33

Olay Tabanlı Eşzamanlılık (İleri Düzey)

Şimdiye kadar, eşzamanlılık hakkında, eşzamanlı uygulamalar oluşturmanın tek yolu iş parçacıklarını kullanmakmış gibi yazdık. Hayattaki birçok şey gibi, bu da tamamen doğru değildir. Özellikle, hem GUI tabanlı uygulamalarda [O96] hem de bazı internet sunucusu türlerinde [PDZ99] farklı bir eşzamanlı programlama stili sıklıkla kullanılır. **Olay tabanlı eşzamanlılık(event-based concurrency)** olarak bilinen bu stil, **node.js** [N13] gibi sunucu tarafı çerçeveleri de dahil olmak üzere bazı modern sistemlerde popüler hale gelmiştir, ancak kökleri aşağıda tartışacağımız C/UNIX sistemlerinde bulunmaktadır.

Olay tabanlı eşzamanlılığın ele aldığı sorun iki yönlüdür. Birincisi, çok iş parçacıklı uygulamalarda eşzamanlılığı doğru bir şekilde yönetmenin zor olabileceğidir; Tartıştığımız gibi, eksik kilitler, kilitlenme ve diğer kötü problemler ortaya çıkabilir. İkincisi, çok iş parçacıklı bir uygulamada, geliştiricinin belirli bir zamanda neyin zamanlanacağı üzerinde çok az kontrolü vardır yada hiç kontrolü yoktur; bunun yerine, programcı sadece iş parçacıkları oluşturur ve ardından altta yatan işletim sisteminin bunları mevcut CPU'lar arasında makul bir şekilde zamanlamasını umar. Tüm iş yükleri için her durumda iyi çalışan genel amaçlı bir zamanlayıcı oluşturmanın zorluğu göz önüne alındığında, işletim sistemi bazı zamanlarda işi optimumdan daha az bir şekilde zamanlayacaktır. Ve böylece, biz ...

TO CRUX:
İŞ PARÇACIKLARI OLMADAN EŞ ZAMANLI SUNUCU NASIL OLUŞTURULUR

İş parçacıkları kullanmadan eşzamanlı bir sunucu nasıl oluşturabiliriz ve böylece eşzamanlılık üzerindeki kontrolü nasıl koruyabiliriz ve çok iş parçacıklı uygulamaları rahatsız eden bazı sorunlardan nasıl kaçınabiliriz?

33.1 Temel Fikir: Bir Olay Döngüsü

Yukarıda belirtildiği gibi kullanacağımız temel yaklaşıma **olay tabanlı eşzamanlılık(event-based concurrency)** denir. Yaklaşım oldukça basittir: sadece bir şeyin (yani bir "olayın") gerçekleşmesini beklersiniz ; ne zaman yaparsa ne tür bir

olması durumunda ve gerektirdiği az miktarda işi yapması (G/Ç istekleri yayınlamak veya gelecekteki kullanımlar için diğer olayları zamanlamak vb.) İşte bu kadar!

