

Concurrency: An Introduction (BİRLİKTELİK: GİRİŞ)

Şimdiye kadar, işletim sisteminin gerçekleştirdiği temel soyutlamaların gelişimini gördük. Tek bir fiziksel CPU'yu alıp birden çok sanal CPU'ya(**virtual CPUs**) nasıl dönüştüreceğimizi gördük, böylece birden çok programın aynı anda çalıştığı yanılsamasını sağladık. Her işlem için büyük, özel bir sanal bellek (**virtual memory**) yanılsamasının nasıl yaratılacağını da gördük; adres uzayının bu soyutlaması, aslında işletim sistemi adres alanlarını fiziksel hafıza (ve bazen disk) boyunca gizlice çoklarken, her programın kendi hafızası varmış gibi davranmasını sağlar.

Bu notta, çalışan tek bir süreç için yeni bir soyutlama tanıtıyoruz: iş parçacığı soyutlaması(**thread**). Bir program içindeki tek bir yürütme noktasına (yani, talimatların alındığı ve yürütüldüğü tek bir PC) ilişkin klasik görüşümüzün yerine, çok iş parçacıklı (**multi-threaded**) bir programın birden fazla yürütme noktası vardır (yani, birden fazla PC, her biri getiriliyor ve yürütülüyor). Belki de bunu düşünmenin başka bir yolu, her iş parçacığının bir fark dışında ayrı bir süreç gibi olduğudur: aynı adres alanını paylaşırlar ve bu nedenle aynı verilere erişebilirler.

Tek bir iş parçacığının durumu bu nedenle bir işlemin durumuna çok benzer. Programın talimatları nereden getirdiğini izleyen bir program sayacına (PC) sahiptir. Her iş parçacığının, hesaplama için kullandığı kendi özel kayıt kümesi vardır; bu nedenle, tek bir işlemci üzerinde çalışan iki iş parçacığı varsa, birini çalıştırmaktan (T1) diğerine (T2) geçiş yaparken, bir bağlam anahtarı (**context switch**) yer almalıdır. T2 çalıştırılmadan önce T1'in kayıt durumu kaydedilmeli ve T2'nin kayıt durumu geri yüklenmelidir (**process control block (PCB)**). Süreçlerle durumu bir süreç kontrol bloğuna (PCB) kaydettik; şimdi, bir işlemin her bir iş parçacığının durumunu depolamak için bir veya daha fazla iş parçacığı kontrol bloğuna (TCB) ihtiyacımız olacak. Bununla birlikte, işlemlerle karşılaştırıldığında iş parçacıkları arasında gerçekleştirdiğimiz bağlam geçişinde önemli bir fark vardır: adres alanı aynı kalır (yani, kullandığımız sayfa tablosunu değiştirmeye gerek yoktur).

İş parçacığı ve işlemler arasındaki bir diğer büyük fark yığınla ilgilidir. Klasik bir sürecin (artık tek iş parçacıklı bir süreç olarak adlandırabileceğimiz) adres uzayına ilişkin basit modelimizde, genellikle adres uzayının en altında yer alan tek bir yığın vardır (Şekil 26.1, sol).

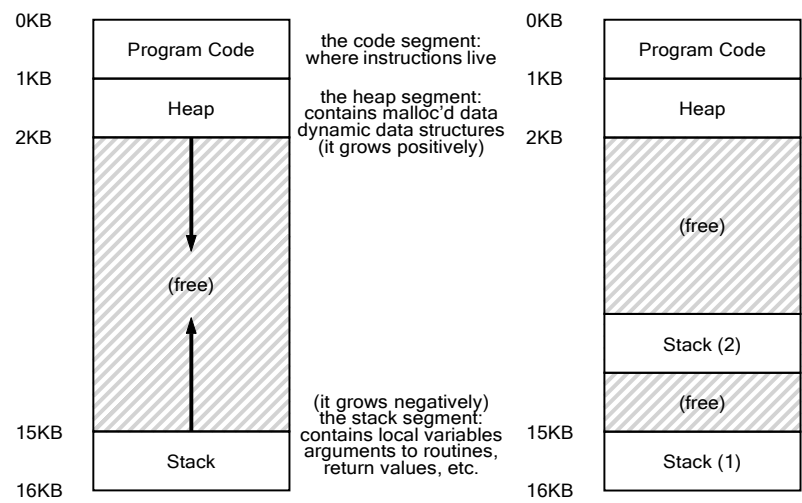


Figure 26.1: Tek İş Parçacığı ve Çok İş Parçacığı Adres Alanları (Single-Threaded And Multi-Threaded Address Spaces)

Bununla birlikte, çok iş parçacıklı bir süreçte, her iş parçacığı bağımsız olarak çalışır ve elbette, yaptığı işi yapmak için çeşitli rutinleri çağırabilir. Adres alanında tek bir yığın yerine, iş parçacığı başına bir tane olacaktır. Diyelim ki içinde iki iş parçacığı olan çok iş parçacıklı bir işlemimiz var; ortaya çıkan adres alanı farklı görünüyor (Şekil 26.1, sağ).

Bu şekilde, işlemin adres alanı boyunca yayılmış iki yığın görebilirsiniz. Bu nedenle, yığına tahsis edilen değişkenler, parametreler, dönüş değerleri ve yığına koyduğumuz diğer şeyler, bazen yerel iş parçacığı (**thread-local**) deposu olarak adlandırılan, yani ilgili iş parçacığının yığına yerleştirilecektir.

Bunun, güzel adres alanı düzenimizi nasıl bozduğunu da fark etmişsinizdir. Önceden, yığın ve öbek bağımsız olarak büyüebiliyordu ve sorun yalnızca adres alanında yer kalmadığında ortaya çıkıyordu. Burada artık böyle güzel bir durumumuz yok. Neyse ki, yığınların genellikle çok büyük olması gerekmediğinden (özyinelemeyi yoğun şekilde kullanan programlarda istisna vardır) bu genellikle uygundur.

26.1 Why Use Threads? (NedenThreads Kullanmalı?)

Çok iş parçacıklı programlar yazarken karşılaşılabileceğiniz sorunların ve iş parçacıklarının ayrıntılarına girmeden önce, daha basit bir soruyu yanıtlayalım. Neden hiç iş parçacığı kullanmalısınız?

