı aracı stat'ı kullanabilirsiniz. Bu örnekte, önce bir dosya (file adında) oluşturacağız ve ardından dosya hakkında bazı şeyleri öğrenmek için stat komut satırı aracını kullanacağız.

```
struct stat {
  dev_t     st_dev;     // Dosyayı içeren cihazın kimliği
  ino_t     st_ino;     // düğüm numarası
  mode_t    st_mode;    // koruma
  nlink_t   st_nlink;   // sabit bağlantı sayısı
  uid_t     st_uid; // sahibin kullanıcı kimliği
  gid_t     st_gid;     // sahibin grup kimliği
  dev_t     st_rdev;  // cihaz kimliği (özel dosya ise)
  off_t     st_size; // toplam boyut, bayt cinsinden
  blksize_t st_blksize;//I/O Dosya sistemi blok boyutu
  blkcnt_t st_blocks;   //Ayrılmış blok sayısı
  time_t    st_atime;   // son erişim zamanı
  time_t    st_mtime;   // Son değişiklik zamanı
  time_t    st_ctime;  // son durum değişikliği zamanı
};
```

Şekil 39.5: **Stat yapısı.**

İşte Linux üzerindeki 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 **düğüm** adı verilen bir yapıda tutar. Dosya sistemi uygulaması hakkında konuşurken düğümler hakkında daha çok şey öğreniyor olacağız. Şimdilik, bir düğüm'ü 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.

39.10 Dosyaların Kaldırılması

Bu bölümde sanallaştırmaya bir kritik parça daha ekliyoruz.Bu noktada, dosyaları nasıl oluşturacağımızı ve bunlara sırayla veya rastgele olarak nasıl erişeceğimizi biliyoruz. Ancak dosyaları nasıl silersiniz? `UNIX` kullandıysanız, muhtemelen bildiğinizi düşünürsünüz: sadece `rm` programını çalıştırın. Ancak `rm` bir dosyayı kaldırmak için hangi sistem çağrısını kullanır?

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

Öğrenmek için eski dostumuz `strace`'i tekrar kullanalım. Burada o sinir bozucu `foo` dosyasını kaldırıyoruz:

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

İzlenen çıktıdan bir sürü alakasız şeyi kaldırdık ve geriye sadece gizemli bir şekilde isimlendirilmiş sistem çağrısı `unlink()`'e yapılan tek bir çağrı kaldı. Gördüğünüz gibi, `unlink()` sadece kaldırılacak dosyanın adını alıyor ve başarılı olduğunda sıfır döndürüyor. Ancak bu bizi büyük bir bilmeceye götürür: bu sistem çağrısı neden `unlink` olarak adlandırılmıştır? Neden sadece `remove` ya da `delete` değil? Bu bilmecenin cevabını anlamak için öncelikle sadece dosyaları değil, dizinleri de anlamamız gerekir.

39.11 Dizinlerin Oluşturulması

Dosyaların ötesinde, dizinlerle ilgili bir dizi sistem çağrısı dizin oluşturmanızı, okumanızı ve silmenizi sağlar. Bir dizine asla doğrudan yazamayacağınızı unutmayın. Dizinin biçimi dosya sistemi meta verisi olarak kabul edildiğinden, dosya sistemi kendisini dizin verilerinin bütünlüğünden sorumlu tutar; bu nedenle, bir dizini yalnızca dolaylı olarak, örneğin içinde dosyalar, dizinler veya diğer nesne türleri oluşturarak güncelleyebilirsiniz. Bu şekilde, dosya sistemi dizin içeriklerinin beklendiği gibi olduğundan emin olur.

Bir dizin oluşturmak için tek bir sistem çağrısı, `mkdir()`, mevcuttur. Adını taşıyan `mkdir` programı böyle bir dizin oluşturmak için kullanılabilir. Şimdi `foo` adında basit bir dizin oluşturmak için `mkdir` programını çalıştırdığımızda neler olduğuna bir göz atalım:

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

Böyle bir dizin oluşturulduğunda, minimum içeriğe sahip olmasına rağmen "boş" olarak kabul edilir. Özellikle, boş bir dizinin iki girdisi vardır: kendisine atıfta bulunan bir girdi ve ebeveynine atıfta bulunan bir girdi. İlki "." (nokta) dizini ve ikincisi ".." (nokta-nokta) olarak adlandırılır. Bu dizinleri `ls` programına bir bayrak (`-a`) geçerek görebilirsiniz:

```
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 komutlara ve bazen çok fazla gücün nasıl kötü bir şey olabileceğine dair harika bir örnek sunuyor. Örneğin, bir grup dosyayı bir kerede kaldırmak için şöyle bir şey yazabilirsiniz:

