33

OLAYA DAYALI EŞ ZAMANLILIK (İLERİ)

Şimdiye kadar eş zamanlılık hakkında sanki eş zamanlı uygulamalar oluşturmanın tek yolu iş parçaları kullanmakmış gibi yazdık. Hayattaki birçok şey gibi bu da tamamen doğru değil. Özellikle hem GUI tabanlı uygulamalarda[O96] hem de bazı internet sunucularında farklı bir eş zamanlı programlama stili sıklıkla kullanılır[PDZ99]. Olaya dayalı eş zamanlılık(**event-based concurrency**) olarak bilinen bu stil, **node.js**[N13] gibi sunucu tarafı çerçeveler de dahil olmak üzere bazı modern sistemlerde popular hale geldi, ancak kökleri aşağıda tartışacağımız C/UNİX sistemlerinde bulunur.

Olay tabanlı eş zamanlılığın ele aldığı sorun iki yönlüdür. Birincisi, çok iş zamanlı uygulamalarda eş zamanlılığı doğru şekilde yönetmenin zor olabileceğidir; tartıştığımız gibi, eksik kilitler, kitlenme ve diğer kötü sorunları ortaya çıkarabilir. İkincisi, çok iş parçacıklı bir uygulamada, geliştiricinin belirli bir zamanda programlanan şey üzerinde çok az kontrolü vardır ya da hiç kontrolü yoktur; bunun yerine programcı basitçe iş parçacıkları oluşturur ve ardından temel işletim sisteminin bunları mevcut CPU'lar arasında makul bir şekilde zamanlamasını bekler. Tüm iş yükleri için her durumda iyi çalışan genel amaçlı bir zamanlayıcı oluşturmanın zorluğu göz önüne alındığında , bazen işletim sistemi çalışmayı optimumdan daha az bir şekilde planlayacaktır, ve böylece elimizde…

Püf Noktası:

İş Parçası Olmadan Eş Zamanlı Sunucular Nasıl Oluşturulur
İş parçacığı kullanmadan eş zamanlı bir sunucuyu nasıl oluşturabiliriz ve böylece eş zamanlılık üzerinde kontrolü nasıl sürdürebiliriz ve aynı zamanda çok iş parçacıklı uygulamalara sorun gibi görünen durumlardan nasıl kaçınabiliriz?

33.1 Temel Fikir : Bir Olay Döngüsü

Yukarıda belirttiğimiz gibi kullanacağımız temel yaklaşıma olaya dayalı eş zamanlılık denir(**event-based concurrency**). Yaklaşım oldukça basittir; sadece bir şeyin(yani bir olayın) gerçekleşmesini beklersiniz; olduğunda, ne tür bir olay olduğunu kontrol eder ve

Gerektirdiği az miktarda işi yaparsınız (bu I/O isteklerini yayınlamayı veya gelecekteki işlemler için diğer olayları planlamayı vb. İçerebilir). Bu kadar!

Ayrıntılara girmeden önce kanonik olay tabanlı bir sunucunun nasıl göründüğüne göz atalım. Bu tür uygulamalar olay döngüsü(**event loop)** olarak bilinen basit bir yapıya dayanır. Bir olay döngüsü için sözde kod şöyle görünür:

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

Gerçekten bu kadar basit. Ana döngü basitçe bir şeylerin yapılmasını bekler (yukarıdaki kodda getEvents() ögesini çağırarak) ve ardından , döndürülen her olay için bunları birer birer işler; her olayı işleyen kod olay işleyici(**event handler)** olarak bilinir. Daha da önemlisi, bir işleyici bir olayı işlerken, sistemde yer alan tek etkinliktir; bu nedenle, hangi olayın daha sonra ele alınacağına karar vermek, planlamaya eşdeğerdir. Programlama üzerindeki bu açık kontrol, olaya dayalı yaklaşımın temel avantajlarından biridir.

Ancak bu tartışma bizi daha büyük bir soruyla baş başa bırakıyor: Olay tabanlı bir sunucu, özellikle ağ ve disk I/O ile ilgili olarak hangi olayların gerçekleştiğini tam olarak nasıl belirler? Spesifik olarak, bir olay sunucusu kendisi için bir mesajın gelip gelmediğini nasıl anlayabilir?

33.4 Önemli Bir API: `select()` (or `poll()`)

Bu temel olay döngüsünü göz önünde bulundurarak, şimdi olayların nasıl alınacağı sorusunu ele almalıyız. Çoğu sistemde, select() veya pool() sistem çağrıları aracılığıyla temel bir API mevcuttur.