Görünen o ki, thread kullanmanız için en az iki önemli sebep var. İlki basit: paralellik(**parallelism**). Çok büyük diziler üzerinde işlemler gerçekleştiren, örneğin iki büyük diziyi birbirine toplayan veya dizideki her bir ögenin değerini bir miktar artıran bir program yazdığınızı hayal edin. Yalnızca tek bir işlemci üzerinde çalışıyorsanız, görev basittir: her işlemi gerçekleştirin ve bitirin. Ancak, programı birden çok işlemcili bir sistemde yürütüyorsanız, işlemcilerin her biri için bir bölümünü gerçekleştirecek şekilde kullanarak bu süreci önemli ölçüde hızlandırma potansiyeline sahipsiniz. Standart tek iş parçacıklı programınızı birden fazla CPU üzerinde bu tür işleri (**single-threaded**) yapan bir programa dönüştürme görevine paralelleştirme (**paralleliza-tion**) denir ve bu işi yapmak için CPU başına bir iş parçacığı kullanmak, programları çalıştırmanın doğal ve tipik bir yoludur. modern donanımda daha hızlı.

İkinci neden biraz daha incelikli: Yavaş G/Ç nedeniyle program ilerlemesini engellemekten kaçınmak. Farklı G/Ç türleri gerçekleştiren bir program yazdığınızı hayal edin: ya bir mesaj göndermeyi veya almayı, açık bir disk G/Ç'nin tamamlanmasını, hatta (dolaylı olarak) bir sayfa hatasının bitmesini bekliyor. Programınız beklemek yerine başka bir şey yapmak isteyebilir, buna hesaplama yapmak için CPU'yu kullanmak veya hatta daha fazla I/O isteği yayınlamak da dahildir. İş parçacığı kullanmak, takılıp kalmamak için doğal bir yoldur; programınızdaki bir iş parçacığı beklerken (yani, G/Ç için beklerken bloke edilir), CPU programlayıcı çalışmaya hazır olan ve faydalı bir şeyler yapan diğer iş parçacıklarına geçebilir. İş parçacığı oluşturma, programlardaki süreçler için çoklu programlamanın yaptığı gibi, G/Ç'nin tek bir program içindeki diğer etkinliklerle çakışmasını (**overlap**) sağlar; sonuç olarak, birçok modern sunucu tabanlı uygulama (web sunucuları,

veritabanı yönetim sistemleri ve benzerleri), uygulamalarında iş parçacıklarından yararlanır.

Tabii ki, yukarıda belirtilen her iki durumda da, iş parçacığı yerine birden çok işlem kullanabilirsiniz. Bununla birlikte, iş parçacıkları bir adres alanını paylaşır ve böylece veri paylaşımını kolaylaştırır ve dolayısıyla bu tür programları oluştururken doğal bir seçimdir. İşlemler, bellekteki veri yapılarının çok az paylaşılmasına ihtiyaç duyulan mantıksal olarak ayrı görevler için daha sağlam bir seçimdir.

26.2 An Example: Thread Creation(Bir Örnek: Thread Oluşturma)

Bazı ayrıntılara girelim. Diyelim ki, her biri bağımsız işler yapan, bu durumda "A" veya "B" yazdıran iki iş parçacığı oluşturan bir program çalıştırmak istiyoruz. Kod, Şekil 26.2'de (sayfa 4) gösterilmektedir.

Ana program, her biri farklı bağımsız değişkenlerle (A veya B dizesi) olsa da, efsane okuma() işlevini çalıştıracak iki iş parçacığı oluşturur. Bir iş parçacığı oluşturulduktan sonra hemen çalışmaya başlayabilir (programlayıcının kaptırmasına bağlı olarak); alternatif olarak, "hazır" ancak "çalışmıyor" durumuna getirilebilir ve bu nedenle henüz çalışmayabilir. Tabii ki, bir çoklu işlemcide, iş parçacıkları aynı anda çalışıyor olabilir, ancak bu olasılık hakkında henüz endişelenmeyelim.

```

1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4  #include "common.h"
5  #include "common_threads.h"
6
7  void *mythread(void *arg) {
8      printf("%s\n", (char *) arg);
9      return NULL;
10 }
11
12 int
13 main(int argc, char *argv[]) {
14     pthread_t p1, p2;
15     int rc;
16     printf("main: begin\n");
17     Pthread_create(&p1, NULL, mythread, "A");
18     Pthread_create(&p2, NULL, mythread, "B");
19     // join waits for the threads to finish
20     Pthread_join(p1, NULL);
21     Pthread_join(p2, NULL);
22     printf("main: end\n");
23     return 0;
24 }
```

Figure 26.2: Simple Thread Creation Code (t0.c)
(Temel Thread Oluşturma Kodu)

İki iş parçacığını oluşturduktan sonra (bunlara T1 ve T2 diyelim), ana iş parçacığı belirli bir iş parçacığının tamamlanmasını bekleyen pthread birleştirme() işlevini çağırır. Bunu iki kez yapar, böylece ana iş

parçacığının tekrar çalışmasına izin vermeden önce T1 ve T2'nin çalışmasını ve tamamlanmasını sağlar; bunu yaptığında, "main: end" yazdırır ve çıkar. Genel olarak, bu çalışma sırasında üç iş parçacığı kullanıldı: ana iş parçacığı, T1 ve T2.

Bu küçük programın olası yürütme sırasını inceleyelim. Yürütme şemasında (Şekil 26.3, sayfa 5), zaman aşağı yönde artar ve her sütun farklı bir iş parçacığının (ana iş parçacığı veya İş Parçacığı 1 veya İş Parçacığı 2) çalıştığını gösterir.

Ancak, bu sıralamanın mümkün olan tek sıralama olmadığını unutmayın. Aslında, bir dizi talimat verildiğinde, programlayıcının belirli bir noktada hangi iş parçacığını çalıştırmaya karar verdiğine bağlı olarak epeyce talimat vardır. Örneğin, bir iş parçacığı oluşturulduktan sonra hemen çalışabilir, bu da Şekil 26.4'te (sayfa 5) gösterilen yürütmeye yol açar.