```
prompt> rm *
```

burada * geçerli dizindeki tüm dosyalarla eşleşecektir. Ancak bazen dizinleri ve aslında tüm içeriklerini de silmek istersiniz. Bunu rm'ye döngüsel olarak her dizine inmesini ve içeriğini de kaldırmasını söyleyerek yapabilirsiniz:

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

Bu küçük karakter dizisiyle başınızın derde girdiği yer, komutu yanlışlıkla bir dosya sisteminin kök dizininden verdiğinizde ve böylece tüm dosya ve dizinleri sistemden kaldırdığınızda ortaya çıkar. Eyvah!

Bu nedenle, güçlü komutların iki ucu keskin kılıcını unutmayın; size az sayıda tuşa basarak çok fazla iş yapma olanağı sağlarken, aynı zamanda hızlı ve kolay bir şekilde büyük ölçüde zarar verebilirler.

39.12 Dizinleri Okuma

Artık bir dizin oluşturduğumuza göre, bir dizini de okumak isteyebiliriz. Gerçekten de `ls` programı tam olarak bunu yapar. Şimdi `ls` gibi kendi küçük aracımızı yazalım ve nasıl yapıldığını görelim.

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

```
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 bir dizin girdisinde bulunan bilgileri göstermektedir:

```
struct dirent {
  char           d_name[256]; // dosya adı
  ino_t          d_ino;       // düğüm numarası
  off_t          d_off;       // sonraki dirent in ofseti
  unsigned short d_reclen;    //   bu   kaydın   uzunluğu
  unsigned char  d_type;      // dosya tipi
};
```

Dizinler çok az bilgi içerdiğinden (temel olarak, birkaç başka ayrıntıyla birlikte yalnızca adı düğüm numarasına eşler), bir program her dosya hakkında uzunluk veya diğer ayrıntılı bilgiler gibi daha fazla bilgi almak için `stat()` işlevini çağırmak isteyebilir. Aslında, `-l` bayrağını verdiğinizde `ls`'nin yaptığı tam olarak budur; kendiniz görmek için bu bayrakla ve bu bayrak olmadan `ls` üzerinde `strace`'i deneyin.

39.13 Dizinlerin Silinmesi

Son olarak, bir dizini `rmdir()` (aynı isimli `rmdir` programı tarafından kullanılır) çağrısı ile silebilirsiniz. Ancak dosya silmenin aksine, dizinleri silmek daha tehlikelidir, çünkü tek bir komutla büyük miktarda veriyi silebilirsiniz. Bu nedenle `rmdir()`, silinmeden önce dizinin boş olması (yani yalnızca "." ve ".." girdileri olması) şartını koşar. Boş olmayan bir dizini silmeye çalışırsanız, `rmdir()` çağrısı başarısız olur.

39.14 Sabit Bağlantılar(Hard Links)

Şimdi, `link()` olarak bilinen bir sistem çağrısı aracılığıyla dosya sistemi ağacında bir giriş yapmanın yeni bir yolunu anlayarak, bir dosyayı kaldırmanın neden `unlink()` aracılığıyla gerçekleştirildiği gizemine geri dönüyoruz. `link()` sistem çağrısı iki değişken alır, eski bir yol adı ve yeni bir yol adı; yeni bir dosya adını eski bir dosya adına "bağladığınızda", aslında aynı dosyaya başvurmak için başka bir yol yaratmış olursunuz. Bu örnekte gördüğümüz gibi, bunu yapmak için komut satırı programı `ln` 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 dosyaya `file2`[2] adını verdik. Daha sonra `ln` programını kullanarak bu dosyaya sabit bir bağlantı oluşturduk. Bundan sonra, `file` ya da `file2`'yi açarak dosyayı inceleyebiliriz.

`link()` işlevinin çalışma şekli, basitçe bağlantıyı oluşturduğunuz dizinde başka bir isim yaratması ve bunu orijinal dosyanın *aynı* düğüm numarasına (yani düşük seviyeli isme) atıfta bulunmasıdır. Dosya herhangi bir şekilde kopyalanmaz; bunun yerine, artık her ikisi de aynı dosyaya atıfta bulunan iki insan tarafından okunabilir adınız (`file ve file2`) vardır. Bunu, her dosyanın düğüm numarasını yazdırarak dizinin kendisinde bile görebiliriz:

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

`ls`'ye `-i` bayrağını geçirerek, her dosyanın düğüm numarasını (dosya adının yanı sıra) yazdırır. Ve böylece link'in gerçekten ne yaptığını görebilirsiniz: sadece aynı düğüm numarasına (bu örnekte 67158084) yeni bir referans oluşturun.

Artık `unlink()` fonksiyonunun neden `unlink()` olarak adlandırıldığını anlamaya başlamış olabilirsiniz. Bir dosya oluşturduğunuzda, gerçekten *iki* şey yaparsınız. İlk olarak, boyutu, bloklarının diskte nerede olduğu ve benzeri dahil olmak üzere dosya hakkında neredeyse tüm ilgili bilgileri izleyecek bir yapı (düğüm) oluşturursunuz. İkinci olarak, bu dosyaya insan tarafından okunabilir bir isim *bağlıyor* ve bu bağlantıyı bir dizine yerleştiriyorsunuz.