Bu arayüzlerin bir programın yapmasını sağladığı şey basittir: İlgilenilmesi gereken herhangi gelen bir I/O olup olmadığını kontrol edin.Örneğin, bir ağ uygulamsının(web sunucusu gibi) herhangi bir ağ paketinin gelip gelmediğini kontrol etmek istediğini düşünün. Bu sistem çağrıları tam olarak bunu yapmazını sağlar.

Örneğin select()'i ele alalım. Kılavuz sayfası (Mac'te) API'yi bu şekilde ele alır:

```
int select(int nfds,
           fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

Kılavuz sayfasındaki asıl açıklama: select(), adreslerini *görmek için readfds, writefds ve errorfds'de iletilen I/O tanımlayıcı kümelerini inceler*

KENAR: BLOKLAMA VS. BLOKLANAMAYAN ARAYÜZLER

Engelleme(veya senkronize(**synchronous**) arabirimleri,arayana geri dönmeden önce tüm işleri yapar; engellenemeyen(veya senkronize olmayan(**asynchronous**) arayüzler bazı işler başlatır ancak hemen geri döner, böylece yapılması gereken iş ne olursa olsun arka planda yapılmasına izin verir.

Aramaları engellemedeki olağan suçlu, bir tür I/O'dur. Örneğin, bir aramanın tamamlanması için diskten okunması gerekiyorsa, diske gönderilen I/O isteğinin geri dönmesini bekleyerek bloke olabilir.

Engellenemeyen arabirimler, herhangi bir programlama stilinde kullanılabilir(örneğin, iş parçacığı ile) ancak olay tabanlı yaklaşımda esastır, çünkü bloke eden bir çağrı tüm ilerlemeyi durduracaktır.

Tanımlayıcılardan bazıları okunmaya ve yazılmaya hazırsa veya bekleyen istisnai bir durumu varsa,sırasıyla. Her sette ilk nfds tanımlayıcıları kontrol edilir, yani tanımlayıcı setlerindeki 0'dan nfds-1'e kadar olan tanımlayıcılar incelenir. Dönüşte select(), verilen tanımlayıcı kümelerini, istenen işlem için hazır olan tanımlayıcılardan oluşan alt kümelerle değiştirir. select() tüm kümelerdeki hazır tanımlayıcıların toplam sayısını döndürür.

select() hakkında birkaç nokta. İlk olarak, tanımlayıcıların okunup yazılamayacağını kontrol etmemize izin verdiğini unutmayın; ilki, bir sunucunun yeni bir paket geldiğini ve işlenmesi gerektiğini belirlemesine izin verirken, ikincisi, hizmetin ne zaman yanıt vermesinin uygun olduğunu bilmesini sağlar(yani, giden kuyruğu dolu değil).

İkincisi, zaman aşımı argümanına dikkat edin. Buradaki yaygın kullanımlardan biri, zaman aşımını NULL olarak ayarlamaktadır; bu, bazı tanımlayıcılar hazır olana kadar select() öğesinin süresiz olarak bloke edilmesine neden olur. Ancak, daha sağlam sunucular genellikle bir tür zaman aşımı belirtecektir; yaygın bir teknik, zaman aşımını sıfıra ayarlamak ve böylece hemen geri dönmek için select() çağrısını kullanmaktır.

pool() sistem çağrısı oldukça benzerdir. Ayrıntılar için kılavuz sayfasına veya Stevens ve Rago'ya(SR05) bakın.

Her iki durumda da, bu temel ilkel öğeler bize gelen paketleri kontrol eden, üzerindeki mesajları içeren yuvalardan okuyan ve gerektiğinde yanıt veren, engelleyici olmayan bir olay döngüsü oluşturmamız için bir yol sağlar.

33.3 `select()`'i kullanma

Bunu daha somut hale getirmek için, hangi ağ tanımlayıcılarının üzerlerinde gelen mesajları olduğunu görmek için select()'i nasıl kullanacağımızı inceleyelim. Figür 33.1 buna basit bir örnektir.

Bu kodu anlamak oldukça basittir. Başlatmadan bir süre sonra, sonucu sonsuz bir döngüye girer. Döngü içinde, önce dosya tanıtıcıları kümesini temizlemek için FD_ZERO makrosunu kullanır ve ardından minFD'den maxFD'ye tüm dosya tanımlayıcıları dahil etmek için FD SET()'ikullanır.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
  // open and set up a bunch of sockets (not shown)
  // main loop
  while (1) {
    // initialize the fd_set to all zero
    fd_set readFDs;
    FD_ZERO(&readFDs);

    // now set the bits for the descriptors
    // this server is interested in
    // (for simplicity, all of them from min to max)
    int fd;
    for (fd = minFD; fd < maxFD; fd++)
      FD_SET(fd, &readFDs);

    // do the select
    int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);

    // check which actually have data using FD_ISSET()
    int fd;
    for (fd = minFD; fd < maxFD; fd++)
      if (FD_ISSET(fd, &readFDs))
        processFD(fd);
  }
}
```

Figür 33.1: **select()Kullanarak Basit Kod**

Bu tanımlayıcı seti, örneğin, sunucunun dikkat ettiği tüm ağ soketlerini temsil edebilir. Son olarak, sunucu hangi bağlantılarda veri bulduğunu görmek için select() işlevini geri çağırır. Daha sonra FD_ISSET()'i bir döngüde kullanarak, olay sunucusu hangi tanımlayıcıların hazır veriye sahip olduğunu görebilir ve gelen verileri işleyebilir.