Diyeelim ki programlayıcı, İş Parçacığı 1 daha önce oluşturulmuş olmasına rağmen önce İş Parçacığı 2'yi çalıştırmaya karar verirse, "B"nin "A"dan önce yazdırıldığını bile görebiliriz; ilk oluşturulan bir iş parçacığının önce çalışacağını varsaymak için hiçbir neden yoktur. Şekil 26.5 (sayfa 6), İş Parçacığı 2'nin İş Parçacığı 1'den önce eşyalarını dikmeye başladığı bu son yürütme sırasını göstermektedir.

| main | Thread 1 | Thread2 |
|----------------------|------------|------------|
| starts running | | |
| prints "main: begin" | | |
| creates Thread 1 | | |
| creates Thread 2 | | |
| waits for T1 | runs | |
| | prints "A" | |
| | returns | |
| waits for T2 | | runs |
| | | prints "B" |
| | | returns |
| prints "main: end" | | |

Figure 26.3: Thread Trace (1)

İşparçacığı İzi(1)

| main | Thread 1 | Thread2 |
|--|------------|------------|
| starts running | | |
| prints "main: begin" | | |
| creates Thread 1 | runs | |
| | prints "A" | |
| | returns | |
| creates Thread 2 | | runs |
| | | prints "B" |
| | | returns |
| waits for T1 | | |
| <i>returns immediately; T1 is done</i> | | |
| waits for T2 | | |
| <i>returns immediately; T2 is done</i> | | |
| prints "main: end" | | |

Figure 26.4: Thread Trace (2)(işparçacığı izi(2))

Görebileceğiniz gibi, iş parçacığı oluşturma hakkında düşünmenin bir yolu biraz işlev çağırısı yapmaya benziyor; ancak, önce işlevi çalıştırıp sonra arayana geri dönmek yerine, sistem bunun yerine çağrılan rutin için yeni bir yürütme iş parçacığı oluşturur ve belki de oluşturmadan dönmenden önce çağırandan bağımsız olarak çalışır, ama belki çok daha sonra. Bundan sonra neyin çalışacağı, işletim sistemi programlayıcısı(**scheduler**) tarafından belirlenir ve programlayıcı muhtemelen bazı mantıklı algoritmalar uygulasa da, herhangi bir zamanda neyin çalışacağını bilmek zordur.

Bu örnekten de anlayabileceğiniz gibi, iş parçacıkları hayatı karmaşık hale getiriyor: Neyin ne zaman çalışacağını söylemek zaten zor! Bilgisayarların eşzamanlılık olmadan anlaşılması yeterince zordur. Ne yazık ki, eşzamanlılık ile durum daha da kötüleşiyor. Çok daha kötü.

| main | Thread 1 | Thread2 |
|--|------------|------------|
| starts running | | |
| prints "main: begin" | | |
| creates Thread 1 | | |
| creates Thread 2 | | runs |
| | | prints "B" |
| | | returns |
| waits for T1 | runs | |
| | prints "A" | |
| | returns | |
| waits for T2 | | |
| <i>returns immediately; T2 is done</i> | | |
| prints "main: end" | | |

Figure 26.5: Thread Trace (3)
(işparçacığı izi(3))

26.3 Why It Gets Worse: Shared Data

(Neden Kötüleşiyor: Paylaşılan Veriler)

Yukarıda gösterdiğimiz basit iş parçacığı örneği, iş parçacıklarının nasıl oluşturulduğunu ve programlayıcının onları nasıl çalıştırmaya karar verdiğine bağlı olarak nasıl farklı sıralarda çalışabileceklerini göstermek açısından faydalıydı. Ancak size göstermediği şey, ileti dizilerinin paylaşılan verilere eriştiklerinde nasıl etkileşime girdiğidir.

İki iş parçacığının genel bir paylaşılan değişkeni güncellemek istediği basit bir örnek düşünelim. Çalışacağımız kod Şekil 26.6'dadır (sayfa 7).

İşte kod hakkında birkaç not. İlk olarak, Stevens'in [SR05] önerdiği gibi, iş parçacığı oluşturma işlemini tamamlarız ve başarısızlık durumunda basitçe çıkmak için rutinleri birleştiririz; Bunun kadar basit bir program için, en azından bir hata oluştuğunu fark etmek isteriz (eğer olduysa), ancak bu konuda çok akıllıca bir şey yapmayız (örneğin, sadece çıkın). Böylece, Pthread create() sadece pthread create()'i çağırır ve dönüş kodunun 0 olduğundan emin olur; değilse, Pthread create() sadece bir mesaj yazdırır ve çıkar.

İkincisi, çalışan iş parçacıkları için iki ayrı işlev gövdesi kullanmak yerine, yalnızca tek bir kod parçası kullanırız ve iş parçacığına bir bağımsız değişken (bu durumda bir dize) iletiriz, böylece her iş parçacığının daha önce farklı bir harf yazdırmasını sağlayabiliriz. mesajları.

Son olarak ve en önemlisi, artık her çalışanın ne yapmaya çalıştığına bakabiliriz: paylaşılan değişken sayacına bir sayı ekleyin ve bunu bir döngüde 10 milyon kez (1e7) yapın. Böylece istenen nihai sonuç: 20.000.000.

Şimdi nasıl davrandığını görmek için programı derleyip çalıştırıyoruz. Bazen her şey beklediğimiz gibi çalışır

```
:prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
```

```
main: done with both (counter = 20000000)
```

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include "common.h"
4 #include "common_threads.h"
5
6 static volatile int counter = 0;
7
8 // mythread()
9 //
```

```

10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *mythread(void *arg) {
15     printf("%s: begin\n", (char *) arg);
16     int i;
17     for (i = 0; i < 1e7; i++) {
18         counter = counter + 1;
19     }
20     printf("%s: done\n", (char *) arg);
21     return NULL;
22 }
23
24 // main()
25 //
26 // Just launches two threads (pthread_create)
27 // and then waits for them (pthread_join)
28 //
29 int main(int argc, char *argv[]) {
30     pthread_t p1, p2;
31     printf("main: begin (counter = %d)\n", counter);
32     Pthread_create(&p1, NULL, mythread, "A");
33     Pthread_create(&p2, NULL, mythread, "B");
34
35     // join waits for the threads to finish
36     Pthread_join(p1, NULL);
37     Pthread_join(p2, NULL);
38     printf("main: done with both (counter = %d)\n",
39           counter);
40     return 0;
41 }

```

Figure 26.6: Sharing Data: Uh Oh (t1.c) (Paylaşılan Veri)

Ne yazık ki, bu kodu tek bir işlemcide bile çalıştırdığımızda, her zaman istediğimiz sonucu alamıyoruz. Bazen şunları alırız:

```

prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)

```

Hadi bir kez daha deneyelim, delirmiş miyiz diye görmek için. Ne de olsa, size öğretildiği gibi, bilgisayarların deterministik sonuçlar üretmesi gerekmiyor mu? Belki de profesörlerin sana yalan söylüyordur?