Bir dosyaya, dosya sistemine sabit bir bağlantı oluşturduktan sonra, orijinal dosya adı (`file`) ile yeni oluşturulan dosya adı (`file2`) arasında hiçbir fark yoktur; aslında her ikisi de 67158084 numaralı kodda bulunan dosya hakkındaki temel meta verilere bağlantıdır.

Bu nedenle, bir dosyayı dosya sisteminden kaldırmak için `unlink()` işlevini çağırırız. Yukarıdaki örnekte, örneğin `file` adlı dosyayı kaldırabilir ve dosyaya zorluk çekmeden erişmeye devam edebiliriz:

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

Bunun çalışmasının nedeni, dosya sistemi dosyanın bağlantısını kaldırdığında, düğüm numarası içindeki bir **referans sayısını (reference count)** kontrol etmesidir. Bu referans sayısı (bazen **bağlantı sayısı (link count)** olarak da adlandırılır), dosya sisteminin bu belirli düğüme kaç farklı dosya adının bağlandığını izlemesini sağlar. `unlink()` işlevi çağrıldığında, insan tarafından okunabilir dosya numarası ile adını (silinmekte olan dosya)

[2]Bu kitabın yazarlarının ne kadar yaratıcı olduğuna bir kez daha dikkat edin. Bizim de eskiden " Cat" adında bir kedimiz vardı (gerçek hikaye). Ancak o öldü ve şimdi "Hammy" adında bir hamsterımız var. Güncelleme: Hammy de artık ölü. Evcil hayvan cesetleri yığılıyor.

verilen düğüm numarasına ekler ve referans sayısını azaltır; yalnızca referans sayısı sıfıra ulaştığında dosya sistemi düğümü ve ilgili veri bloklarını da serbest bırakır ve böylece dosyayı gerçekten "siler"

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

```
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(Symbolic Links)

Gerçekten kullanışlı olan bir başka bağlantı türü daha vardır ve buna **sembolik bağlantı (symbolic link)** ya da bazen **yumuşak bağlantı (soft link)** denir. Sabit bağlantılar biraz sınırlıdır: bir dizine sabit bağlantı oluşturamazsınız (dizin ağacında bir döngü oluşturacağınızdan korktuğunuz için); diğer disk bölümlerindeki dosyalara sabit bağlantı oluşturamazsınız (çünkü inode numaraları yalnızca belirli bir dosya sistemi içinde benzersizdir, dosya sistemleri arasında değil); vb. Böylece, sembolik bağlantı adı verilen yeni bir bağlantı türü oluşturulmuştur [MJLF84].

Böyle bir bağlantı oluşturmak için aynı ln programını kullanabilirsiniz, ancak -s bayrağı ile. İşte bir örnek:

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

Gördüğünüz gibi, bir yazılım bağlantısı oluşturmak hemen hemen aynıdır ve orijinal dosyaya artık dosya adı `file` yanı sıra sembolik bağlantı adı `file2` aracılığıyla da erişilebilir.

Ancak, bu yüzeysel benzerliğin ötesinde, sembolik bağlantılar aslında sabit bağlantılardan oldukça farklıdır. İlk fark, bir sembolik bağlantının aslında farklı türde bir dosyanın kendisi olmasıdır. Normal dosya ve dizinlerden zaten bahsetmiştik; sembolik bağlantılar dosya sisteminin bildiği üçüncü bir türdür. Sembolik bağlantı üzerinde yapılan bir `stat` her şeyi ortaya çıkarır:

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

`ls` çalıştırıldığında da bu gerçek ortaya çıkar. Eğer `ls` çıktısının uzun formunun ilk karakterine yakından bakarsanız, en soldaki sütundaki ilk karakterin normal dosyalar için -, yönlendirmeler için `d` ve yazılımsal bağlantılar için `l` olduğunu görebilirsiniz. Ayrıca sembolik bağlantının boyutunu (bu durumda 4 bayt) ve bağlantının neye işaret ettiğini (`file` adlı dosya) 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, bağlantı verilen dosyanın yol adını bağlantı dosyasının verisi olarak tutmasıdır. File adlı bir dosyaya bağlantı verdiğimiz için, bağlantı dosyamız file2 küçüktür (4 bayt). Eğer daha uzun bir yol adına bağlantı verseydik, bağlantı dosyamız daha büyük olurdu:

```
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 referans(dangling reference)** olarak bilinen bir olasılık 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 farklı olarak, dosya adlı orijinal dosyanın kaldırılması, bağlantının artık var olmayan bir yol adına işaret etmesine neden olur.

39.16 İzin Bitleri ve Erişim Kontrol Listeleri

Bir işlemin soyutlanması iki merkezi sanallaştırma sağlamıştır: CPU ve bellek. Bunların her biri, bir işleme kendi *özel* CPU'suna ve kendi *özel* belleğine sahip olduğu yanılsamasını veriyordu; gerçekte, altındaki işletim sistemi sınırlı fiziksel kaynakları rakip varlıklar arasında güvenli ve emniyetli bir şekilde paylaşmak için çeşitli teknikler kullanıyordu.

