26

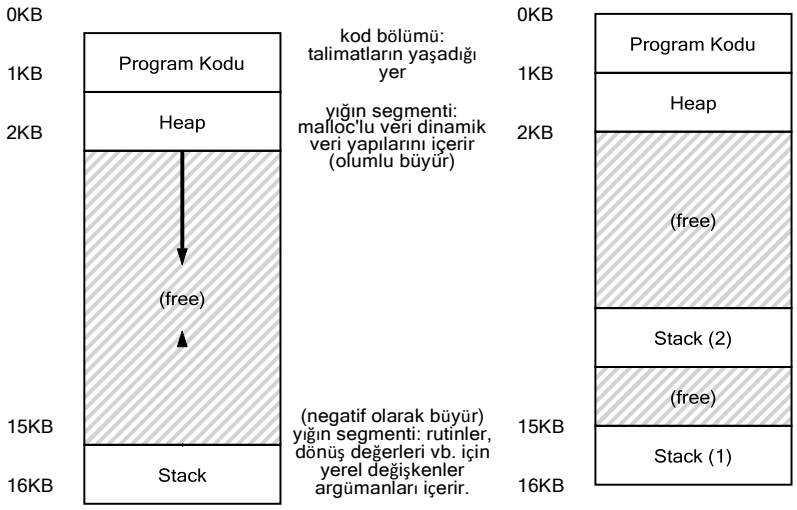
Eşzamanlılık (Concurrency): Giriş

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.Ayrıca her işlem için büyük, özel bir sanal bellek (virtual memory) yanılsamasının nasıl yaratılacağını 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 (thread) soyutlaması. Bir program içindeki tek bir yürütme noktasına (yani, talimatların alındığı ve yürütüldüğü tek bir PC) ilişkin klasik görüşümüzün yerine, çoklu iş parçacıklı (a 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 aldığını 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. Süreçlerle durumu bir süreç control bloğuna (a process control block) (PCB) kaydettik; şimdi, bir işlemin her bir iş parçacığının durumunu depolamak için bir veya daha fazla iş parçacığı kontrol bloğuna (a thread control block) (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.



Şekil 26.1: Tek İş Parçacığı ve Çok İş Parçacığı Adres Alanları

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). 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 yığın olacaktır. Diyelim ki içinde iki iş parçacığı olan çok iş parçacıklı bir işlemimiz var; ortaya çıkan adres alanı farklı görünür (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 yığınına yerleştirilecektir.

Bunun, güzel adres alanı düzenimizi nasıl bozduğunu da fark etmişsinizdir. Önceden, yığın ve öbek bağımsız olarak büyüyebiliyordu 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 Neden İş Parçacıkları(Threads) kullanmalı?

Çok iş parçacıklı programlar yazarken karşılaşabileceğiniz 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 öğenin 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 işin bir bölümünü gerçekleştirecek şekilde kullanarak bu süreci önemli ölçüde hızlandırma potansiyeline sahipsiniz. Standart **tek iş parçacıklı (single-threaded)** programınızı birden fazla CPU üzerinde bu tür işleri yapan bir programa dönüştürme görevine **paralelleştirme (parallelization)** 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. Beklemek yerine, programınız hesaplama yapmak için CPU'yu kullanmak veya hatta daha fazla G/Ç isteği yayınlamak da dahil olmak üzere başka bir şey yapmak isteyebilir. 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** (**multiprogramming**) yaptığı gibi, G/Ç'nin tek bir program içindeki diğer **etkinliklerle (overlap)** çakışmasını 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 Örnek: İş Parçacğı(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, `mythread()` işlevini çalıştıracak iki iş parçacığı oluşturur. Bir iş parçacığı oluşturulduktan sonra hemen çalışmaya başlayabilir (programlayıcının kaprislerine 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.

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Şekil 26.2: Basit Konu Oluşturma Kodu (t0.c)

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.

Diyelim 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.

Görebileceğiniz gibi, iş parçacığı oluşturma hakkında düşünmenin bir yolu,

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”		

Şekil 26.4: İplik İzleme (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		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Şekil 26.4: İplik İzleme (2)

bunun bir işlev çağrısı yapmak gibi olduğudur; bununla birlikte, önce işlevi yürütmek ve ardından arayana geri dönmek yerine, sistem bunun yerine çağrılan rutin için yeni bir yürütme iş parçacığı oluşturur ve çağırandan bağımsız olarak çalışır, belki de oluşturma işleminden dönmeden önce, ama belki de çok sonra. Bundan sonra neyin çalışacağı, işletim sistemi (OS) programlayıcısı tarafından **belirlenir (scheduler,)** 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 getirir: 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		
returns immediately; T2 is done		
prints “main: end”		

Şekil 26.5: İplik İzleme (3)

26.3 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'da (sayfa 7).

İşte kod hakkında birkaç not. İlk olarak, Stevens'ın [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 öyleyse), ancak bu konuda çok akıllıca bir şey yapmayız (örneğin, sadece çıkın). Böylece, `Pthread create()` sadece `pthread create()`'i çağırır ve dönüş kodunun 0 olmasını sağlar; 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** (**counter**) 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; ./mainmain:
begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Basitçe, bir döngüde sayaca art arda 1 ekler
// Hayır, 10.000.000'i bu şekilde ekleyemezsiniz.
// bir sayaç, ancak sorunu güzel bir şekilde
   gösteriyor.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Sadece iki iş parçacığı başlatır (pthread_create)
// ve sonra onları bekler (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
           counter);
    return 0;
}
```

Şekil 26.6: Veri Paylaşımı: Uh Oh (t1.c)

Ne yazık ki, bu kodu tek bir işlemcide bile çalıştırdığımızda, mutlaka istenen sonucu alamıyoruz. Bazen 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? *(gasp)*

```
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 anlamanıza yardımcı olan yeni araçları her zaman öğrenmelisiniz. Burada, sökücü 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