```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Her çalıştırma yanlış olmakla kalmaz, aynı zamanda farklı bir sonuç verir! Geriye büyük bir soru kalıyor: bu neden oluyor?

İPUCU: ARAÇLARINIZI BİLİN VE KULLANIN

Bilgisayar sistemlerini yazmanıza, hata ayıklamanıza ve anlamınıza yardımcı olan yeni araçları her zaman öğrenmelisiniz. Burada, demonte (**disassem-bler**) adı verilen temiz bir araç kullanıyoruz. Yürütülebilir bir dosya üzerinde bir sökme aracı çalıştırdığınızda, programı hangi montaj yönergelerinin oluşturduğunu gösterir. Örneğin, bir sayacı güncellemek için düşük seviyeli kodu anlamak istersek (örneğimizde olduğu gibi), montaj kodunu görmek için `objdump` (Linux) çalıştırırız:

```
prompt> objdump -d main
```

Bunu yapmak, programdaki tüm talimatların düzgün bir şekilde etiketlenmiş (özellikle `-g` bayrağıyla derlediyseniz) programdaki sembol bilgilerini içeren uzun bir listesini oluşturur. `objdump` programı, nasıl kullanılacağını öğrenmeniz gereken birçok araçtan yalnızca biridir; `gdb` gibi bir hata ayıklayıcı, `valgrind` veya `purify` gibi bellek profili oluşturucular ve elbette derleyicinin kendisi, hakkında daha fazla bilgi edinmek için zaman ayırmanız gereken diğerleridir; araçlarınızı ne kadar iyi kullanırsanız, o kadar iyi sistemler kurabilirsiniz.

26.4 The Heart Of The Problem: Uncontrolled Scheduling

(Sorunun Kalbi: Kontrolsüz Programlama)

Bunun neden olduğunu anlamak için, derleyicinin sayaç güncellemesi için ürettiği kod dizisini anlamamız gerekir. Bu durumda, karşı koymak için basitçe bir sayı (1) eklemek istiyoruz. Bu nedenle, bunu yapmak için kod dizisi şöyle görünebilir (x86'da);

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Bu örnek, sayaç değişkeninin 0x8049a1c adresinde bulunduğunu varsayar. Bu üç komut dizisinde, x86 mov komutu, adresteki bellek değerini almak ve eax yazmacına koymak için önce kullanılır. Ardından, eax kaydının içeriğine 1 (0x1) eklenerek ekleme işlemi gerçekleştirilir ve son olarak, eax içeriği aynı adreste belleğe geri depolanır.

İki iş parçacığımızdan birinin (İplik 1) bu kod bölgesine girdiğini ve bu nedenle sayacı birer birer artırmak üzere olduğunu düşünelim. Sayacın değerini (başlangıçta 50 diyelim) register eax'ına yükler. Böylece Thread 1 için eax=50 olur. Sonra register'a bir ekler; böylece eax=51. Şimdi talihsiz bir şey oluyor: bir zamanlayıcı kesintisi geliyor; bu nedenle işletim sistemi, o anda çalışan iş parçacığının durumunu (PC'si, eax dahil kayıtları vb.) iş parçacığının TCB'sine kaydeder.

Şimdi daha kötü bir şey olur: Thread 2 çalıştırmak için seçilir ve aynı kod parçasını girer. Ayrıca ilk talimatı yürütür, sayacın değerini alır ve eax'ına koyar (unutmayın: çalışırken her iş parçacığının kendi özel kayıtları vardır; kayıtlar, onları kaydeden ve geri yükleyen bağlam değiştirme kodu tarafından sanallaştırılır). Sayacın değeri bu noktada hala 50'dir ve dolayısıyla Thread 2 eax=50'dir. O halde, İş Parçacığı 2'nin eax'i 1 artırarak (böylece eax=51) sonraki iki talimatı yürüttüğünü ve ardından eax'in içeriğini sayaca (adres 0x8049a1c) kaydettiğini varsayalım. Böylece, global değişken sayacı artık 51 değerine sahiptir.

Son olarak, başka bir içerik anahtarı gerçekleşir ve Thread 1 çalışmaya devam eder. Hareketi ve eklemeyi henüz yürüttüğünü ve şimdi son hareket talimatını gerçekleştirmek üzere olduğunu hatırlayın. eax=51 olduğunu da hatırlayın. Böylece, son mov komutu yürütülür ve değeri belleğe kaydeder; sayaç tekrar 51'e ayarlanır.

Basitçe söylemek gerekirse, olan şu: sayacı artırma kodu iki kez çalıştırıldı, ancak 50'de başlayan sayaç şimdi yalnızca 51'e eşit. Bu programın "doğru" bir versiyonu, sayaç değişkeninin eşit olmasıyla sonuçlanmalıydı. 52'ye.