Ayrıntılara girmeden önce, önce standart bir olay tabanlı sunucunun nasıl göründüğünü inceleyelim . 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(ler);
}
```

Gerçekten bu kadar basit. Ana döngü sadece bir şeyler yapmayı bekler (yukarıdaki kodda getEvents() öğesini çağırarak) ve sonra, yeniden döndürülen her olay için, bunları birer birer işler; her olayı işleyen kod, **olay işleyicisi(event handler)** olarak bilinir. Daha da önemlisi, bir işleyici bir olayı işlediğinde, sistemde gerçekleşen tek etkinliktir; bu nedenle, bir sonraki olayın hangi olay olacağına karar vermek, zamanlamaya eşdeğerdir. Zamanlama üzerindeki bu açık kontrol, olay tabanlı yaklaşımın temel avantajlarından biridir.

Ancak bu tartışma bizi daha büyük bir soruyla karşı karşıya bırakıyor: Olay tabanlı bir sunucu, özellikle ağ ve disk G/Ç açısından hangi olayların gerçekleştiğini tam olarak nasıl belirler? Özellikle, bir olay sunucusu kendisine bir mesaj ulaşıp ulaşmadığını nasıl anlayabilir?

33.2 Önemli bir API: select() (veya poll())

Bu temel olay döngüsünü göz önünde bulundurarak, daha sonra olayların nasıl alınacağı sorusunu ele almalıyız. Çoğu sistemde, select() veya poll() sistem çağrıları aracılığıyla temel bir API kullanılabilir .

Bu arayüzlerin bir programın yapmasını sağladığı şey basittir: İlgilenmesi gereken herhangi bir G/Ç olup olmadığını kontrol etmektir. Örneğin, bir ağ uygulamasının (bir web sunucusu gibi) herhangi bir ağ paketinin gelip gelmediğini kontrol etmek istediğini, bunlara hizmet vermek için sipariş verdiğini imgeleyin. Bu sistem çağrıları tam olarak bunu yapmanızı sağlar.

Örneğin select() öğesini ele alalım. Kılavuz sayfası (Mac'te) API'yi şu şekilde açıklar:

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

Man sayfasındaki gerçek açıklama: *select()*, adresleri readfds, *writefds ve errorfd'lerde geçirilen G/Ç tanımlayıcı kümelerini inceleyerek görür.*

BİR KENARA: ENGELLEME VE ENGELLEMESİZ ARAYÜZLER

Bloklayan (veya **senkron(synchronous)**) arayüzler, arayana dönmeden önce tüm işlerini yapar ; Engellemeyen (veya **asenkron(asynchronous)**) arayüzler bazı çalışmalara başlar, ancak hemen geri döner, böylece yapılması gereken her işin arka planda yapılmasına izin verir.

Çağrıların bloke olmasının olağan sorumlusu bir tür G/Ç'dir. Örneğin, bir çağrının tamamlanması için diskten okunması gerekiyorsa, diske gönderilen G/Ç isteğinin geri dönmesini bekleyerek bloke olabilir.

Bloklamayan arayüzler herhangi bir programlama tarzında (örneğin, iş parçacıklarıyla) kullanılabilir, ancak bloklayan bir çağrı tüm ilerlemeyi durduracağından olay tabanlı yaklaşımda çok önemlidir.

tanımlayıcılarından bazıları sırasıyla okumaya hazırsa, yazmaya hazırsa veya bekleyen istisnai bir durum varsa. Her kümede ilk nfds tanımlayıcıları kontrol edilir, yani tanımlayıcı kümelerde 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 yazıldığı gibi okunup *okunamayacağını* kontrol etmenize izin verdiğine dikkat edin; birincisi bir sunucunun yeni bir paketin geldiğini ve işlenmesi gerektiğini belirlemesine izin verirken, ikincisi hizmetin yanıtlamanın ne zaman uygun olduğunu bilmesini sağlar (yani, giden kuyruk dolu değildir).

İkinci olarak, zaman aşımı argümanına dikkat edin. Buradaki yaygın kullanımlardan biri, zaman aşımını NULL olarak ayarlamaktır, bu da select() işlevinin bazı tanımlayıcılar hazır olana kadar süresiz olarak bloke olmasına neden olur. Bununla birlikte, daha sağlam sunucular genellikle bir tür zaman aşımı belirtir; yaygın bir teknik, zaman aşımını sıfıra ayarlamak ve böylece hemen geri dönmek için select() çağrısını kullanmaktır.

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

Her iki durumda da, bu temel ilkeller bize, basitçe gelen paketleri kontrol eden, üzerlerinde mesajlar bulunan soketlerden okuyan ve gerektiğinde yanıt veren, engellemeyen bir olay döngüsü oluşturmanın bir yolunu sunar.

33.3 select() öğesini kullanma

Bunu daha somut hale getirmek için, hangi ağ tanımlayıcılarının üzerinde gelen iletiler olduğunu görmek için select() öğesinin nasıl kullanılacağını inceleyelim. Şekil 33.1'de basit bir örnek gösterilmektedir.

Bu kodun anlaşılması aslında oldukça basittir. Bazı başlangıç işlemlerinden sonra, sunucu sonsuz bir döngüye girer. Döngünün içinde, önce dosya tanımlayıcıları kümesini temizlemek için FD ZERO() makrosunu kullanır ve ardından minFD'den maxFD'ye kadar tüm dosya tanımlayıcılarını dahil etmek için FD SET() makrosunu kullanır.

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

int main(void) {
    // bir grup soketi açın ve kurun (gösterilmemiştir)
    // ana döngü
    while (1) {
        // fd_set'i tümüyle sıfır olarak başlatın
        fd_set readFDs;
        FD_ZERO(&readFDs);

        // şimdi tanımlayıcılar için bitleri ayarlayın
        // bu sunucu şunlarla ilgileniyor
        // (basitlik için, hepsi min'den max'a)
        int fd;
        for (fd = minFD; fd < maxFD; fd++)
            FD_SET(fd, &readFDs);

        // seçme işlemini yap
        int rc = select(maxFD+1, &readFDs, NULL, NULL , NULL);

        // FD_ISSET() kullanarak gerçekte hangi verilerin bulunduğunu kontrol edin
        int fd;
        for (fd = minFD; fd < maxFD; fd++)
          if(FD_ISSET(fd, &readFDs))
            processFD(fd);
    }
}
```