Sorunun Kalbi (asıl): Kontrolsüz Zamanlama

Bunun neden olduğunu anlamak için, derleyicinin güncellemenin karşı koyması için oluşturduğu kod sırasını anlamamız gerekir. Bu durumda, sadece sayaca (`counter`) 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 (`counter`) 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) ekleyerek 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 (`counter`) birer birer artırmak üzere olduğunu düşünelim. Sayacın değerini (başlangıçta 50 diyelim) kayıt 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 çalıyor; 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: 2. Konu çalıştırmak için seçilir ve aynı kod parçasını girer. Ayrıca, sayacın değerini alarak ve eax'ına koyarak ilk talimatı yürütür (unutmayın: çalışırken her iş parçacığının kendi özel kayıtları vardır; kayıtlar, onları kaydeden ve geri yükleyen bağlam değiştirme kodu tarafından sanallaştırılır). Sayacın (`counter`) değeri bu noktada hala 50'dir ve dolayısıyla Thread 2 `eax`=50'dir. O halde, İş Parçacığı 2'nin eax'i 1 artırarak (dolayısıyla `eax`=51) sonraki iki talimatı yürüttüğünü ve ardından `eax` içeriğini sayaca (adres 0x8049a1c) kaydettiğini varsayalım. Böylece, global değişken sayacı (`counter`) artık 51 değerine sahiptir.

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

Basitçe söylemek gerekirse, olan şudur: sayacı (`counter`) artırma kodu iki kez çalıştırılmıştır, ancak 50'de başlayan sayaç şimdi yalnızca 51'e eşittir. Bu programın "doğru" bir versiyonu, sayaç (`counter`) değişkeninin 52'ye eşit olmasıyla sonuçlanmalıydı.

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 setlerine alışkın olanlarınız için not: x86, değişken uzunluklu komutlara sahiptir; bu `mov` komutu, 5 bayt bellek ve yalnızca ekleme 3):

OS	Thread 1	Thread 2	PC	eax	counter (talimattan sonra)
	kritik bölümden önce		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
yarıda kesmek					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
Yarıda kesmek					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Şekil 26.7: Sorun: Yakından ve Kişisel

```
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 yürütmede 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 olmasının muhtemel olduğu bu sonucu **belirsiz** - **indeterminate** olarak adlandırırız.

Bu kodu çalıştıran birden fazla iş parçacığı bir yarış durumuna neden olabileceğinden, bu kodu kritik bir bölüm olarak adlandırıyoruz. 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 dediğimiz şeydir. Bu özellik, kritik bölüm içinde bir iş parçacığının yürütülmesi durumunda diğerlerinin bunu yapmasının engelleneceğini garanti eder.

Bu arada, bu terimlerin neredeyse tamamı, alanında öncü olan ve bu ve diğer çalışmaları nedeniyle gerçekten de 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 inceleyeceğimiz), veritabanı yönetim sistemlerine ve hatta dağıtılmış bilgisayar sistemleri oluşturmanın altında yatan en güçlü tekniklerden biridir. sistemler [L+93].

Bir dizi eylemi **atomic** - **atomic** hale getirmenin arkasındaki fikir, basitçe "ya hep ya hiç" ifadesiyle ifade edilir; ya birlikte gruplandırmak istediğiniz tüm eylemler gerçekleşmiş gibi görünmelidir ya da hiçbir ara durum görünmeden hiçbiri gerçekleşmemiş gibi görünmelidir. Bazen, birçok **eylemin** - **transaction**, tek bir atomik eylemde gruplandırılmasına işlem 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 ilkellerini 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 kopyala-yazma gibi teknikleri kullanır. Bu bir anlam ifade etmiyorsa endişelenmeyin - gelecek bir bölümde mantıklı olacaktır..

26.4 Atomiklik Dileği - The Wish for Atomicity

Bu sorunu çözmenin bir yolu, tek bir adımda tam olarak yapmamız gerekeni yapan ve böylece zamansız bir kesinti olasılığını ortadan kaldıran daha güçlü talimatlara sahip olmak olabilir. 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 istendiği gibi gerçekleştirir. Talimatın ortasında kesilemez, çünkü donanımdan aldığımız garanti tam olarak budur: bir kesme meydana geldiğinde, talimat ya hiç çalışmamıştır ya da tamamlanana kadar çalışmıştır; ara durum yoktur. Donanım güzel bir şey olabilir, hayır?

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

atomik olarak yürütmektir:

```
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 aklı başında bir talimat setinde.

Bu nedenle, bunun yerine yapacağımız şey, üzerine **senkronizasyon ilkelleri** – **synchronization 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, doğru?

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

DOĞRULUK: SENKRONİZASYON NASIL DESTEKLENİR

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

26.5 Bir Sorun 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, sürecin devam edebilmesi için 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 kısa sürede yeterli olacaktır. O zamana kadar olmazsa, o zaman daha az uygun demektir ve mantıklı olana kadar o bölümü tekrar (ve tekrar) okumalısınız.

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

Bu dört terim, eşzamanlı kod için o kadar merkezi ki, onları açıkça belirtmenin faydalı olduğunu 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 koşulu** - **race condition** (veya **veri yarışı** [NM92]), birden çok 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 **deterministic - deterministic** değil, genellikle bilgisayar sistemlerinden beklediğimiz bir şey.
- 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.

26.6 Özet: Neden işletim sistem(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 OS *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 iç yapıları nasıl güncellediği konusunda endişelenmek zorunda kaldı. 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 ilkelleriyle dikkatlice erişilmesi gerekir.

References - Kaynaklar

[D65] "Solution of a problem in concurrent programming control" by E. W. Dijkstra. Commu- nications 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 amaz- ing 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 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 transac- tions 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 im- portantly, you learn a little more about how to become a serious* UNIX *programmer.*

Homework (Simulation) – Ödev(Simülasyon)

Bu program, x86.py, farklı iş parçacığı (**thread**) serpiştirmelerinin **(interleavings)** nasıl yarış (**race**) koşullarına neden olduğunu veya bunlardan nasıl kaçındığını görmenizi sağlar. Programın nasıl çalıştığına ilişkin ayrıntılar için README'ya bakın, ardından aşağıdaki soruları yanıtlayın.

