Niçin Thread'ler?

Thread ile program için pek çok neden vardır. Bu sınıfın önemli olan iki bağlamı vardır:

1. Senkronize ve verimli bir şekilde uyumsuz olaylar ile başa çıkmak için.
2. Paylaşılan bellek çok işlemcili paralel performansı almak için.

Thread ler bir işletim sistemi yazımı için çok yardımcı olduğunu göreceksiniz. Bu nedenle, burada thread leri öğrenerek, size CS460 için bir destek katacaktır.

Thread nedir?

Thread lere genellikle "hafif işlemler" denir. Unix tipik bir süreç, CPU durumu (yani kayıt), bellek (kod, globals, yığın ve) ve OS bilgisi (örneğin, açık dosyalar, bir süreç kimliği, vb gibi) oluşur, bir thread sisteminde "task" olarak adlandırılan daha büyük bir varlık olan "pod" yada "heavyweight process" bazen olmayabiliyor.

"Task" bir bellek (kod, globals, yığın), OS bilgisi ve threadlerden oluşur. Her thread bir yürütme birimi ve CPU (yani kayıt) ve herhangi bir yığından oluşur. Birden çok iş parçacığı, çoklu süreçleri benzer bir görev içinde birden çok iş parçacığı dışında aynı kodu, globals ve yığın olarak kullanmaktadır. Böylece Unix iki süreç üzerinden iletişimkurabilir, biri sadece işletim sistemi (dosyalar, borular ya da prizler aracılığıyla gibi) üzerinden diğeri thread belleği üzerinden iletişim kurabiliyor.

Threadler zaman program konuları ile, aynı anda yürütülüyor varsayalım. Diğer bir deyişle her konu kendi CPU su üzerinden yürütmeli ve tüm konuları aynı bellek paylaşıyormuş gibi görünmelidir.

Bir thread sisteminde sağlamanız gereken çeşitli ilkeler vardır. Iki temel thread ile başlayalım. Bu ilk tartışma genel bir thread sisteminden bahsedeceğiz. Daha sonra belirli (specific) olanlar (örneğin, POSIX ve Solaris threadleri gibi) hakkında konuşacağız.

- `tcb thread_fork (prosedür, tartışmalar);`

  Bu verilen argümanlar ile verilen prosedürü çalışan yeni bir thread oluşturmak için söylüyor. Bazen argümanlar ihmal edilirse,sadece bir bağımsız değişkene (**(void *)**) izin verilir. Bu yeni thread (bir thread kontrolü bloğu veya TCB) bir pointer ile döner.

- `thread_join (TCB)`

tcb çalışmalarını bitirinceye kadar bu threadı
beklemelidir. Genellikle **thread_join ()** kendi çıkış değeri olarak bir tam sayı ya da (void *) döndürür. Unix **wait ()** i analog olarak **thread_join ()** düşünebiliriz. Belirtilen thread i tamamlamak için bekler ve thread çıkış durumu hakkında bilgi toplar.

Posix Thread leri

Hydra / Cetus makinelerimizde , kullanabileceğiniz bir thread sistemi vardır. ``Solaris threads" olarak adlandırılır. Bir standarttır `` Posix threads" adı verilen başka bir thread sistemi yoktur. Neyse ki, Hydra / Cetus makinelerimiz de Posix threads destekliyor. Posix threads çağrıları sadece Solaris threads üstüne bir üst tabaka oluyor.