Dosya sistemi de diskin sanal bir görünümünü sunar ve bu bölümde açıklandığı gibi diski bir grup ham bloktan çok daha kullanıcı dostu dosyalara ve dizinlere dönüştürür. Bununla birlikte, bu soyutlama CPU ve bellekten önemli ölçüde farklıdır, çünkü dosyalar farklı kullanıcılar ve işlemler arasında *paylaşılır* ve (her zaman) özel değildir. Bu nedenle, dosya sistemlerinde genellikle çeşitli derecelerde paylaşımı mümkün kılan daha kapsamlı bir dizi mekanizma mevcuttur.

Bu tür mekanizmaların ilk şekli klasik UNIX izin bitleridir. Bir `foo.txt` dosyasının 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--` kısmına dikkat edeceğiz. Buradaki ilk karakter sadece dosyanın türünü gösterir: normal bir dosya için (ki `foo.txt` öyledir), bir dizin için `d`, sembolik bir bağlantı için `l` ve benzeri; bu (çoğunlukla) izinlerle ilgili değildir, bu yüzden şimdilik görmezden geleceğiz.

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

İzinler üç gruptan oluşur: dosya **sahibinin(owner)** dosyaya yapabilecekleri, bir **gruptaki(group)** birinin dosyaya yapabilecekleri ve son olarak herhangi birinin (bazen **diğer(other)** olarak da adlandırılır) yapabilecekleri. Sahibin, grup üyesinin veya diğerlerinin sahip olabileceği yetenekler arasında dosyayı okuma, yazma veya yürütme yer alır.

Yukarıdaki örnekte, ls çıktısının ilk üç karakteri dosyanın sahibi tarafından hem okunabilir hem de yazılabilir olduğunu `(rw-)` ve yalnızca wheel grubu üyeleri ve sistemdeki diğer herkes tarafından okunabilir olduğunu gösterir (`r--` ardından `r--`).

Dosyanın sahibi bu izinleri kolayca değiştirebilir, örneğin chmod komutunu kullanarak (**dosya modunu(file mode)** değiştirmek için). Dosya sahibi dışında herhangi birinin dosyaya erişimini kaldırmak için şunu yazabilirsiniz:

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

Dosya Sistemleri için Yetkili Kullanıcı(Superuser)

Dosya sistemini yönetmeye yardımcı olmak için hangi kullanıcının ayrıcalıklı işlemler yapmasına izin verilir? Örneğin, etkin olmayan bir kullanıcının dosyalarının yer kazanmak için silinmesi gerekiyorsa, bunu yapmaya kimin hakkı vardır?

Yerel dosya sistemlerinde, genel varsayılan, ayrıcalıklardan bağımsız olarak tüm dosyalara erişebilen bir tür **yetkili kullanıcı** (yani **root**) olmasıdır. AFS gibi dağıtılmış bir dosya sisteminde (erişim kontrol listeleri olan), `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, saldırgan sistemdeki tüm bilgilere erişebilir ve böylece öngörülen gizlilik ve koruma garantilerini ihlal edebilir.

Bu komut, sahip için okunabilir biti (4) ve yazılabilir biti (2) etkinleştirir (bunları birlikte OR'ladığınızda yukarıdaki 6 elde edilir), ancak grup ve diğer izin bitlerini sırasıyla 0 ve 0 olarak ayarlar, böylece izinleri rw olarak ayarlar.

Yürütme biti özellikle ilginçtir. Normal dosyalar için, varlığı bir programın çalıştırılıp çalıştırılamayacağını belirler. Örneğin, `hello.csh` adında basit bir shell scriptimiz varsa, bunu yazarak çalıştırmak isteyebiliriz:

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

Ancak, bu dosya için execute bitini doğru şekilde ayarlamazsak, aşağıdakiler gerçekleşir:

```
prompt> chmod 600 hello.csh
prompt> ./hello.csh
./hello.csh: Erişim reddedildi.
```

Dizinler için, execute biti biraz farklı davranır. Özellikle, bir kullanıcının (veya grubun veya herkesin) verilen dizinde dizin değiştirme (yani `cd`) gibi şeyler yapmasını ve yazılabilir bit ile birlikte burada dosya oluşturmasını sağlar. Bu konuda daha fazla bilgi edinmenin en iyi yolu: kendiniz oynayın! Endişelenmeyin, (muhtemelen) hiçbir şeyi çok kötü karıştırmayacaksınız.

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

Örneğin, `fs listacl` komutuyla gösterildiği gibi, bir yazarın AFS hesabındaki özel bir dizin için erişim denetimleri aşağıda verilmiştir:

```
prompt> fs listacl private
Access list for private is
Normal rights:
  system:administrators rlidwka
  remzi rlidwka
```

Liste, hem sistem yöneticilerinin hem de `remzi` kullanıcısının bu dizindeki dosyaları arayabileceğini, ekleyebileceğini, silebileceğini ve yönetebileceğini, ayrıca bu dosyaları okuyabileceğini, yazabileceğini ve kilitleyebileceğini göstermektedir.