Sorular (Questions)

1. Basit bir program olan "loop.s"yi inceleyelim. Önce 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 **(thread)**, 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.

Cevab:

Basitçe, `sub $1, %dx`, dx değerini 1 çıkarır ve -1'de bırakır. Daha sonra `test $0, %dx` ulaştığında jgte .top ile karşılaştırılır ve karşılaştırma sonucu değerlendirilir, bunun için başarısız olur (0'dan küçüktür), böylece durmaya devam eder. 0'dan (varsayılan) başlayacak ve bir sonraki komut için -1'e değişecek ve bu şekilde kalacak.

Şekil- 1.1

Şekil- 1.2

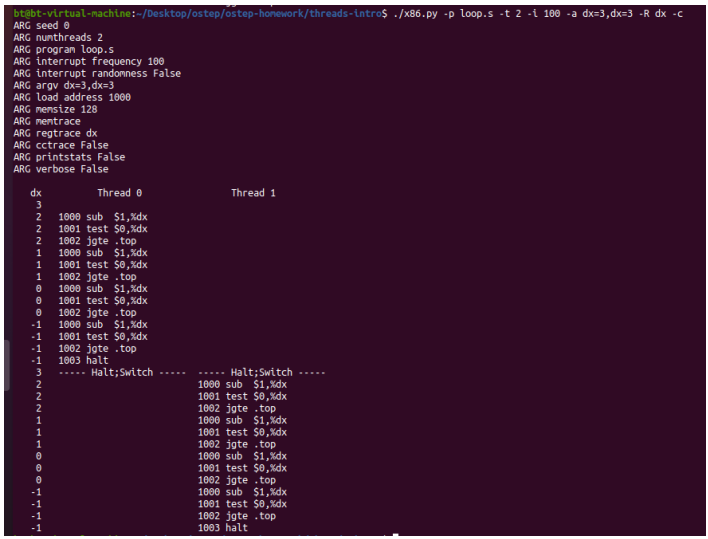
2. Aynı kod farklı bayraklar:

```
./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx
```

Bu, iki iş parçacığını belirtir ve her %dx'i 3 olarak başlatır. %dx hangi değerleri görecek? Kontrol etmek için -c ile çalıştırın. Birden çok iş parçacığının varlığı hesaplamalarınızı etkiler mi? Bu kodda bir yarış (**race**) var mı?

Cevab:

Sonuçların ilk kısmı için hiçbir şey değişmeyecek: tamamen sıralı / seriler ve paralel çalışmıyorlar, bu da yarış koşullarını oluşturmuyor. Geçiş, durana kadar gerçekleşmeyecek. Ancak, herhangi bir yarış (**race**) koşulu olmasa da, her konu için sonuçlar değişecektir. dx'in başlangıç değeri 3 olarak değiştirildiği için jgte. top ifadesi değerlendirecek ve "**loop**"un üstten devam etmesine neden olacaktır. Bu, dx'in değeri -1 (0'dan küçük) olana kadar 3 kez gerçekleşir.



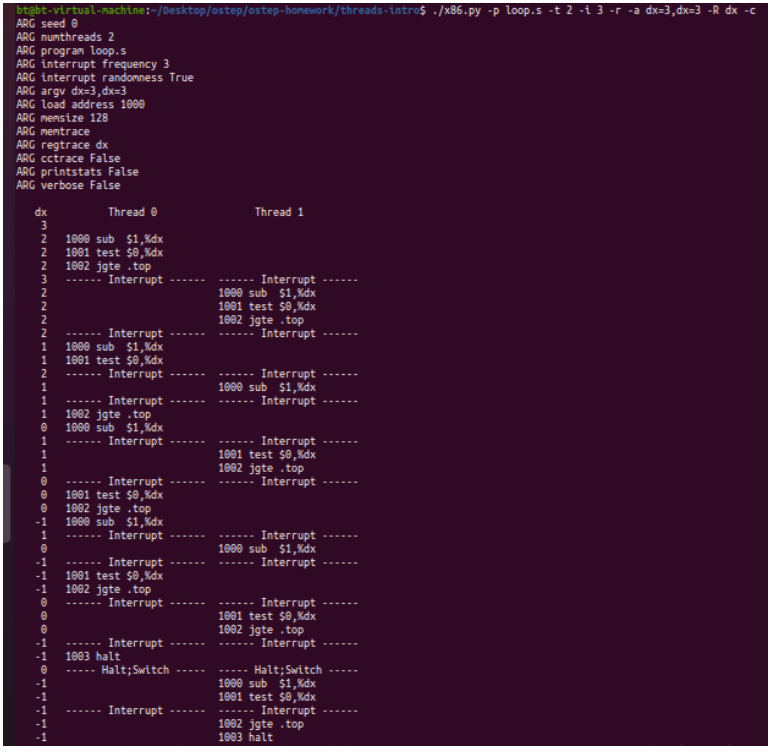
Şekil- 2

3. Bunu çalıştır:

`./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 **(interleavings)** görmek için farklı çekirdekler (-s) kullanın. Kesinti frekansı herhangi bir şeyi değiştirir mi?

