Preemption (Çağrı)

Önceki ders notlarında preemptionlar hakkında biraz konuştuk. Burada Solaris'teki çağrılar hakkında daha açıklayıcı bilgiler verilecektir. Bu kafa karıştırıcı bir türdür, ama tüm detayları anladığınızda, Solaris Threads sistemi çok iyi tasarlanmış olduğunu görürsünüz.

Solaris sisteminde iki çeşit thread var: User -level threads (Kullanıcı düzeyindeki çağrılar) ve system-level threads (sistem düzeyinde çağrılar). Bu ikisi arasındaki kafa karıştırıcı farkı anlatmaya çalışacak. User-level threads yalnızca çalışan processler içinde mevcuttur, bunlar işletim sistemi desteğine sahip değiler. Yani, eğer bir program bir çok user-level threads de sahipse, bir thread ile bir "normal" Unix programı, işletim sistemi ile eşitmiş gibi bakılır. Solaris'te , user-level threads non-preemptivedir. Diğer bir değişle, bir thread çalıştığı zaman **pthread_exit()** yada **pthread_join()** gibi bir çağrı gelmediğinde bir user-level threads güvenli bloklar da kesme yapmayacaktır.

Bir threadin yürütülmesi durduğu zaman başka bir thread çalışmaya başlar, biz bir thread içerik anahtarını çağırırız. Yukarıdaki düzeltilmede, güvenli bloklar yaptıkları zaman user-level threads yanlızca içerik bağlam yapar. Bu konuda düşünüyorsanız , **setjmp () / longjmp ()** ile thread içerik anahtarını yerine getirebilirsiniz. Bunun anlamı thread içerik anahtarlama yapmak için işletim sistemine ihtiyaç duyulmamasıdır. Bu da dahil onlar sistem çağrısı itemediğinde içerik anahtarlama ve user-level threads arasında çok hızlı yapılabilir.

Yani system-level thread nedir? Bu işletim sistemi tarafından yürütme birimidir. Standart non-threaded Unix programların her biri bir ayrı system-level thread tarafından yönetilir. İşletim sistemi peryodik kesmeleri system-level thread 'te halen çalışan,onun kayıt durumunu ve farklı systen-level thread çalışmasını zaman dilimlemeli olarak yönetir. Bu, aynı anda bir çok programın nasıl olduğunu çalıştırabilirsin. Bu işlem ayrıca bağlam anahtarlamayı çeğırdı.

Biz **pthread_create ()** fonksiyonu çağırdığımız zaman, aynı system-level thread tarafından yönetilen yeni bir user-lvel thread oluşturarak thread çağırılıyor.ilişkisi bulunan her iki thread non-preemptively çalıştıracak. Aslında, her koleksiyon system-level thread tarafından servis edilen user-level threads'dir. Butun çalışılabilir olanlar birbiriyle ilişki içerisindedir. Bütün programlar bu yolla çalışan prvious threads lecture içindedir.

Bir kaçtanesine bakalım . İlk olarak, bakmak **preempt1.c** . Bu iki thread ile çalışan bir fork programıdır, bunların herbiri sonsuz bir döngüde çalışır. Çalıştırdığımız *zaman:*

```
UNIX> preempt1
thread 0.  i =          0
```