Birinin (bu durumda diğer yazarın) bu dizine erişmesine izin vermek için `remzi` kullanıcısı aşağıdaki komutu yazabilir

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

İşte `remzi`'nin gizliliği! Ama şimdi daha da önemli bir ders öğrendiniz: iyi bir evlilikte sır olmaz, dosya sistemi[3] içinde bile.

39.17 Dosya Sistemi Oluşturma ve Bağlama(Mounting)

Şimdi dosyalara, dizinlere ve bazı özel bağlantı türlerine erişmek için temel arayüzleri gezdik. Ancak tartışmamız gereken bir konu daha var: altta yatan birçok dosya sisteminden tam bir dizin ağacının nasıl bir araya getirileceği. Bu görev, önce dosya sistemleri oluşturarak ve daha sonra içeriklerini erişilebilir hale getirmek için bunları bağlayarak gerçekleştirilir.

Bir dosya sistemi oluşturmak için çoğu dosya sistemi, genellikle `mkfs` ("make fs" olarak okunur) olarak adlandırılan ve tam olarak bu görevi yerine getiren bir araç sağlar. Fikir şu şekildedir: araca girdi olarak bir aygıt (disk bölümü gibi, örneğin `/dev/sda1`) ve bir dosya sistemi türü (örneğin ext3) verin ve o disk bölümüne bir kök dizinle başlayan boş bir dosya sistemi yazsın. Ve mkfs dedi ki, bir dosya sistemi olsun!

Ancak, böyle bir dosya sistemi oluşturulduktan sonra, tek tip dosya sistemi ağacı içinde erişilebilir hale getirilmesi gerekir. Bu görev `mount` programı (asıl işi yapmak için altta yatan sistemin `mount()` çağrısı yapmasını sağlar) aracılığıyla gerçekleştirilir. Mount programının yaptığı şey, oldukça basit bir şekilde, mevcut bir dizini hedef **bağlama noktası(mount point)** olarak almak ve esasen bu noktadaki dizin ağacına yeni bir dosya sistemi yapıştırmaktır.

Burada bir örnek faydalı olabilir. Aygıt bölümü `/dev/sda1`'de saklanan, aşağıdaki içeriğe sahip, bağlanmamış bir ext3 dosya sistemimiz olduğunu düşünün: `a` ve `b` adında iki alt dizin içeren bir kök dizin, her biri sırayla foo adında tek bir dosya içerir. Diyelim ki bu dosya sistemini `/home/users` bağlama noktasına bağlamak istiyoruz. Şöyle bir şey yazacağız:

[3] Eğer merak ediyorsan söyleyeyim, 1996'dan beri evliyim. Biliyorum etmiyordun.

İPUCU: TOCTTOU'YA KARŞI DİKKATLİ OLUN

1974 yılında McPhee bilgisayar sistemlerinde bir sorun olduğunu fark etmiştir. McPhee özellikle "... bir geçerlilik kontrolü ile bu geçerlilik kontrolüne bağlı işlem arasında bir zaman aralığı varsa, [ve] çoklu görev yoluyla, geçerlilik kontrolü değişkenleri bu zaman aralığında kasıtlı olarak değiştirilebilir ve bu da kontrol programı tarafından geçersiz bir işlem gerçekleştirilmesine neden olabilir." Bugün buna Kontrol Zamanı ile **Kullanım Zamanı(Time Of Check To Time Of Use) (TOCTTOU)** sorunu diyoruz ve ne yazık ki hala ortaya çıkabiliyor.

Bishop ve Dilger [BD96] tarafından açıklanan basit bir örnek, bir kullanıcının daha güvenilir bir hizmeti nasıl kandırabileceğini ve böylece nasıl sorun yaratabileceğini göstermektedir. Örneğin, bir posta hizmetinin root olarak çalıştığını (ve dolayısıyla sistemdeki tüm dosyalara erişim ayrıcalığına sahip olduğunu) düşünün. Bu hizmet gelen bir mesajı kullanıcının gelen kutusu dosyasına aşağıdaki şekilde ekler. İlk olarak, dosya hakkında bilgi almak için `lstat()` işlevini çağırır, özellikle de dosyanın hedef kullanıcıya ait normal bir dosya olduğundan ve posta sunucusunun güncellememesi gereken başka bir dosyaya bağlantı olmadığından emin olur. Daha sonra, kontrol başarılı olduktan sonra, sunucu dosyayı yeni mesajla günceller.