Tabii ki, gerçek bir sunucu bundan daha karmaşık olacaktır ve mesaj gönderirken, disk I/O yayınlarken ve diğer birçok ayrıntıyı kullanırken mantığa ihtiyaç duyar. Daha fazla bilgi için API bilgileri için bkz. Stevens ve Rago[SR05] veya Pai et. al veya Welsh ve ark. olay tabanlı sunucuların genel akışına iyi bir genel bakış için[PDZ99, WCB01].

İpucu: Olay Tabanlı Sunucularda Bloklamayın
Olay tabanlı sunucular görevlerin programlanması üzerinde ayrıntılı kontrol sağlar. Ancak, böyle bir kontrolü sürdürmek için arayanın yürütülmesini engelleyen hiçbir arama yapılamaz; bu tasarım ipucuna uymamak, olay tabanlı bir sunucunun engellenmesine, müşterilerin hüsrana uğramasına ve kitabın bu bölümünü hiç okuyup okumadığınıza dair ciddi sorulara yol açacaktır.

33.4 Neden Daha Basit? Kilit Gerekmez

Tek bir CPU ve olay tabanlı bir uygulama ile eş zamanlı programlarda bulunan sorunlar artık mevcut değil. Spesifik olarak, aynı anda yalnızca bir olay işlendiğinden, kilitleri almaya veya serbest bırakmaya gerek yoktur; olay tabanlı sunucu başka bir iş parçacığı tarafından kesilemez çünkü kesinlikle tek iş parçacıklıdır. Bu nedenle, iş parçacığı programlarında yaygın olan eşzamanlılık hataları, temel olay tabanlı yaklaşımda kendini göstermez.

33.5 Problem: Sistem Çağrılarını Bloklama

Şimdiye kadar olay tabanlı programlama kulağa harika geliyor değil mi? Basit bir döngü programlarsınız ve olayları ortaya çıktıkça yönetirsiniz. Kilitlemeyi düşünmenize bile gerek yok! Ancak bir sorun var: Ya bir olay, sistemi engelleyebilecek bir sistem çağrısı yapmanızı gerektiriyorsa?

Örneğin, bir istemciden bir sunucuya diskten bir dosyayı okumak ve içeriğini istekte bulunan istemciye döndürmek için bir istek geldiğini hayal edin(basit bir HTTP isteği gibi). Böyle bir isteğe hizmet vermek için, bazı olay işleyicilerinin sonunda dosyayı açmak için bir open() sistem çağrısı ve ardından dosyayı okumak için bir dizi read() çağrısı yapması gerekecektir. Dosya belleğe okunduğunda, sunucu muhtemelen sonuçları istemciye göndermeye başlayacaktır.

Hem open() hem de read() çağrıları, depolama sistemine I/O istekleri gönderebilir(gerekli metadata veya veriler zaten bellekte olmadığında) ve bu nedenle hizmet verilmesi uzun zaman alabilir. İş parçacığı tabanlı bir sunucuda bu bir sorun değildir: I/O isteğini yayınlayan iş parçacığı askıya alınırken(I/O'nun tamamlanmasını bekler), diğer iş parçacıkları çalışabilir, böylece sunucunun ilerleme kaydetmesi sağlanır. Aslında, I/O ve diğer hesaplamaların bu doğal örtüşmesi (**overlap**), iş parçacığı tabanlı programlamayı oldukça doğal ve anlaşılır kılan şeydir.

Bununla birlikte, olay tabanlı bir yaklaşımla, çalıştırılacak başka bir iş parçacığı yoktur: Yalnızca ana olay döngüsü. Ve bu, bir olay işleyicisi engelleyen bir çağrı gönderirse, tüm sunucunun tam olarak şunu yapacağı anlamına gelir: Çağrı tamamlanana kadar engelle. Olay döngüsü bloke olduğunda, sistem boşta kalır ve bu büyük bir potansiyel kaynak israfıdır. Dolayısıyla, olay tabanlı sistemlerde uyulması gereken bir kuralımız var: Engelleme çağrılarına izin verilmez.