Sorunu daha iyi anlamak için ayrıntılı bir yürütme izine bakalım. Bu örnek için, yukarıdaki kodun, aşağıdaki dizi gibi bellekteki adres 100'e yüklendiğini varsayalım (nice, RISC benzeri komut setlerin alışkın olanlarınız için not: x86, değişken uzunluklu komutlara sahiptir; bu mov komutu, 5 bayt bellek ve yalnızca 3 ekleyin):

| OS | Thread 1 | Thread 2 | (after instruction) | | |
|-------------------|--------------------------------|------------------|---------------------|-----|---------|
| | | | PC | eax | counter |
| | <i>before critical section</i> | | 100 | 0 | 50 |
| | mov 8049a1c,%eax | | 105 | 50 | 50 |
| | add \$0x1,%eax | | 108 | 51 | 50 |
| interrupt | | | | | |
| <i>save T1</i> | | | | | |
| <i>restore T2</i> | | | 100 | 0 | 50 |
| | | mov 8049a1c,%eax | 105 | 50 | 50 |
| | | add \$0x1,%eax | 108 | 51 | 50 |
| | | mov %eax,8049a1c | 113 | 51 | 51 |
| interrupt | | | | | |
| <i>save T2</i> | | | | | |
| <i>restore T1</i> | | | 108 | 51 | 51 |
| | mov %eax,8049a1c | | 113 | 51 | 51 |

Figure 26.7: **The Problem: Up Close and Personal**(Up Close ve Personal Problemi)

```

100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c

```

Bu varsayımlarla ne olduğu Şekil 26.7'de (sayfa 10) gösterilmektedir. Sayacın 50 değerinde başladığını varsayalım ve neler olduğunu anladığınızdan emin olmak için bu örneği izleyin.

Burada gösterdiğimiz şeye bir yarış koşulu (**race condition**) (veya daha spesifik olarak bir veri yarışı) denir: sonuçlar, kodun zamanlama yürütmesine bağlıdır. Biraz şanssızlıkla (yani yürütmeye zamansız noktalarda meydana gelen bağlam değişiklikleri), yanlış sonuca ulaşırız. Aslında her seferinde farklı bir sonuç alabiliriz; bu nedenle, (bilgisayarlardan alışık olduğumuz) güzel bir deterministik(**deterministic**) hesaplama yerine, çıktının ne olacağının bilinmediği ve gerçekten de çalıştırmalar arasında farklı olma ihtimalinin yüksek olduğu bu sonuca belirsiz diyoruz.

Bu kodu çalıştıran birden çok iş parçacığı bir yarış durumuna neden olabileceğinden, bu kodu kritik bölüm (**critical section**) olarak adlandırırız. Kritik bölüm, paylaşılan bir değişkene (veya daha genel olarak paylaşılan bir kaynağa) erişen ve birden fazla iş parçacığı tarafından aynı anda yürütülmemesi gereken bir kod parçasıdır.

Bu kod için gerçekten istediğimiz şey, karşılıklı dışlama (**mutual exclusion**) dediğimiz şeydir. Bu özellik, kritik bölüm içinde bir iş parçacığının yürütülmesi durumunda diğerlerinin bunu yapmasının engellenmesini garanti eder.

Bu arada, bu terimlerin neredeyse tamamı, bu alanda öncü olan ve gerçekten de bu ve diğer çalışmaları nedeniyle Turing Ödülü'nü kazanan Edsger Dijkstra tarafından icat edildi; sorunun şaşırtıcı derecede net bir açıklaması için "İşbirliği Yapan Sıralı Süreçler" [D68] hakkındaki 1968 makalesine bakın. Kitabın bu bölümünde Dijkstra hakkında daha fazla şey

duyacağız.

İPUCU: ATOMİK İŞLEMLERİ KULLANIN

Atomik işlemler, bilgisayar mimarisinden eşzamanlı koda (burada incelediğimiz şey), dosya sistemlerine (yakında inceleyeceğiz), veri tabanı yönetim sistemlerine, bilgisayar sistemleri oluşturma'nın altında yatan en güçlü tekniklerden biridir. ve hatta dağıtılmış sistemler [L+93].

Bir dizi eylemi atomik(**atomic**) hale getirmenin ardındaki fikir, basitçe "ya hep ya hiç" ifadesiyle ifade edilir; ya birlikte gruplandırmak istediğiniz tüm eylemler gerçekleşmiş gibi görünmelidir ya da hiçbir ara durum görünmeden hiçbir gerçekleşmemiş gibi görünmelidir. Bazen, birçok eylemin tek bir atomik eylemde gruplandırılmasına işlem (**transaction**) denir, veritabanları ve işlem işleme dünyasında çok ayrıntılı olarak geliştirilen bir fikir [GR92].

Eşzamanlılığı keşfetme temamızda, kısa talimat dizilerini atomik yürütme bloklarına dönüştürmek için senkronizasyon ilkelerini kullanacağız, ancak atomiklik fikri, göreceğimiz gibi bundan çok daha büyük. Örneğin, dosya sistemleri, sistem arızaları karşısında doğru şekilde çalışmak için kritik olan disk üzerindeki durumlarını atomik olarak değiştirmek için günlük kaydı veya yazma üzerine kopyalama gibi teknikleri kullanır. Bu bir anlam ifade etmiyorsa endişelenmeyin - gelecek bir bölümde mantıklı olacaktır.

26.5 The Wish For Atomicity (Atomiklik Arzusu)

Bu sorunu çözenin bir yolu, tek bir adımda tam olarak yapmamız gerekeni yapan ve böylece zamansız bir kesinti olasılığını ortadan kaldıran daha güçlü talimatlara sahip olmaktır. Örneğin, şuna benzeyen bir süper talimatımız olsaydı:

```
memory-add 0x8049a1c, $0x1
```

Bu talimatın bir bellek konumuna bir değer eklediğini ve donanımın atomik olarak (atomically) yürütüldüğünü garanti ettiğini varsayalım; komut yürütüldüğünde, güncellemeyi istediği gibi gerçekleştirdi. Talimatın ortasında kesilemez, çünkü donanımdan aldığımız garanti tam olarak budur: bir kesme meydana geldiğinde, talimat ya hiç çalışmaz ya da tamamlanana kadar çalışır; ara durum yoktur. Donanım güzel bir şey olabilir, değil mi?

Atomik olarak, bu bağlamda, bazen "ya hep ya hiç" olarak aldığımız "bir birim olarak" anlamına gelir. İstedığımız şey, üç talimat dizisini atomik olarak yürütmek:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Söylediğimiz gibi, bunu yapmak için tek bir talimatımız olsaydı, o talimatı verir ve bitirirdik. Ancak genel durumda böyle bir talimatımız olmayacak. Eşzamanlı bir B-ağacı oluşturduğumuzu ve onu güncellemek istediğimizi hayal edin; donanımın "B-ağacının atomik güncellemesi" talimatını desteklemesini gerçekten ister miydik? Muhtemelen hayır, en azından akli başında bir talimat setinde.