thread 0. i = 1
thread 0. i = 2
thread 0. i = 3
thread 0. i = 4
thread 0. i = 5
...
Konu 0. i = 1
Sadece 0 iş parçacığının çalıştığını göreceksin.(eğer bunu ctrl-c ile iptal edemiyorsanız, başka bir pencere açın ve işlemi sona erdir komutuyla etkisiz hale getir.) 1iş parçacığının hiç işlemesinin sebebi, 0 iş parçacığının cpu'dan asla ayrılamamasıdır. Bu starvasyon olarak adlandırılır.
Şimdi, farklı kullanıcı-seviye iş parçacıklarını farklı sistem-seviye iş parçacıklarına bağlayabilirsiniz. Eğer bir kullanıcı –seviye iş parçacığı işliyorsa, öyleyse aynı noktada işletim sisteminin onu yarıda bırakacağı ve başka bir kullanıcı-seviye iş parçacığı başlatacağı anlamına geliyor bu. Çünkü her iki kullanıcı-seviye iş parçacığı farklı sistem seviye iş parçacıklarına bağlıdır.
Bir kullanıcı seviye iş parçacığını farklı bir başka sistem seviye iş parçacığına bağlamanın yolu pthread_create() komutunu farklı bir yolla çağırmaktır. preempt2.c.'e bakınız. Size pthread_create() uyarısıyla ,'' bu iş parçacığını farklı bir sistem seviye iş parçacığıyla oluşturun' uyarısı gelecektir . onu çalıştırdığınızda iki iş parçacığının işlediğini göreceksiniz; ve bundan böyle her işleyen iş parçacığının devreye girmesiyle diğer iş parçacığı çalışmaya başlar:
UNIX> preempt2
thread 0. i = 0
thread 1. i = 0
thread 0. i = 1
thread 1. i = 1
thread 1. i = 2
thread 0. i = 2
thread 0. i = 3
thread 1. i = 3
thread 0. i = 4
thread 1. i = 4
thread 0. i = 5
thread 1. i = 5

bu Solaris'in ustalık isteyen bölümü. Eğer bir iş parçacığı bloke edici sistem çağrısında bulunursa, o zaman eğer diğer kullanıcı-seviye iş parçacıkları aynı sistem-seviye iş parçacığına bağlılarsa, yeni bir sistem-seviye iş parçacığı oluşturulur ve bloke edici iş parçacığı ona bağlanır. Bu şekilde işlemesi iş parçacıkları bloke edildiğinde, diğer kullanıcı-seviye iş parçacıklarının işlemesine olanak sağlar.
Bu Solaris ve SunOS gibi daha eski işletim sistemlerin in temel farkıdır. SunOS, işlem başına sadece bir sistem-seviye iş parçacığına izin verir. Bu yüzden, eğer bir kullanıcı-seviye iş parçacığı SunOS'de bloke edici sistem çağrısında bulunursa, bütün iş parçacıkları sistem çağrısı tamamlanana kadar bloke eder. Bu bir taramadır. Solaristeki tasarım çok iyidir.
Bu yüzden, preempt3.c.'e bakınız şimdi. Öncelikle, iş parçacıklarının bir kullanıcı-seviye iş parçacıkları olarak aynı sistem-seviye iş parçacığına bağlı bir şekilde oluşturulduğunu görmeniz gerekmektedir. Daha sonra, 0 iş parçacığının döngüden başlamadan önce, standart giren bilgiden bir karakteri okur öncelikle. Bu bir bloke edici sistem çağrısıdır. Bu yüzden; o, bu iş parçacıklarının temel iş parçacığı ve iş parçacığı 1'den ayrı bir sistem iş parçacığına

bağlı olmasıyla sonuçlanır. Bu yüzden, bloke ettiğinde, iş parçacığı 1 devreye girebilir. Devam edin ve çalıştırın:

```
UNIX> preempt3
Thread 0: stopping to read
thread 1. i = 0
thread 1. i = 1
thread 1. i = 2
thread 1. i = 3
```

böylelikle, 0 iş parçacığı bloke edilir ve iş parçacığı 1 devreye girer. Onlar bu yüzden ayrı sistem iş parçacıklarına bağlıdırlar. Şimdi RETURN yazalım, ve ardından 0 iş parçacığ tekrar başlayacaktır ve onların preempt2'te olduğu gibi araya nasıl girdiklerini göreceksiniz.

```
...
thread 1. i = 3
( RETURN burada yazıldı )
Thread 0: Starting up again
thread 0. i = 0
thread 1. i = 4
thread 0. i = 1
thread 1. i = 5
thread 0. i = 2
thread 1. i = 6
thread 0. i = 3
...
```

Bu kullanıcı/sistem seviye iş parçacıkları ve kısaca bir önalım. Yukarıdaki örneklere bakın tekrar aklınz karıştıysa.