33.6 Çözüm: Eşzamansız I/O

Bu sınırın üstesinden gelmek için birçok modern işletim sistemi disk sistemine I/O isteklerini göndermek için genel olarak eşzamansız (**asynchronous**) I/O olarak adlandırılan yeni yollar getirmiştir. Bu arabirimler, bir uygulamanın I/O talebi yayınlamasına ve I/O tamamlanmadan hemen arayana kontrolü geri vermesine olanak tanır; ek arabirimler, bir uygulamanın çeşitli I/O'ların tamamlanıp tamamlanmadığını belirlemesine olanak tanır.

Örneğin, bir Mac'te sağlanan arayüzü inceleyelim (diğer sistemlerde benzer API'ler bulunur). API'ler temel bir yapı olarak terminolojide `struct aiocb` veya AIO kontrol bloğu (**AIO control block**) etrafında döner . Yapının basitleştirilmiş bir versiyonu şöyle görünür (daha fazla bilgi için kılavuz sayfalarına bakınız):

```
struct aiocb {
    int             aio_fildes;    // File descriptor
    off_t           aio_offset;    // File offset
    volatile void   *aio_buf;      // Location of buffer
    size_t          aio_nbytes;    // Length of transfer
};
```

Bir dosyaya eşzamansız okuma yapmak için önce bir uygulama bu yapıyı ilgili birimlerle doldurun: okunacak dosyanın dosya tanımlayıcısı (`aio_fildes`), dosya içindeki ofset (`aio_ofset`) ve ayrıca isteğin uzunluğu (`aio_nbytes`) ve son olarak okuma sonuçlarının kopyalanması gereken hedef bellek konumunu alın (`aio_buf`) .

Bu yapı doldurulduktan sonra, uygulamanın dosyayı okumak için asenkron çağrı yapması gerekir; bir Mac'te, bu API basiteçe eşzamansız okuma (**asynchronos read**) API'sidir:

```
int aio_read(struct aiocb *aiocbp);
```

Bu çağrı, I/O'yu düzenlemeye çalışır; başarılı olursa, hemen geri döner ve uygulama (yani olay tabanlı sunucu) işine devam edebilir.

Ancak bulmacanın çözmemiz gereken son bir parçası daha var. Bir I/O'nun ne zaman tamamlandığını ve böylece ara belleğin (`aio_buf` tarafından gösterilen) artık istenen verielere sahip olduğunu nasıl anlayabiliriz?

Son bir API'ye ihtiyacımız var. Mac'te buna (biraz kafa karıştırıcı şekilde) `aio_hatası()` denir. API şuna benziyor:

```
int aio_error(const struct aiocb *aiocbp);
```

Bu sistem çağrısı, `aiocbp` tarafından atıfta bulunulan talebin tamamlanıp tamamlanmadığını kontrol eder. Varsa, rutin başarıyı döndürür (sıfır ile gösterilir); değilse, EINPROGRESS döndürülür. Böylece, bekleyen her eşzamansız I/O için, bir uygulama söz konusu I/O'nun henüz tamamlanıp tamamlanmadığını belirlemek için aio error() çağrısı yoluyla sistemi periyodik olarak yoklayabilir (**poll**).

Fark etmiş olabileceğiniz şey, bir I/O'nun tamamlanıp tamamlanmadığını kontrol etmenin sancılı olduğudur; bir program belirli bir zamanda onlarca veya yüzlerce I/O yayınlamışsa, her birini tekrar tekrar kontrol etmeye devam etmeli mi yoksa önce biraz beklemeli mi, yoksa…?

Bu sorunu çözmek için, bazı sistemler kesmeye (**interrupt**) dayalı bir yaklaşım sunar. Bu yöntem, eşzamansız birI/O tamamlandığında uygulamaları bilgilendirmek için UNIX sinyalleri (**signals**) kullanır ve böylece sisteme tekrar tekrar sorma ihtiyacını ortadan kaldırır. Bu yoklama ve kesme sorunu , I/O cihazlarıyla ilgili bölümde göreceğiniz (ya da daha önce gördüğünüz) gibi cihazlarda da görülmektedir.

Eşzamansız I/O olmayan sistemlerde, saf olay tabalı yaklaşım uygulanamaz. Bununla birlikte, akıllı araştırmacılar, kendi yerlerinde oldukça iyi çalışan yöntemler türetmişlerdir. Örneğin, Pai et al. [PDZ99] olayların ağ paketlerini işlemek için kullanıldığı ve bekleyen I/O'ları yönetmek için bir iş parçacığı havuzunun kullanıldığı hibrit bir yaklaşımı açıklar. Ayrıntılar için makaleleri okuyun.