Bu nedenle, bunun yerine yapacağımız şey, üzerine senkronizasyon ilkeleri (**syn-chronization primitives**) dediğimiz genel bir set oluşturabileceğimiz birkaç yararlı talimat için donanım istemek olacaktır. Bu donanım desteğini işletim sisteminden biraz yardım alarak kullanarak, kritik bölümlere senkronize ve kontrollü bir şekilde erişen ve böylece zorlu yapıya rağmen güvenilir bir şekilde doğru sonucu üreten çok iş parçacıklı kod oluşturabileceğiz. eşzamanlı yürütme. Oldukça harika, değil mi?

Kitabın bu bölümünde inceleyeceğimiz sorun budur. Bu harika ve zor bir problem ve zihninizi (biraz) incitmeli. Eğer değilse, o zaman anlamıyorsunuz! Başınız ağrıyana kadar çalışmaya devam edin; o zaman doğru yöne gittiğinizi bilirsiniz. Bu noktada bir ara verin; Başınızın çok fazla ağrımasını istemiyoruz.

ÖNEMLİ NOKTA: SENKRONİZASYON NASIL DESTEKLENİR

Yararlı senkronizasyon ilkelerini oluşturmak için donanımdan hangi desteğe ihtiyacımız var? İşletim sisteminden hangi desteğe ihtiyacımız var? Bu ilkeleri nasıl doğru ve verimli bir şekilde inşa edebiliriz? Programlar, istenen sonuçları elde etmek için bunları nasıl kullanabilir?

26.6 One More Problem: Waiting For Another (Bir Problem Daha: Bir Diğerini Beklemek)

Bu bölüm, eş zamanlılık problemini, sanki iş parçacıkları arasında yalnızca bir tür etkileşim oluyormuş gibi, paylaşılan değişkenlere erişim ve kritik bölümler için atomikliği destekleme ihtiyacı olarak kurmuştur. Görünüşe göre, ortaya çıkan başka bir ortak etkileşim var, burada bir iş parçacığı devam etmeden önce bazı eylemleri tamamlamak için diğerini beklemek zorunda. Bu etkileşim, örneğin, bir işlem bir disk G/Ç gerçekleştirdiğinde ve uyku moduna alındığında ortaya çıkar; G/Ç tamamlandığında, devam edebilmesi için sürecin uykusundan uyandırılması gerekir.

Bu nedenle, önümüzdeki bölümlerde, yalnızca atomikliği desteklemek için senkronizasyon ilkelleri için nasıl destek oluşturulacağını değil, aynı zamanda çok iş parçacıklı programlarda yaygın olan bu tür uyku/uyanma etkileşimini destekleyen mekanizmaları da inceleyeceğiz. Bu şu anda mantıklı gelmiyorsa, sorun değil! Koşul değişkenleri (**condition variables**) ile ilgili bölümü okuduğunuzda çok yakında olacak. O zamana kadar olmazsa, o zaman daha az uygun demektir ve mantıklı olana kadar o bölümü tekrar (ve tekrar) okumalısınız.

KENARA: ANAHTAR EŞ ZAMANLILIK ŞARTLARI KRİTİK BÖLÜM, YARIŞ DURUMU, BELİRSİZ, KARŞILIKLI İSTİSNA

Bu dört terim, eşzamanlı kod için o kadar merkezidir ki, onları açıkça belirtmeye değeceğini düşündük. Daha fazla ayrıntı için Dijkstra'nın erken çalışmalarından bazılarına [D65, D68] bakın.

- Kritik bölüm (**critical section**), paylaşılan bir kaynağa, genellikle bir değişkene veya veri yapısına erişen bir kod parçasıdır.
- Bir yarış durumu (**race condition**) (veya veri yarışı (**data race**)) [NM92]), birden fazla yürütme iş parçacığı kritik bölüme kabaca aynı anda girerse ortaya çıkar; her ikisi de paylaşılan veri yapısını güncellemeye çalışarak şaşırtıcı (ve belki de istenmeyen) bir sonuca yol açar.
- Belirsiz (**indeterminate**) bir program, bir veya daha fazla yarış koşulundan oluşur; programın çıktısı, hangi iş parçacığının ne zaman çalıştığına bağlı olarak çalıştırmadan çalıştırmaya değişir. Dolayısıyla sonuç, genellikle bilgisayar sistemlerinden beklediğimiz bir şey olan deterministik(**deterministic**) değildir.
- Bu sorunlardan kaçınmak için, iş parçacıkları bir tür karşılıklı dışlama (**mutual exclusion**) ilkelleri kullanmalıdır; bunu yapmak, kritik bir bölüme yalnızca tek bir iş parçacığının girmesini garanti eder, böylece yarışlardan kaçınılır ve deterministik program çıktıları elde edilir.

26.7 Summary: Why in OS Class? (Özet: Neden OS Sınıfında?)

Bitirmeden önce, sahip olabileceğiniz bir soru şudur: Bunu neden OS sınıfında çalışıyoruz? "Tarih" tek kelimelik cevaptır; işletim sistemi ilk

eşzamanlı programdı ve işletim sistemi içinde kullanılmak üzere birçok teknik oluşturuldu. Daha sonra, çok iş parçacıklı süreçlerle, uygulama programcıları da bu tür şeyleri dikkate almak zorunda kaldı.

Örneğin, çalışan iki işlemin olduğu durumu hayal edin. Her ikisinin de dosyaya yazmak için write()'ı çağırdığını ve her ikisinin de verileri dosyaya eklemek istediğini varsayalım (yani, verileri dosyanın sonuna ekleyerek uzunluğunu artırın). Bunu yapmak için, her ikisinin de yeni bir blok ayırması, bu bloğun yaşadığı dosyanın inode'una kaydetmesi ve dosyanın boyutunu yeni daha büyük boyutu yansıtacak şekilde değiştirmesi gerekir (diğer şeylerin yanı sıra; dosyalar hakkında daha fazlasını öğreneceğiz. kitabın üçüncü bölümü). Herhangi bir zamanda bir kesinti meydana gelebileceğinden, bu paylaşılan yapıları güncelleyen kod (örneğin, ayırma için bir bitmap veya dosyanın inode'u) kritik bölümlerdir; bu nedenle, işletim sistemi tasarımcıları, kesintinin tanıtılmasının en başından itibaren, işletim sisteminin dahili yapıları nasıl güncellediği konusunda endişelenmek zorunda kaldılar. Zamansız bir kesinti, yukarıda açıklanan tüm sorunlara neden olur. Şaşırtıcı olmayan bir şekilde, düzgün çalışması için sayfa tablolarına, işlem listelerine, dosya sistemi yapılarına ve hemen hemen her çekirdek veri yapısına, uygun senkronizasyon ilkeleriyle dikkatlice erişilmesi gerekir.