Şekil 33.1: **select() Kullanan Basit Kod**

Bu tanımlayıcılar kümesi, örneğin, sunucunun dikkat ettiği tüm ağ çalışma soketlerini temsil edebilir. Son olarak, sunucu hangi bağlantıların üzerinde kullanılabilir veri olduğunu görmek için select() öğesini çağırır. Daha sonra bir döngüde FD ISSET() 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 G/Ç verirken ve diğer birçok ayrıntıda kullanmak için mantık gerektirir . Daha fazla bilgi için, API bilgileri için Stevens ve Rago [SR05] veya olay tabanlı sunucuların genel akışına iyi bir genel bakış için Pai ve diğerlerine veya Welsh ve diğerlerine bakın [PDZ99, WCB01].

İPUCU: OLAY TABANLI SUNUCULARDA ENGELLEME YAPMAYIN
Olay tabanlı sunucular, görevlerin zamanlanması üzerinde ayrıntılı denetim sağlar. Bununla birlikte, böyle bir kontrolü sürdürmek için, arayanın yürütülmesini engelleyen hiçbir çağrı yapılamaz; Bu tasarım ipucuna uymamak, engellenmiş bir olay tabanlı sunucuya, hayal kırıklığına uğramış istemcilere ve kitabın bu bölümünü hiç okuyup okumadığınıza dair ciddi sorulara neden olacaktır.

33.4 Neden Daha Basit? Kilit Gerektirmez

Tek bir CPU ve olay tabanlı bir uygulamayla, eşzamanlı programlarda bulunan sorunlar artık mevcut değildir. Özellikle, aynı anda yalnızca bir olay işlendiğinden, kilitleri almaya veya serbest bırakmaya gerek yoktur; olay tabanlı sunucu başka bir iş parçacığı tarafından kesintiye uğratılamaz , çünkü kesinlikle tek iş parçacıklıdır. Bu nedenle, iş parçacıklı programlarda yaygın olan eşzamanlılık hataları temel olay tabanlı yaklaşımda ortaya çıkmaz.

33.5 Bir Sorun: Sistem Çağrılarını Engelleme

Şimdiye kadar, olay tabanlı programlama kulağa harika geliyor, değil mi? Basit bir döngü programlıyorsunuz ve olayları ortaya çıktıkça ele alıyorsunuz. Kilitlemeyi düşünmenize bile gerek yok! Ancak bir sorun var: Bir olay, bloklanabilecek bir sistem çağrısı yapmanızı gerektiriyorsa ne olur?

Örneğin, bir istemciden bir dosyayı diskten okumak ve içeriğini istekte bulunan istemciye döndürmek için bir istemciden sunucuya bir istek geldiğini düşünün (basit bir HTTP isteği gibi). Böyle bir isteği karşılamak için, bazı olay yöneticilerinin dosyayı açmak için bir open() sistem çağrısı yapması ve ardından dosyayı okumak için bir dizi read() çağrısı yapması gerekir. Dosya belleğe okunduğunda , sunucu büyük olasılıkla sonuçları istemciye göndermeye başlayacaktır.

Hem open() hem de read() çağrıları , depolama sistemine G/Ç istekleri gönderebilir (gerekli meta veriler veya veriler zaten bellekte olmadığında) ve bu nedenle hizmet vermesi uzun zaman alabilir. İş parçacığı tabanlı bir sunucuda, bu sorun değildir: G/Ç isteğini veren iş parçacığı askıya alınırken (G/Ç'nin tamamlanmasını beklerken), diğer iş parçacıkları çalışabilir ve böylece sunucunun ilerleme kaydetmesini sağlar. Gerçekten de G/Ç ve diğer hesaplamaların bu doğal **örtüşmesi(overlap)** , iş parçacığı tabanlı programlamayı oldukça doğal ve basit kılan şeydir.