Ne yazık ki, kontrol ve güncelleme arasındaki boşluk bir probleme yol açar: saldırgan (bu durumda, postayı alan ve dolayısıyla gelen kutusuna erişim izinlerine sahip olan kullanıcı) gelen kutusu dosyasını (`rename()` çağrısı yoluyla) `/etc/passwd` (kullanıcılar ve parolaları hakkında bilgi tutan) gibi hassas bir dosyaya işaret edecek şekilde değiştirir. Eğer bu geçiş doğru zamanda (kontrol ve erişim arasında) gerçekleşirse, sunucu hassas dosyayı postanın içeriği ile kayıtsızca güncelleyecektir. Saldırgan artık bir e-posta göndererek hassas dosyaya yazabilir; `/etc/passwd` dosyasını güncelleyerek `root` ayrıcalıklarına sahip bir hesap ekleyebilir ve böylece sistemin kontrolünü ele geçirebilir.

TOCTTOU sorununa yönelik basit ve mükemmel çözümler bulunmamaktadır [T+08]. Bir yöntem, çalıştırmak için kök ayrıcalıklarına ihtiyaç duyan hizmetlerin sayısını azaltmaktır, bu da yardımcı olur. O `NOFOLLOW` bayrağı, hedefin sembolik bir bağlantı olması durumunda `open()` işlevinin başarısız olmasını sağlar, böylece söz konusu bağlantıları gerektiren saldırıları önler. İşlemsel bir dosya sistemi [H+18] kullanmak gibi daha radikal yaklaşımlar sorunu çözebilir, ancak geniş çapta kullanılan çok fazla **işlemsel dosya sistemi(transactional file system)** yoktur. Bu nedenle, her zamanki (klasik) tavsiye: yüksek ayrıcalıklarla çalışan kod yazarken dikkatli olun!

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

Eğer başarılı olursa, bağlama işlemi bu yeni dosya sistemini kullanılabilir hale getirecektir. Ancak, yeni dosya sistemine şimdi nasıl erişildiğine dikkat edin. Kök dizinin içeriğine bakmak için `ls` komutunu şu şekilde kullanırız:

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

AGördüğünüz gibi, /home/users/ yol adı artık yeni bağlanan dizinin köküne atıfta bulunuyor. Benzer şekilde, a ve b dizinlerine /home/users/a ve /home/users/b yol adlarıyla erişebiliriz. Son olarak, foo isimli dosyalara /home/users/a/foo ve /home/users/b/foo yol adlarıyla erişilebilir. Ve böylece mount'un güzelliği: mount, bir dizi ayrı dosya sistemine sahip olmak yerine, tüm dosya sistemlerini tek bir ağaçta birleştirerek adlandırmayı tek tip ve kullanışlı hale getirir.