Cevap:

Bu bir şeyleri değiştirir. Önceki sorudan farklı olarak, daha küçük aralık, iş parçacıklarını **(threads)** tümü dx kaydını yöneten daha küçük parçalara ayıracaktır. Bu, kesme aralığı programı tamamlamak/durdurmak için gereken adım sayısından az olduğunda gerçekleşir.


btgbt-virtual-machine:~/Desktop/ostep/ostep-homework/threads-intro$ ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx -c
ARG seed 0
ARG numthreads 2
ARG program loop.s
ARG interrupt frequency 3
ARG interrupt randomness True
ARG argv dx=3,dx=3
ARG load address 1000
ARG memsize 128
ARG memtrace
ARG regtrace dx
ARG cctrace False
ARG printstats False
ARG verbose False
dx Thread 0 Thread 1
3
2 1000 sub $1,%dx
2 1001 test $0,%dx
2 1002 jgte .top
3 ------ Interrupt ------ ------ Interrupt ------
2 1000 sub $1,%dx
2 1001 test $0,%dx
2 1002 jgte .top
2 ------ Interrupt ------ ------ Interrupt ------
1 1000 sub $1,%dx
1 1001 test $0,%dx
2 ------ Interrupt ------ ------ Interrupt ------
1 1000 sub $1,%dx
1 ------ Interrupt ------ ------ Interrupt ------
1 1002 jgte .top
0 1000 sub $1,%dx
1 ------ Interrupt ------ ------ Interrupt ------
1 1001 test $0,%dx
1 1002 jgte .top
0 ------ Interrupt ------ ------ Interrupt ------
0 1001 test $0,%dx
0 1002 jgte .top
-1 1000 sub $1,%dx
1 ------ Interrupt ------ ------ Interrupt ------
0 1000 sub $1,%dx
-1 ------ Interrupt ------ ------ Interrupt ------
-1 1001 test $0,%dx
-1 1002 jgte .top
0 ------ Interrupt ------ ------ Interrupt ------
0 1001 test $0,%dx
0 1002 jgte .top
-1 ------ Interrupt ------ ------ Interrupt ------
-1 1003 halt
0 ----- Halt;Switch ----- ----- Halt;Switch -----
-1 1000 sub $1,%dx
-1 1001 test $0,%dx
-1 ------ Interrupt ------ ------ Interrupt ------
-1 1002 jgte .top
-1 1003 halt

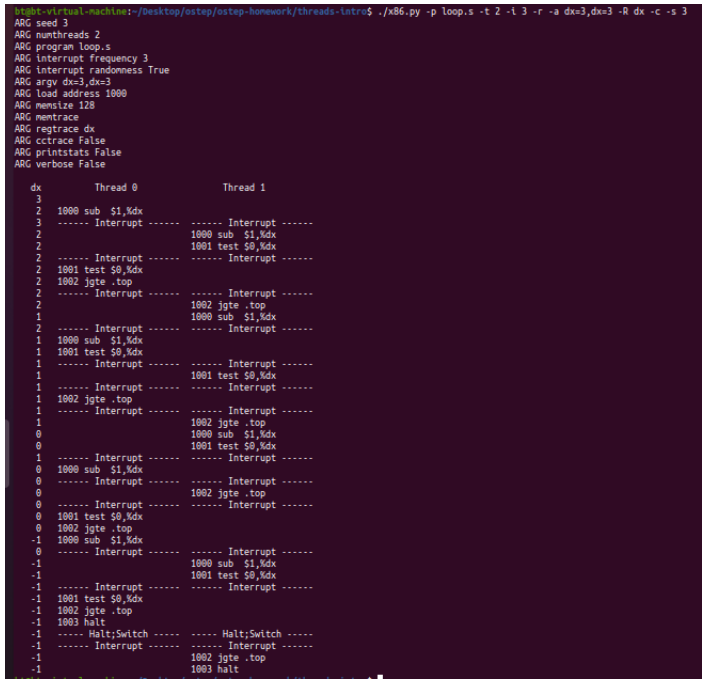
Şekil- 3.1

```
btgbt-virtual-machine:~/Desktop/ostep/ostep-homework/threads-intro$ ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx -c -s 1
ARG seed 1
ARG numthreads 2
ARG program loop.s
ARG interrupt frequency 3
ARG interrupt randomness True
ARG argv dx=3,dx=3
ARG load address 1000
ARG memsize 128
ARG memtrace
ARG regtrace dx
ARG cctrace False
ARG printstats False
ARG verbose False

  dx          Thread 0                Thread 1
   3
   2   1000 sub  $1,%dx
   3   ------ Interrupt ------  ------ Interrupt ------
   2                           1000 sub  $1,%dx
   2                           1001 test $0,%dx
   2                           1002 jgte .top
   2   ------ Interrupt ------  ------ Interrupt ------
   2   1001 test $0,%dx
   2   1002 jgte .top
   1   1000 sub  $1,%dx
   2   ------ Interrupt ------  ------ Interrupt ------
   1                           1000 sub  $1,%dx
   1   ------ Interrupt ------  ------ Interrupt ------
   1   1001 test $0,%dx
   1   1002 jgte .top
   1   ------ Interrupt ------  ------ Interrupt ------
   1                           1001 test $0,%dx
   1                           1002 jgte .top
   1   ------ Interrupt ------  ------ Interrupt ------
   0   1000 sub  $1,%dx
   0   1001 test $0,%dx
   1   ------ Interrupt ------  ------ Interrupt ------
   0                           1000 sub  $1,%dx
   0                           1001 test $0,%dx
   0                           1002 jgte .top
   0   ------ Interrupt ------  ------ Interrupt ------
   0   1002 jgte .top
   0   ------ Interrupt ------  ------ Interrupt ------
  -1                           1000 sub  $1,%dx
   0   ------ Interrupt ------  ------ Interrupt ------
  -1   1000 sub  $1,%dx
  -1   1001 test $0,%dx
  -1   1002 jgte .top
  -1   ------ Interrupt ------  ------ Interrupt ------
  -1                           1001 test $0,%dx
  -1                           1002 jgte .top
  -1   ------ Interrupt ------  ------ Interrupt ------
  -1   1003 halt
  -1   ----- Halt;Switch -----  ----- Halt;Switch -----
  -1                           1003 halt
```