Bununla birlikte, olay tabanlı bir yaklaşımla, çalıştırılacak başka bir iş parçacığı yoktur : yalnızca ana olay döngüsü vardır. Ve bu, bir olay işleyicisi engelleyen bir çağrı yayınlarsa, *tüm* sunucunun tam olarak şunu yapacağı anlamına gelir: çağrı tamamlanana kadar engelle. Olay döngüsü bloke edildiğinde, sistem boşta oturur ve bu nedenle büyük bir kaynak kaybı potansiyeli vardır. Bu nedenle, olay tabanlı sistemlerde uyulması gereken bir kuralımız var : engelleme çağrılarına izin verilmez.

33.6 Bir Çözüm: Asenkron G/Ç

Bu sınırın üstesinden gelmek için, birçok modern işletim sistemi disk sistemine G/Ç istekleri göndermek için genel olarak **asenkron G/Ç (asynchronous I/O)** olarak adlandırılan yeni yollar geliştirmiştir. Bu arabirimler, bir uygulamanın bir G/Ç isteği yayınlamasını ve G/Ç'nin tamamlandığını belirten çağırana derhal kontrol göndermesini sağlar; ek arabirimler bir uygulamanın çeşitli G/Ç'lerin tamamlanıp tamamlanmadığını belirlemesini sağlar .

Örneğin, bir Mac'te sağlanan arayüzü inceleyelim (diğer sistemler de benzer API'lere sahiptir). API'ler temel bir yapı, struct aiocb veya yaygın termolojide **AIO kontrol bloğu (AIO control block)** etrafında döner. Yapının basitleştirilmiş bir versiyonu şu şekildedir(daha fazla bilgi için kılavuz sayfaları):

```
struct aiocb {
    int             aio_fildes;   // Dosya tanımlayıcısı
    off_t           aio_offset    // Dosya seti
    volatile void   *aio_buf;     // Tamponun konumu
    size_t          aio_nbytes;   // Transfer süreci

};
```

Bir dosyaya eşzamansız okuma yapmak için, bir uygulama öncelikle bu yapıyı ilgili bilgilerle doldurun: okunacak dosyanın dosya tanımlayıcısı (aio fildes), dosya içindeki ofset (aio offset) ve isteğin uzunluğu (aio nbytes) ve son olarak okuma sonuçlarının içine girdiği bellek konumunu alır ve okunan kopyalalanmalıdır (aio buf).

Bu yapı doldurulduktan sonra, uygulamanın dosyayı okumak için asenkron çağrısını vermesi gerekir ; bir Mac'te, bu API basitçe **asenkron okuma (asynchronous read)** API'sidir:

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

Bu çağrı G/Ç'yi vermeye çalışır; başarılı olursa, hemen geri döner ve uygulama (yani olay tabanlı sunucu) çalışmalarına devam edebilir.

Bununla birlikte , çözmemiz gereken bulmacanın son bir parçası var. Bir G/Ç'nin ne zaman tamamlandığını ve böylece arabelleğin (aio buf tarafından işaret edilen) artık içinde istenen verilere sahip olduğunu nasıl anlayabiliriz ?

Son bir API'ye daha ihtiyaç vardır. Mac'te, (biraz kafa karıştırıcı bir şekilde) aio error() olarak adlandırılır. API şöyle görünür:

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

Bu sistem çağrısı, aiocbp tarafından başvurulan isteğin tamamlanıp tamamlanmadığını denetler. Varsa, rutin başarıyı döndürür (sıfır ile gösterilir); değilse, EINPROGRESS döndürülür. Böylece, bekleyen her eş zamanlı G/Ç için, bir uygulama, söz konusu G/ Ç'nin henüz tamamlanıp tamamlanmadığını belirlemek için aio error() çağrısı yoluyla sistemi periyodik olarak **sorgulayabilir(poll).**

Fark etmiş olabileceğiniz bir şey, bir G/Ç'nin tamamlanıp tamamlanmadığını kontrol etmenin acı verici olduğudur; bir programda belirli bir zamanda onlarca veya yüzlerce G/Ç yayınlanmışsa, her birini tekrar tekrar kontrol etmeye devam etmeli veya önce biraz beklemeli mi, veya ... ?

Bu sorunu gidermek için, bazı sistemler **kesmeye(interrupt)** dayalı bir yaklaşım sağlar. Bu yöntem, eşzamansız bir G/Ç tamamlandığında uygulamaları bilgilendirmek için UNIX **sinyallerini (signals)** kullanır, böylece sisteme tekrar tekrar sorma ihtiyacını ortadan kaldırır. G/Ç aygıtları bölümünde göreceğiniz (ya da zaten görmüş olduğunuz) gibi, bu yoklama ve kesme sorunu aygıtlarda da görülür.