Referanslar (References)

[D65] “Solution of a problem in concurrent programming control” by E. W. Dijkstra. Communications of the ACM, 8(9):569, September 1965. *Pointed to as the first paper of Dijkstra’s where he outlines the mutual exclusion problem and a solution. The solution, however, is not widely used; advanced hardware and OS support is needed, as we will see in the coming chapters.*

[D68] “Cooperating sequential processes” by Edsger W. Dijkstra. 1968. Available at this site: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>. *Dijkstra has an amazing number of his old papers, notes, and thoughts recorded (for posterity) on this website at the last place he worked, the University of Texas. Much of his foundational work, however, was done years earlier while he was at the Technische Hogeschool Eindhoven (THE), including this famous paper on “cooperating sequential processes”, which basically outlines all of the thinking that has to go into writing multi-threaded programs. Dijkstra discovered much of this while working on an operating system named after his school: the “THE” operating system (said “T”, “H”, “E”, and not like the word “the”).*

[GR92] “Transaction Processing: Concepts and Techniques” by Jim Gray and Andreas Reuter. Morgan Kaufmann, September 1992. *This book is the bible of transaction processing, written by one of the legends of the field, Jim Gray. It is, for this reason, also considered Jim Gray’s “brain dump”, in which he wrote down everything he knows about how database management systems work. Sadly, Gray passed away tragically a few years back, and many of us lost a friend and great mentor, including the co-authors of said book, who were lucky enough to interact with Gray during their graduate school years.*

[L+93] “Atomic Transactions” by Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete. Morgan Kaufmann, August 1993. *A nice text on some of the theory and practice of atomic transactions for distributed systems. Perhaps a bit formal for some, but lots of good material is found herein.*

[NM92] “What Are Race Conditions? Some Issues and Formalizations” by Robert H. B. Netzer and Barton P. Miller. ACM Letters on Programming Languages and Systems, Volume 1:1, March 1992. *An excellent discussion of the different types of races found in concurrent programs. In this chapter (and the next few), we focus on data races, but later we will broaden to discuss **general races** as well.*

[SR05] “Advanced Programming in the UNIX Environment” by W. Richard Stevens and Stephen A. Rago. Addison-Wesley, 2005. *As we’ve said many times, buy this book, and read it, in little chunks, preferably before going to bed. This way, you will actually fall asleep more quickly; more importantly, you learn a little more about how to become a serious UNIX programmer.*

Ödev (Simülasyon)

Bu program, x86.py, farklı iş parçacığı geçişlerinin nasıl yarış koşullarına neden olduğunu veya bunlardan nasıl kaçındığını görmenizi sağlar. Programın nasıl çalıştığıyla ilgili ayrıntılar için README (BENİOKU) 'ya bakın, ardından aşağıdaki soruları yanıtlayın.

Sorular

1.Let's examine a simple program, "loop.s". First, just read and understand it. Then, run it with these arguments (./x86.py -p loop.s -t 1 -i 100 -R dx) This specifies a single thread, an interrupt every 100 instructions, and tracing of register %dx. What will %dx be during the run? Use the -c flag to check your answers; the an- swers, on the left, show the value of the register (or memory value) after the instruction on the right has run.

- Basit bir program olan "loop.s"yi inceleyelim. İlk olarak, sadece okuyun ve anlayın. Ardından, şu bağımsız değişkenlerle çalıştırın (./x86.py -p loop.s -t 1 -i 100 -R dx) Bu, tek bir iş parçacığını, her 100 yönergede bir kesmeyi ve %dx yazmacının izlenmesini belirtir. Çalıştırma sırasında %dx ne olacak? Cevaplarınızı kontrol etmek için -c bayrağını kullanın; soldaki cevaplar, sağdaki komut çalıştırıldıktan sonra yazmacın değerini (veya hafıza değerini) gösterir.

Çözüm:

```
.main
    .top
    sub $1,%dx
    test $0,%dx
    jgte .top
    halt
```

x86 terimi, bir işlemcinin mimarisini ifade eder. x86 işlemcileri, en yaygın olarak kullanılan işlemci mimarisidir ve bilgisayarınızın işlemcisi muhtemelen bu türden bir işlemcidir. x86 bir dosya uzantısı değildir ve bu nedenle bir Python dosyası olarak kullanılamaz.

- Same code, different flags: (./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx) This specifies two threads, and initializes each %dx to 3. What values will %dx see? Run with -c to check. Does the presence of multiple threads affect your calculations? Is there a race in this code?

- Aynı kod, farklı bayraklar: (./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx) Bu, iki iş parçacığını belirtir ve her birini başlatır %dx ile 3. %dx hangi değerleri görecektir? Kontrol etmek için -c ile çalıştırın. Birden çok iş parçacığının varlığı hesaplamalarınızı etkiler mi? Bu kodda bir yarış var mı?

Çözüm:

```

# assumes %bx has loop count in it

.main

.top

# critical section

mov 2000, %ax # get 'value' at address 2000

add $1, %ax # increment it

mov %ax, 2000 # store it back


# see if we're still looping

sub $1, %bx

test $0, %bx

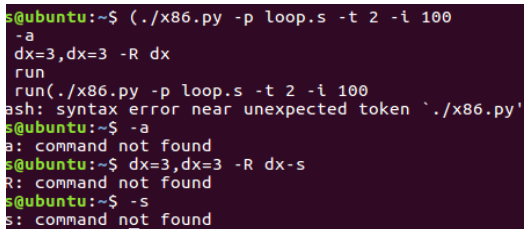
jgt .top

halt

```

3. Run this: `./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx` This makes the interrupt interval small/random; use different seeds (-s) to see different interleavings. Does the interrupt frequency change anything?
3. Bunu çalıştırın: `./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx` Bu, kesinti aralığını küçük/rastgele yapar; farklı serpiştirmeleri görmek için farklı tohumlar (-s) kullanın. Kesinti frekansı herhangi bir şeyi değiştirir mi?