Şekil- 3.2

```
btgbt-virtual-machine:~/Desktop/ostep/ostep-homework/threads-intro$ ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx -c -s 2
ARG seed 2
ARG numthreads 2
ARG program loop.s
ARG interrupt frequency 3
ARG interrupt randomness True
ARG argv dx=3,dx=3
ARG load address 1000
ARG memsize 128
ARG memtrace
ARG regtrace dx
ARG cctrace False
ARG printstats False
ARG verbose False

  dx          Thread 0                Thread 1
   3
   2   1000 sub  $1,%dx
   2   1001 test $0,%dx
   2   1002 jgte .top
   3   ------ Interrupt ------  ------ Interrupt ------
   2                           1000 sub  $1,%dx
   2                           1001 test $0,%dx
   2                           1002 jgte .top
   2   ------ Interrupt ------  ------ Interrupt ------
   1   1000 sub  $1,%dx
   2   ------ Interrupt ------  ------ Interrupt ------
   1                           1000 sub  $1,%dx
   1   ------ Interrupt ------  ------ Interrupt ------
   1   1001 test $0,%dx
   1   1002 jgte .top
   0   1000 sub  $1,%dx
   1   ------ Interrupt ------  ------ Interrupt ------
   1                           1001 test $0,%dx
   1                           1002 jgte .top
   0                           1000 sub  $1,%dx
   0   ------ Interrupt ------  ------ Interrupt ------
   0   1001 test $0,%dx
   0   1002 jgte .top
  -1   1000 sub  $1,%dx
   0   ------ Interrupt ------  ------ Interrupt ------
   0                           1001 test $0,%dx
  -1   ------ Interrupt ------  ------ Interrupt ------
  -1   1001 test $0,%dx
  -1   1002 jgte .top
   0   ------ Interrupt ------  ------ Interrupt ------
   0                           1002 jgte .top
  -1                           1000 sub  $1,%dx
  -1   ------ Interrupt ------  ------ Interrupt ------
  -1   1003 halt
  -1   ----- Halt;Switch -----  ----- Halt;Switch -----
  -1                           1001 test $0,%dx
  -1   ------ Interrupt ------  ------ Interrupt ------
  -1   ------ Interrupt ------  ------ Interrupt ------
  -1                           1002 jgte .top
  -1                           1003 halt
```

Şekil- 3.3



Şekil- 3.4

4. Şimdi farklı bir program, `looping-race-nolock.s`, 2000 adresindeki paylaşılan bir değişkene erişiyor; bu değişkene değer diyeceğiz. Anladığınızı doğrulamak için tek bir iş parçacığı ile çalıştırın:

   ```
   ./x86.py -p looping-race-nolock.s -t 1 -M 2000
   ```

   Çalıştırma boyunca değer (yani, bellek adresi 2000'de) nedir? Kontrol etmek için -c'yi kullanın.

   Cevap:

   `looping-race-nolock.s` kaynak kodunu inceledikten sonra, `add $1, %ax` adımının yürütülmesinden sonra değerin 1 olacağını tahmin ediyorum.

   Bunu kontrol ederek haklı olduğumu görüyorum:

Şekil- 4

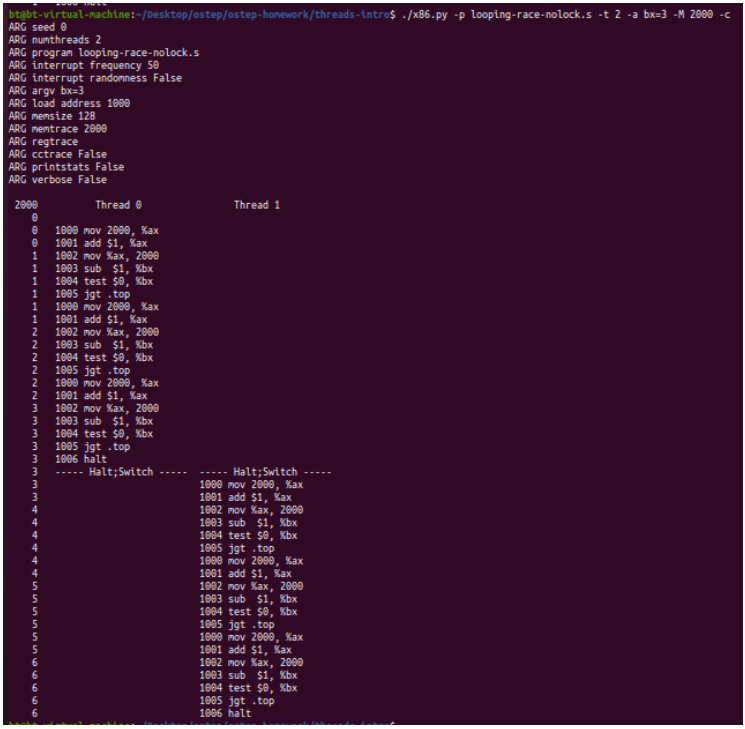
5. Çoklu yineleme/iş parçacığı ile çalıştır:

```
./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000
```

Neden her iş parçacığı (**thread**) üç kez döngü yapıyor? Değerin son değeri nedir?

Cevap:

`Test $0, %bx` dan sonra jgt. top ifadesi, bx'in 0'dan büyük olup olmadığını test ediyor. bx'in başlangıç değeri 3 olduğundan ve `sub $1, %bx` ile 1 azaltıldığından, bu etkili bir şekilde 3 yinelemeli bir döngü oluşturur. Her iki iş parçacığı da 3 kez döngüye girer ve her yineleme 2000 adresindeki değeri 1 artırır, bu nedenle nihai (**final**) değer 6'dır (2*3*1=6).



Şekil- 5

6. Rastgele kesinti aralıklarıyla çalıştırın:

```
./x86.py -p looping-race-nolock.s -t 2 -M 2000 -I 4 -r -s 0
```

farklı tohumlarla (-s 1, -s 2, vb.) İş parçacığının serpiştirilmesine bakarak, value'nin son değerinin ne olacağını söyleyebilir misiniz? Kesintinin zamanlaması önemli mi? Güvenli bir şekilde nerede meydana gelebilir? Nerede değil? Başka bir deyişle, kritik bölüm tam olarak nerede?