Eşzamansız G/Ç olmayan sistemlerde, saf olay tabanlı yaklaşım uygulanamaz. Bununla birlikte, zeki araştırmacılar, yerlerinde oldukça iyi çalışan yöntemler türetmişlerdir. Örneğin, Pai ve diğerleri [PDZ99], olayların ağ paketlerini işlemek için kullanıldığı ve bekleyen G/Ç'leri yönetmek için bir iş parçacığı havuzunun kullanıldığı karma bir yaklaşımı açıklar. Ayrıntılar için makalelerini okuyun.

33.7 Başka Bir Sorun: Devlet Yönetimi

Olay tabanlı yaklaşımla ilgili bir diğer sorun , bu tür kodların yazılmasının geleneksel iş parçacığı tabanlı koddan genel olarak daha karmaşık olmasıdır. Bunun nedeni şudur: Bir olay işleyicisi eşzamansız bir G/Ç yayınladığında, G/Ç nihayet tamamlandığında bir sonraki olay işleyicisinin kullanması için bazı program durumlarını paketlemesi gerekir ; programın ihtiyaç duyduğu durum iş parçacığının yığınında olduğundan, iş parçacığı tabanlı programlarda bu ek işe gerek yoktur. Adya ve arkadaşları bu işi **manuel yığın yönetimi (manual stack management)** olarak adlandırır ve olay tabanlı programlamanın temelini oluşturur. [A+02].

Bu noktayı daha somut hale getirmek için, iş parçacığı tabanlı bir sunucunun bir dosya tanımlayıcısından (fd) okuması ve tamamlandıktan sonra dosyadan okuduğu verileri bir ağ soketi tanımlayıcısına (sd) yazması gereken basit bir örneğe bakalım. Kod (hata denetimini yoksayarak) şöyle görünür:

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

Gördüğünüz gibi, çok iş parçacıklı bir programda, bu tür bir işi yapmak önemsizdir; read() nihayet geri döndüğünde, kod hangi sokete yazılacağını hemen bilir, çünkü bu bilgi iş parçacığının yığınındadır (sd değişkeninde).

Olaya tabanlı bir sistemde, hayat o kadar kolay değildir. Aynı görevi gerçekleştirmek için, önce yukarıda açıklanan AIO çağrılarını kullanarak okumayı eşzamansız olarak gerçekleştiririz. Diyelim ki aio error() çağrısını kullanarak okumanın tamamlanıp tamamlanmadığını periyodik olarak kontrol ediyoruz; bu çağrı bize okumanın tamamlandığını bildirdiğinde, olay tabanlı sunucu ne yapacağını nasıl biliyor?

BİR KENARA: UNIX SİNYALLERİ

Sinyaller (signals) olarak bilinen devasa ve büyüleyici bir altyapı, tüm modern UNIX varyantlarında mevcuttur. En basit haliyle, sinyaller bir süreçle iletişim kurmanın bir yolunu sağlar. Spesifik olarak, bir sinyal bir uygulamaya iletilebilir; bunu yapmak, uygulamanın bir **sinyal işleyicisini (signal handler)** çalıştırmak için ne yaptığını , yani bu sinyali işlemek için uygulamadaki bazı kodları durdurur. Tamamlandığında, işlem sadece önceki davranışına devam eder.

Her sinyalin **HUP** (kapatma), **INT** (kesme), **SEGV** (segmentasyon ihlali) vb. gibi bir adı vardır; ayrıntılar için man sayfasına bakın. İlginçtir ki, bazen sinyalizasyonu yapan çekirdeğin kendisidir. Örneğin, programınız bir segmentasyon ihlaliyle karşılaştığında, işletim sistemi ona bir **SIGSEGV** gönderir (sinyal adlarına **SIG** eklenmesi yaygındır); programınız bu sinyali yakalayacak şekilde yapılandırılmışsa, bu hatalı program davranışına yanıt olarak bazı kodlar çalıştırabilirsiniz (bu, hata ayıklama için yararlıdır). Bir sinyali işlemek üzere yapılandırılmamış bir işleme sinyal gönderildiğinde, varsayılan davranış yürürlüğe girer; SEGV için süreç öldürülür.