33.7 Başka Bir Problem: Durum Yönetimi

Olaya dayalı yaklaşımla ilgili diğer bir sorun, bu tür bi kodun yazılmasının geleksel iş parçacığı tabanlı bir koda göre genellikle daha karmaşık olmasıdır. Bunun nedeni şudur: Bir olay işleyici eşzamansız bir I/O yayınladığında, I/O nihayet tamamlandığında bir sonraki olay işleyicinin kullanması için bazı program durumlarını paketlemilidir; programın ihtiyaç duyduğu durum iş parçacığının yığınında olduğundan, iş parçacığı tabanlı programlarda bu ek çalışmaya gerek yoktur. Adya et al. bu çalışma kılavuzu manuel yığın yönetimi (**manuel stack management**) adlandırılır ve olay tabanlı programlama için temeldir[A+02].

Bu noktayı daha somut hale getirmek için, iş parçacığı tabanlı bir sunucunun bir dosya tanımlayıcıdan (`fd`) okuması ve tamamlandıktan sonra dosya okuduğu verileri bir ağ soket tanımlayıcısına yazması gereken (`sd`) bir örneğe bakalım (hata kontrolünü yok sayarak) şöyle görünür:

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

Gördüğünüz gibi, çok iş parçacıklı bir programda bu tür işleri yapmak önemsizdir; `read ()` sonunda döndüğünde, kod hangi sokette yazılacağını hemen bilir çünkü bu bilgi iş parçacığının yığınındadır (`sd` değişkeninde).

Olay tabanlı bir sistemde hayat o kadar kolay değil. Aynı görevi gerçekleştirmek için önce yukarıda açıklanan AIO çağrılarını kullanarak okumayı eşzamansız olarak yayınlarız. Daha sonra aio error() çağrısını kullanarak okumanın tamamlanıp tamamlanmadığını periyodik olarak kontrol ettiğimizi varsayalım; bu çağrı bize okumanın tamamlandığını bildirdiğinde, olay tabanlı sunucu ne yapacağını nasıl biliyor?

Kenara: UNIX Sinyalleri

Tüm modern UNIX varyantlarında sinyaller (**signals**) olarak bilinen devasa ve büyüleyici bir altyapı vardır. En basit haliyle, sinyaller bir süreçle iletişim kurmanın yolunu sağlar. Spesifik olarak, bir uygulamaya bir sinyal iletilebilir; bunu yapmak, uygulamayı bir sinyal işleyici (signal handler), yani uygulamadki bu sinyali işlemek için bazı kodları çalıştırmak için ne yapıyorsa durdurur. Bittiğinde, süreç sadece önceki davranışına devam eder.

Her sinyalin **HUP** (kapatma), **INT** (kesme) **SEGV** (segmentasyon ihlali), vb. Gibi bir adı vardır; ayrıntılar için man sayfasına bakınız. İlginç bir şekilde, bazen sinyali yapan çekirdeğin kendisidir. Örneğin, programınız bir segmentasyon ihlali ile karşılaştığında, işletim sistemi ona bir **SIGSEGV** gönderir (sinyal adlarının başına SIG eklenmesi yaygındır); programınız bu siyali yakalayacak şekilde yapılandırılmışsa, bu hatalı program davranışına yanıt olarak gerçekten bazı kodlar çalıştırabilirsiniz (bu, hata ayıklamaya yardımcı olur). Bir sinyali işlemek için yapılandırılmamış bir işleme sinya gönderildiğinde, varsayılan davranış uygulanır; SEGV için süreç öldürüldü.