Sisteminize nelerin ve hangi noktalara bağlandığını görmek için mount programını çalıştırmanız yeterlidir. Bunun gibi bir şey göreceksiniz:

```
/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 disk tabanlı bir dosya sistemi), proc dosya sistemi (mevcut işlemler hakkındaki bilgilere erişmek için kullanılan bir dosya sistemi), tmpfs (yalnızca geçici dosyalar için kullanılan bir dosya sistemi) ve AFS (dağıtılmış bir dosya sistemi) dahil olmak üzere çok sayıda farklı dosya sisteminin bu makinenin dosya sistemi ağacında bir araya getirildiğini göstermektedir.

39.18 Özet

UNIX sistemlerindeki (ve aslında herhangi bir sistemdeki) dosya sistemi arayüzü görünüşte oldukça ilkeldir, ancak ustalaşmak istiyorsanız anlamanız gereken çok şey vardır. Elbette hiçbir şey onu (çok) kullanmaktan daha iyi değildir. Bu yüzden lütfen öyle yapın! Elbette, daha fazlasını okuyun; her zaman olduğu gibi, Stevens [SR05] başlamak için en uygun yerdir.

Anahtar Dosya Sistemi Terimleri

- **Dosya(file)**, oluşturulabilen, okunabilen, yazılabilen ve silinebilen bir bayt dizisidir. Kendisini benzersiz bir şekilde ifade eden düşük seviyeli bir adı (yani bir sayı) vardır. Düşük seviyeli isim genellikle **i-numarası** olarak adlandırılır.
- **Dizin(directory)**, her biri insan tarafından okunabilir bir ad ve eşleştiği düşük seviyeli bir ad içeren bir küme koleksiyonudur. Her girdi ya başka bir dizine ya da bir dosyaya atıfta bulunur. Her dizinin kendisinin de düşük seviyeli bir adı (i-numarası) vardır. Bir dizin her zaman iki özel girdiye sahiptir: kendisine atıfta bulunan . girdisi ve ebeveynine atıfta bulunan ... girdisi.
- **Dizin ağacı(directory tree)** veya **dizin hiyerarşisi(directory hierarchy)**, tüm dosya ve dizinleri **kökten(root)** başlayarak büyük bir ağaç halinde düzenler
- Bir dosyaya erişmek için, bir işlem işletim sisteminden izin istemek üzere bir sistem çağrısı (genellikle `open()`) kullanmalıdır. İzin verilirse, işletim sistemi bir dosya **tanımlayıcısı(file descriptor)** döndürür; bu tanımlayıcı, izinler ve amaç izin verdiği ölçüde okuma veya yazma erişimi için kullanılabilir.
- Her dosya tanımlayıcısı, **açık dosya tablosundaki(open file table)** bir girdiye atıfta bulunan özel, işlem başına bir varlıktır. Buradaki giriş, bu erişimin hangi dosyaya atıfta bulunduğunu, dosyanın **geçerli ofsetini(current offset)** (yani, bir sonraki okuma veya yazma işleminin dosyanın hangi bölümüne erişeceğini) ve diğer ilgili bilgileri izler.
- `read()` ve `write()` çağrıları doğal olarak geçerli ofseti günceller; aksi takdirde, süreçler değerini değiştirmek için `lseek()` işlevini kullanarak dosyanın farklı bölümlerine rastgele erişim sağlayabilir.
- Güncellemeleri kalıcı ortamlara zorlamak için , bir işlem `fsync()` veya ilgili çağrıları kullanmalıdır. Ancak, yüksek performansı korurken bunu doğru bir şekilde yapmak zordur [P+14], bu nedenle bunu yaparken dikkatli düşünün.
- Dosya sistemindeki birden fazla insan tarafından okunabilir ismin aynı temel dosyaya atıfta bulunması için **sabit bağlantılar(hard links)** veya **sembolik bağlantılar(symbolic links)** kullanın. Her biri farklı durumlarda kullanışlıdır, kullanmadan önce güçlü ve zayıf yönlerini göz önünde bulundurun. Ve unutmayın, bir dosyayı silmek sadece dizin hiyerarşisinden son bir `unlink()` oluşturmaktır
- Çoğu dosya sistemi paylaşımı etkinleştirmek ve devre dışı bırakmak için mekanizmalara sahiptir. Bu tür kontrollerin ilkel bir biçimi **izin bitleri(permission bits)** tarafından sağlanır; daha karmaşık **erişim kontrol listeleri(access control lists)**, bilgiye tam olarak kimin erişebileceği ve manipüle edebileceği üzerinde daha hassas kontrol sağlar.

Kaynak

[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. Arpaci-Dusseau. 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.*

Ev Ödevi (KOD)

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

Questions

1. **Stat:** Belirli bir dosya veya dizin üzerinde `stat()` sistem çağrısını çağıran komut satırı programı stat'ın kendinize ait versiyonunu yazın. Dosya boyutunu, tahsis edilen blok sayısını, referans (bağlantı) sayısını ve benzerlerini yazdırın. Dizindeki giriş sayısı değiştikçe, bir dizinin bağlantı sayısı nedir? Faydalı arayüzler: `stat()`, doğal olarak.

```c
#define _DEFAULT_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/sysmacros.h>


int main(int argc, char** argv) {
    if (argc < 2) {
        printf("Dosya\Dizin belirtin\n");
        exit(-1);
    }
    struct stat statbuf;
    if (stat(argv[1], &statbuf) < 0) {
        printf("İstatikler alınamadı\n");
        exit(-1);
    }
    printf("Düğüm       : %lu\n", statbuf.st_ino);
    printf("Boyutu       : %'ld\n", statbuf.st_size);
    printf("Tipi        : ");
    switch (statbuf.st_mode & S_IFMT) {
        case S_IFBLK:  printf("blok cihazıe\n");            break;
        case S_IFDIR:  printf("dizin\n");               break;
        case S_IFLNK:  printf("symlink\n");                 break;
        case S_IFREG:  printf("dÜzenli dosya\n");           break;
        case S_IFSOCK: printf("soket\n");               break;
        default:       printf("bilinmeyen?\n");                break;
    }
    printf("Bağlantı Sayısı  : %lu\n", statbuf.st_nlink);
    return 0;
}
```

Girişler arttıkça bağlantı sayısı artar, azaldıkça azalır.

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

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include <string.h>


void join_path(char* path1, char* path2, char* final) {
    sprintf(final, "%s%s%s", path1, path1[strlen(path1)-1] == '/' ? "" : "/", path2);
}

void print_stats(char* path) {
    struct stat statbuf;
    if (stat(path, &statbuf) < 0) {
        printf("Failed to get stats\n");
        exit(-1);
    }
    printf("%5u %5u %5lu %12ld   ", statbuf.st_uid, statbuf.st_uid, statbuf.st_nlink, statbuf.st_size);
    if (S_ISDIR(statbuf.st_mode)) printf("📁 ");
    else if (S_ISREG(statbuf.st_mode)) printf("📄 ");
    else if (S_ISLNK(statbuf.st_mode)) printf("🔗 ");
    else printf("👽 ");
}


int main(int argc, char** argv) {
    int show_details = 0, opt;
    while ((opt = getopt(argc, argv, "l")) != -1) {
        if (opt == 'l') show_details = 1;
    }
    char* dir = optind < argc ? argv[optind] : ".";

    char path[1024];
    struct dirent* pDirent;
    DIR *pDir;

    if ((pDir = opendir(dir)) == NULL) {
        fprintf(stderr, "Failed to open directory\n");
        exit(-1);
    }
    if (show_details) {
        printf("  UID   GID NLink          Size Type Name\n");
    }
    while ((pDirent = readdir(pDir)) != NULL ) {
        if (show_details) {
            join_path(dir, pDirent->d_name, path);
            print_stats(path);
        }
        printf("%s\n", pDirent->d_name);
    }
    return 0;
}
```