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

```
void handle(int arg) {
        printf("beni uyandırmayı bırak ...\n");
}

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

Kill komut satırı aracıyla sinyal gönderebilirsiniz (evet, bu garip ve agresif bir isimdir). Bunu yapmak, programdaki ana while döngüsünü kesecek ve handle() işleyici kodunu çalıştırıracaktır:

```
prompt> . /main &

[3] 36705
prompt> kill -HUP 36705
beni uyandırmayı bırak...
prompt> kill -HUP 36705
beni uyandırmayı bırak...
```

Sinyaller hakkında öğrenilecek çok şey var, o kadar ki tek bir bölüm, tek bir sayfa bile neredeyse yeterli değil. Her zaman olduğu gibi, harika bir kaynak var: Stevens ve Rago [SR05]. İlgileniyorsanız daha fazla bilgi edinin.

Adya ve diğerleri [A+02] tarafından tanımlandığı gibi çözüm, **devamlılık(continuation)** olarak bilinen eski bir programlama dili yapısını kullanmaktır[FHK84]. Karmaşık gibi görünse de , fikir oldukça basittir: temel olarak, bazı veri yapılarında olayın işlenmesini bitirmek için gerekli bilgileri kaydedin; olay gerçekleştiğinde (yani, disk G/Ç tamamlandığında), gerekli bilgileri arayın ve olayı işleyin .

Bu özel durumda, çözüm, soket tanımlayıcısını (sd) dosya tanımlayıcısı (fd) tarafından dizine eklenmiş bir tür veri klasörüne (örneğin, bir hash tablo)kaydetmek olacaktır. Disk G/Ç işlemi tamamlandığında, olay yöneticisi, devamlılığı aramak için dosya tanımlayıcısını kullanır ve bu da soket tanımlayıcısının değerini arayan kişiye geri döndürür. Bu noktada (son olarak), sunucu daha sonra verileri sokete yazmak için son işi yapabilir.

33.8 Etkinliklerle İlgili Hala Zor Olan Nedir?

Olay temelli yaklaşımda bahsetmemiz gereken birkaç zorluk daha var. Örneğin, sistemler tek bir CPU'dan birden fazla CPU'ya geçtiğinde, olay tabanlı yaklaşımın basitliğinin bir kısmı ortadan kalktı. Özellikle, birden fazla CPU kullanmak için, olay sunucusunun birden fazla olay işleyicisini paralel olarak çalıştırması gerekir; bunu yaparken, olağan senkronizasyon sorunları (örneğin, kritik bölümler) ortaya çıkar ve olağan çözümler (örneğin, kilitler) kullanılmalıdır. Bu nedenle, modern çok çekirdekli sistemlerde, kilitler olmadan basit olay işleme artık mümkün değildir.

Olay tabanlı yaklaşımla ilgili bir başka sorunda, **sayfalama(paging)** gibi belirli sistem etkinliği türleriyle iyi entegre olmamasıdır. Örneğin, bir olay işleyici sayfası hata verirse, engellenir ve böylece sayfa hatası tamamlanana kadar sunucu ilerleme kaydetmez. Sunucu açık engellemeyi önleyecek *şekilde* yapılandırılmış olsa da, sayfa hataları nedeniyle bu tür *örtülü* engellemelerden kaçınmak zordur ve bu nedenle yaygın olduğunda büyük performans sorunlarına yol açabilir .

Üçüncü bir sorun ise, çeşitli rutinlerin tam semantikleri değiştiğinden, olay tabanlı kodun zaman içinde yönetilmesinin zor olabileceğidir [A + 02]. Örneğin, bir rutin engellememeden engellemeye geçerse, bu yordamı çağıran olay işleyicisinin de kendisini iki parçaya bölerek yeni doğasına uyacak şekilde değişmesi gerekir. Engelleme, olay tabanlı sunucular için çok feci olduğundan, bir program , her olayın kullandığı API'lerin semantiklerinde bu tür değişikliklere her zaman dikkat etmelidir.

Son olarak, asenkron disk G/Ç artık çoğu plat formunda mümkün olsa da, oraya ulaşmak uzun zaman aldı [PDZ99] ve asenkron ağ G/Ç ile hiçbir zaman olabildiğince basit ve tekdüze bir şekilde bütünleşmez. Örneğin, biri tüm bekleyen tüm G/Ç'leri yönetmek için select() arabirimini kullanmak isterken , genellikle ağ iletişimi için select() ve disk G/Ç için AIO çağrılarının bir kombinasyonu gereklidir.

33.9 Özet

Olaylara dayalı farklı bir para birimi tarzına basit bir giriş sunduk. Olay tabanlı sunucular, zamanlamanın kontrolünü uygulamanın kendisine verir, ancak bunu karmaşıklık ve modern sistemlerin diğer yönleriyle (örneğin, sayfalama) entegrasyon zorluğu gibi bazı maliyetlerle yapar. Bu zorluklar nedeniyle, tek bir yaklaşım en iyisi olarak ortaya çıkmamıştır; bu nedenle hem iş parçacıkları hem de olayların uzun yıllar boyunca aynı eşzamanlılık sorununa iki farklı yaklaşım olarak devam etmesi muhtemeldir. Daha fazla bilgi edinmek için bazı araştırma makalelerini okuyun (örneğin, [A + 02, PDZ99, vB + 03, WCB01]) veya daha iyisi , biraz 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) ödevde, 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 yapacağınızı bilmiyorsanız, internette biraz araştırmanız gerekecek. Bunu bir seferde tam olarak bir istek sunacak şekilde oluşturun; her isteğin çok basit olmasını sağlayın, örneğin, günün geçerli saatini almak gibi.

server.py