İşte sonsuz bir döngüye giren, ancak önce SIGHUP'I yakalamak için sinyal işleyici kuran basit bir program:

```
void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

kill komut satırı aracılığıyla ona sinyal gönderebilirsiniz (evet, bu garip ve agresif bir isim). Bunu yapmak, programdaki ana while döngüsünü kesintiye uğratır ve işleyici kod tutmacını çalıştırır `handle ()`:

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

Sinyaller hakkında öğrenilecek daha çok şey var, o kadar çok ki, tek bir sayfa bir yana, tek bir bölüm bile neredeyse yeterli değil. Herzaman olduğu gibi harika bir kaynak var: Steven ve Rago [SR05]. İlgileniyorsanız daha fazlasını okuyun.

Çözüm, Adya et al. [A+02], devamı (**continuation**) olarak bilinen eski bir programlama dili yapısının kullanılmasıdır [FHK84]. Kulağa karmaşık gelse de, fikir oldukça basit: temel olarak, bu olayı işlemeyi bitirmek için gerekli bilgileri bir veri yapısında kaydedin; olay gerçekleştiğinde (yani, disk I/O tamamlandığında), gerekli bilgilere bakın ve olayı işleyin.

Bu özel durumda çözüm, soket tanımlayıcısını `(sd)` dosya tanımlayıcısı `(fd)` tarafından indekslenen bir tür veri yapısına (örneğin bir hash tablosu) kaydetmek olacaktır. Disk I/O tamamlandığında, olay işleyici dosya tanıtıcıyı kullanarak devamı arar ve bu dosya tanımlayıcının değerini arayana döndürür. Bu noktada (nihayet), sunucu verileri sokete yazmak için son işi yapabilir.

33.8 Olaylarda Hala Zor Olanlar

Olay tabanlı yaklaşımla ilgili bahsetmemiz gereken birkaç başka zorluk daha var. Örneğin, sistemler tek bir CPU'dan birden fazla CPU'ya geçtiğinde, olay tabanlı yaklaşımın basitliği ortadan kalktı. Spesifik olarak, birden fazla CPU kullanmak için olay sunucusunun birden çok olay işleyiciyi paralel olarak çalıştırılması gerekir; bunu yaparken olağan senkronizasyon sorunları (örn. kritik bölümler) ortaya çıkar ve olağan çözümler (örn. kilitler) kullanılmalıdır. Bu nedenle, modern çekirdekli sistemlerde, kilitler olmadan basit olay yönetimi artık mümkün değildir.

Olaya dayalı yaklaşımla ilgili diğer bir sorun, sayfalama (paging) gibi belirli sistem etkinliği türleri ile iyi bütünleşmemesidir. Örneğin, bir olay işleyici sayfası arızalanırsa engellenir ve bu nedenle sunucu, sayfa hatası tamamlanana kadar ilerleme kaydetmez. Sunucu, açık engellemeyi önleyecek şekilde yapılandırılmış olsa da, sayfa hatalarından kaynaklanan bu tür örtülü engellemenin önlenmesi zordur ve bu nedenle yaygın olduğundan büyük performans sorunlarına yol açabilir.

Üçüncü bir sorun, çeşitli rutinlerin tam semantiği değiştikçe, olay tabanlı kodun zaman içerisinde yönetilmesinin zor olabilmesidir [A+02]. Örneğin, bir rutin engellememe durumundan engellemeye dönüşürse, bu rutini çağıran olay işleyiciside kendisini iki parçaya ayırarak yeni doğasına uyum sağlamak için değişmelidir. Engelleme, olay tabanlı sunucular için çok feci olduğundan, bir programcı her olayın kullandığı API'lerin anlambilimindeki bu tür değişiklikleri her zaman göz önünde bulundurmalıdır.

Son olarak, çoğu platformda eşzamansız disk I/O artık mümkün olsa da, oraya ulaşmak uzun zaman aldı[PDZ99] ve hiçbir zaman eşzamansız ağ I/O ile sizin kadar basit ve tek biçimli şekilde tam olarak bütünleşmez düşünülebilir. Örneğin,, bekleyen tüm I/O'ları yönetmek için basitçe select () arayüzünü kullanmak isterken, genellikle ağ iletişimi için bazı select () ve disk I/O için AIO çağrıları gerekir.

33.9 Özet

Olaylara dayalı farklı bir eşzamanlılık tarzına temel bir giriş sunduk. Olay tabanlı sunucular, zamanlama kontrolünü uygulamanın kendisine verir,ancak bunu karmaşıklık ve modern sistemlerin diğier yönleriyle(örn. pag-ing). Bu zorluklar nedeniyle, tek bir yaklaşım en iyisi olarak ortaya çıkmadı; bu nedenle, hem ileti dizilerinin hem de olayların aynı eşzamanlılık sorununa iki farklı yaklaşım olarak uzun yıllar devam etmesi muhtemeldir. Daha fazla bilgi edinmek için bazı araştırma makalelerini okuyun (örn. [A+03, PDZ99, Vb+03, WCB01]) veya daha iyisi olay tabanlı kod yazın.

References

[A+02] "Cooperative Task Management Without Manual Stack Management" by Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur. USENIX ATC '02, Monterey, CA, June 2002. *This gem of a paper is the first to clearly articulate some of the difficulties of event-based concurrency, and suggests some simple solutions, as well explores the even crazier idea of combining the two types of concurrency management into a single application!*

[FHK84] "Programming With Continuations" by Daniel P. Friedman, Christopher T. Haynes, Eugene E. Kohlbecker. In Program Transformation and Programming Environments, Springer Verlag, 1984. *The classic reference to this old idea from the world of programming languages. Now increasingly popular in some modern languages.*

[N13] "Node.js Documentation" by the folks who built node.js. Available: `nodejs.org/api`. *One of the many cool new frameworks that help you readily build web services and applications. Every modern systems hacker should be proficient in frameworks such as this one (and likely, more than one). Spend the time and do some development in one of these worlds and become an expert.*

[O96] "Why Threads Are A Bad Idea (for most purposes)" by John Ousterhout. Invited Talk at USENIX '96, San Diego, CA, January 1996. *A great talk about how threads aren't a great match for GUI-based applications (but the ideas are more general). Ousterhout formed many of these opinions while he was developing Tcl/Tk, a cool scripting language and toolkit that made it 100x easier to develop GUI-based applications than the state of the art at the time. While the Tk GUI toolkit lives on (in Python for example), Tcl seems to be slowly dying (unfortunately).*

[PDZ99] "Flash: An Efficient and Portable Web Server" by Vivek S. Pai, Peter Druschel, Willy Zwaenepoel. USENIX '99, Monterey, CA, June 1999. *A pioneering paper on how to structure web servers in the then-burgeoning Internet era. Read it to understand the basics as well as to see the authors' ideas on how to build hybrids when support for asynchronous I/O is lacking.*

[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens and Stephen A. Rago. Addison-Wesley, 2005. *Once again, we refer to the classic must-have-on-your-bookshelf book of UNIX systems programming. If there is some detail you need to know, it is in here.*

[vB+03] "Capriccio: Scalable Threads for Internet Services" by Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric Brewer. SOSP '03, Lake George, New York, October 2003. *A paper about how to make threads work at extreme scale; a counter to all the event-based work ongoing at the time.*

[WCB01] "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services" by Matt Welsh, David Culler, and Eric Brewer. SOSP '01, Banff, Canada, October 2001. *A nice twist on event-based serving that combines threads, queues, and event-based handling into one streamlined whole. Some of these ideas have found their way into the infrastructures of companies such as Google, Amazon, and elsewhere.*

Ödev (Kod)

Bu (kısa) ev ödevinde, olay tabanlı kod ve bazı temel kavramları hakkında biraz deneyim kazanacaksınız. İyi şanslar!

Sorular

1. İlk olarak, TCP bağlantılarını kabul edebilen ve sunabilen basit bir sunucu yazın. Bunu nasıl yapabileceğinizi bilmiyorsanız, internette biraz gezinmeniz gerekecek. Her seferinde tam olarak bir isteğe hizmet etmek için bunu oluşturun; isteğin çok basit olmasını sağlayın, örneğin günün o anki saatini almak için.

İşte Python'da TCP bağlantılarını kabul edebilen ve sunabilen basit bir sunucu örneği:

```
import socket
from datetime import datetime

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
server_address = ('localhost', 10000)
print('Starting up on {} port {}'.format(*server_address))
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)

while True:
    # Wait for a connection
    print('Waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('Connection from', client_address)

        # Receive the data in small chunks and retransmit it
        while True:
```

```
        # Receive the data in small chunks and retransmit it
        while True:
            data = connection.recv(16)
            print('Received "{}"'.format(data))
            if data:
                # Get the current time and send it back to the client
                now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
                print('Sending time: "{}"'.format(now))
                connection.sendall(now.encode('utf-8'))
            else:
                print('No more data from', client_address)
                break

    finally:
        # Clean up the connection
        connection.close()
```

Bu sunucu, yerel makinede 10000 numaralı bağlantı noktasından gelen bağlantıları dinler. Bir bağlantı kabul edildiğinde istemciden küçük parçalar halinde veri alır ve istemciye geçerli saati göndererek yanıt verir. Bu sunucunun aynı anda yalnızca bir istek sunabileceğini unutmayın. Aynı anda birden çok isteğe hizmet etmesini sağlamak için, her isteği paralel olarak işlemek için bir iş parçacığı veya işlem havuzu kullanabilirsiniz.

2. Şimdi select () arabirimi ekleyin. Birden çok bağlantıyı kabul edebilen bir ana program ve hangi dosya tanıtıcılarında veri bulunduğunu denetleyen bir olay döngüsü oluşturun ve ardından bu istekleri okuyup işleyin. select () işlevini doğru kullandığınızdan emin olun.