Programda Posix threads faydalanmak için, aşağıdaki direktifin dahil olması gerekir:

```
# Include <pthread.h>
```

libpthread.a nesne dosyalarına bağlanmak zorundadır. (Program main.c içinde ise, yani threadi çalıştırılabilmek için aşağıdakileri yapmanız gerekir):

```
UNIX> cc-c main.c
UNIX> cc-o ana main.o-lpthread
```

Sizde **gcc** kullanabilirsiniz. **Pthread** kütüphanesinde önemsiz bir sürü var. Çeşitli man sayfalarında bu konuyu okuyabilirsiniz. `` **man pthreads** " ile başlayın. Yukarıda tanımlanan iki temel ilkelerin Posix dizileri aşağıdaki gibidir:

```
int pthread_create (pthread_t * new_thread_ID,
                    const pthread_attr_t * attr
                    void * (* start_func) (void *),
                    void * arg);

int pthread_join (pthread_t target_thread,
                  void ** durumu);
```

Bu çokta kötü değildir, ve genel açıklamalar biraz kapalı.Bunun yerine bir thread kontrol bloğu için bir pointer döndürmek, **pthread_create()** yerine birinin adresini geçer, ve onun içini doldurur. **attr** argümanı merak etmeyin sadece **NULL** kullanın. Sonra **func** fonksiyonu ve **arg** bir **(void*)** işlevine argümandır. Dönen **pthread_create** TCB olduğu zaman *** new_thread_ID** ve yeni thread **func (arg)** olarak çalışıyor.

pthread_join () bir threadbelirtmek ve bir **(void *)** işaretçisi vermektedir. Belirtilen thread çıktığında, **pthread_join ()** çağrısı geri dönecek ve *** status** iadesi veya bir thread çıkış değeri olacaktır.

Tüm Posix thread çağrıları olarak, bir tamsayıdır döndürülür. Sıfır, herşey yolunda gitti. Aksi takdirde, bir hata oluştu. Sistem çağrıları gibi, her zaman bir hata olmuşsa hatayı görmek için bu çağrıların dönüş değerlerini kontrol etmek iyidir. Burada ders notları benim olduğu için kodumda hata denetimi ihmal edeceğiz, ama dosyalar olduğunda bunu yapmalıyım.

Bir thread çıkışı nasıl ? return veya **pthread_exit ()** çağırarak

Tamam, böylece aşağıdaki program kontrol edelim (hw.c)

```
# Include <pthread.h>

# Include <stdio.h>

void *printme ()
{
  printf ("Merhaba dünya \ n");
   return NULL;
}

main()
{
  pthread_t tcb;
  void *status;

  if (pthread_create(&tcb, NULL, printme, NULL) != 0) {
    perror("pthread create");
    exit(1);
  }
  if (pthread_join(tcb, &status) != 0) { perror("pthread_join"); exit(1); }
}
```

hw.c kodunu kopyalayıp derleyip çalıştırmayı deneyin.

Birden fazla thread çatallamak

Şimdi, **print4.c**.bakalım. Print4 çıktısı **"Hi. I'm thread n"** olur, n 0 ile 3 arasında bir değere gider. Bu **pthread** kütüphanesinin nasıl çalıştığına iyi bir fikir verecektir. Bir thread sistemin nasıl çalıştığını öğrenmek için bu kütüphane ile oynamaktan çekinmeyin. Unix okuyuculu olmadığından makinelerinizin multiprocessors ve threadten herhangi bir ekstra performans alamazsınız. Senin thred ile oynamaya izin verir.

Buradan **print4.c** çıktısı :

```
Trying to join with tid 0
Hi.  I'm thread 0
Hi.  I'm thread 1
Hi.  I'm thread 2
Hi.  I'm thread 3
Joined with tid 0
Trying to join with tid 1
```

```
Joined with tid 1
Trying to join with tid 2
Joined with tid 2
Trying to join with tid 3
Joined with tid 3
```

Peki şu ne oldu. **Main ()** programı 4 thread sonra kapalı çatallama kontrolü var. Buranın adı **pthread_join** thread sıfır ve engelledi. Daha sonra thread sıfır, kontrolü var çıktısını çizdi ve çıkıtı. Sonraki threadler bir, iki ve üç geldi. Onlar bittiğinde **main()** threadini tekrar kontrole aldık ve thread sıfır yapıldı yanına **pthread_join ()** döndürdü. Sonra **pthread_join ()** threadı yapıldı yana dönen bütün threadleri bir, iki ve üç için çağırır. Ne zaman **main ()** döner, bütün threadler yapılır ve programdan çıkar.

exit () ve pthread_exit ()

Pthreads thread/ program sonlanması hakkında bilmeniz gereken iki şey vardır. Birincisi **pthread_exit ()** bir thread çıkışı yapan olmasıdır , **exit ()** görevi sona erer. Bütün threadler (ve **main()** programı bir thread ı dikkate alır)sona erer , daha sonra görevi sonlandırır.Bak **p4a.c** .