3. **Kuyruk:** Bir dosyanın son birkaç satırını yazdıran bir program yazın. Program şu şekilde erimli olmalıdır, çünkü dosyanın sonuna yaklaşır, bir veri bloğu okur ve sonra istenen satır sayısını bulana kadar geriye doğru gider; bu noktada, dosyanın başından sonuna kadar bu satırları yazdırmalıdır. Programı çağırmak için şu yazılmalıdır: mytail -n file, burada n yazdırılacak dosyanın sonundaki satır sayısıdır. Faydalı arayüzler: stat(), lseek(), open(), read(), close().

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#define BLOCKSIZE 5
#define MAX_SIZE 1<<20

#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))

void reverse(char* s, int len) {
    char tmp;
    for(int i = 0; i < len / 2; i++) {
        tmp = s[i];
        s[i] = s[len-i-1];
        s[len-i-1] = tmp;
    }
}
int main(int argc, char** argv) {
    int lines = 1, opt;
    char* path;
    while ((opt = getopt(argc, argv, "n:")) != -1) {
        if (opt == 'n') lines = atoi(optarg);
    }
    if (optind >= argc) {
        fprintf(stderr, "File path is required\n");
        exit(-1);
    } else {
        path = argv[optind];
    }
    int fd;
    struct stat statbuf;
    if ((fd = open(path, O_RDONLY)) < 0) {
        fprintf(stderr, "Can't read file %s\n", path);
    }
    fstat(fd, &statbuf);

    int done = 0, seek_pos = BLOCKSIZE, pos = 0, len, res_pos = 0, read_size;
    char buff[BLOCKSIZE+1], res[MAX_SIZE];
    while (done < lines) {
        lseek(fd, MAX(-statbuf.st_size, -seek_pos), SEEK_END);
        read_size = MIN(BLOCKSIZE, statbuf.st_size - pos);
        read(fd, buff, read_size);
        len = strlen(buff);
        len = MIN(len, read_size);
        pos+= read_size;
        for(int i = len-1; i >= 0; i--) {
            if (buff[i] == '\n' && res_pos != 0) done++;
            if (done >= lines) break;
            res[res_pos++] = buff[i];
        }
        if (len < BLOCKSIZE) break;
        seek_pos += BLOCKSIZE;
    }
    res[res_pos] = '\0';
    reverse(res, res_pos);
    printf("%s\n", res);
    return 0;
```

4. **Özyinelemeli(recursive) Arama :** Ağacın belirli bir noktasından başlayarak dosya sistemi ağacındaki her dosya ve dizinin adını yazdıran bir program yazın. Örneğin, argümanlar olmadan çalıştırıldığında, program geçerli çalışma dizini ile başlamalı ve CWD'deki kök olmak üzere tüm ağaç yazdırılana kadar içeriğinin yanı sıra herhangi bir alt dizinin vb. içeriğini de yazdırmalıdır. Tek bir argüman (bir dizin adı) verilirse, bunun yerine ağacın kökü olarak bunu kullanın. Güçlü find komut satırı aracına benzer şekilde, özyinelemeli aramanızı daha eğlenceli seçeneklerle hassaslaştırın. Faydalı arayüzler: Kendiniz çözün.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include <string.h>

#define INDENT 6

char indent[1024];

void join_path(char* path1, char* path2, char* final) {
    sprintf(final, "%s%s%s", path1, path1[strlen(path1)-1] == '/' ? "" : "/", path2);
}

void print_contents(char* path, int level) {
    struct dirent* pDirent;
    DIR *pDir;
    struct stat statbuf;
    char new_path[1024];

    if ((pDir = opendir(path)) == NULL) {
        return;
    }
    while ((pDirent = readdir(pDir)) != NULL ) {
        if (!strcmp(pDirent->d_name, ".") || !strcmp(pDirent->d_name, "..")) {
            continue;
        }

        printf("%s", indent);

        printf("%s\n", pDirent->d_name);
        join_path(path, pDirent->d_name, new_path);
        if (stat(new_path, &statbuf) < 0) {
            continue;
        }
        if (S_ISDIR(statbuf.st_mode)) {
            sprintf(indent+(level*INDENT), "\u2502   \u251c\u2500");
            print_contents(new_path, level+1);
            sprintf(indent+(level*INDENT), "\u251c\u2500");
        }
    }
}

int main(int argc, char** argv) {
    char* dir = argc >= 2 ? argv[1] : ".";
    sprintf(indent, "\u251c\u2500");
    print_contents(dir, 0);
    return 0;
}
```