```
import select
import socket
from datetime import datetime

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
server_address = ('localhost', 10000)
print('Starting up on {} port {}'.format(*server_address))
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)

# Keep track of the sockets that we need to check for events
inputs = [sock]

while True:
    # Wait for some sockets to be ready for I/O
    readable, _, _ = select.select(inputs, [], [])

    for s in readable:
        if s is sock:
            # A new connection is available
            connection, client_address = s.accept()
```

```
    for s in readable:
        if s is sock:
            # A new connection is available
            connection, client_address = s.accept()
            print('Connection from', client_address)
            inputs.append(connection)
        else:
            # Handle an existing connection
            try:
                # Receive the data in small chunks and retransmit it
                data = s.recv(16)
                print('Received "{}" from {}'.format(data, s.getpeername()))
                if data:
                    # Get the current time and send it back to the client
                    now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
                    print('Sending time: "{}" to {}'.format(now,
s.getpeername()))
                    s.sendall(now.encode('utf-8'))
                else:
                    # No more data is available, so close the connection
                    print('Closing connection with', s.getpeername())
                    inputs.remove(s)
                    s.close()
            except socket.error:
                # An error occurred, so close the connection
                print('Closing connection with', s.getpeername())
                inputs.remove(s)
```

3. Ardından, basit bir web veya dosya sunucusunu taklit etmek için istekleri daha ilginç hale getirelim. Her istek, bir dosyanın içeriğini (istekte adı geçen) okumak için olmalıdır ve sunucu, dosyayı bir ara belleğe okuyarak ve ardından içeriği istemciye geri göndererek yanıt vermelidir. Bu özelliği kullanmak için standart `open()`, `read()`, `close()` sistem çağrılarını kullanın. Burada biraz dikkatli olun: Bunu uzun süre çalışır durumda bırakırsanız, birisi bilgisayarınızdaki tüm dosyaları okumak için onu nasıl kullanacağını anlayabilir!

```
import os
import select
import socket

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
server_address = ('localhost', 10000)
print('Starting up on {} port {}'.format(*server_address))
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)

# Keep track of the sockets that we need to check for events
inputs = [sock]

while True:
    # Wait for some sockets to be ready for I/O
    readable, _, _ = select.select(inputs, [], [])

    for s in readable:
        if s is sock:
            # A new connection is available
            connection, client_address = s.accept()
            print('Connection from', client_address)
            inputs.append(connection)
        else:
            # Handle an existing connection
            try:
                # Receive the data in small chunks and retransmit it
                data = s.recv(16)
                print('Received "{}" from {}'.format(data, s.getpeername()))
                if data:
                    # Parse the file name from the request
                    file_name = data.decode('utf-8').strip()
                    print("Reading file: "{}"".format(file_name))
                    # Read the contents of the file and send them back to the client
                    with open(file_name, 'r') as f:
                        contents = f.read()
                        s.sendall(contents.encode('utf-8'))
                else:
                    # No more data is available, so close the connection
                    print('Closing connection with', s.getpeername())
                    inputs.remove(s)
                    s.close()
            except (OSError, socket.error):
                # An error occurred, so close the connection
                print('Closing connection with', s.getpeername())
                inputs.remove(s)
                s.close()
```

4. Şimdi, standart I/O sistem çağrıları kullanmak yerine, bölümde açıklandığı gibi asenkron I/O arayüzleri kullanın. Eşzamansız arabirimleri programınıza dahil etmek ne kadar zordu?

Python'da aynı anda birden çok bağlantıyı işlemek için eşzamansız I/O arabirimlerini kullanan bir sunucu örneğini burada bulabilirsiniz.

```
import asyncio
import os

async def handle_connection(reader, writer):
    # Receive the data in small chunks and retransmit it
    data = await reader.read(16)
    print('Received "{}" from {}'.format(data,
writer.get_extra_info('peername')))
    if data:
        # Parse the file name from the request
        file_name = data.decode('utf-8').strip()
        print('Reading file: "{}"'.format(file_name))
        # Read the contents of the file and send them back to the client
        with open(file_name, 'r') as f:
            contents = f.read()
            writer.write(contents.encode('utf-8'))
            await writer.drain()
    print('Closing connection with', writer.get_extra_info('peername'))
    writer.close()

async def main():
    # Create a TCP/IP server
    server = await asyncio.start_server(handle_connection, 'localhost', 10000)

    # Wait for incoming connections
    addr = server.sockets[0].getsockname()
```

5. Eğlenmek için kodunuza biraz sinyal işleme ekleyin. Sinyallerin yaygın kullanımlarından biri, bu tür yapılandırma dosyasını yeniden yüklemek veya başka türde bir yönetim eylemi gerçekleştirmek için bir sunucuyu dürtmektir. Belki de bununla oynamanın doğal bir yolu, sunucunuza son erişen dosyaları depolayan kullanıcı düzeyinde bir dosya önbelleği eklemektir. Sunucu işlemine sinyal gönderildiğinde önbelleği temizleyen sinyal işleyici uygulayın.
6. Son olarak, zor kısmımız var: Eşzamansız, olay tabanlı bir yaklaşım oluşturma çabasının buna değip değmediğini nasıl anlarsınız? Faydaları göstermek için bir deney oluşturabilir misiniz? Yaklaşımınız uygulama karmaşıklığını ne kadar arttırdı?