Cevap:

Evet, serpiştirilmiş iş parçacığının **(thread)** yürütülecek ifadelerinde adım adım ilerlerseniz, x'in son değerini belirleyebilirsiniz. Değer 2 olmalıdır (iş parçacıkları **(threads)** sırayla çalıştırıldığında). Kesilmemesi gereken kritik bölüm `sub $1, %ax` ve `test $0, %bx` arasındadır.

Bunun nedeni, `sub $1, %ax` ifadesinin balta kaydını azaltmasıdır ve eğer takip eden test `$0,%bx` kesilirse, diğer iş parçacığı **(thread)** da `sub $1,%bx'i` çalıştırabilir ve bu, tüm yinelemenin atlanacağı anlamına gelir (bx 2'den 0'a gitti, örneğin, karşılaştırılmadan iki kez azaltıldı).

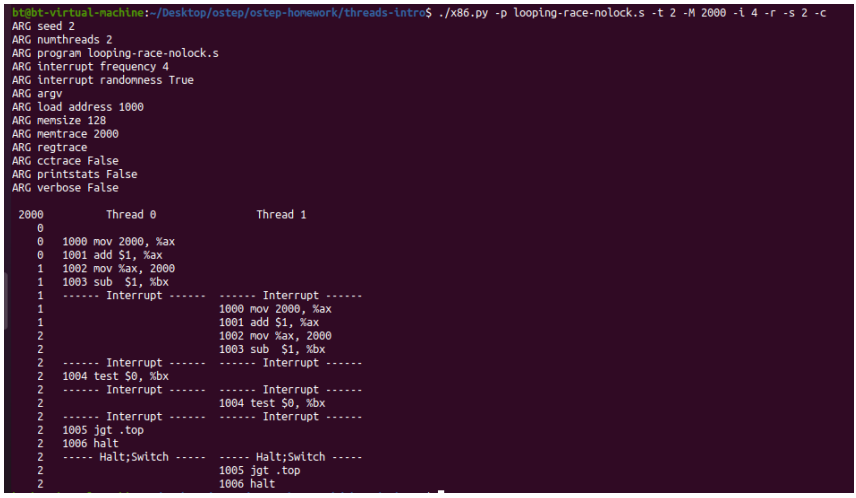
Örneğin :

```
bt@bt-virtual-machine:~/Desktop/ostep/ostep-homework/threads-intro$ ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0 -c
ARG seed 0
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 4
ARG interrupt randomness True
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False

 2000          Thread 0                Thread 1
    0
    0   1000 mov 2000, %ax
    0   1001 add $1, %ax
    1   1002 mov %ax, 2000
    1   1003 sub  $1, %bx
    1   ------ Interrupt ------  ------ Interrupt ------
    1                           1000 mov 2000, %ax
    1                           1001 add $1, %ax
    2                           1002 mov %ax, 2000
    2                           1003 sub  $1, %bx
    2   ------ Interrupt ------  ------ Interrupt ------
    2   1004 test $0, %bx
    2   1005 jgt .top
    2   ------ Interrupt ------  ------ Interrupt ------
    2                           1004 test $0, %bx
    2                           1005 jgt .top
    2   ------ Interrupt ------  ------ Interrupt ------
    2   1006 halt
    2   ----- Halt;Switch -----  ----- Halt;Switch -----
    2                           1006 halt
```

Şekil- 6.1

```
bt@bt-virtual-machine:~/Desktop/ostep/ostep-homework/threads-intro$ ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 1 -c
ARG seed 1
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 4
ARG interrupt randomness True
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False

 2000          Thread 0                Thread 1
    0
    0   1000 mov 2000, %ax
    0   ------ Interrupt ------  ------ Interrupt ------
    0                           1000 mov 2000, %ax
    0                           1001 add $1, %ax
    1                           1002 mov %ax, 2000
    1                           1003 sub  $1, %bx
    1   ------ Interrupt ------  ------ Interrupt ------
    1   1001 add $1, %ax
    1   1002 mov %ax, 2000
    1   1003 sub  $1, %bx
    1   1004 test $0, %bx
    1   ------ Interrupt ------  ------ Interrupt ------
    1                           1004 test $0, %bx
    1                           1005 jgt .top
    1   ------ Interrupt ------  ------ Interrupt ------
    1   1005 jgt .top
    1   1006 halt
    1   ----- Halt;Switch -----  ----- Halt;Switch -----
    1   ------ Interrupt ------  ------ Interrupt ------
    1                           1006 halt
```

Şekil- 6.2



Şekil- 6.3

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" yanıtı veriyor?

   Cevap:

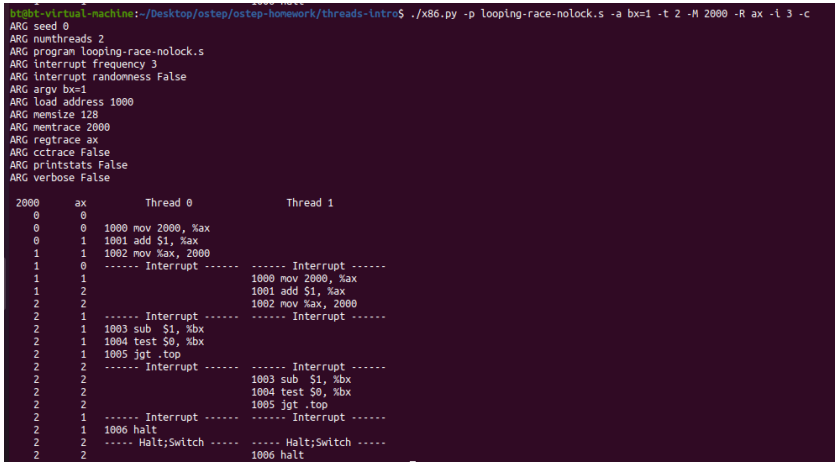
   i 1 olduğunda, x'in son değeri 1'dir. i 2 olduğunda, x'in son değeri 1'dir. i 3 olduğunda, x'in son değeri "doğru cevap" olan 2'dir.

```
bt@bt-virtual-machine:~/Desktop/ostep/ostep-homework/threads-intro$  ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -R ax -i 1 -c
ARG seed 0
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 1
ARG interrupt randomness False
ARG argv bx=1
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace ax
ARG cctrace False
ARG printstats False
ARG verbose False

 2000      ax          Thread 0                Thread 1
    0       0
    0       0   1000 mov 2000, %ax
    0       0   ------ Interrupt ------  ------ Interrupt ------
    0       0                           1000 mov 2000, %ax
    0       0   ------ Interrupt ------  ------ Interrupt ------
    0       1   1001 add $1, %ax
    0       0   ------ Interrupt ------  ------ Interrupt ------
    0       1                           1001 add $1, %ax
    0       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1002 mov %ax, 2000
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1                           1002 mov %ax, 2000
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1003 sub  $1, %bx
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1                           1003 sub  $1, %bx
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1004 test $0, %bx
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1                           1004 test $0, %bx
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1005 jgt .top
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1                           1005 jgt .top
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1006 halt
    1       1   ----- Halt;Switch -----  ----- Halt;Switch -----
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1                           1006 halt
```