Çözüm:



```

s@ubuntu:~$ ./x86.py -p loop.s -t 2 -i 100
-a
dx=3,dx=3 -R dx
run
run(./x86.py -p loop.s -t 2 -i 100
ash: syntax error near unexpected token `./x86.py'
s@ubuntu:~$ -a
a: command not found
s@ubuntu:~$ dx=3,dx=3 -R dx-s
R: command not found
s@ubuntu:~$ -s
s: command not found
s@ubuntu:~$

```

```

.main

# this is a critical section

mov 2000(%bx), %ax # get the value at the address

add $1, %ax # increment it

mov %ax, 2000(%bx) # store it back


halt

```

4. A different program, `looping-race-nolock.s`, which accesses a shared variable located at address 2000; we'll call this variable value. Run it with a single thread to confirm your understanding: `./x86.py -p looping-race-nolock.s -t 1 -M 2000` What is value (i.e., at memory address 2000) throughout the run? Use `-c` to check.

4.Şimdi, 2000 adresinde bulunan paylaşılan bir değişkene erişen farklı bir program, `looping-race-nolock.s`; bu değişkene değer diyeceğiz. Anladığınızı doğrulamak için tek bir iş parçacığı ile çalıştırın: `./x86.py -p looping-race-nolock.s -t 1 -M 2000` Çalışma boyunca değer (yani, bellek adresi 2000'de) nedir? Kontrol etmek için `-c`'yi kullanın.

Çözüm:

```
@ubuntu:~$ x86.py -p
x86.py: command not found
@ubuntu:~$ looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1
looping-race-nolock.s: command not found
@ubuntu:~$
@ubuntu:~$ runx86.py -p
runx86.py: command not found
@ubuntu:~$ looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1
```

5. Run with multiple iterations/threads: `./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000` Why does each thread loop three times? What is final value of value?
5. Birden çok yineleme/iş parçacığı ile çalıştırın: `./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000` Neden her iş parçacığı üç kez dönüyor? Değerin nihai değeri nedir?

Çözüm:

```
@ubuntu:~$
@ubuntu:~$ if printstats:
@ubuntu:~$     print('')
ash: syntax error near unexpected token ''
@ubuntu:~$     print('STATS:: Instructions    %d' % lc)
ash: syntax error near unexpected token ''STATS:: Instructions    %d'
@ubuntu:~$     print('STATS:: Emulation Rate  %.2f kinst/sec' % (Float(lc) / float(t2 - t1) / 1000.0))
ash: syntax error near unexpected token ''STATS:: Emulation Rate  %.2f kinst/sec'
@ubuntu:~$
@ubuntu:~$ # use this for profiling
@ubuntu:~$ # import cProfile
@ubuntu:~$ # cProfile.run('run()')
@ubuntu:~$
```

6. Run with random interrupt intervals: `./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0` with different seeds (`-s 1`, `-s 2`, etc.) Can you tell by looking at the thread interleaving what the final value of value will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?
- 6.Rastgele kesme aralıklarıyla çalıştırın: `./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0` ile farklı tohumlar (`-s 1`, `-s 2`, vb.) değeri olacak? Kesintinin zamanlaması önemli mi? Güvenli bir şekilde nerede meydana gelebilir? Nerede değil? Başka bir

deyişle, kritik bölüm tam olarak nerede?

Çözüm:

7. Now examine fixed interrupt intervals: `./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1` What will the final value of the shared variable value be? What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the “correct” answer?

7. Şimdi sabit kesinti aralıklarını inceleyin: `./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1` Paylaşılan değişken değerinin son değeri ne olacak? Peki ya `-i 2`, `-i 3` vb. değiştirdiğinizde? Program hangi kesme aralıkları için “doğru” yanıt veriyor?

Çözüm: `# assumes %bx has loop count in it`

```
.main
.top
# critical section
mov 2000, %ax # get 'value' at address 2000
add $1, %ax   # increment it
mov %ax, 2000 # store it back
```

```
# see if we're still looping
sub $1, %bx
test $0, %bx
jgt .top
```

```
halt
```

8. Run the same for more loops (e.g., set `-a bx=100`). What interrupt intervals (`-i`) lead to a correct outcome? Which intervals are surprising?

8. Aynısını daha fazla döngü için çalıştırın (örneğin, `-a bx=100` olarak ayarlayın). Hangi kesme aralıkları (`-i`) doğru sonuca götürür? Hangi aralıklar şaşırtıcı?

Çözüm:

9. One last program: `wait-for-me.s`. Run: `./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000` This sets the `%ax` register to 1 for thread 0, and 0 for thread 1, and watches `%ax` and memory location 2000. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

9. Son bir program: `wait-for-me.s`. Çalıştır: `./x86.py -p`

`wait-for-me.s -a ax=1,ax=0 -R ax -M 2000` Bu, %ax, iş parçacığı 0 için 1'e ve iş parçacığı 1 için 0'a kayıt olur ve %ax ile bellek konumu 2000'i izler. Kod nasıl davranmalı? 2000 konumundaki değer iş parçacıkları tarafından nasıl kullanılıyor? Nihai değeri ne olacak?

Çözüm:

```
.main
test $1, %ax    # ax should be 1 (signaller) or 0 (waiter)
je .signaller
.waiter
mov 2000, %cx
test $1, %cx
jne .waiter
halt

.signaller
mov $1, 2000
halt
```

10. Now switch the inputs: `./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000` How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., `-i 1000`, or perhaps to use random intervals) change the trace out- come? Is the program efficiently using the CPU?

10. Şimdi girişleri değiştirin: `./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000` İş parçacıkları nasıl davranıyor? Konu 0 ne yapıyor? Kesme aralığı nasıl değiştirilir (örn.-i 1000, ya da belki rastgele aralıklar kullanmak için) iz sonucunu değiştirmek? Program CPU'yu verimli bir şekilde kullanıyor mu?

Çözüm: `.main`

```
test $1, %ax    # ax should be 1 (signaller) or 0 (waiter)
je .signaller

.waiter
mov 2000, %cx
test $1, %cx
jne .waiter
halt

.signaller
mov $1, 2000
halt
```