```python
import socket

host = "localhost"
port = 12345


try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print("socket oluşturuldu")

    s.bind((host, port))
    print("socket {} nolu porta bağlandı".format(port))

    s.listen(5)
    print("socket dinleniyor")
except socket.error as msg:
    print("Hata:",msg)

while True:

    # Client ile bağlantı kurulursa
    c, addr = s.accept()
    print('Gelen bağlantı:', addr)

    # Bağlanan client tarafına hoşgeldin mesajı gönderelim.
    mesaj = 'Bağlantı için teşekkürler'
    c.send(mesaj.encode('utf-8'))



    # Bağlantıyı sonlandıralım
    c.close()
```

client.py

```python
import socket
import datetime

# Socket oluşturulması
s = socket.socket()

# Bağlanılacak adres ve port
host = "localhost"
port = 12345

try:
    # Bağlantıyı yap
    s.connect((host, port))

    # serverden yanıtı al
    yanit = s.recv(1024)
    print(yanit.decode("utf-8"))

    now = datetime.datetime.now()
    print ("Şu anki Tarih ve Saat : ")
    print (now.strftime("%Y-%m-%d %H:%M:%S"))

    # bağlantıyı kapat
    s.close()
except socket.error as msg:
    print("[Server aktif değil.] Mesaj:", msg)
```

Komut İstemi - python server.py

```
Microsoft Windows [Version 10.0.19044.2364]
(c) Microsoft Corporation. Tüm hakları saklıdır.

C:\Users\developers>cd desktop\python

C:\Users\developers\Desktop\python>python server.py
socket oluşturuldu
socket 12345 nolu porta bağlandı
socket dinleniyor
Gelen bağlantı: ('127.0.0.1', 53939)
Gelen bağlantı: ('127.0.0.1', 53940)
```

Komut İstemi

```
Microsoft Windows [Version 10.0.19044.2364]
(c) Microsoft Corporation. Tüm hakları saklıdır.

C:\Users\developers>cd desktop\python

C:\Users\developers\Desktop\python>python client.py
Bağlantı için teşekkürler
Şu anki Tarih ve Saat :
2022-12-15 17:17:24

C:\Users\developers\Desktop\python>client.py
Bağlantı için teşekkürler
Şu anki Tarih ve Saat :
2022-12-15 17:17:34

C:\Users\developers\Desktop\python>
```

Burda TCP bağlantılarını kullanan bir server kurduk ve bağlantıya geçtik. Serverda günün saatini kullanan bir ifade ekledik ve böylelikle bağlantı saatini öğrenmiş olduk.

2. Şimdi, select() arabirimini ekleyin. Birden çok bağlantıyı kabul edebilen bir ana program ve hangi dosya tanımlayıcılarında veri olduğunu kontrol eden ve ardından bu istekleri okuyup işleyen bir olay döngüsü oluşturun. select() işlevini doğru kullandığınızı dikkatlice test ettiğinizden emin olun.

soru2

```
while True:
    # Dosya tanımlayıcılarını seçmeyi bekle
    readable, writable, exceptional = select.select(inputs, outputs, inputs)

    # Gelen istekleri işle
    for r in readable:
        if r is server:
            # Yeni bir bağlantı kabul et
            connection, client_address = r.accept()
            inputs.append(connection)
        else:
            # Bağlı bir istemciden veri oku
            data = r.recv(1024)
            if data:
                # Veriyi işle ve cevapla
                process_data(data)
                r.send(response)
            else:
                # Bağlantıyı kapat
                inputs.remove(r)
                r.close()