Şekil- 7.1

```
bt@bt-virtual-machine:~/Desktop/ostep/ostep-homework/threads-intro$ ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -R ax -i 2 -c
ARG seed 0
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 2
ARG interrupt randomness False
ARG argv bx=1
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace ax
ARG cctrace False
ARG printstats False
ARG verbose False

 2000      ax          Thread 0                Thread 1
    0       0
    0       0   1000 mov 2000, %ax
    0       1   1001 add $1, %ax
    0       0   ------ Interrupt ------  ------ Interrupt ------
    0       0                           1000 mov 2000, %ax
    0       1                           1001 add $1, %ax
    0       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1002 mov %ax, 2000
    1       1   1003 sub  $1, %bx
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1                           1002 mov %ax, 2000
    1       1                           1003 sub  $1, %bx
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1004 test $0, %bx
    1       1   1005 jgt .top
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1                           1004 test $0, %bx
    1       1                           1005 jgt .top
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1006 halt
    1       1   ----- Halt;Switch -----  ----- Halt;Switch -----
    1       1                           1006 halt
```

Şekil- 7.2



Şekil- 7.3

8. Daha fazla döngü için aynısını çalıştırın (örneğin, set -a bx=100). Hangi kesinti aralıkları (-i) doğru sonuca götürür? Hangi aralıklar şaşırtıcı?

   Cevap:

   "Doğru" sonuç 200'dür. Daha fazla döngü olduğundan, artık kritik bölümün kesintiye uğraması için daha fazla fırsat vardır. İlginç olan, 3'ün katlarının (3, 6, 9, 12, vb.) hepsinin 200'lük doğru yanıtı vermesidir. Diğerlerinin tümü 100'den 200'ün altına kadar bazı farklılıklar verir.

   Yani, 3'ün katı olan aralıkların doğru cevaba götürdüğünü buldum. Bunun nedeni, talimatların ilk üçünün birlikte yürütülmesi gerektiğidir.

```
bt@bt-virtual-machine:~/Desktop/ostep/ostep-homework/threads-intro$ ./x86.py -p looping-race-nolock.s -a bx=100 -t 2 -M 2000 -R ax -i 2 -c
ARG seed 0
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 2
ARG interrupt randomness False
ARG argv bx=100
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace ax
ARG cctrace False
ARG printstats False
ARG verbose False

 2000      ax          Thread 0                Thread 1
    0       0
    0       0   1000 mov 2000, %ax
    0       1   1001 add $1, %ax
    0       0   ------ Interrupt ------  ------ Interrupt ------
    0       0                           1000 mov 2000, %ax
    0       1                           1001 add $1, %ax
    0       1   ------ Interrupt ------  ------ Interrupt ------
    1       1   1002 mov %ax, 2000
    1       1   1003 sub  $1, %bx
    1       1   ------ Interrupt ------  ------ Interrupt ------
    1       1                           1002 mov %ax, 2000
    1       1                           1003 sub  $1, %bx
    1       1          Interrupt                Interrupt
```

```
  196     196                           1002 mov %ax, 2000
  196     195   ------ Interrupt ------  ------ Interrupt ------
  196     195   1003 sub  $1, %bx
  196     195   1004 test $0, %bx
  196     195   1005 jgt .top
  196     196   ------ Interrupt ------  ------ Interrupt ------
  196     196                           1003 sub  $1, %bx
  196     196                           1004 test $0, %bx
  196     196                           1005 jgt .top
  196     195   ------ Interrupt ------  ------ Interrupt ------
  196     196   1000 mov 2000, %ax
  196     197   1001 add $1, %ax
  197     197   1002 mov %ax, 2000
  197     196   ------ Interrupt ------  ------ Interrupt ------
  197     197                           1000 mov 2000, %ax
  197     198                           1001 add $1, %ax
  198     198                           1002 mov %ax, 2000
  198     197   ------ Interrupt ------  ------ Interrupt ------
  198     197   1003 sub  $1, %bx
  198     197   1004 test $0, %bx
  198     197   1005 jgt .top
  198     198   ------ Interrupt ------  ------ Interrupt ------
  198     198                           1003 sub  $1, %bx
  198     198                           1004 test $0, %bx
  198     198                           1005 jgt .top
  198     197   ------ Interrupt ------  ------ Interrupt ------
  198     198   1000 mov 2000, %ax
  198     199   1001 add $1, %ax
  199     199   1002 mov %ax, 2000
  199     198   ------ Interrupt ------  ------ Interrupt ------
  199     199                           1000 mov 2000, %ax
  199     200                           1001 add $1, %ax
  200     200                           1002 mov %ax, 2000
  200     199   ------ Interrupt ------  ------ Interrupt ------
  200     199   1003 sub  $1, %bx
  200     199   1004 test $0, %bx
  200     199   1005 jgt .top
  200     200   ------ Interrupt ------  ------ Interrupt ------
  200     200                           1003 sub  $1, %bx
  200     200                           1004 test $0, %bx
  200     200                           1005 jgt .top
  200     199   ------ Interrupt ------  ------ Interrupt ------
  200     199   1006 halt
  200     200   ----- Halt;Switch -----  ----- Halt;Switch -----
  200     200                           1006 halt
```