Yarış Koşulları ve Muteksler

race1.c'e bir göz atın. Bu güzel basit bir program. Komut çizgi tartışmaları, kullanıcının iş parçacıkları numarasını ,dizilim ölçüsü ve tekrarlama sayısını özelleştirmeyi gerektirir. Daha sonar program aşağıdaki gibi işler. Dizgiyi dizim ölçüsü karakterlerinden ayırır. Daha sonra, her iş parçacığına kimliğini, yinelemeleri ve karakter dizisini benimseterek, bu nthreads iş parçacıklarını kollara ayırır. Her iş parçacığı bir kullanıcı-seviye iş parçacığıdır, bu yüzden iş parçacıkları önlenemezler. Şimdi her iş parçacığı belli yinelemeleri işletir. Her yinelemede, karakter dizisinde başka bir karakterle yer değiştirilir—0 iş parçacığı 'A', iş parçacığı 1 'B' vs. kullanır. Bir yinelemenin sonunda iş parçacığı karakter dizisini işleme sokar. Bu yüzden, 4,4,1, tartışmalarıyla çağrıda bulunursak, aşağıdaki çıktıyı elde ederiz, daha doğrusu bu elde ettiğimiz sonuçtur:

```
UNIX> race1 4 4 1
Thread 0: AAA
Thread 1: BBB
Thread 2: CCC
Thread 3: DDD
```

Benzer şekilde, aşağıdaki su sekildedir:

```
UNIX> race1 4 4 2
Thread 0: AAA
Thread 0: AAA
Thread 1: BBB
Thread 1: BBB
Thread 2: CCC
Thread 2: CCC
```

```
Thread 3: DDD
Thread 3: DDD
UNIX> race1 4 30 2
Thread 0: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread 0: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Thread 1: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread 1: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Thread 2: CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Thread 2: CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Thread 3: DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
Thread 3: DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
UNIX
```

Ne beklenen midir? Açıkçası, ilk ve üçüncü aramalar tuhaf durmuyor, ama ikinci çağrı çok berbat: Ben konuları rastgele bir sırayla yazdırmak görmeyi beklediğiniz ama Konu 0 C 'nin, ya da konu 1 yazdırmak için beklemek değildir D's yazdırın. Neler oluyor?

Neler oluyor konuları yerde preempted olmasıdır. Özellikle, sağ önce preempted olabilir **printf** başka bir iş parçacığı değiştirilebileceği anlamına gelir tablosu, **s** özgün iş parçacığı aslında çağırdığında, sonra ve **printf ()**değerleri tablosu, **s** iplik olduklarını düşündüm değildir.

Dosyası **output.txt** başka bir şekilde, aynı noktada göstermektedir. Bu yaratılmıştır:

```
UNIX> race2 10 70 100> output.txt
```

Bu dosyanın satır 181 olduğunu fark edeceksiniz:

```
Konu 1:
HHHHHHHHHHHHHHHHHHHHHHHHHHHHFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Yani, 'B' dışındaki karakterler bir hat çıkışı konu 1 baskı, fakat H vardır sadece *ve* F kullanıcısının. Bu kontrol akışı aşağıdaki gibi bir şey olduğu anlamına gelir:

- Konu 1 doldurulacak **s** B ile. Sonra bir içerik anahtarı oldu.
- Sonraki Konu 5 doldurulur **s** yine F Kullanıcı, ve, bir bağlam anahtarı oldu.
- Konu 7 kontrolünü kazandı ve yarı dolu **s** H ile ve sonra başka bir içerik anahtarı oldu.
- Konu 1 kontrolü tekrar kalktı ve baskılı **s** .

Alt satırda olduğunu **race2.c** bir sahip *yarış durumu* . Bu konuları tarafından paylaşılan bazı bellek, bu durumda dize anlamına gelir **ler** çok şaşırtıcı sonuçlara yol açan, kontrolsüz bir şekilde ulaşılır. Eğer konuları ile program, paylaşılan bellek dikkat etmelisiniz zaman. Birden fazla iş parçacığı paylaşılan bellek değiştirebilirsiniz, o zaman sık sık garip şeyler belleğe olmaz böylece bellek korumak gerekir.

Bizim **yarış** programı, biz onu değiştirme ve yazdırma yaparken hiçbir iplik başka bir iş parçacığı tarafından kesildi edilebileceğini uygulayarak yarış durumu düzeltebilirsiniz **s** . Bu yapılabilir **mutex** . bazen bir `` kilit" ya da `` ikili semafor adı verilen," pthreads yılında muteksler başa çıkmak için üç prosedürü vardır:

```
pthread_mutex_init (pthread_mutex_t * muteks, NULL);
pthread_mutex_lock (pthread_mutex_t * muteks);
pthread_mutex_unlock (pthread_mutex_t * muteks);
```

Sizinle bir mutex oluşturmak **pthread_mutex_init ()** . Sonra herhangi bir iş parçacığı muteksi kilitlemek veya kilidini açabilir. Bir iş parçacığı mutex kilitler zaman, başka hiçbir iş parçacığı kilitlenebilir. Onlar ararsanız**pthread_mutex_lock ()** Konu kilitli iken iplik kilidi açılana kadar, sonra da engeller. Sadece bir iş parçacığının bir anda mutex kilitlenebilir.

Yani, biz tamir **yarışı** olan programı **race3.c** . Bir konu sadece değiştirmeden önce mutex kilitler fark edeceksiniz**ler** ve sadece baskıdan sonra muteks kilidini **s** . Çıkış mantıklı, böylece, bu program giderir:

```
UNIX> race3 4 4 1
Konu 0: AAA
Konu 1: BBB
Konu 2: CCC
Konu 3: DDD
UNIX> race3 4 4 2
Konu 0: AAA
Konu 0: AAA
Konu 2: CCC
Konu 2: CCC
Konu 1: BBB
Konu 1: BBB
Konu 3: DDD
Konu 3: DDD
UNIX> race3 4 70 1
Konu: 0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Konu 1:
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Konu 2:
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Konu 3:
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
UNIX> race3 10 70 100> output3.txt
```

Muteksler üzerinde terse tavsiye

Senkronizasyon ilkelleri ile ilgili zorluklardan biri çok fazla konuları sistemi daraltıcı olmadan istediklerini elde etmektir. Örneğin, içinde **jtalk_server** programı, mevcut tüm bağlantıları tutan bir veri yapısına sahip olmak zorunda kalacaktır. Birisi soket verdiği zaman, bu veri yapısı için bağlantı ekleyin. Birisi / onu sonlandırıldığında **jtalk**oturumunda, daha sonra veri yapısı arasındaki bağlantıyı silin. Ve birisi sunucuya bir çizgi gönderdiğinde, veri yapısı çapraz ve tüm

bağlantıları için hat göndereceğiz. Bir mutex ile veri yapısını korumak gerekir. Örneğin, veri yapısı geçme ve aynı zamanda bir bağlantı silme olmak istemiyorum. Seni düşünmek istediğiniz bir şey veri yapısı nasıl koruyacağınızı, ama aynı zamanda onlar gerçekten yoksa çok fazla iş parçacığı mutex üzerinde engellemeye neden olmaz. Biz daha sonra bu konuda daha fazla konuşacağız.

Dişli telnet

Bak **cat1.c** . Bu basit bir **kedi** bir değişmeze kullanarak program **inout ()** biz biraz kullanacağı söyledi.

Sonra, bakmak **cat2.c** . Bu başka bir şeydir **kedi** ayrı kullanır programı **inout** konu.

Son olarak, bakmak **th_telnet1.c** . Bu istekleri istenilen sunucu ve port bağlantısı, ve sonra iki kapalı çatal **inout ()**konuları. İlk standart girdiden okur ve soket bağlantısı için çıkış gönderir. Ikinci bir soket okur, ve standart bir çıkışa gönderir. Ya soket veya standart girdi kapalı olduğunu algılarsa, sonra telnet oturumunu sona erdirir.

Bu çok basit bir kod olduğunu ve konuları gücünü göstermelidir. Bu yazı çok daha basittir **telnet** ile **seçin** , değil mi?