    # Yazılabilir dosya tanımlayıcıları için veri gönder
    for w in writable:
        # Veriyi gönder
        w.send(data)

    # Özel durumlar için işlem yap
    for e in exceptional:
        # Bağlantıyı kapat ve inputs listesinden kaldır
        inputs.remove(e)
        e.close()
```

Bu kod, bir sunucunun birden çok bağlantıyı kabul edebilmesini ve hangi bağlantılarda veri olduğunu kontrol eder.

3. Ardından, basit bir web veya dosya sunucusunu taklit etmek için istekleri biraz daha ilginç hale getirelim . Her istek, bir dosyanın (istekte adlandırılan) içeriğini okumak için olmalı ve sunucu , dosyayı bir arabelleğe okuyarak ve ardından içeriği istemciye döndürerek yanıt vermelidir. Bu özelliği uygulamak için standart open(), read(), close() sistem çağrılarını kullanın . Burada biraz dikkatli olun: Bunu uzun süre çalışır durumda bırakırsanız , birisi bilgisayarınızdaki tüm dosyaları okumak için nasıl kullanılacağını bulabilir!

soru3

```
# İstemci tarafından gönderilen istekleri oku ve dosya adını çıkart
request = r.recv(1024)
filename = parse_request(request)

# Dosyayı aç ve içeriğini bir arabellekte oku
with open(filename, 'r') as f:
    file_content = f.read()

# Arabellekteki dosya içeriğini istemciye döndür
r.send(file_content)

# Dosyayı ve bağlantıyı kapat
f.close()
r.close()
```

İstemci tarafından gönderilen istekleri okuruz ve dosya adına çıkartırız. Dosya adını kullanarak dosyayı açar ve içeriğini arabelleğe okuturuz. Arabellekteki dosya içeriğinide istemciye döndürürüz.

4. Şimdi, standart G/Ç sistem çağrılarını kullanmak yerine, bölümde açıklandığı gibi eşzamanlı G/Ç arayüzlerini kullanın. Asenkron arayüzleri programınıza dahil etmek ne kadar zor oldu?

5. Eğlenmek için, kodunuza biraz sinyal işleme ekleyin. Sinyallerin yaygın kullanımlarından biri, bir sunucuyu bir tür yapılandırma dosyasını yeniden yüklemek veya başka bir tür yönetim eylemi gerçekleştirmek için dürtmektir. Belki de bununla oynamanın doğal bir yolu , sunucunuza son erişilen dosyaları depolayan kullanıcı düzeyinde bir dosya önbelleği eklemektir. Sinyal sunucu işlemine gönderildiğinde önbelleği temizleyen bir sinyal işleyicisi uygulayın.

soru5

```
# Sinyal işleyicisi oluştur
def clear_cache(signum, frame):
    # Önbelleği temizle
    cache.clear()

# Sinyal işleyicisini tanımla
signal.signal(signal.SIGUSR1, clear_cache)

# Önbelleği doldur
cache = {}
while True:
    # İstemci tarafından gönderilen istekleri oku ve dosya adını çıkart
    request = r.recv(1024)
    filename = parse_request(request)

    # Dosyayı aç ve içeriğini bir arabellekte oku
    if filename in cache:
        # Önbellekte varsa dosya içeriğini kullan
        file_content = cache[filename]
    else:
        # Önbellekte yoksa dosyayı oku ve önbelleğe ekle
        with open(filename, 'r') as f:
            file_content = f.read()
        cache[filename] = file_content

    # Arabellekteki dosya içeriğini istemciye döndür
    r.send(file_content)

    # Dosyayı ve bağlantıyı kapat
    f.close()
    r.close()
```

Bu kod, bir sinyal işleyicisi oluşturarak sunucunuza bir dosya önebllеği ekler. İstemci tarafından gelen istekleri okur ve dosya adını çıkartır. Bu

sinyal işleyicisi SIGUSR1 sinyali gönderildiğinde önbelleği temizler. Bu sayede bir yapılandırma dosyasını yeniden yüklemek veya başka bir yöntem eylemi gerçekleştirmek için sunucuya bir sinyal gönderebilirsiniz.

6. Son olarak, zor kısmı: Eşzamansız, olay tabanlı bir yaklaşım oluşturmak için harcadığınız çabaya değip değmeyeceğini nasıl anlayabilirsiniz? Faydaları göstermek için bir deney oluşturabilir misiniz? Yaklaşımınız ne kadar uygulama karmaşıklığı ekledi?