Burada tüm threadler, **main ()** ile programın çıkışı **pthread_exit ()** . Bu çıkışın **print4** aynı olduğunu görürsünüz .

Şimdi, bakın **p4b.c** . Burada, join çağrısı yapmadan önce bir **pthread_exit () main ()** çağrısı yapmalıyız. Çıktısı:

```
Merhaba. Ben thread 0 değilim
Merhaba. Ben thread 1 değilim
Merhaba. Ben thread 2 değilim
Merhaba. Ben thread 3 değilim
```

Bu ana iş parçacığı çıkıldığı gerekçesiyle "birleştirilmesi" satırları hiçbiri dışarı basıldı dikkat edeceğiz. Ancak, diğer konuları sadece iyi koştu ve tüm konuları çıktıktan olduğu zaman program sonlandırıldı.

Bilmeniz gereken ikinci şey bir forked thread döndüğünde bu ilk görüşme prosedürüdür (**printme** içindeki **print4.c**, daha sonra bu **pthread_exit()** çağrılarıyla aynıdır.). Bununla birlikte eğer **main()** thread dönderirse, o **exit ()** çğrısı aynıdır ve görev biter. **p4c.c** içinden çıkış nedeni budur . Threadler 3 , 0 ile forked edildiğinde ana thread çıkar, ama henüz çalışmamaktadır. Ana thread geri döner , görevi sona erer ve böylece threadler çalıştırılmaz.

Son olarak **p4d.c** bakınız. Burada threadler için **pthread_exit ()** yerine **exit ()** diyoruz. Bunun da çıkış olduğuna dikkat edeceğiz:

```
Trying to join with tid 0
Hi.  I'm thread 0
```

Thread 0 ile **exit ()** çağrısı sonlandırılmış oluyor.

Preemption a karşı olmayan preemption

Şimdi, **iloop.c** göz atın. Burada, dört thread forked edilir , ve daha sonra **main()** parçacığı sonsuz bir döngüye girer. Bunu çalıştırdığınızda, bir şey göremezsiniz. Thread 3 thread 0 sayesinde çalıştırılır. Makinelerimizde thread sistemi önleyici olmayan olmasıdır . Diğer bir deyişle, tek bir işlemci var ve bir thread gönüllü CPU vermedikçe (Pthread_join gibi bir engelleme çağrısı üzerinden veya sonlandırma ile), CPU tutacaktır. Bir önleyici sistem içerisinde , threadler her zaman kesilir ve itfa planına bağlanan olabilir, **iloop** aslında kendi id dışarı 3 baskı (program sonlandırmasına rağmen asla) aracılığı ile threadler 0 olacak. Bazı makşneler birden fazla CPU tek bir bellek aktarıyordu.Bu sistemler doğa tarafından önleyici olarak, farklı threadleri farklı CPU üzerinde yürütmektedir. Böyle bir makine kendi id dışarı 3. baskı (program sonlandırmasına rağmen asla) aracılığıyla thread 0 olacak.

Daha sonra preemption hakkında daha fazla konuşacağız

Tek bir CPU (bir "tek işlemcili" denir) bir sistemde olmayan bir önleyici thread sistemi işe yaramaz gibi görünebilir, ama gerçekte son derece yararlıdır. Çok işlemcili bir sistem (yani birden fazla CPU bir bellek takılı oluyor) üzerinde, sizin programlar paralel speedup ulaşmak için threadleri kullanabileceğiniz açık olmalı.