Şekil- 8

-*Kod çok uzun olduğu için tümünü alamadım*

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`

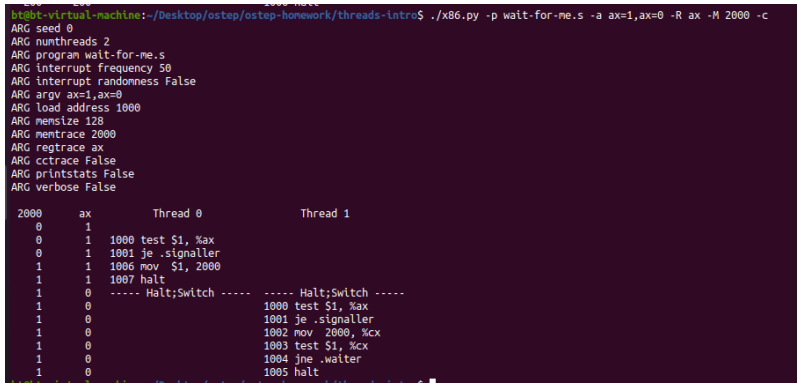
   Bu, %ax kaydını iş parçacığı 0 için 1'e ve iş parçacığı 1 için 0'a ayarlar ve %ax ile bellek konumu 2000'i izler. Kod nasıl davranmalı? 2000 konumundaki değer iş parçacıkları tarafından nasıl kullanılıyor? Nihai değeri ne olacak?

   Cevap:

   Konum 2000'deki değer, işaretçinin mi yoksa garsonun mı yürütmesi gerektiğini belirlemek için bir bayrak olarak kullanılıyor.

   `ax=1, ax=0` ile ilk çalıştırma, ilk iş parçacığına **(thread)** işaretçi **(signaller)** ve ikinci iş parçacığına garson **(waiter)** olmasını söyler. İlk iş parçacığı yürütür ve 2000 konumundaki değeri 1'e ayarlar, bu da garsonun **(waiter)** çalışabileceği anlamına gelir. İkinci iş parçacığı çalıştığında, çalışıp çalışamayacağını görmek için konum 2000'deki değeri kontrol eder: çalışabilir, çünkü sinyalci zaten yapabileceğini söylemiştir.

   `ax=0, ax=1` ile ikinci çalıştırma, ilk iş parçacığının garson **(waiter)** olmasını ve ikinci iş parçacığının **(thread)** garson **(waiter)** olmasını söyler. Bununla birlikte, kesme frekansı varsayılan olarak 50'dir ve bu nedenle, sinyal vericide temsil edilen ikinci iş parçacığı yürütülmeden önce garson **(waiter)** birçok kez döngü yapar. İkinci, işaretçi **(signaller)**, iş parçacığı nihayet **(finally)** yürütülüp durduğunda, birinci, garson **(waiter)**, iş parçacığı artık devam edebilir ve durabilir.



Şekil- 9

Kod yukarıdaki gibi davranır. 2000'deki değer 0'dan 1'e değişir.

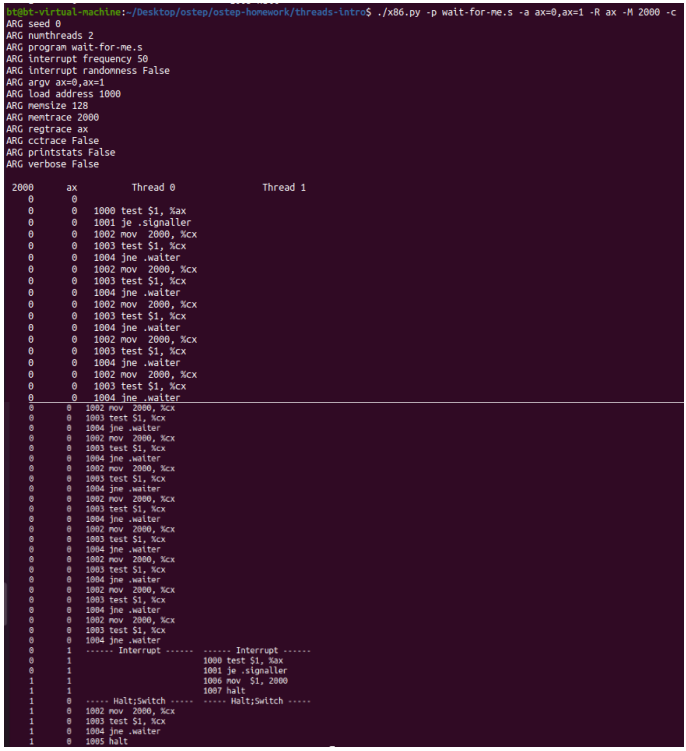
10. Şimdi girişleri değiştirin:

```
./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -
M 2000
```

İş parçacıklar **(Threads)** nasıl davranır? Konu 0 ne yapıyor? Kesinti aralığını değiştirmek (örneğin, -i 1000 veya rastgele aralıklar kullanmak) izleme sonucunu nasıl değiştirir? Program CPU'yu verimli bir şekilde kullanıyor mu?

Cevap:

İş parçacığı **(Thread)** davranışı aşadaki gibidir, iş parçacığı 0, bir kesinti oluşana kadar 2000'deki değeri kontrol etmek için döngü halindedir. Zaman aralığı değiştirilirse, döngü sayısı da değişecektir, döngü kodu verileri değiştirmediğinden, kesme aralığı ne kadar uzunsa, döngü sayısı o kadar fazla olur ve **CPU** kullanım verimliliği o kadar düşük olur.



Şekil- 10